Hierarchical Methods for Filtering and Visualization Based on Graphics Hardware

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

vorgelegt von

Matthias Hopf

aus München

Hauptberichter:	Prof. Dr. Thomas Ertl
Mitberichter:	Prof. Dr. Martin Rumpf
Tag der mündlichen Prüfung:	27.05.2004

Institut für Visualisierung und Interaktive Systeme
der Universität Stuttgart

2004

To my beloved parents and my dear friends.

G'Kar: (to Sakai)
Narns, Humans, Centauri ...
we all do what we do for the same reason:
because it seems like a good idea at the time.

Babylon 5, Mind War

Contents

G'Kar: *The gift of time.*
The gift of life.
The gift of wisdom.
The gift of light.
For these things we are thankful.
For these things we pray.

Babylon 5, By Any Means Necessary

Acknowledgments

This work would never have seen the light without the help and support from a lot of people. I am most thankful to my advisor Thomas Ertl for constantly pushing me towards this final goal. I am also very thankful to the German Research Foundation[1], which made the funding of this work possible with their concept of Center of Excellence[2] groups, SFB 382[3] D6 in my case.

Several aspects of this work were joint work with other scientific groups or were strongly influenced by others. Using graphics hardware for morphological operators was one of the key algorithms for a paper by Sabine Iserhardt-Bauer [2001]. The mathematical basis of using Whitney forms with sparse grids was discussed with Vasile Gradinaru from the mathematical institute of the University of Tübingen. The development of algorithms on hierarchical radial basis functions was joint work with Manfred Weiler (VIS group), Yun Jang, Jingshu Huang, and David Ebert (all from Purdue University), and Kelly P. Gaither (University of Texas). I especially would like to thank Manfred for his great support with regard to this project. Yun helped me a lot with encoding data sets for the comparison in Section 9.2.

There are other projects that have been completed successfully during my time at the VIS group, though they are not part of this thesis. Several aspects of sparse grids have also been discussed together with Christian Teitzel et al. [1998b, 1999, 2000]. There was also some basic work about flow visualization [Teitzel and Hopf 2000]. More investigations in this area have been performed together with Daniel Weiskopf [Weiskopf et al. 2001, Weiskopf et al. 2002, Weiskopf and Hopf 2002], using graphics hardware for texture advection. Joint work with Marcelo Magallón [2001] deals with the use of Commodity Of The Shelf (COTS) graphics hardware in PC cluster systems for volume visualization. Splatting of uncorrelated points in Chapter 8 revealed a brand new research topic in the VIS group, which is now actively worked on by Guido Reina. I can only say that working with all these people was truly a great pleasure.

Using graphics hardware for other purposes than graphics was a constant source for vivid discussion. Here I would like to point out Marcelo Magallón, Daniel Weiskopf, and my former colleague Rüdiger Westermann, as well as Robert Strzodka and Martin Rumpf during my visit at the University of Bonn, for their helpful discussion and ideas regarding hardware implementation issues. Additionally, I would like to thank my former colleague Christoph Lürig for providing ideas about how to accelerate hardware-based wavelet transformations. Guido Reina was

[1]German title: Deutsche Forschungsgesellschaft (DFG), `http://www.dfg.de/`
[2]German title: Sonderforschungsbereich (SFB)
[3]`http://www.uni-tuebingen.de/uni/opx/`

a constant source for inspiration regarding splatting of uncorrelated data, both during development of my system and afterwards during his own research. Peter Leinen from the University of Tübingen made the timing measurements possible by helping me with the new PC cluster ”Kepler”[4], on which several systems have been evaluated. Volker Springel and Martin Jubelgas from the Max-Planck Society in Garching gave me many details about their needs for good visualization software.

Visualization always depends on sources for data sets in order to provide meaningful images for real-life data. The 1-form data set in Chapter 6 has been supplied by Vasile Gradinaru from the mathematical institute of the University of Tübingen.

The X38 data set that has been used to test the accuracy of the RBFs (Chapter 7) was computed at the Engineering Research Center at Mississippi State University by the Simulation and Design Center. The blunt fin data set is courtesy of C. M. Hung and P. G. Buning. The bunny data set has been published by the University of Stanford. The natural convection simulation has been developed by the Computational Fluid Dynamics Laboratory at the University of Texas at Austin (UT-Austin). The oil reservoir data set has been computed by the Center for Subsurface Modeling at UT-Austin.

Some of the data sets shown in Chapter 8 are based on simulations carried out by the Virgo Supercomputing Consortium using computers based at the Computing Center of the Max-Planck Society in Garching and at the Edinburgh Parallel Computing Center. The data sets are publicly available[5]. Another data set used in this chapter shows a cosmological simulation performed at the Texas Advanced Computing Center by Hugo Martel, Paul R. Shapiro, and Marcelo Alvarez, all of the Galaxy Formation and Intergalactic Medium Research Group, Department of Astronomy, UT-Austin. The data set was provided by Bong-Soo Sohn and downloaded from the Center for Computational Visualization, Institute of Computational Engineering and Sciences (ICES), UT-Austin. Please see their web site[6] for more details. The Diesel injection data set was provided by Sebastian Niedworok from Theoretical Astrophysics (TAT) and Sven Ganzenmüller from Technical Computer Science (TI), both University of Tübingen. The galaxy formation simulation is courtesy of Martin Jubelgas from the Max-Planck Society in Garching. The reversible Apollonian packing and the molecular dynamics simulation have been created by Reza Mahmoodi Baram from the Institute of Computer Applications (ICA) and Johannes Roth from the Institute of Theoretical and Applied Physics (ITAP), respectively, both University of Stuttgart. I am very grateful for their support with data sets and insight into their visualization needs.

For proofreading I would like to thank my colleagues and friends Katrin Bidmon, Thomas Klein, Guido Reina, Bettina Salzer, and Manfred Weiler. Your suggestions have been very valuable!

Many thanks to our secretary Ulrike Ritzmann for all her support, and to Hermann Kreppein for his help in administrative work and purchasing of hardware.

I was really blessed with my room mates Martin Kraus and Thomas Klein. Thank you both for being great and supportive friends. We will always remember our time in ”Tom’s Waiting

[4]`http://kepler.sfb382-zdv.uni-tuebingen.de/`
[5]`http://www.mpa-garching.mpg.de/NumCos/`
[6]`http://www.ticam.utexas.edu/CCV/`

Room” in the old building with all its deficiencies and smells, as well as the temperature in the new building during hot summer days. My sincere condolences to Manfred for his tropic aquarium office.

A big warm thank you goes to my parents Hermann and Renate Hopf for supporting me all the time to the best of their possibilities. The whole VIS group is especially thankful to Renate for her great cooking abilities, and awarded her with an informal ”Dr. rer. Back”.

All introductory cites have been taken from Babylon 5, the best Science Fiction show of all times, created by J. Michael Straczynski. Thank you for your inspiration!

And finally I want to thank all my dear friends of the last years — for being my friends.

Abstract and Chapter Summaries

Abstract

Interactive visualization of large data sets is only possible with efficient algorithms for all parts of the visualization pipeline. This thesis analyzes the filtering and the rendering steps of this pipeline for several fundamentally different data types. Two key techniques that are employed throughout this work are the use of hierarchical methods and graphics-hardware-based implementations of the presented algorithms.

In order to improve the efficiency of filtering, both linear and nonlinear filters are accelerated using graphics hardware for the computation. For many algorithms hierarchical filters based on wavelets are needed and, therefore, inspected as well. Finally, the quality of the achieved results is analyzed, as the accuracy of graphics-card-based approaches is limited by register sizes and framebuffer depths.

During rendering hierarchical approaches allow for a compact representation of partially detailed data. Additionally, the user can trade visualization speed for quality. Sparse grids allow for extremely compact representations, thus interpolated data from sparse grids would no longer fit into main memory. This raises the need for visualization algorithms working directly on the sparse grid coefficients. The interpolation process is expensive, but graphics hardware is usually too inaccurate to be used for acceleration. Thus, the rendering process is parallelized with MPI, using a ray distribution scheme that implicitly generates previews with lower resolution during rendering. For unstructured data a compact hierarchical representation with radial basis functions is introduced that can be employed for rendering at interactive frame rates using graphics hardware.

Completely uncorrelated data like astrophysical n-body problems have high spatial resolution which is lost during resampling for volume rendering. A new hierarchical splatting approach is presented that is able to visualize tens of millions of points interactively for steady and time-dependent data sets.

The used data representations have different approximation capabilities, thus the properties of the different data encodings are analyzed by comparing several data sets that exhibit different amounts of features.

Chapter Summaries

Chapter 1 introduces the visualization pipeline and motivates hierarchical methods and graphics-hardware-based algorithms. Additionally, it lists all cooperations that have been involved in the creation process of this thesis.

Chapter 2 explains basic technologies and algorithmic approaches that are needed throughout this work. Overviews about graphics hardware and its rendering pipeline are presented as well as general ideas about hierarchical methods and the mathematical basis of volume visualization. As most of this work deals with three-dimensional data sets, the employed volume rendering algorithms are explained and characterized according to the order of their basic operation steps.

Linear filters are addressed in Chapter 3. Filtering is widely used for reducing noise, enhancing detail structures, and feature extraction. Segmentation processes are usually based on filters as well. Discrete linear filters (convolutions) for two-dimensional images are part of the imaging extension of OpenGL, this chapter deals with the implementation of a three-dimensional volume filter based on these image filters. Texture coordinate details are analyzed, followed by the performance comparison to a software solution.

Chapter 4 deals with one special kind of nonlinear filters, the so-called morphological operators. Compared to the previous approach this type of filter cannot be expressed with convolutions. As applying morphological filters is a completely memory bandwidth bound problem, the speedup gained by using graphics hardware for the filtering process is much higher compared to linear filters. The chapter deals with the decomposition of morphological operators into lower dimensional filters, hardware-based filtering implementation details, and overlapping texture memory writes. Finally, the performance of hardware- and software-based algorithms is evaluated for the SGI Octane and a modern PC with a GeForce FX graphics card.

More complex operators are often described with hierarchical filter types. One of the best understood multiresolution techniques is the wavelet analysis. In Chapter 5 its discrete decomposition and reconstruction steps are analyzed and accelerated using graphics hardware. For implementing wavelets on fixed function graphics pipelines, a mathematical model of the OpenGL pipeline is developed, and the wavelet filtering steps are mapped to this model. Three different approaches are examined; two use different data mappings on the imaging pipeline, and the third uses the programmable fragment pipeline of modern PC graphics cards to further accelerate the process. During wavelet decomposition the evaluated coefficients may exceed the range $[0,1]$ supported by the framebuffer, thus a scaled and biased version of the wavelet analysis is developed, leading to different filter specifications for even and odd pixel positions. Finally, performance and accuracy is analyzed on both SGI Octane and modern PC hardware with respect to register sizes and framebuffer depths.

Another multiresolution analysis is investigated in Chapter 6. The so-called sparse grids have theoretically ideal approximation quality with respect to the number of basis functions, which makes them perfect candidates for computational mathematics. In this chapter the visualization of data on these sparse grids is accelerated by parallelization using MPI, as graphics hardware is usually not accurate enough to cope with sparse grids of higher levels. New approaches employ Whitney forms on sparse grids for the representation of 1-forms like magnetic fields. As interpolation on these data sets is extremely expensive, efficient visualization is only possible if previews can be presented to the user during interaction. The developed ray assignment function for the used raycasting algorithm is able to generate these previews implicitly and balances the load almost perfectly without the need for an additional master node for distributing the rays to the render nodes. Finally, some figures demonstrating the performance scaling and load balancing quality are presented.

Sparse grids are not good at approximating data sets given on structured or unstructured grids, which leads to the introduction of radial basis functions as another hierarchical data representation in Chapter 7. The parameters of the radial basis functions are determined by a clustering approach that leads to octree-based grid structures. In order to exploit the graphics adapter for rendering data sets based on these basis functions the data has to be encoded in textures for fast lookup from the decoding fragment programs. Often a single octree cell contains more basis functions than can be handled in a single rendering step, and intermediate images have to be rendered to a P-buffer. The evaluation of the basis functions itself is embedded in the fragment program used for decoding the texture data. Finally, several data sets are encoded and analyzed using this approach.

For some data sets like solutions of astrophysical n-body problems the most important information is not correlated with the scalar or vectorial data represented by the basis functions, but the mere positions of the basis functions themselves. In this case any interpolation-based representation of the data set reduces the spatial resolution inherent to the data. Chapter 8 introduces a splatting approach that renders the basis functions directly as points and is thus not affected by interpolation. The points are stored in a hierarchy that is created using principal component analysis and compressed employing quantization of relative coordinates. In the time-dependent case, the quantized data is additionally Lempel-Ziv compressed and loaded on demand from the hard disk. These out-of-core techniques are needed for the interactive visualization of time-varying data sets, because their size is usually too large for the available main memory, and only one set of spline parameters is necessary for evaluation of particle positions at any given time step. The combination of these techniques allow for a smooth visualization of four-dimensional data sets. In order to render several points with a single OpenGL call, point data and hierarchical cluster structures are stored separately. During visualization the hierarchy is traversed recursively and rendered adaptively according to an appropriate maximum screen error metrics. However, rendering semitransparent objects requires the objects to be sorted before drawing. Several sorting techniques are implemented and compared, and the shortcomings of the presented approximative approaches are analyzed. Finally, vertex programs are used for decoding relative coordinates, dequantization, and attenuation in order to accelerate the rendering process with graphics hardware. For comparison several other rendering techniques are presented and evaluated.

Chapter 9 concludes this thesis with an analysis of the presented algorithms and approaches. As several fundamentally different data types have been encountered in the previous chapters, their properties and suitability for different kinds of data is analyzed considering four different data sets representing the varying information density in the data. Finally, an overview over topics of future research is presented.

Zusammenfassung und Kapitelübersicht

Zusammenfassung

Die interaktive Visualisierung großer Datensätze kann nur durch den Einsatz effizienter Algorithmen in allen Teilen der Visualisierungspipeline gelingen. In dieser Dissertation werden die Schritte Filterung und Rendering für einige fundamental unterschiedliche Datentypen analysiert. Die beiden Haupttechniken, die in dieser Arbeit eingesetzt werden, sind die Verwendung hierarchischer Methoden sowie die Anpassung der präsentierten Algorithmen für die Ausführung des Programmcodes auf Graphikkarten.

Um die Effizienz von Filtertechniken zu steigern, können sowohl lineare als auch nichtlineare Filter mit Hilfe von Graphikhardware beschleunigt werden. Viele Algorithmen bauen auf hierarchischen Filtern wie den Wavelets auf, deshalb werden auch diese untersucht. Ein weiteres Augenmerk liegt auf der Qualität der erreichten Ergebnisse, da alle Ansätze, die Graphikkarten für ihre Berechnungen benutzen, bezüglich ihrer Genauigkeit durch die Registerbreite und die Framebuffertiefe limitiert sind.

Hierarchische Ansätze können bei nur partiell hochaufgelösten Daten helfen, diese in einer kompakten Repräsentation zu speichern. Außerdem kann der Benutzer / die Benutzerin sich zugunsten einer schnelleren Darstellung für niedrigere Qualitätsstufen entscheiden. Dünne Gitter erlauben eine extrem kompakte Speicherung und würden auf vollen Gittern interpoliert nicht mehr in den Hauptspeicher passen. Deshalb müssen Datensätze auf diesen Gittern direkt dargestellt werden. Die Interpolation ist aufwändig, und Graphikhardware ist normalerweise zu ungenau, um für die Beschleunigung der Berechnungen benutzt werden zu können. Deshalb wird der Rendering-Prozess mit Hilfe von MPI parallelisiert. Dabei wird ein Strahlverteilungsverfahren benutzt, das während der Berechnung implizit eine progressive Vorschau in niedrigerer Auflösung erzeugt. Für unstrukturierte Daten wird eine kompakte hierarchische Repräsentation eingeführt, die radiale Basisfunktionen einsetzt. Diese Darstellung kann für die Visualisierung mit Hilfe von Graphikhardware verwendet werden, wobei interaktive Frameraten erzielt werden.

Vollkommen unkorrelierte Daten, wie sie in der Astrophysik bei n-Körper-Problemen auftreten, haben eine hohe Ortsauflösung, welche bei der Neuabtastung für die direkte Volumenvisualisierung verloren geht. Ein neuer Splatting-basierter hierarchischer Ansatz wird vorgestellt, der

es ermöglicht, mehrere Millionen Punkte interaktiv zu visualisieren, sowohl für den statischen als auch den zeitabhängigen Fall.

Die benutzten Datenrepräsentationen haben unterschiedliche Näherungseigenschaften. Aus diesem Grund werden die Charakteristiken dieser Repräsentationen mit Hilfe verschiedener Datensätze mit unterschiedlich vielen Merkmalen untersucht.

Kapitelübersicht

In Kapitel 1 wird die Visualisierungpipeline eingeführt, sowie die Verwendung von hierarchischen Methoden und das Ausnutzen von Grafikhardware durch spezielle Algorithmen motiviert. Außerdem werden alle Kooperationen aufgeführt, die während der Erstellung dieser Dissertation eine Rolle gespielt haben.

Kapitel 2 erläutert die Basistechnologien und algorithmischen Ansätze, die in der gesamten Arbeit benutzt werden. Dabei werden sowohl Graphikhardware samt dazugehöriger Renderingpipeline dargestellt, als auch allgemeine Ideen über hierarchische Methoden erörtert und die mathematische Basis der Volumenvisualisierung gelegt. Da diese Arbeit im Wesentlichen mit dreidimensionalen Datensätzen arbeitet, werden die benutzen Rendering-Algorithmen erklärt und anhand der Reihenfolge ihrer Hauptoperationen verglichen.

Lineare Filter werden in Kapitel 3 behandelt. Filterung wird hauptsächlich dazu benutzt, Rauschen zu reduzieren, Details hervorzuheben und Merkmale zu extrahieren. Segmentierung baut typischerweise auch auf Filterprozessen auf. Diskrete lineare Filter (Faltungen) für zweidimensionale Bilder sind bereits in der Imaging Erweiterung von OpenGL enthalten. Dieses Kapitel behandelt die Implementierung dreidimensionaler Volumenfilter basierend auf diesen zweidimensionalen Bildfiltern. Details der Texturkoordinatengenerierung werden analysiert, und die erzielte Geschwindigkeit des Algorithmus wird mit einer Softwarelösung verglichen.

Kapitel 4 behandelt einen Sonderfall nichtlinearer Filter, die sogenannten morphologischen Operatoren. Verglichen mit dem vorhergehenden Ansatz kann diese Art Filter nicht mit Hilfe von Faltungen dargestellt werden. Da die Anwendung morphologischer Operatoren vollständig speicherbandbreitenbegrenzt ist, kann man im Vergleich zu linearen Filtern eine deutlich höhere Beschleunigung erhalten, wenn man Graphikhardware für den Filterprozess einsetzt. In diesem Kapitel wird die Zerlegung morphologischer Operatoren in niederdimensionale Operatoren behandelt, Details der Implementierung einer hardwarebasierten Filterung aufgezeigt und das Problem überlappender Texturspeicherzugriffe behandelt. Zum Schluß wird die Geschwindigkeit von Hardware- und Softwarelösungen verglichen, sowohl für SGI Octanes, als auch für moderne PCs mit einer GeForce FX Karte.

Komplexere Operatoren werden oft mit Hilfe hierarchischer Filter beschrieben. Eines der bestverstandenen Multiresolution-Verfahren ist die Wavelet Analysis. Die dazugehörigen Dekompositions- und Rekonstruktionsschritte werden in Kapitel 5 analysiert und anschließend mit Hilfe von Graphikhardware beschleunigt. Um Wavelets auf Graphikpipelines mit statischer Funktionalität implementieren zu können, wird ein mathematisches Modell der OpenGL Pipeline erstellt und die Wavelet-Filterschritte auf dieses abgebildet. Es werden drei verschiedene Ansätze untersucht, zwei, die auf der Imaging Pipeline mit unterschiedlichen Datenmodellen arbeiten, und eines, das mit Hilfe programmierbarer Fragmenteinheiten moderner PC-Graphikkarten den Prozess noch weiter beschleunigt. Während der Wavelet-Zerlegung können die ausgewerteten Koeffizienten den Bereich $[0;1]$ des Framebuffers verlassen, deshalb wird eine skalierte und verschobene Version der Wavelet Analysis entwickelt. Dies führt zu unterschiedlichen Filterspezifikationen für gerade und ungerade Pixelpositionen. Schließlich wird die Performanz und die Genauigkeit sowohl für die SGI Octane Serie als auch für moderne PC Hardware in Abhängigkeit von Registergröße und Framebuffertiefe analysiert.

Eine weitere Multiresolution Analysis wird in Kapitel 6 untersucht. Die sogenannten Dünnen Gitter haben theoretisch optimale Approximationseigenschaften für die verwendete Anzahl an Basisfunktionen. Deshalb sind sie ideale Kandidaten für mathematische Berechnungen. In diesem Kapitel wird die Visualisierung von Daten, die auf diesen Gittern gegeben sind, mit Hilfe von Parallelisierung beschleunigt, da Graphikhardware für Dünne Gitter höherer Level meist nicht genau genug arbeitet. Neue Ansätze benutzen Whitney Formen auf Dünnen Gittern für die Darstellung von 1-Formen wie zum Beispiel magnetischen Feldern. Da die Interpolation auf diesen Datensätzen extrem aufwändig ist, können sie nur effizient visualisiert werden, wenn dem Benutzer / der Benutzerin während der Interaktion eine Vorschau auf das finale Ergebnis geliefert werden kann. Das präsentierte Strahlenselektionsverfahren für den verwendeten Raycasting-Algorithmus kann eine progressive Vorschau implizit erstellen und sorgt dabei gleichzeitig für eine fast perfekte Lastverteilung. Das Verfahren hat die Eigenschaft, keinen dedizierten Master-Knoten für die Strahlverteilung an die Render-Knoten zu benötigen. Zum Schluß werden die Skalierungseigenschaften und die Qualität der Lastverteilung des vorgestellten Algorithmus dargestellt.

Dünne Gitter können Daten, die auf uniformen oder unstrukturierten Gittern vorliegen, nur schlecht approximieren, deshalb werden in Kapitel 7 radiale Basisfunktionen als eine weitere hierarchische Datenrepräsentation eingeführt. Die Parameter der radialen Basisfunktionen werden dabei mit Hilfe eines Clustering-Verfahrens ermittelt, welches zu einer Octree-basierten Gitterstruktur führt. Um Graphikkarten für die Darstellung dieser Datensätze verwenden zu können, müssen die Daten in Texturen gespeichert werden, damit auf sie vom dekodierenden Fragmentprogramm aus schnell zugegriffen werden kann. Oft enthält eine einzelne Octree-Zelle mehr Basisfunktionen als in einem Rendering-Schritt ausgewertet werden können. In diesem Fall müssen Zwischenbilder in einen P-Buffer gezeichnet werden. Die Auswertung der Basisfunktionen selber ist zusammen mit der Dekodierung der Texturdaten als Fragmentprogramm implementiert. Schließlich wird das Verfahren anhand mehrerer Datensätze analysiert.

Bei manchen Datensätzen ist die wichtigste Information nicht das Skalarfeld oder die Vektordaten, die durch die Basisfunktionen repräsentiert werden, sondern die Positionen eben dieser Basisfunktionen. Ein prominentes Beispiel für derartige Datensätze sind Lösungen des astrophysikalischen n-Körper-Problems. Für diese Daten schlecht geeignet sind Repräsentationen, die auf der Interpolation zwischen festen Stützstellen aufbauen, da sie die räumliche Auflösung reduzieren. In Kapitel 8 wird ein Splatting-Verfahren eingeführt, das die Basisfunktionen direkt als Punkte zeichnet und deshalb nicht durch Interpolation beeinflusst wird. Die Punkte sind in einer

Hierarchie gespeichert, die durch eine Hauptachsenzerlegung erzeugt und anschließend mit Hilfe von Quantisierung und relativen Koordinaten komprimiert wird. Für zeitabhängige Datensätze werden die Daten zusätzlich noch mit Hilfe des Lempel-Ziv Algorithmus komprimiert. Um bei der Visualisierung zeitabhängiger Daten glatte Partikelbahnen zu erhalten, werden Splines für die Ermittlung der Punktpositionen ausgewertet. Mit einem einzigen OpenGL-Aufruf sollen dabei mehrere Punkte gezeichnet werden. Dazu müssen Punktdaten und Clusterinformationen in getrennten Datenstrukturen gespeichert werden. Während der Visualisierung wird dann rekursiv in der Hierarchie abgestiegen und die Daten abhängig von einer Metrik, die den maximalen Bildfehler berücksichtigt, adaptiv gerendert. Zeitabhängige Datensätze passen normalerweise nicht in den Hauptspeicher, deshalb müssen in diesem Fall Out-of-Core-Techniken benutzt und die Daten erst bei Bedarf von der Festplatte geladen werden. Für semitransparente Darstellungen müssen alle zu zeichnenden Objekte bezüglich des Betrachterabstands sortiert werden. Für den präsentierten Algorithmus werden daher mehrere Sortierverfahren betrachtet und verglichen. Dabei werden vor allem die Schwächen der vorgestellten nur näherungsweise korrekten Verfahren untersucht. Des weiteren werden Vertexprogramme für die Transformation der relativen Koordinaten, die Dequantisierung und die abstandsabhängige Abschwächung eingesetzt, um den Darstellungsprozess mit Hilfe von Graphikhardware zu beschleunigen. Zum Vergleich werden außerdem einige andere Darstellungsmethoden präsentiert und bewertet.

Kapitel 9 schließt die Dissertation mit einer Analyse der dargestellten Algorithmen und Ansätze ab. Da in der Arbeit fundamental unterschiedliche Basen zur Darstellung der Daten benutzt wurden, werden die Eigenschaften dieser Basen sowie ihre Eignung zur Repräsentation unterschiedlicher Arten von Daten analysiert. Dafür werden exemplarisch vier unterschiedliche Datensätze zur Visualisierung herangezogen, die verschiedene Merkmalsdichten aufweisen. Zum Schluß wird eine Übersicht über zukünftige Forschungsmöglichkeiten gegeben.

Delenn: (to Sheridan)
Universe puts us in places where we can learn.
They are never easy places, but they are right.
Wherever we are, it's the right place ... and the right time.
Pain that sometimes comes is part of the process
of constantly being born.

Babylon 5, A Distant Star

Chapter 1

Introduction

Visualization of scientific data has become an integral part in research projects as well as in real-world product design and development. The data sets, coming from numerical simulations and real measurements, are very often so large, that analyzing them without visualization software is no longer an option. Due to the development of faster and larger supercomputers, better numerical algorithms, and improved high-resolution scanners, the data sizes are increasing at a pace trivial visualization algorithms cannot keep up with.

It is one of the most fundamental convictions of the visualization community that one can get the best impressions about large and complicated structures when the according data sets can be viewed interactively. Due to their complexity, it is often not sufficient to accelerate the rendering process itself to at least 10 frames per second. Instead, the speed of a full visualization cycle has to be improved. Figure 1.1 shows that interaction influences all steps of the visualization pipeline, and not only the rendering itself.

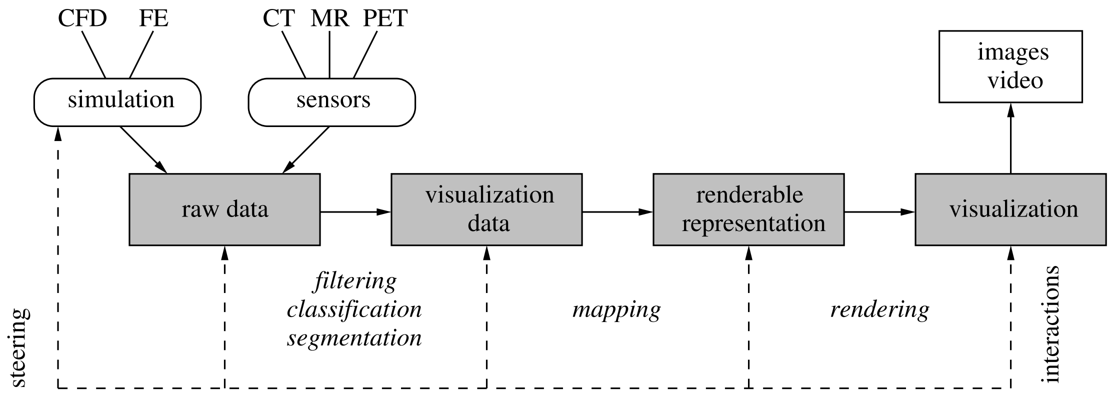


Figure 1.1: The visualization pipeline.

So far many researchers have concentrated on accelerating the rendering of different types of data. For many applications, even state-of-the-art hardware is barely capable of displaying medium-sized data sets, let alone at interactive speed. Thus, more sophisticated techniques have to be developed to match the ever increasing requirements.

1.1 Motivation

In order to visualize extremely large data sets interactively, we have to access this problem from several opposite directions. On the one hand, we have to develop new algorithms that reduce the amount of work done for invisible details of the data, which leads to hierarchical methods. On the other hand, we have to approach these algorithms from the implementation point of view, and accelerate them using the available resources, which leads to the implementation of graphics-hardware-based algorithms.

As mentioned before, the complete visualization pipeline has to be accelerated. By implementing algorithms for the opposite ends of the pipeline, the filtering and the rendering step, I will demonstrate that modern graphics hardware can be used for this aim. In the long term we do not have to pass the data from main memory to graphics memory and back, if we manage to implement the complete data flow on the graphics processing unit (GPU).

1.1.1 Hierarchical Methods

Large data sets impose two major problems on visualization algorithms, especially on those that are based on regular grids: memory consumption and processing speed. Note that memory requirements have a large impact on processing times, as larger memory systems (like main memory or even hard discs) have typically much higher latencies and lower throughput compared to smaller memory systems (first and second level cache, graphics memory).

In order to achieve higher local memory coherency for better cache utilization, data should be stored and processed in different resolutions. With hierarchical techniques we can easily adapt the used resolution to the available processing power, and we can even improve the visual quality of renderings as we reduce aliasing by adapting the data decomposition to the screen resolution. In some cases we can even lower the memory requirements with this approach, though usually the introduction of hierarchies increases the memory footprints of data sets. Often visualization quality can be tracded for speed with adaptive algorithms, which can be used for improving the system's interaction behavior.

By representing the original data using hierarchical basis functions, another aspect of multiresolution analysis is important, especially for filtering and segmentation processes. Each level of the hierarchy represents different frequencies of the original data due to the different scales of the basis functions. Using this information e.g. with a wavelet basis (Chapter 5) the features can be separated with much higher confidence compared to simpler filtering techniques [Westermann and Ertl 1997].

In some cases (e.g. with Sparse Grids, Chapter 6) the used simulation systems are already producing hierarchical data. Here it would be necessary to resample these data on a regular grid for most visualization systems. Especially in the case of Sparse Grids an enormous increase of memory is required, thus algorithms have to be developed that work directly on these hierarchical data structures.

While using hierarchical data structures has been combined successfully with graphics hardware in the case of structured grids [LaMar et al. 1999, Weiler et al. 2000a], there are many other areas that have not been addressed so far.

1.1.2 Graphics-Hardware-Based Algorithms

Due to the demand for improved realism in computer games, GPUs are changing from former fixed function pipeline accelerators into almost general purpose data processing units. So far, graphics hardware has been employed for the rendering step of the visualization pipeline for a long time, and more recent researches [Rezk-Salama et al. 2000, Roettger et al. 2000, Engel et al. 2001, Weiler et al. 2003b] have successfully implemented the mapping step of many algorithms on GPUs as well. As filtering can be seen as a mass data processing step, it is also a perfect candidate for being implemented using a SIMD equivalent processing paradigm on graphics hardware. Until the beginning of this work in 1998 surprisingly little work was done in this direction. A good overview over recent researches can be found on Mark Harris' web site[1] about "General Purpose Computation Using Graphics Hardware".

There are several conditions that motivate the use of graphics hardware for all steps of the visualization pipeline:

- GPUs have more transistors than CPUs (Pentium 4: 42 million, GeForce FX5900: 135 million), and, due to their less complicated structure, they have many more functional units. Additionally, the speed of graphics hardware is currently increasing faster than Moore's law — though it is foreseeable, that this effect will not last much longer.
- GPUs have wider and faster data paths to their memory. Additionally, the chips are directly soldered on the boards, which reduces parasitic capacities, leading to higher clock frequencies and extremely high memory bandwidths.
- Graphics hardware is mainly optimized for image and texture processing, which leads to deep pipelining and special caching algorithms that will decrease the chance of stalls for more complicated arithmetic operations.

Optimized data paths are placed in the fragment path which can only be used for very regular data access. For more irregular data, per-vertex operations have a greater flexibility but are less optimized in terms of speed and memory access. However, their arithmetic throughput is much higher than the raw processing power of the main processor.

The superior performance of graphics hardware compared to CPU-based algorithms can be seen most prominently in the large success of texture-based volume rendering, for example in [Rezk-Salama et al. 2000, Kniss et al. 2001, Guthe et al. 2002]. As we will see in Chapter 9, the gap between purely processor-based and graphics-hardware-based implementations is still widening.

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

As soon as we have a GPU-based implementation of the algorithms, we can accelerate it even more by balancing the CPU and the GPU work load. This has already been shown as feasible, e.g. in [Weiler and Ertl 2001].

1.2 Context and Cooperation

Most of the work for this dissertation has been performed at the Visualization and Interactive Systems Group (VIS) in Stuttgart, financed by the SFB 382 project D6 from the German Research Foundation (DFG), though it has been started at the Computer Graphics Group in Erlangen, financed by the SFB 603.

The former SFB had a great influence on this work, as its research aims at adaptive methods and hierarchical data structures for the integration of data analysis and visualization[2].

Due to the structure of the SFB (combining projects related to physics, mathematics, and computer science) several cooperations were possible. The visualization of sparse grids in Chapter 6, inspired by work in my diploma thesis [Hopf 1998], has been greatly influenced by cooperations with project C10, which deals with sparse grids for the calculation of electromagnetic fields. Projects C14 (MD simulations of quasi crystals), C6 (Object oriented parallelization), C13 (PIC-MC particle simulations), and C16 (Smoothed particle hydrodynamics) had a vital need for visualization algorithms capable of displaying millions of scattered data points (Chapter 8) and were very helpful with providing point-based data sets.

Another exquisite source of grid-less data sets has been the Max-Planck Institute for Astro Physics in Garching. Many of their data sets presented in this paper are publicly available on the web pages of the Virgo Consortium as stated in the acknowledgments.

The hardware acceleration of mapping techniques, especially in the context of volume rendering, is ongoing research of a number of people in the VIS group, namely Katrin Bidmon, Joachim Diepstraten, Sabine Iserhard-Bauer, Guido Reina, Stefan Röttger, Simon Stegmaier, and Manfred Weiler. It has been an important part of the Ph.D. theses of former colleagues, namely Klaus Engel, Martin Kraus, and Jürgen Schulze-Doebold. Additionally, there is currently research done by Stefan Guthe (SFB 382, project D1, University of Tübingen) and Christoph Rezk-Salama (University of Siegen). Therefore, I decided to focus on the two ends of the visualization pipeline and to completely dismiss research on the acceleration of the mapping step.

Many aspects on the visualization of time-dependent grid-less data[3] have been analyzed in a student's work by Michael Luttenberger [2003].

[2]Original German title: Adaptive Verfahren und Hierarchische Datenstrukturen zur Integration von Datenanalyse und Visualisierung

[3]Original German title: Visualisierung zeitabhängiger gitterloser Daten

G'Kar: *The universe is driven by the complex interaction between three ingredients: matter, energy, and enlightened self-interest.*

Babylon 5, Survivors

Chapter 2

Background

2.1 3D Graphics Revisited

Some decades ago graphics hardware was only used for displaying two-dimensional vector plots and later two-dimensional raster images on a monitor. The technological advances since then made this problem trivial, and whenever we refer to graphics hardware nowadays, we generally talk about hardware that is able to help the CPU in rendering 2D projections of three-dimensional scenes.

In this context 3D scenes are usually represented as surfaces by a set of primitives with accompanying vertices and meta information about surface attributes. The primitives can be given explicitely or are generated by subdividing complex objects. This tessellation is currently not accelerated by any hardware architecture and has to be performed on the CPU. The meta information usually consists of color, opacity, and normals that are specified per vertex on the one hand, and of blending and rendering modes on the other hand.

As the processing of primitives takes a lot of time, realistic surfaces cannot be modeled by changing attributes per vertex and interpolating them in-between. Instead we have to change them for each and every fragment. This can be done by using images that are projected onto the drawn primitives. At first, these so-called textures were only used for changing the fragment color, but the potential of changing any attributes (e.g. opacity for billboards, normals for bump mapping) was recognized soon.

2.1.1 The Rendering Pipeline

Figure 2.1 shows an overview over the standard rendering pipeline. The individual stages are described here briefly, for a more thorough theoretical introduction I refer to a graphics programming text book like [Foley et al. 1993]. The pipeline description in the following sections is mainly based on OpenGL [Shreiner et al. 2004], but the basic principles hold for other programming interfaces as well.

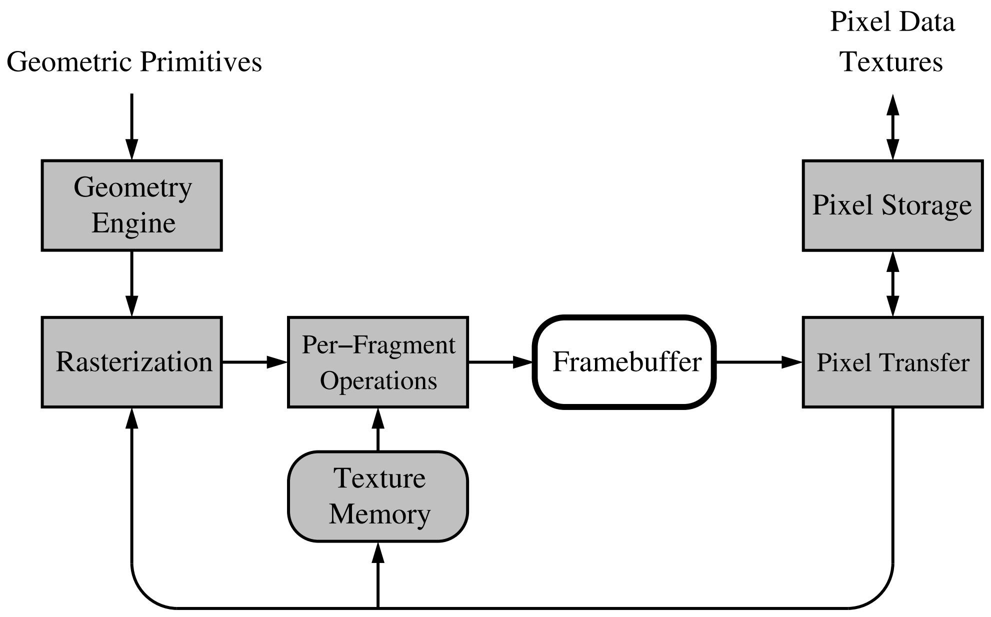


Figure 2.1: Overview over the standard rendering pipeline.

2.1.2 Vertex Processing

All geometric primitives enter the pipeline as a stream of homogeneous vertices with attributes. The vertices are transformed from object space to screen space coordinates and lighting information is calculated:

Modelview transformation

Primitives are usually given in object space, and their vertices have to be converted into world space, as e.g. light positions are known in world coordinates as well. This modeling transformation is regularly coupled with the viewing transformation, transforming world space into eye space. This so-called modelview transformation can be expressed as an affine 4×4 matrix.

Lighting

Usually, rendered polygons are lit using a Phong model. The calculation of lighting conditions has to be performed before the projection transformation as it needs a linear space for its computations.

Perspective projection

Finally, the vertices are projected onto the image plane into clip space by a 4×4 projection matrix.

2.1.3 Primitive Assembly & Rasterization

After projection the vertices are converted into fragments. Almost all currently available graphics hardware uses triangles for rendering only. One notable exception is the PowerVR chip series[1] from Imagination Technologies, which uses tile-based rendering of scenes described by plane equations. Some early PC graphics cards were able to render higher order surfaces, for instance NVIDIA's NV1, but the complexity of interpolation, rasterization, and blending of self-intersecting surfaces almost ruined the company at that time. Thus, higher order surfaces are broken down into triangles before rasterization for all current high-performance GPUs.

The rasterization process changes the number of primitives, and the complete graphics pipeline is split into two different parts (vertex and fragment pipeline) at this step. Primitive assembly and rasterization can not be reconfigured or even reprogrammed with current hardware as compared to the previous and next building blocks of the pipeline. Even though this block spans both parts of the pipeline, its steps are closely coupled:

Primitive assembly
For rasterization the individual vertices of the vertex pipeline have to be grouped into primitives again. At this point, tessellation of more complex primitives into triangles is performed, too.

Clipping
All primitives are clipped against the viewing frustum. This can — again — increase the number of triangles. The attributes of the vertices have to be adjusted as well.

Homogeneous division & viewport transformation
After clipping the homogeneous coordinates can be transformed into window space.

Rasterization
All fragments that lie in the interior of the incoming triangles are generated. This is the start of the fragment pipeline.

Attribute interpolation
As attributes are only given at vertices so far, they have to be interpolated for the fragments in-between.

2.1.4 Fragment Processing

For changing the attributes on a per-fragment level, textures can be applied to the fragments. As more fragments than vertices are usually involved in rendering even complex 3D scenes, this part of the pipeline is often the bottleneck of hardware-based visualization algorithms:

[1]`http://www.powervr.com/`

Texture lookup
The interpolated texture coordinates (which are attributes themselves) can be used to look up additional attributes for the current fragment. As the texture coordinates usually do not hit texel positions exactly, nearest neighbor or bilinear interpolation is used in this process.

Texture application
Additional attributes looked up in the previous step can be used in various modes to modulate other attributes.

2.1.5 Compositing

Finally, the fragment color and depth attributes have to be combined with the data that is already present in the framebuffer:

Alpha, stencil, depth test
The current fragment's opacity, depth, and stencil values can be tested against their counterparts in the framebuffer. The fragment is only processed further if all tests succeeded. For 3D graphics the most important test is the depth test, which ensures the correct drawing order of the rendered primitives.

Blending
The incoming fragment color is either written directly to the framebuffer or blended with the framebuffer color using the over operator (alpha blending). Some other less intuitive blending functions are available as well.

2.1.6 Imaging Pipeline

The pipeline described so far is used for rendering three-dimensional primitives to the screen. For rendering two-dimensional images, for uploading textures, and for downloading parts of the framebuffer another more flexible rasterization process is used. Usually, this is referred to as the imaging pipeline:

Pixel storage
During reading and writing pixels from and to application memory, the data has to be converted from the application defined memory layout into the basic data types understood by the GPU and vice versa. Additionally, this stage is responsible for transforming different color spaces into each other, e.g. luminance into RGBA.

Pixel transfer
The pixel transfer stage contains several image manipulation operations. It allows for scaling and biasing the original data, several different color table lookups, 1D and 2D convolutions, color matrix multiplication, and the production of histograms as well as the determination of maxima. In OpenGL most of these steps are part of the optional imaging subset. In many cases, they are not hardware-accelerated on PC graphics cards.

2.1.7 Application Programming Interfaces

For accessing the graphics system in a hardware-independent manner, all device specific programming has to be encapsulated in system libraries that are accessible through an Application Programming Interface (API). APIs can be divided into two categories, high-level APIs such as Cosmo, Optimizer, Inventor, Performer, OpenSceneGraph, OpenSG, and low-level APIs, of which the most important are OpenGL and DirectX. High-level interfaces use low-level interfaces for the actual rendering internally, and as the presented algorithms do not depend on specific features of high-level APIs, I will concentrate on OpenGL in this work. As DirectX is only a de-facto standard and not available for other operating systems than Windows I will not discuss it any further.

As OpenGL is defined by a vendor-neutral Architecture Review Board (ARB), the introduction of new features into the core functionality is rather slow. Therefore many vendors have added additional functions to their implementation of OpenGL by so-called extensions. Many features of modern PC graphics hardware are still accessible by these extensions only.

2.1.8 Programmable Graphics Hardware

For many rendering styles and lighting effects a fixed function pipeline as described above is not sufficient. Especially the game industry demanded new functionalities for greater realism, for instance bump mapping and Phong instead of Gouraud shading.

Register combiners

In order to improve the situation, NVIDIA introduced register combiners with their GeForce256 graphics card which implement the concept of a configurable fragment pipeline that would replace the former fixed function pipeline (Section 2.1.4). With this extension the programmer could route incoming fragment attributes through a set of combiners that resembled certain selectable mathematical operators, e.g. dot products. The main limitations of this concept besides its complexity was the number of combiners and the small range of operators to choose from. Additionally, the use of dependent texture lookups (texture coordinates for a lookup depend on other texture lookups) was only possible with additional extensions that complicated the programming model even more.

Vertex programs

The next logical step was to improve the flexibility of the vertex pipeline. Vertex programs were introduced with later GeForce drivers, which resemble short assembler-type programs without conditionals, jumps, and loops. These programs are evaluated instead of the fixed function pipeline (Section 2.1.2). The instruction set is optimized for 4D vertex processing in floating point only, with many special instructions that carry out complex operations needed for lighting calculations. Vertex programs can be fully pipelined, as vertices can be neither absorbed nor generated in the program, and no state information can be passed to successive vertices.

Fragment programs

It turned out soon that a configurable fragment pipeline was not flexible enough. Thus pixel shader, fragment shader, or fragment programs — naming depending on the vendor and API — were implemented as a replacement of the configurable fragment pipeline. On the bottom line, fragment programs are much like vertex programs, with special instructions for accessing textures and more flexibility with the used data types (e.g. 16 bit floats vs. 32 bit floats). As it is now possible to calculate arbitrary functions on a per-fragment level, textures and framebuffers with a higher data resolution had to be implemented as well, for instance floating point textures.

So far, rasterization (Section 2.1.3) and compositing (Section 2.1.5) remain a fixed-function part of the pipeline, but it is foreseeable that the functionality of compositing and tests will be integrated into the fragment pipeline. Other recent advances concern conditionals and loops that will be available at least for vertex programs.

2.1.9 High Level Shading Languages

Programming the vertex and fragment pipeline with assembler is time consuming and error-prone, especially as different APIs and vendor extensions have slightly different pre-conditions, syntaxes, and semantics for implementing the same functionality. High level shading languages (HLSLs) have been used in the offline rendering community for a long time (e.g. Renderman [Hanrahan and Lawson 1990]) and had a great influence on the development of shading languages for graphics hardware. C for graphics (Cg) [Mark et al. 2003] was the first HLSL in the OpenGL context with a working framework for graphics hardware with decent performance. In contrast to the recently supposed official OpenGL HLSL, Cg is not really vendor-independent, despite the advertisement from NVIDIA, as the different backends of the language can change or enhance the language specification. On the other hand, Cg has a backend for DirectX as well, and shader code may be compiled for both OpenGL and DirectX. As DirectX has its own HLSL model now Cg will be less used here in the future. For OpenGL, it will still play an important role, as the official HLSL is not available yet.

2.2 Hierarchical Methods

The basic idea of hierarchical data sets is that data is represented on different scales, creating a multiresolution analysis. Mathematically speaking, a set of subspaces is constructed from the original data space $\bar{S}$

$$S_0 \subset S_1 \subset S_2 \subset \ldots \subset S_n = \bar{S} \quad . \tag{2.1}$$

By projecting the data set into these subspaces we get a level-of-detail representation of the data. The projection into S_j is called the representation of the data set in level j, or just the data set of level j. Because the subspaces S_m contain all functions of subspaces S_n with $m > n$, (2.1) spawns up a hierarchy. The enclosing subspace S_m representing higher frequency data is called the upper or higher level, while the contained subspace S_n representing low-pass filtered data is called the lower level.

Usually, only S_0 and the differences between two levels are stored in order to save memory. For accessing the original data, a recursive traversal is necessary. Some hierarchical basis functions like sparse grids (see Chapter 6) allow for an iterative evaluation, utilizing certain properties of the used basis.

The most well-known multiresolution analysis is the wavelet theory (see Chapter 5). Within this theory, a data set is decomposed into a low-pass filtered data set of smaller resolution and several high-pass filtered data sets that represent higher frequencies only.

During analysis, the development of features in the data set can be tracked through the levels. This helps e.g. with the robust detection of edges in noisy data [Westermann and Ertl 1997], robust segmentation [Lürig et al. 1997a], and compact storage of inhomogeneously structured data [LaMar et al. 1999]. During rendering, the recursive traversal allows for fast and smooth display of inhomogeneous data [Weiler et al. 2000a], and by changing the recursion depth on-the-fly according to some screen-space criteria, we get adaptive rendering [Lürig et al. 1997b, Rusinkiewicz and Levoy 2000].

2.3 Volume Visualization

As this work mostly deals with scalar three-dimensional data sets, the presented results will be usually rendered with volume visualization. In this section the basic ideas behind this technique will be introduced, as there exists a vast number of published volume rendering algorithms, e.g. [Schröder and Stoll 1992, Williams 1992, Malzbender 1993, Cabral et al. 1994, Lacroute and Levoy 1994, Lippert et al. 1997, Rezk-Salama et al. 2000].

2.3.1 Mathematical Basics

The basic principle behind volume rendering is an approximate evaluation of the volume rendering integral for each pixel of the image plane. This integral has been derived from the transport theory of light [Hege et al. 1993] for the case of completely neglected scattering.

The integral form of the equation of transfer describes the intensity $I(x)$ at the position x along a ray of sight:

$$I(x) = I_{\text{far}}\, e^{-\tau(x_{\text{far}},x)} + \int_{x_{\text{far}}}^{x} \eta(x')\, e^{-\tau(x',x)} dx' \quad , \qquad \tau(x_1,x_2) = \int_{x_1}^{x_2} \kappa(x) dx$$

where I_{far} denotes the specific intensity at the volume boundary at x_{far}, $\tau(x_1,x_2)$ the optical depth, $\kappa(x)$ the absorption, and $\eta(x)$ the total emission coefficient.

This ray integral can be used for all physically based illumination models, which influence only the transfer function. Thus different visualization techniques can be implemented by different mappings from scalars to RGBA values. Together with the discretization of the integral this leads to the compositing formula for computing the intensity contribution:

$$I = \sum_{k=1}^{n} C_k \prod_{i=0}^{k-1} (1 - \alpha_i) \quad .$$

The emission of the voxel C_k and its opacity α_k are derived from the transfer functions after the interpolation of the scalar value from the given volume function representation. One example of a commonly used illumination model that cannot be modeled with this approach is maximum intensity projection (MIP), which maps the maximum value along a ray of sight to the final pixel value.

The total emission coefficient η is modeled by C_k, and the optical depth $\tau(x_1, x_2)$ is approximated by the product of transparencies of the voxels the ray has already passed. The later is computed in an iterative process in actual implementations, compositing the C_k back to front to evaluate the final pixel intensity I_0:

$$I_i = C_i + I_{i+1} \cdot (1 - \alpha_i) \quad , \qquad i = n-1, \ldots, 1 \quad , \qquad I_n = I_{\text{far}} \quad . \tag{2.2}$$

Note that in some applications other non physically based rendering models may be more appropriate, e.g. maximum intensity projection for some medical indications. Other applications can benefit from neglecting any opacity ($\tau \equiv 0$), because this changes the model to a commutative one in the sense that the order of summation does not affect the result.

2.3.2 Algorithms

Volume rendering algorithms can be subdivided into three major classes: frequency space, image space, and object space methods. Due to major limitations regarding blending operator flexibility the frequency domain approach will not be discussed any further.

Image space methods comprise raycasting, shear warp, and 3D-hardware-based methods. All methods have in common that for each pixel in the image plane the given input function is resampled along the viewing ray and the ray equation is evaluated for these samples. Their main difference is the exact order of loops and the sampling and projection steps.

The most common object space methods are cell projection and splatting. These techniques compute the contribution of a single object or basis function to the complete image plane, thus evaluating the outer loops of the rendering algorithm in the opposite order compared to image space methods. Figure 2.2 lists the order of the basic steps for the different types of algorithms for comparison. In the following the three algorithms used in this work will be explained in more detail.

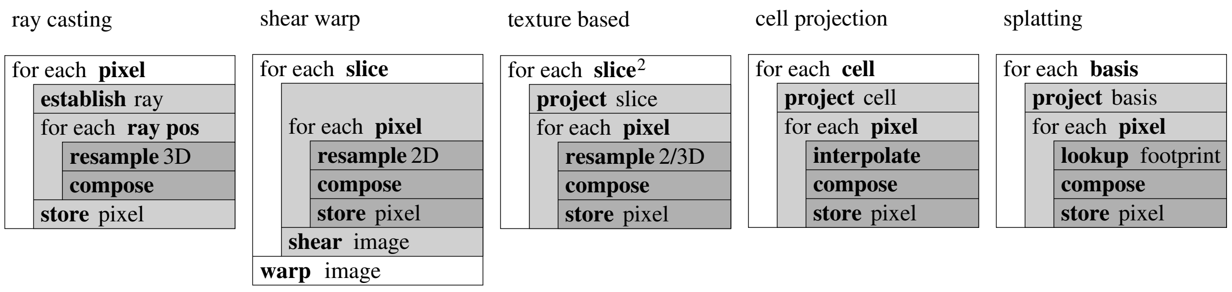


Figure 2.2: Different volume rendering algorithms and the order of their basic steps.

Raycasting

The basic idea of raycasting is to trace viewing rays from the eye through the volumetric data set (see Figure 2.3). Along each ray, the volume data is sampled and composed according to (2.2). This algorithm resembles the basic idea of a physically based volume visualization very closely. Thus raycasting algorithms have been one of the first volume visualization techniques [Levoy 1988].

In this work this approach is used for algorithms that need the highest flexibility and that are mainly limited by the evaluation speed of the basis functions.

Splatting

Westover introduced in [1990] this object-order volume rendering technique. In this class of algorithms the contribution of a single basis function to the image plane is evaluated or defined in a preprocessing step, and stored as a footprint of the basis function. During rendering scaled versions of this footprint for each basis function are drawn to the image plane and composited (see Figure 2.4). Most splatting algorithms can only deal with rotational invariant symmetric footprints, but recently methods have been invented that can deal with elliptic splats like in [Zwicker et al. 2001a].

Splatting is usually slower and leads to lower quality images compared to texture-based volume rendering, except when special measures are taken as e.g. in [Swan et al. 1997, Mueller et al. 1999, Zwicker et al. 2001a]. Other recent advances of 3D-hardware-based rendering like pre-integration have not been incorporated into the splatting approach so far. On the other hand, splatting is much more flexible when it comes to volume representations, thus this is the choice for applications that need a higher positional resolution than the previous approaches can handle.

Quite a lot of work has been done for rendering solid objects with points, as in [Rusinkiewicz and Levoy 2000] for example — for more references please see Chapter 8. In this work this approach is used for algorithms that work with irregularly sampled scalar fields. In the particular cases one of the major parameters of the visualization is the exact position of the basis functions, which can only be preserved if the volume is not resampled on a Cartesian grid.

[2]This simplification only covers slice-based proxy geometries as the ones described.

Texture-based volume rendering

Texture-hardware-based volume rendering has been presented for SGI graphics workstations first [Cabral et al. 1994]. One of its major merits is that it made volume rendering on PCs capable of rendering several frames per second for decently sized volumes [Rezk-Salama et al. 2000].

Basically, hardware-accelerated volume rendering is a highly parallel version of raycasting, which evaluates all rays simultaneously. Practically, it renders the volume data using proxy geometries, onto which the appropriate part of the volume data is mapped. Figures 2.5 and 2.6 show the two most commonly used approaches using 2D and 3D textures, that differ by the used proxy geometry. 3D-hardware-based techniques are usually declared to be a parallel implementation of a raycasting algorithm, however, they share many aspects with object space algorithms taking into account these proxy geometries. For a more detailed discussion of the advantages and disadvantages of the different rendering techniques see [Rezk-Salama 2002].

With the use of transfer functions and clever utilization of OpenGL tests, mapping techniques can be incorporated into volume rendering as well. Westermann [1998] has introduced the rendering of isosurfaces on a per-fragment level with the help of the alpha test in order to render only the part of the volume that represents the chosen isosurface. More recent advances like pre-integration [Roettger et al. 2000] improved the visual appearance of the renderings almost to the quality of analytically integrating raycasting models.

In this work texture-based volume rendering is used for algorithms that work with regularly sampled scalar fields on a Cartesian grid and for functionally encoded data sets that are reconstructed during rendering.

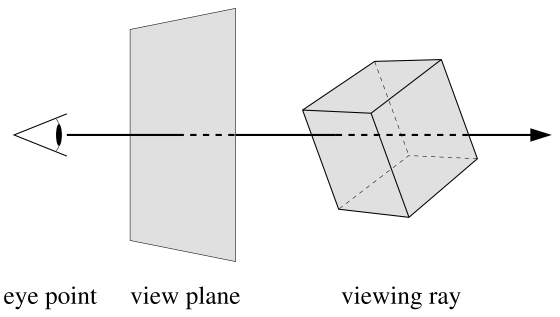


Figure 2.3: Raycasting: one viewing ray is traced per pixel.

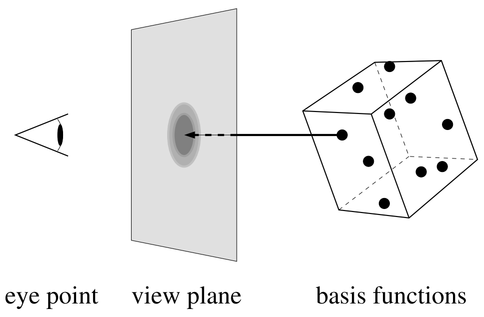


Figure 2.4: Splatting: one footprint is rendered to the image plane per basis function.

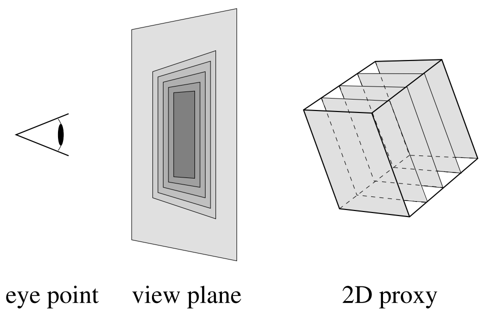


Figure 2.5: 2D texturing: proxy geometry is aligned to the object axes.

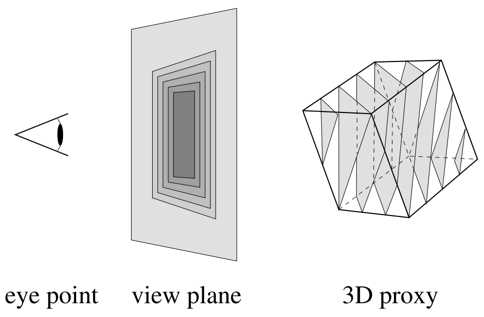


Figure 2.6: 3D texturing: proxy geometry is aligned to the image plane.

Delenn: (to the Grey Council)
Summoned, I come.
In Valen's name I take the place
that has been prepared for me.
I'm Grey.
I stand between the candle and the star.
We are Grey.
We stand between the darkness ... and the light.

Babylon 5, Babylon Squared

Chapter 3

Linear Filters

Many volume filtering operations used for image enhancement, data processing or feature detection can be written in terms of three-dimensional convolutions. It is not possible to yield interactive frame rates on todays hardware when applying such convolutions on volume data using software filter routines. As modern graphics workstations have the ability to render two-dimensional convoluted images to the framebuffer, this feature can be used to accelerate the process significantly. This way generic 3D convolution can be added as a powerful tool in interactive volume visualization toolkits.

Filtering is a major part of the visualization pipeline. It is widely used for improving images, reducing noise, and enhancing detail structure. Volume rendering can benefit from filter operations, as low-pass filters reduce noise, e.g. in sampled medical volume images, and high-pass filters can be used for edge extraction, visualizing prominent data features.

Filters can be classified as linear or nonlinear. Discrete linear filters can be written as convolutions with filter kernels that completely specify the filtering operation. Nonlinear filters include for instance morphological operators, which are covered in the next chapter.

For texture-based volume rendering the data set has to be loaded into special texture memory, which can be addressed by the GPU very fast. The loading process itself is relatively slow, taking several seconds for large data sets even on the fastest available SGI workstations. PCs are a bit faster due to the recent technological advances. However, the AGP bus is still one of the main bottlenecks with respect to texture downloading, and it does not allow for much higher clock rates due to physical constraints. Future changes in the PC infrastructure with high bandwidth processor / north bridge busses and the upcoming PCI-Express standard may change the situation, though.

As the data set has to be reloaded after a filter operation has been performed in software, interactive filtering will benefit a lot from convolution algorithms that directly work on the texture hardware. Additionally, it will be shown in the following that computing the convolution with graphics hardware is much faster than software solutions. This work has first been published at the Visualization conference in [Hopf and Ertl 1999a].

3.1 Convolution

The general three-dimensional discrete convolution can be written as

$$\tilde{f}(x,y,z) = \sum_{i_1,i_2,i_3} k(i_1,i_2,i_3) \cdot f(x+i_1-c_1,\ y+i_2-c_2,\ z+i_3-c_3)$$

with f being the input data function and k being the filter kernel, resulting in the convoluted data $\tilde{f}$. $\mathbf{c}$ is called the center of the filter.

In the following examination it is assumed without loss of generality that $k(i_1,i_2,i_3)$ is given for $0 \le i_1,i_2,i_3 < n$ and vanishes outside this interval. Also, it is assumed that the input data function vanishes for (x,y,z) outside the interval $[0,m)^3$. Most filters are either symmetric or antisymmetric, in these cases the center of the filter is given by $\mathbf{c} = \lfloor \frac{1}{2}\mathbf{n} \rfloor$.

In the special case of $k(i_1,i_2,i_3) = \overline{k}_1(i_1) \cdot \overline{k}_2(i_2) \cdot \overline{k}_3(i_3)$ the kernel is called separable. In this case the number of operations necessary for the convolution can be reduced down to $O(m^3n)$, from $O(m^3n^3)$ in the non-separable case:

$$\begin{aligned}
\tilde{f}_1(x,y,z) &= \sum_{i_1} \overline{k}_1(i_1) \cdot f\ (x+i_1-c_1,\ y,\ z) && (3.1)\\
\tilde{f}_2(x,y,z) &= \sum_{i_2} \overline{k}_2(i_2) \cdot \tilde{f}_1(x,\ y+i_2-c_2,\ z) && (3.2)\\
\tilde{f}(x,y,z) &= \sum_{i_3} \overline{k}_3(i_3) \cdot \tilde{f}_2(x,\ y,\ z+i_3-c_3) && (3.3)
\end{aligned}$$

Of course special care has to be taken near the boundaries of the input data function, as convolution routines are generally written on a very low language level for speed purposes.

Figures 3.1 and 3.2 shows two well known convolution filters, the Gaussian filter and its second derivative, both in their continuous and discrete forms. They can be used for noise reduction and edge detection, respectively. An example image that has been filtered with these kernels can be seen in Figure 3.3.

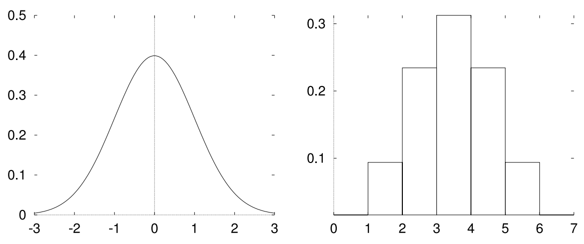


Figure 3.1: The Gaussian filter function.

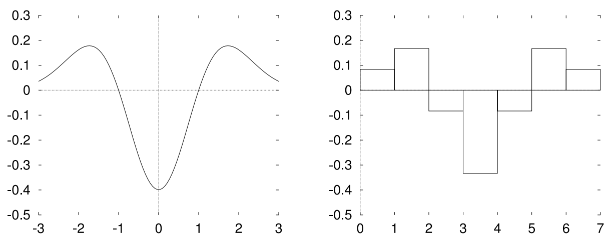


Figure 3.2: The second derivative of Gaussian.

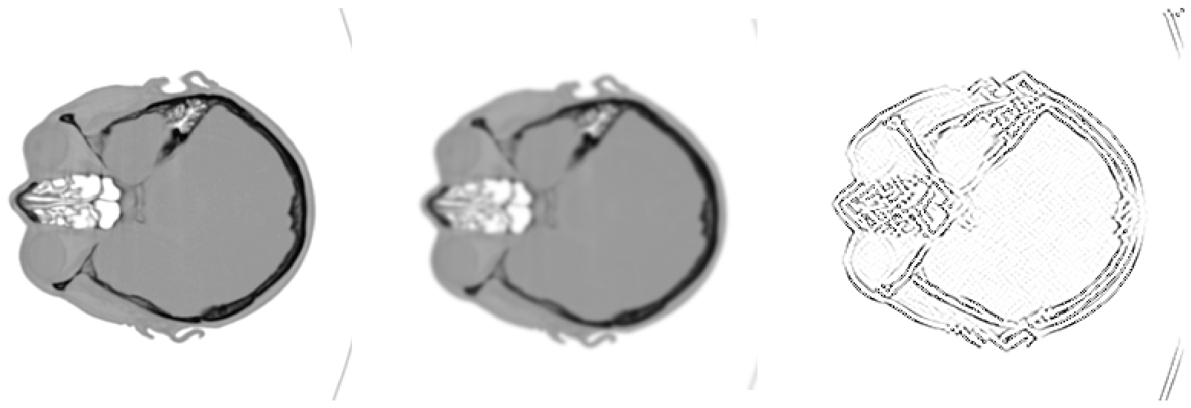


Figure 3.3: Example image, filtered with Gaussian, and filtered with its second derivative.

3.2 Related Work

Until 1999, only few algorithms performing mathematical operations using standard graphics hardware APIs for acceleration had been published, and even less (if any at all) that employed OpenGL. Mark Harris maintains a web site[1] with references on ”General Purpose Computation Using Graphics Hardware”, which shows that the majority of papers have been published in the early 21st century. Newer researches related to convolution include the use of PC graphics hardware for higher order texture filtering [Hadwiger et al. 2001, Hadwiger et al. 2002] and thoughts about quality issues [Teitzel et al. 1999, Strzodka 2002, Hadwiger et al. 2003].

Multiscale approaches as [Westermann and Ertl 1997] regularly use disjunct filtering and downsampling steps and can benefit from any speedups of the filtering process. Segmentation and classification depend heavily on filtering operations as well. Bro-Nielson [1996] already thought about using convolution hardware for accelerating the registration process.

3.3 Implementing Convolution with OpenGL

In order to accelerate the convolution process, special purpose hardware can be used. On systems that have built-in Digital Signal Processors (DSPs), for example for multimedia acceleration, a specialized convolution subroutine could be downloaded to the signal processor. On the other hand, most times these DSPs are not well documented or the run-time system can not be modified by the user. In general they are not faster than the main processor as well. Additionally, there exists a wide range of different DSP systems, all of which are incompatible to each other.

Another approach that has been taken in this work is to combine a 2D and a 1D convolution kernel in order to calculate three-dimensional separable convolutions. Several vendors of the graphics API OpenGL — as for example Silicon Graphics Inc. [SGI 1996] — have included extensions for one- and two-dimensional filtering, which is now an optional part of the official

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

OpenGL specification. In contrast to most implementations that emulate these extensions only in software, the SGI graphics pipes MXE, V8, V12, BasicReality, and InfiniteReality calculate the convolutions on-board, boosting performance by an order of magnitude even for reasonably sized filters. The CRM graphics system of the O_2 is capable of rendering convolutions in hardware as well, but it does not support volume textures, which are crucial for the algorithm. PC graphics hardware would be capable of doing convolutions in graphics hardware, however, their drivers are usually not optimized for the imaging pipeline, as it is rarely used in games. On PCs an implementation comparable to the one shown in Chapter 4 would have much better performance, but as Hadwiger [2003] points out for using higher order texture filters this approach has several error sources that would need some investigation first.

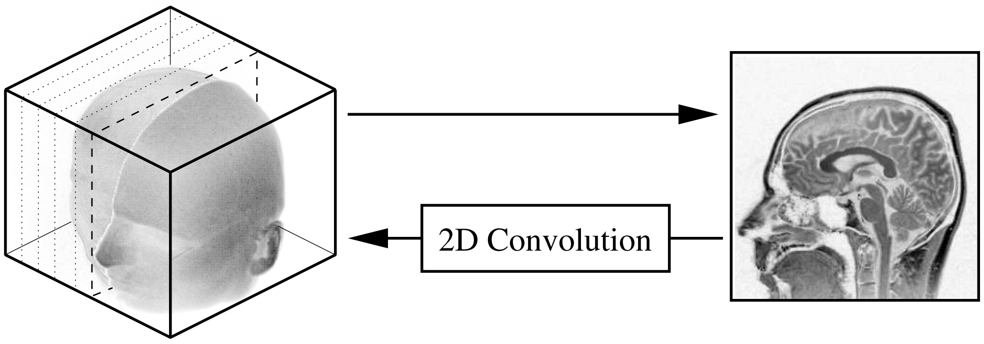


Figure 3.4: The first pass of the hardware filtering algorithm.

Recall that the volume data is already stored in texture memory for visualization using texture hardware. Now (3.1) and (3.2) are combined to one 2D convolution that is to be applied to every plane of the volume data perpendicular to the z-axis. Therefore, plane by plane is drawn by rendering quads into the framebuffer as it is sketched in Figure 3.4. The texture coordinates assigned to the vertices of the triangle strips are specified in such a way that no interpolation of the texture is necessary (see Figure 3.5) as long as vertex coordinates are set to raster the planes in a pixel-exact layout on the screen. The offset of the filter center **c** has to be compensated in this step as well, as convolution in the imaging pipeline does not allow for kernels coefficients with negative indices. In order to increase the potential speedup and to avoid rounding problems,

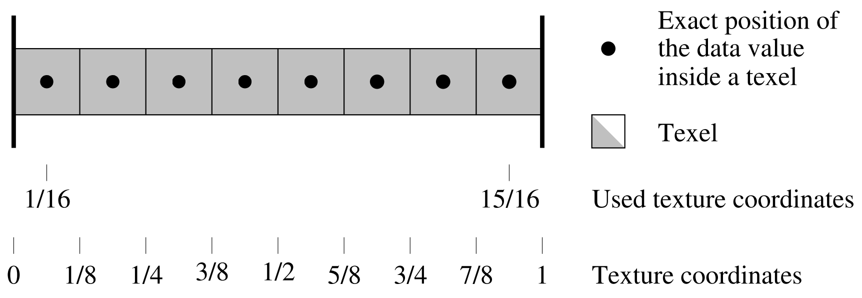


Figure 3.5: Texture coordinates used for exact texel hits.

nearest neighbor interpolation is activated during the rendering process. Each plane is then read back with the OpenGL routine `glCopyTexSubImage3DEXT`, which replaces one plane of the active volume texture orthogonal to the z-axis by data that is read directly from the framebuffer. While transfering the data to the texture memory, the separable 2D convolution filter is activated using `glSeparableFilter2DEXT`. After this first pass, the volume texture contains data filtered along the x- and y-axes.

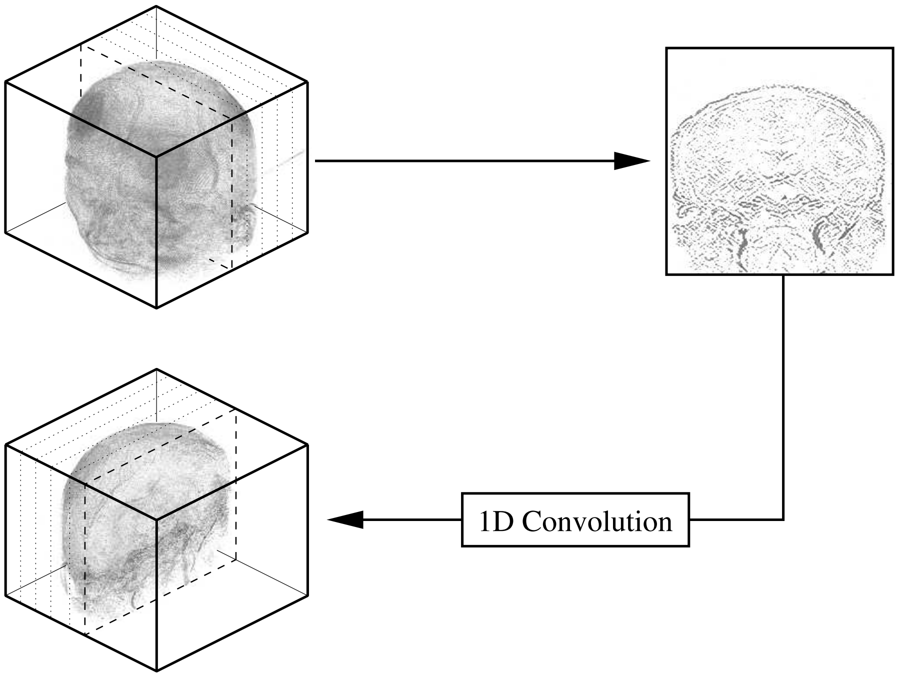


Figure 3.6: The second pass.

Applying the convolution to the third axis is more complicated and depicted in Figure 3.6. In this second pass planes are rendered perpendicular to one of the other axes. Assume that the y-axis has been chosen. As `glCopyTexSubImage3DEXT` can not write planes orthogonal to any other axis than the z-axis to the texture memory, they have to be transfered to a second volume texture. OpenGL's texture objects are used for switching between them, which implies only a very small overhead. While copying the data from the framebuffer to the texture memory, a one-dimensional convolution filter is activated. As we are dealing with two-dimensional image data, a 2D convolution filter is specified with `glConvolutionFilter2DEXT`, using a filter kernel that is exactly one pixel wide.

After this second pass the convoluted volume data has been mirrored at the plane $y - z = 0$. For texture-based volume renderers this does not impose any restrictions, as they only have to swap texture coordinates. When data order is crucial for the application the algorithm of pass two can be used for both passes, thus restoring the data order in the second pass. However, the textured planes have to be drawn two times perpendicular to the y-axis. Cache misses are much more

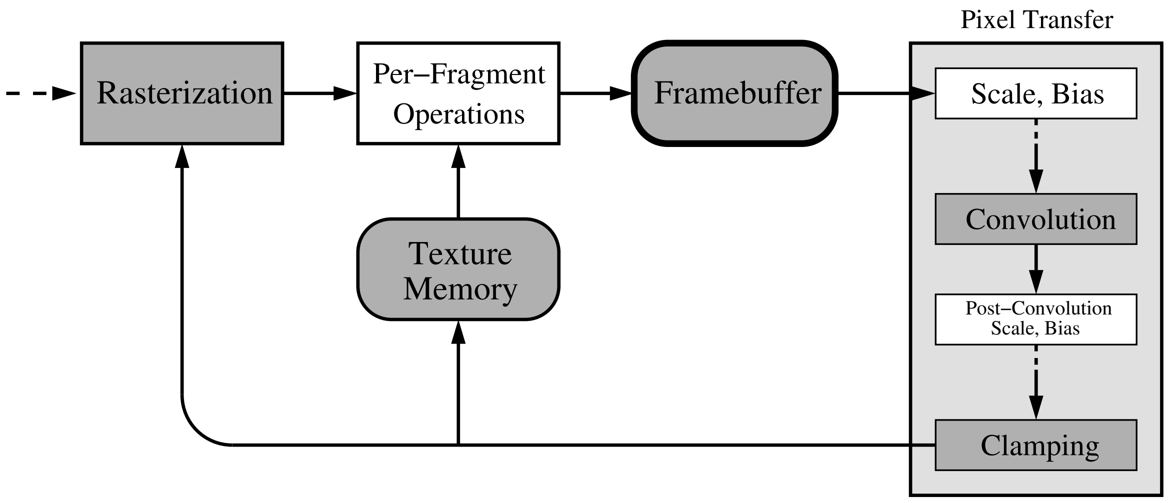


Figure 3.7: The imaging pipeline of OpenGL.

likely in this case compared to planes rendered orthogonal to the z-axis. This can increase the convolution times on big volumes by up to 50% even on fast graphics hardware.

Figure 3.7 depicts the relevant parts of the OpenGL pipeline. It reveals, that pixels read from the framebuffer are clamped to $[0,1)$ before they can be written back to the framebuffer or into the texture memory. Filter kernels with negative coefficients can compute negative intermediary values during the two-dimensional convolution pass, which will not contribute to the final 1D convolution. These intermediary values are especially needed when the filter kernel is not symmetrical. Additionally, no negative results can be stored in the output volume.

The strategy for avoiding these effects depends on the particular application. For edge detection the absolute maxima of the filtered volume data are of interest. In this case, calculating the absolute value can be performed in hardware as well, further reducing necessary computations on the CPU. In most other cases post-convolution scaling and bias can be used to map the expected results to the interval $[0,1)$ just before the clamping takes place. OpenGL provides the `GL_POST_CONVOLUTION_c_SCALE_EXT` and `GL_POST_CONVOLUTION_c_BIAS_EXT` parameters, which are applied to pixel color values after convolution and before clamping as depicted in Figure 3.7.

3.4 Results and Comparison

Figures 3.8 to 3.13 show slices of a head data set of size 128^3. Figure 3.8 depicts the unfiltered data set, whereas Figures 3.9 and 3.10 present slices of the software and hardware low-pass filtered volume data, respectively. Here, a Gaussian filter kernel of size 5^3 has been used. The results of the hardware-based algorithm differs only in the two lowest significant bits, compared to the purely software-based system. A contrast enhanced difference image can be seen in Figure 3.11. Figures 3.12 and 3.13 reveal the results for high-pass filtering using the second deriva-

Filter size	2^3	3^3	5^3	7^3
`head`[†]	0.33/0.72	0.33/1.02	0.33/1.56	0.48/2.0
`angio`[‡]	2.5/6.0	2.5/8.7	2.5/14.7	3.7/21.3

[†] Data set created by computer tomography, 128^3
[‡] Data set created by MR angiography, 256^3

All times were measured on a Silicon Graphics Onyx2 equipped with a BasicReality graphics pipe. The system has two R10000/195 MHz processors and 640 MB main memory.

Table 3.1: Convolution times in seconds using hardware / software.

tive of the Gaussian filter, again computed in software and in hardware. The hardware-convoluted volume displays noticeable artifacts that occur due to the already mentioned clamping step in the OpenGL pipeline. By using post-convolution scaling and bias the artifacts disappear completely.

The Figures 3.16 to 3.15 picture another data set that has been used for testing the implementation. They have been visualized with the hardware-based volume rendering toolkit TiVOR [Sommer et al. 1998], again with the first picture being rendered with the original data set. While the noise reduction effect of the Gaussian filter is rather bothering in Figure 3.17 by smearing volume details, it has remarkably positive effects on isosurface generation (compare Figures 3.14 and 3.15). Note that the isosurfaces are rendered in real-time using a hardware-accelerated volume rendering approach as described in [Westermann and Ertl 1998].

Noise interferes with high-pass filters, which can be seen in Figure 3.18. Using a high-pass filter on the already low-pass filtered data set reveals by far more and better separable details (see Figure 3.19) compared to the directly filtered volume.

The speed of the hardware-based convolution algorithm has been compared to a well tuned software convolution filter. Unsurprisingly, the software convolution is almost completely memory bandwidth bound. Even workstations with a high performance backplane as the Octane are limited by the main memory bandwidth, as today's caches are far too small for the values needed for convolution along the z-axis. High end machines as the Onyx2 perform huge 3D convolutions three times faster than the Octane, even when equipped with slower CPUs. Standard PCs can only recently cope with the memory bandwidth of the Onyx2 system, and multiprocessor options will not accelerate the process because it is not CPU bound.

Table 3.1 shows convolution times for different data sets and filter sizes, using software and hardware convolution. All times have been measured on an Onyx2 equipped with a BasicReality graphics pipe. The maximum filter size supported by the graphics system is 7^2. Therefore, the maximum 3D convolution that can be performed in hardware on this system is 7^3. Noteworthy is the fact that the BasicReality graphics system is optimized for filter kernels of size 5^2. Convolutions with smaller kernels need exactly the same computation time. Filters of size 6^2 and 7^2 share their timing results as well. The x and y coordinates of the volume are swapped during the hardware-based convolution process, which is a side effect of the presented 3D convolution algorithm.

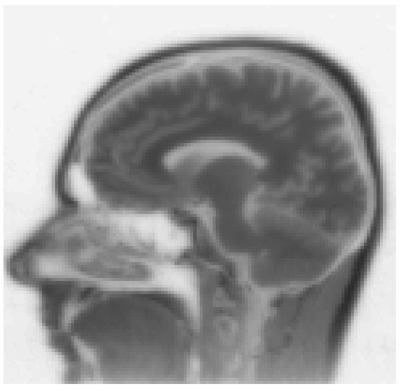


Figure 3.8: The unfiltered head data set.

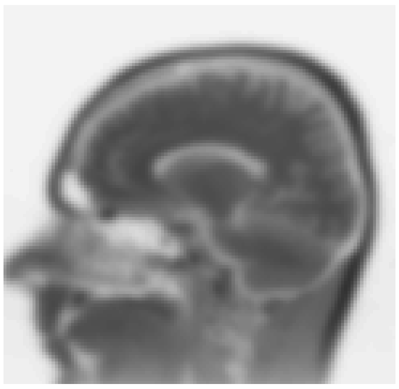


Figure 3.9: Head, low-pass filtered in software.

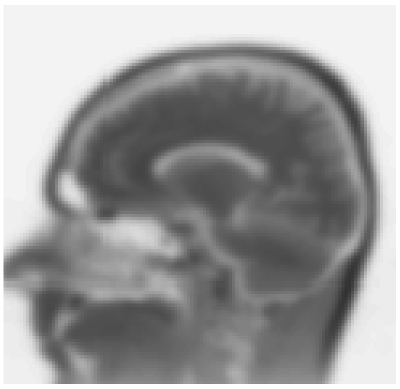


Figure 3.10: Head, low-pass filtered in hardware.

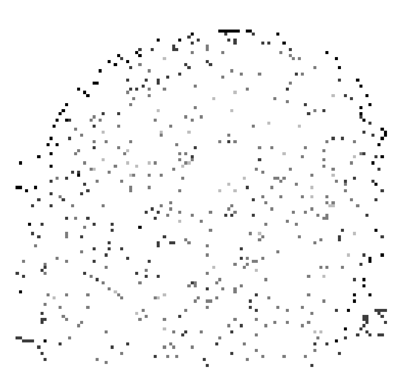


Figure 3.11: Differences of software- and hardware-based filtering, showing the two lowest significant bits.

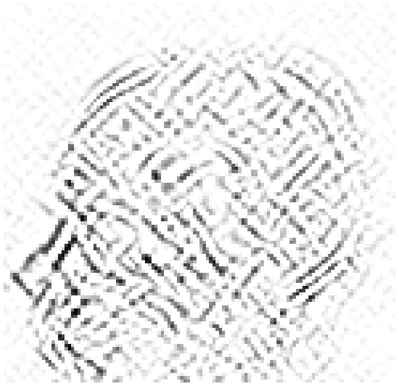


Figure 3.12: Head, high-pass filtered in software.

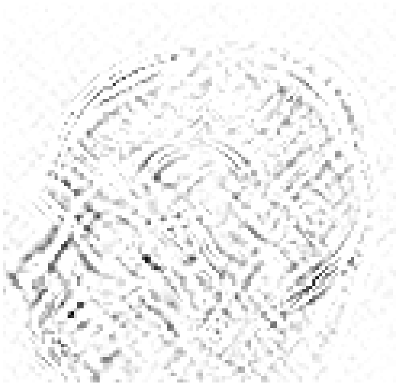


Figure 3.13: Head, high-pass filtered in hardware.

As several of today's graphics workstation vendors have added two-dimensional convolution to their OpenGL pipeline, using this capability for accelerating 3D convolution is an almost straightforward approach. By using the implemented algorithm three-dimensional convolution can be performed even on big data sets with nearly interactive rates.

All intermediary data is transfered to the framebuffer, thus clamping can suppress negative values that result from the two-dimensional convolution as well as final negative results. Therefore, this approach is currently most useful for symmetrical filter kernels. By using post-convolution scaling and bias extensions these problems can be easily overcome. This technique will be demonstrated on a more elaborate problem in Chapter 5.

Non-separable convolutions are not possible right now with this algorithm. However, by applying several two-dimensional filter kernels and blending convoluted images in the framebuffer with an algorithm similar to the one presented in Chapter 4, the use of non-separable 3D kernels would be a possibility for the future as well. Due to the extremely small number of applications that

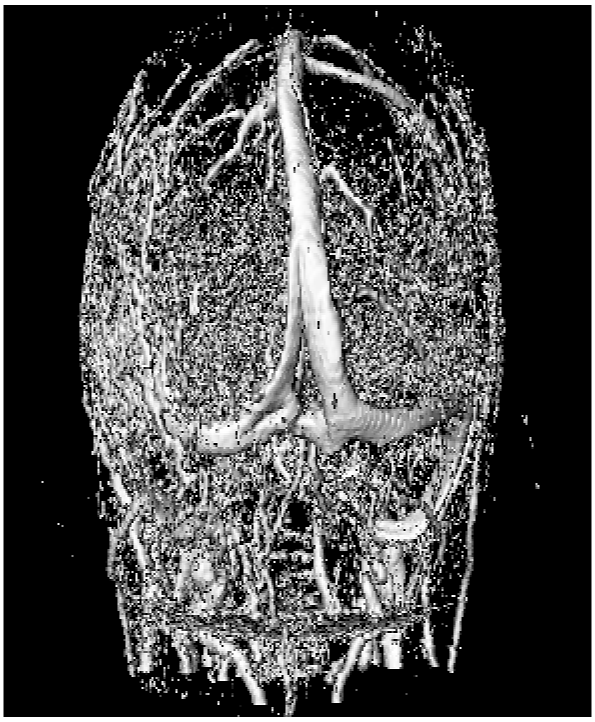


Figure 3.14: Isosurfaces on the original angiography data set.

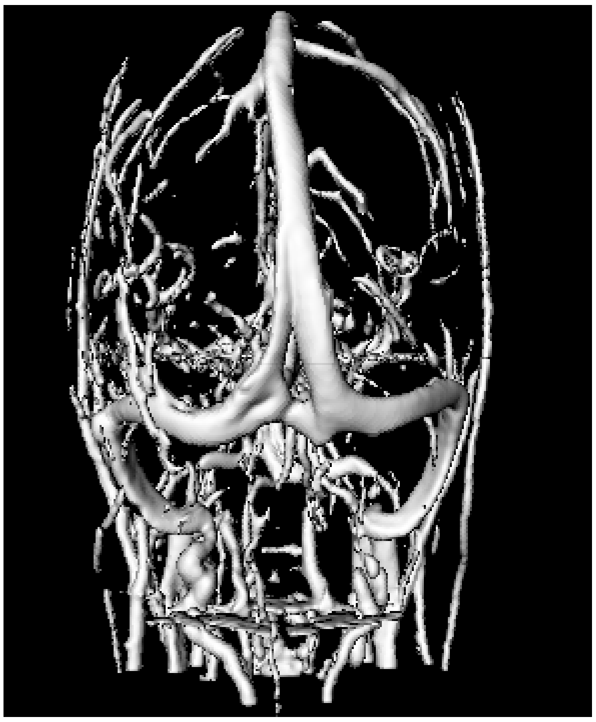


Figure 3.15: Isosurfaces on the Gaussian filtered data set.

depend on non-separable kernels, this problem has not been investigated any further. The same approach would enable the hardware-accelerated use of linear filters on standard PCs as well. An implementation will have exactly the same performance figures as in the presented nonlinear case, because only the blending mode has to be changed.

As we now have a basis for accelerating linear filtering operations using graphics hardware, we can extend this approach to nonlinear filters first, before we analyze how to accelerate hierarchical filters.

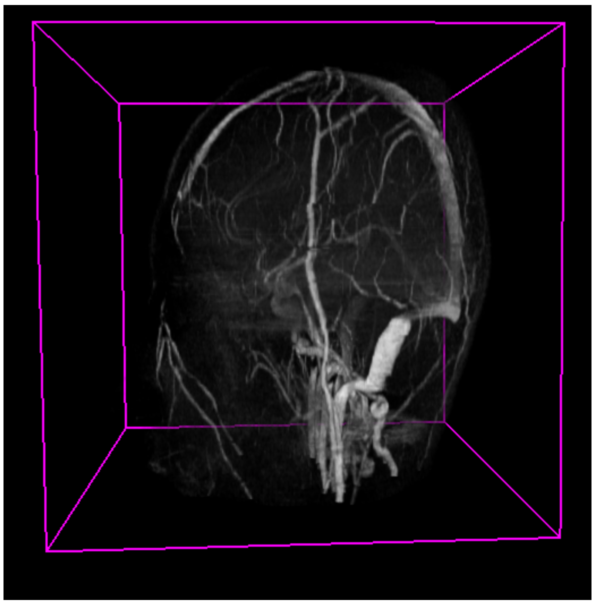


Figure 3.16: The original angiography data set.

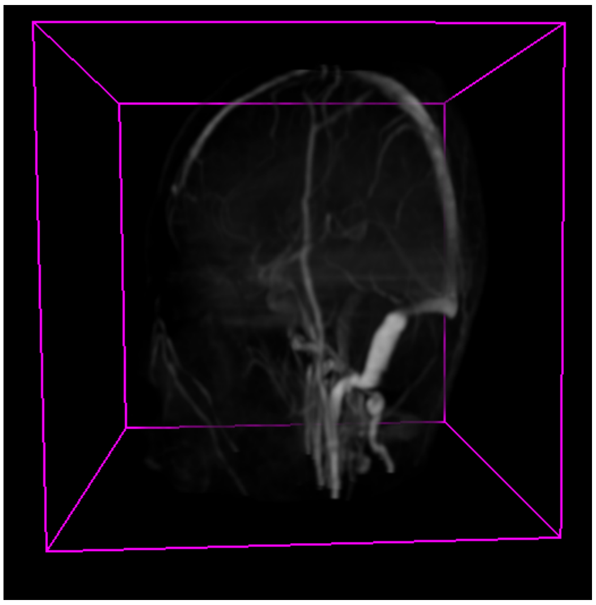


Figure 3.17: The Gaussian filtered data set.

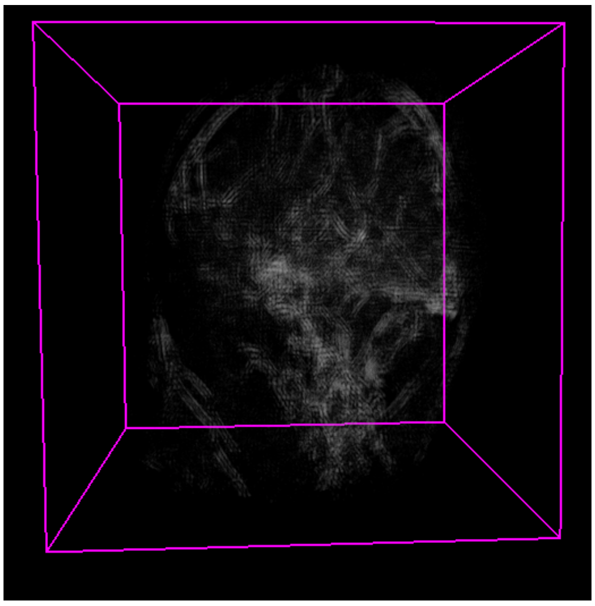


Figure 3.18: Data, after direct filtering with Gaussian's second derivative.

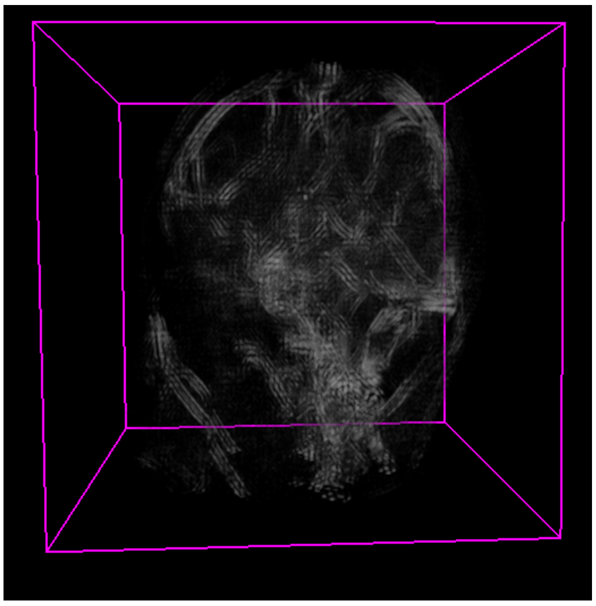


Figure 3.19: First low-pass, then high-pass filtered data.

Ivanova: *I can only conclude that I'm paying off karma at a vastly accelerated rate.*

Babylon 5, Points of Departure

Chapter 4

Nonlinear Filters

Before a given data set is rendered, several filtering and mapping steps can be applied to the data in the visualization pipeline in order to expose prominent features more precisely. Even with the broad knowledge about filtering techniques we have today, for volume rendering this only means adapting the mapping step in most cases. The proper adjustment of this process is one of the major problems for the practical use of volume visualization. It is commonly accomplished by the interactive generation of a more or less complicated transfer function. More complex approaches are either too slow or beyond the capabilities even of modern graphics hardware and therefore not very useful for interactive volume rendering.

Alternatively, the volume data can be preprocessed in order to reveal certain features more properly with relatively simple transfer functions. This approach requires fewer input parameters from the user during analysis and rendering, simplifying the visualization cycle. One promising approach for semi-automatic frequency-based volume analysis are morphological operators. Despite all improvements in processor technology one serious drawback of this approach remains: almost all operators working on three-dimensional data are computationally complex, reducing their usefulness for interactive visualization.

The disadvantage of filters based on linear combinations of the input data is the fact that the analyzed structures become distorted. This can be seen especially on lower frequency scales, accomplished by large filter kernels. One class of special nonlinear filters that do not flatten the contours of the original data set are the morphological operators. This filter type computes the minimum respectively maximum of pixel values within a given scope. Before the actual maxima are calculated, the values of the so-called structuring element are added to the pixel values.

Graphics hardware can be used to accelerate this important type of operation by using the GPU's superior memory coupling. Additionally, we will see that in contrast to other filters the implementation of morphological operators on graphics hardware does not impose any additional errors on the filtered data. This work has first been published at the Workshop on Vision, Modeling, and Visualization VMV in [Hopf and Ertl 2000a].

4.1 Morphological Operators

Morphological operators are a special form of nonlinear filters that are capable to separate or to combine different regions in an image with a minimal distortion of the contours. At first, these operators are defined on binary images only. Let X and Y be two one-dimensional binary data sets. Then the erosion $Z^- := X \ominus Y$ is defined as

$$Z_i^- = \begin{cases} 0: & \exists j: \; Y_j = 1 \wedge X_{i-j} = 0 \\ 1: & \text{otherwise} \end{cases} \quad .$$

The other basic operator, the dilation operator $Z^+ := X \oplus Y$ is defined as the dual operator to the erosion:

$$Z_i^+ = \begin{cases} 1: & \exists j: \; Y_j = 1 \wedge X_{i+j} = 1 \\ 0: & \text{otherwise} \end{cases} \quad .$$

As one can see in Figure 4.1, the erosion operator cuts away parts of the boundary of the analyzed image X. The amount that is removed is defined by the structuring element Y, which is typically much smaller than the input image. The dilation operator on the other hand enlarges the set parts of the input data.

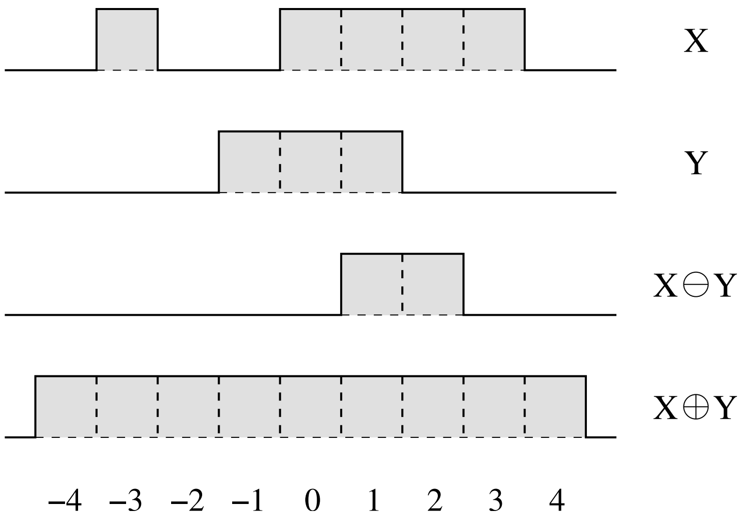


Figure 4.1: Results of the binary erosion and dilation operators.

By combining these basic operators to more complex ones we get the so-called opening and closing operators. The opening operator breaks up small bridges between connected regions while the closing operator tends to fill small gaps in solid components. The opening operator is defined as

$$X \bigcirc Y := (X \ominus Y) \oplus Y$$

and the closing operator is defined as

$$X \circ Y := (X \oplus Y) \ominus Y$$

Gray-scale morphological operators, which are used in this chapter, are defined by transforming the gray-scale data to a binary data set with an extra dimension, representing the gray level. This lifting into the extra dimension can be done implicitly by defining according dilation and erosion operators for gray-scale data. This way we get the dilation operator $Z^+ := X \oplus Y$ as

$$Z_i^+ = \max_j \{X_{i-j} + Y_j\} \tag{4.1}$$

and the erosion $Z^- := X \ominus Y$ accordingly as

$$Z_i^- = \min_j \{X_{i+j} - Y_j\} \quad . \tag{4.2}$$

In the binary forms of the operators, the parts of the structuring element Y with $Y_j = 0$ have no effect at all, and thus we will call them neutral elements. In gray-scale morphology this corresponds to parts of the structuring element with $Y_j = N := -\infty$, because in the way they are used they are invariants to the maximum and minimum operators. In the following we will assume that all structuring elements have only a finite size, i. e. all values outside a given domain are equal to N.

When we remember that all input data as well as the structuring element is bound for problems related to image and volume data sets to a fixed domain $[0,m]$, we can define $\overline{Y}_j = m - Y_j$, and using this we can shift the range of (4.1) and (4.2) to $[0,m]$ as well by using

$$Z_i^+ = \max_j \{X_{i-j} - \overline{Y}_j\} \tag{4.3}$$

for the dilation and

$$Z_i^- = \min_j \{X_{i+j} + \overline{Y}_j\} \tag{4.4}$$

for the erosion operator, provided that there exists a j_m with $Y_{j_\mathrm{m}} = m$. This requirement assures, that $X_{i-j_\mathrm{m}} - \overline{Y}_{j_\mathrm{m}} \geq 0$, because $\overline{Y}_{j_\mathrm{m}} = 0$. Additionally, $X_{i-j} - \overline{Y}_j \leq m$ holds for all i, j, because $\overline{Y}_j \geq 0$. The neutral element in this case is $\overline{N} := m$, because $X_j - m \leq 0$ and is thus not contributing to the maximum value in (4.3). Equivalent inequalities hold for the erosion operator as well.

As the range of the morphological operators can be shifted to the domain range and only integer operations are needed during computation, it is clear, that these operators can be implemented using graphics hardware without precision loss, when the equations can be mapped onto the graphics pipeline.

The most problematic aspect of morphologic volume analysis are the high memory access costs of dilation and erosion operators, when they are invoked for volume data. For structuring elements of size n, $2n^3$ data values have to be addressed per voxel. The general approach to cope with this efficiency problem is to decompose a large structuring element into several smaller ones. This is possible because the dilation and erasion operators obey the following relations as described in [Zhuang and Haralick 1986]:

$$\begin{aligned} X \ominus (Y \oplus Z) &= (X \ominus Y) \ominus Z \\ X \oplus (Y \oplus Z) &= (X \oplus Y) \oplus Z \end{aligned}$$

When a decomposition of a large structuring element into several smaller ones can be found, the morphological operation with the large element can be accomplished by the consecutive application of the smaller ones:

$$\begin{aligned} S &= H_1 \oplus H_2 \oplus \cdots \oplus H_n \quad , \\ X \ominus S &= (((X \ominus H_1) \ominus H_2) \cdots) \ominus H_n \quad , \\ X \oplus S &= (((X \oplus H_1) \oplus H_2) \cdots) \oplus H_n \quad . \end{aligned}$$

With this technique a diamond shaped structuring element of the form

$$Y_{j_1,\ldots,j_d} = c_1 \cdot \left(c_2 - \sum_k |j_k| \right)$$

with constants c_1 and c_2 can be decomposed into several smaller diamond shaped structures. For instance, if we decompose a structuring element of size n into several smaller elements of size 3, only $27(n-1)$ data values have to be addressed per voxel for one basic morphological operation, as 27 filter and 27 data values have to be addressed for a single voxel with a filter of size 3.

But there is still the better possibility to decompose the structuring element into several one-dimensional elements in a tensor product like fashion: $Y = ((Y^1 \oplus Y^2) \cdots) \oplus Y^d$ with

$$Y^i_{j_1,\ldots,j_d} = \begin{cases} c_1 \cdot \left(\frac{c_2}{d} - |j_i|\right) & j_k = 0 \;\; \forall\, k \neq i \\ N & \text{otherwise} \end{cases} \tag{4.5}$$

for which an example is depicted in Figure 4.2.

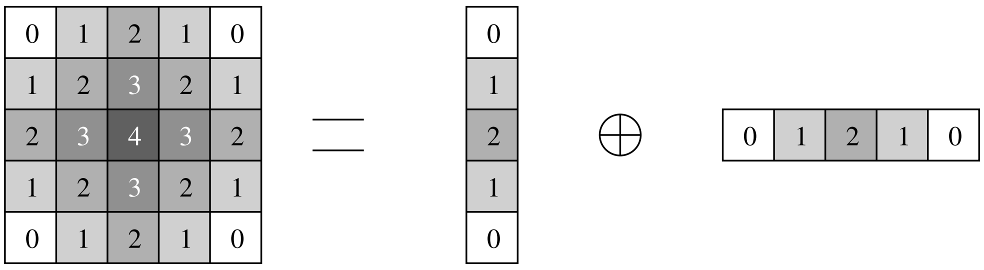


Figure 4.2: Example decomposition of a structuring element of size 5.

The structuring elements of the range shifted operators (4.3) and (4.4) can be decomposed similarly.

That way the computational complexity can be reduced to $6n$ addressed data cells per voxel. This is the approach that will be investigated in this chapter, though the basic algorithms are in no way restricted to this type of decomposition. For more information about structuring element decomposition I refer to [Zhuang and Haralick 1986]. [Steinberg 1986] gives a more thorough introduction to gray-scale morphology.

4.2 Related Work

Morphological operators are mainly used in pattern recognition to perform some kind of pre-segmentation filtering. Often they are used in combination with region growing methods [Hoehne and Hanson 1992]. Recently they have gained interest in visualization for processing raw data in order to reveal certain structures. One approach uses structuring elements of different sizes in order to do some kind of a hierarchical analysis, determining areas of different spatial frequencies of the data set as in [Lürig and Ertl 1998]. The filtering process takes two minutes and more for big data sets, despite the efforts to parallelize the operations.

With modern graphics hardware, nonlinear filters can be implemented using the programmable fragment pipeline [Viola et al. 2003]. With current graphics cards this approach is often slower than a solution based on standard OpenGL 1.3 calls, but there are a lot of other nonlinear filters that can only be implemented easily with the use of fragment programs.

One other highly effective filter is nonlinear diffusion, which is mainly used for noise reduction and feature enhancement. This filter is computationally very expensive, but it can be implemented using graphics hardware as well [Diewald et al. 2001, Rumpf and Strzodka 2001]. However, one cannot expect the same acceleration properties with this approach, as it is much more involved and less memory bandwidth bound.

4.3 Implementing Morphology with OpenGL

As I have already shown the feasibility of implementing graphics-hardware-accelerated three-dimensional separable filters in the last chapter, I will now address morphological operators. With the ability of almost all graphics accelerators to use a minimum/maximum blending function while rendering textured triangles to the framebuffer, these operators can be mapped perfectly onto the hardware pixel path, accelerating the time consuming filtering steps while retaining the volume data in the texture memory of the graphics hardware for the following visualization step. That way the volume data does not have to be reloaded for consecutive visualization steps, as it is the case with software-based approaches.

Most PC-based graphics cards support 3D textures nowadays. As access to this type of texture is still slower than to regular 2D textures, they are still often used for volume rendering with the 2D

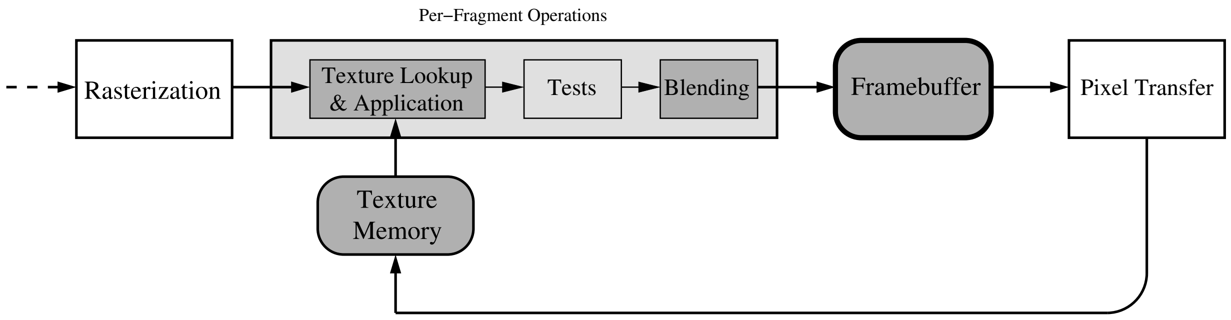


Figure 4.3: The per-fragment part of the OpenGL pipeline.

slicing approach. The algorithm has been implemented for both types of textures, but a serious bug in partial 3D texture reloading on a number of machines effectively disables the possible use right now. Therefore, we should concentrate on the 2D texture solution in this chapter. Note that we are still dealing with 3D data sets with this approach as well. The 3D texture version of the algorithm looks almost the same, but instead of switching the texture object the third texture coordinate would be changed, and the intermediate result would replace a slice in the 3D texture instead being stored in a 2D texture object like it has been shown in Chapter 3. Without loss of generality it can be assumed, that the texture planes contain data information perpendicular to the z axis.

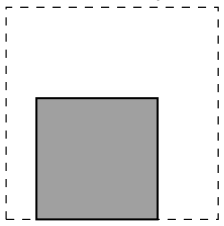
In order to map morphological operators onto the graphics pipeline (see Figure 4.3), we first have to find out where minimum and maximum operators are supported. It turns out, that only the per-fragment operations can perform minimum and maximum blending in the framebuffer, if the corresponding OpenGL extension is supported by the graphics driver.

As the basic algorithmic overview in Figure 4.4 shows, we can use the structure of the structuring element decomposition (4.5) to sweep over the volume in three passes. Note that the described approach can also be applied to morphological filters, that cannot be decomposed into several one-dimensional filters. However, the necessary computation time usually prohibits the usage of such operations. Nevertheless, the possible speedup would be even higher in these cases.

In the first two passes the filtering is performed along the x and y axes. These steps only access data from within one single texture, which contains data perpendicular to the z axis, as defined in the previous section. In the third pass we have to filter along the z axis. As the data along this axis is spread in multiple textures, triangles textured with several different images have to be rendered. The first two passes are combined in order to minimize texture binding changes.

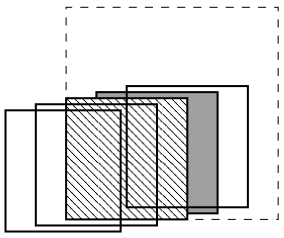
In every pass textured triangles that are translated by several pixels along the filtering axis are rendered into the framebuffer. There the incoming fragments are blended with the pixel data that is already contained in the buffer. By using the `minmax` blending extension the compositing step effectively calculates the maximum or minimum value of all texels within the filter scope. After a one-dimensional filter operation has been performed, the resulting image is read back from the framebuffer into texture memory.

```
1. Pass: For all z:
  – Clear frame buffer, select texture z
  – For all j: $\bar{Y}_j \neq \bar{N}$
    – Render textured polygons, shifted by (j,0),
      biased by $\bar{Y}_j$
```



(Example for j=1)

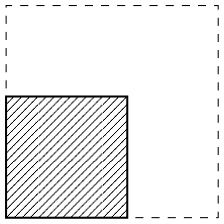
```
    – Blend results in the frame buffer
      to calculate minimum/maximum
```



(The shift along the y axis in this diagram is only used to clearify the situation)

```
  – Copy indicated region back to texture z
  – For all j: $\bar{Y}_j \neq \bar{N}$
    – Do the same, this time shifted by (0,j)

2. Pass: For all z:
  – Clear frame buffer
  – For all j: $\bar{Y}_j \neq \bar{N}$
    – Render polygons textured with texture z+j,
      biased by $\bar{Y}_j$
```



```
    – Blend results in the frame buffer
      to calculate minimum/maximum
  – Copy indicated region back to texture $z - j_{neg}$
```

Figure 4.4: The basic algorithmic structure.

In order to perform the dilation (4.3), we have to subtract a per polygon constant value from the fragments after texturing and before blending. This can be accomplished on Silicon Graphics systems by using the texture color lookup table with different lookup tables that only contain linear ramps, shifted by the according structuring element values. The lookup table has to be changed for every rendered polygon, therefore the loading process is accelerated by using predefined lookup tables contained in display lists. On NVIDIA systems, the same effect can be accomplished with the paletted texture extension, or even better with the register combiners extension. With the recent advances in chip design and driver technology NVIDIA clearly outperforms the filter rates that have been measured on an SGI Octane system.

In order to save texture memory, which is a scarce resource, we want to do the filtering in place, replacing the previously used textures. In the first pass this is accomplished implicitly, because the filter operations are performed for all pixels along the filter axis in one step. Therefore, we do not need the original data any more after the operator was invoked.

In the second pass the filter operations cannot be performed for all pixels along the filter axis in one step, as this involves several textures, and framebuffer contents cannot be copied back to multiple textures in one step. Therefore, we either have to use a second set of textures, which we wanted to avoid, or we have to store the resulting data to parts of the volume we will not address any more. As we are dealing with usually small or at least finite filters, we can see that we need j_{neg} spare textures, with

$$j_{\text{neg}} = \max\left\{ \min_{j}\left\{ j : \; \bar{Y}_j \neq \bar{N} \right\}, \; 0 \right\}$$

being the number of coefficients the structuring element extends to negative indices. Figure 4.5 shows an example, where a structuring element of size 3 is used, with the filter coefficients $\bar{Y}_{-1}$, $\bar{Y}_0$ and $\bar{Y}_1$. If no precautions had been taken, the textures that are needed in the next step would be overwritten by values from the framebuffer, which is indicated in the figure by dashed lines.

The basic algorithm can be optimized even more by removing the framebuffer clear command. When the first polygon of the inner loop is rendered with blending disabled, it will implicitly clear the relevant part of the framebuffer, and rendering polygons without blending is faster anyway. We just have to be sure, that the first polygon rendered is the one related to $\bar{Y}_0$, because it is drawn exactly at the position of the image to be read back after the filtering is complete (compare Figure 4.4).

For many applications of morphological operators, the same data set has to be filtered several times, using different operators of different sizes. When using the graphics hardware approach, it is clear that we can use the color channels to perform three or four operators in one filtering step, provided that we use the biggest morphological operator as the basic filter, combine it with the other operators and fill the remaining empty slots with the neutral element $\bar{N}$.

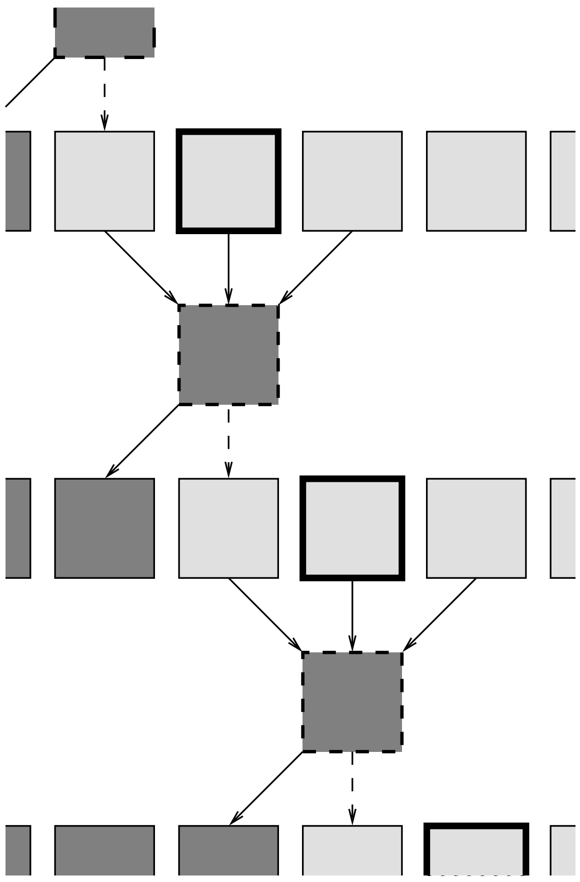


Figure 4.5: Handling texture copy overlaps.

4.4 Results and Comparison

Table 4.1 reveals that hardware-based morphological operators are much faster than well tuned software implementations. Compared with [Lürig and Ertl 1998] they easily beat multiprocessor implementations as well. Admittedly, the times measured there include two basic morphological operators with four different elements and the additional overhead for difference signal creation, thresholding and transfer function invocation. But with hardware-based operators, three or even four (when RGBA texture lookup tables are supported) filters can be applied in one cycle, reducing comparable times even more. By performing four different morphological operations in one cycle, we get filtering speedups of 20 times and more for large data sets and small to mid-sized filters on SGI systems, and even speedups of 120 times and more for mid-size data sets and large filters on modern PC hardware. These numbers clearly indicate, that the gap between software implementations and hardware-based solutions is still widening. Note that all methods work on 8 bit data structures in order to minimize memory consumption and to speed up data access.

The Intel-based systems seem to be superior to this kind of image processing, and modern graphics cards for PC systems have high fill rates that can perform morphological operators extremely

Data set size: 256^3

Kernel	pIII[†]	mips[*]	GFX[*]/RGBA[◇]	Speedup	P4[‡]	GFX[‡]/Lum	GFX[‡]/RGBA[◇]	Speedup
3^3	13.4	27.3	5.7	4.8×	6.29	1.3	0.54	11.6×
7^3	24.0	32.7	10.1	3.2×	7.5	1.46	0.54	13.9×
15^3	43.5	46.3	18.8	2.5×	27.1	1.76	1.09	24.8×
25^3	62.8	90.6	29.5	3.1×	59.0	2.16	1.78	33.1×

Data set size: $512^2 \times 154$

Kernel	pIII[†]	mips[*]	GFX[*]/Lum	Speedup	P4[‡]	GFX[‡]/Lum	Speedup
3^3	33.3	67.7	11.8	5.7×	15.8	3.18	5.0×
7^3	58.8	101.5	19.5	5.2×	18.8	3.47	5.4×
15^3	104.2	172.3	34.3	5.0×	69.7	4.24	16.4×
25^3	149.9	251.2	54.5	4.6×	113.1	5.2	21.7×

[*] SGI Octane with R10000, 250MHz, MXE graphics
[†] PC with Pentium III, 500 MHz, Intel BX chipset
[‡] PC with P4, 2.8 GHz, Intel 7205 chipset, NVIDIA GeForce FX5800 Ultra graphics
[◇] Four different morphological operations can be performed simultaneously with RGBA textures

Table 4.1: Times in seconds per basic operator.

fast. The performance of the software-based method is not processor architecture depended, though, as a faster Athlon processor produced inferior results due to the influences of the main board chip set, RAM technology, and cache sizes. Still, the performance of the modern architecture with dual channel DDR RAM is a bit disappointing, compared to the performance of the relatively old Pentium III system. It seems to be, that the higher memory throughput is mostly wasted by filling up the larger cache lines of the P4.

The performance of the software filtering system can be clearly improved by adapting the algorithm to deal with workloads in cache line size and by using the SSE prefetch operations of the P4. However, both optimizations are extremely dependent on the used processor, even more than the presented approach is dependent on the GPU. Even with these optimizations, it is very unlikely that the software filter can be accelerated by two orders of magnitude in order to keep up with the graphics-hardware-based approach.

Mathematical computations in the framebuffer are usually susceptible to accuracy loss due to the limited framebuffer depth [Teitzel et al. 1999, Hopf and Ertl 2000b, Teitzel et al. 2000, Strzodka 2002, Hadwiger et al. 2003]. As morphological operators only need integer operations and the range of the results does not exceed the domain, no precision loss occurs at all. Due to this exactness the algorithm is perfectly suited for medical applications as well [Iserhardt-Bauer et al. 2001]. Figure 4.6 shows how morphology can be used for the segmentation of aneurysms. The correctness of the implementation has been verified by several tests, of course, yielding the desired results. Figures 4.7 to 4.10 show sample result slices of the opening and closing operator working on a data set of size 256^3, using a filter of size 7^3.

In this chapter an OpenGL-hardware-based algorithm for morphological operators has been introduced, that utilizes the high fill rates of modern graphics hardware for accelerating time con-

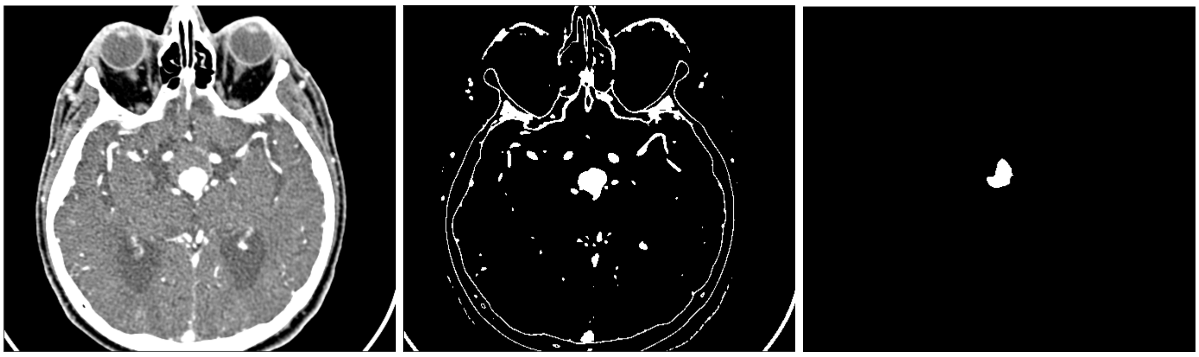


Figure 4.6: Segmentation of an aneurysm as explained in [Iserhardt-Bauer et al. 2001]. The images show one slide of the original data set, after applying the transfer function and thresholding, and after filtering with an morphological open operator.

suming 3D filtering up to 120 times and more for large data sets. Because of the structure of the operators, no loss of precision occurs at all, making this method attractive for all analysis performing morphological operations.

The approach that has been taken can be used for non-decomposed morphological operators and for non-separable linear filters as well, although these kinds of filters are not as often used as the ones already addressed. These and other types of filtering systems are subject of future work.

PC-based graphics outclassed high end workstations in the last few years. Recent advances show that the gap of the processing power between GPUs and CPUs is still widening. Current graphics cards are able to perform this type of filtering operation up to two orders of magnitude faster than the main processor, if not special measures are taken to accelerate the software solution as well.

As already indicated, there exist several hierarchical filter operations one would like to use in a GPU-accelerated form for large data sets. As we now have algorithms to use plain filters with graphics hardware, it is the next logical step to extend them to multiresolution techniques, which is the topic of the next chapter.

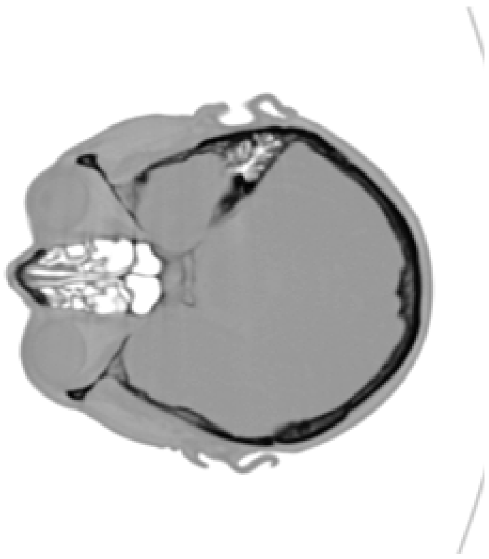


Figure 4.7: The original data set.

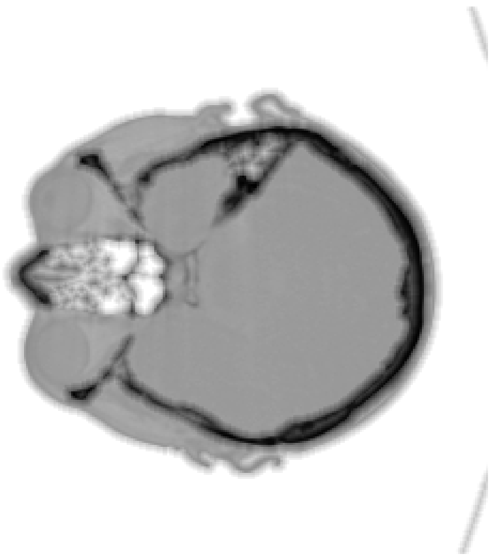


Figure 4.8: The opened data set.

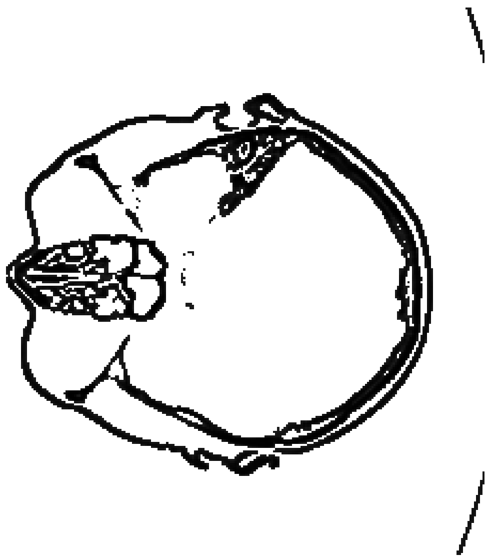


Figure 4.9: Difference to closed data set.

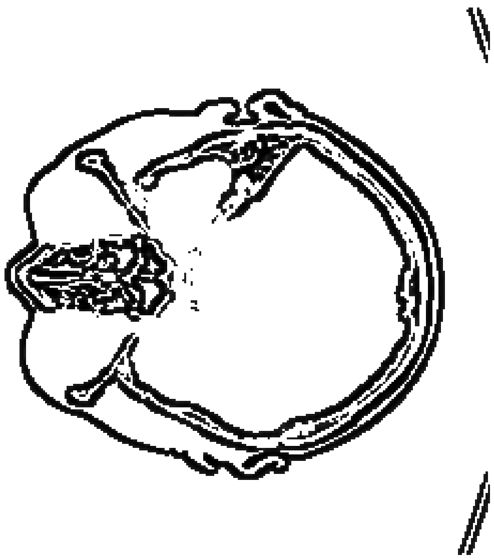


Figure 4.10: Difference to opened data set.

Sheridan: *I guess it's the old joke.*
You don't have to be crazy to work here,
but it helps.
I suppose there is a ... certain attraction
to being out here on the edge like this,
a new frontier.

Babylon 5, And Now For a Word

Chapter 5

Hierarchical Filters

For a lot of algorithmic problems, regular filters are not sufficient for the work with real-life data. As an example, feature detection with linear filters either finds more features than are really present, or it misses important edges in noisy data sets. These data sets are difficult to handle, one approach to deal with them is to exploit the observation that many types of features appear on many scales of the data set, representing different frequencies. This leads to multiresolution approaches with hierarchical filters. Other topics that are using multiscale bases in their computations include segmentation, registration, image enhancement, and compression techniques.

The most important multiresolution analysis nowadays is the wavelet theory. Due to the computational complexity of this approach no interactive visualization of the extraction process has been possible for large data sets. By using the hardware of modern graphics workstations for accelerating wavelet decomposition and reconstruction a first important step for reducing lags in the visualization cycle has been realized.

Wavelet analysis is a mainly memory-bandwidth bound problem. Graphics hardware on the other hand usually has memory systems that can be accessed extremely fast. With the support for two-dimensional convolution and the ability to scale bitmaps by arbitrary factors, all necessary steps needed for wavelet decomposition and reconstruction are available on graphics hardware. Additionally, three-dimensional convolution with separable filter kernels can be implemented by using these hardware supported convolution filters along with volume textures as shown in Chapter 3, paving the way to 3D wavelet analysis, which will benefit from the high memory bandwidth of the graphics hardware even more. As some issues of hardware-accelerated wavelet analysis get quite technical, I avoid the notation of 3D wavelet analysis in this chapter, and deal with 2D images only for clarity. Work on the first algorithm presented in this chapter has first been published at the Workshop on Vision, Modeling and Visualization VMV in [Hopf and Ertl 1999b], and the second, faster algorithm has first been published in an additional paper at the Eurographics Symposium on Visualization in [Hopf and Ertl 2000b]. The third algorithm, working with programmable graphics hardware, is unpublished so far.

5.1 Wavelets

In the past decades, wavelet analysis has grown from a mathematical curiosity into a major source of new basis decomposition and signal processing algorithms [Wickerhauser 1994, Strang and Nguyen 1996]. The importance of wavelets and multiresolution analysis resides in their hierarchical nature, which offers a mathematical framework for describing functions at different levels of resolution. Using basis functions with good approximation properties, i.e. with many vanishing moments, one can represent functions by keeping only the important coefficients and discarding all others. This section gives a short introduction into the basics of wavelet theory. A more detailed description can be found e.g. in [Chui 1992, Daubechies 1992, Louis et al. 1994].

A multiresolution analysis can be thought of as a ladder of approximating closed subspaces $(V_j)_{j\in\mathbb{Z}}$ of $L^2(\mathbb{R})$. The functions in these subspaces have well defined scaling and translation properties. Furthermore, there exists a function $\phi \in V_0$ such that $\{\phi_{j,n};\, j,n \in \mathbb{Z}\}$ with $\phi_{j,n} = 2^{j/2}\phi(2^j x - n)$ is an orthonormal basis of V_0. Under these conditions one can construct an orthonormal wavelet basis $\{\psi_{j,n};\, j,n \in \mathbb{Z}\}$ with $\psi_{j,n} = 2^{j/2}\psi(2^j x - n)$, such that for any function f in $L^2(\mathbb{R})$

$$P_j f = P_{j-1} f + Q_{j-1} f \ , \tag{5.1}$$

where P_j and Q_j are the orthogonal projections onto V_j and W_j, respectively:

$$P_j f = \sum_{n\in\mathbb{Z}} < f, \phi_{j,n} > \phi_{j,n} \ , \quad Q_j f = \sum_{n\in\mathbb{Z}} < f, \psi_{j,n} > \psi_{j,n} \ .$$

The function ψ is sometimes called the *mother* wavelet. The projection $P_j f$ onto the subspaces V_j corresponds to the different resolution levels in which the function f can be decomposed. These projections contain the *smooth* information of f at a given level of resolution. The projections $Q_j f$ onto the subspaces W_j spanned by the $\psi_{j,n}$ represent the *detail* information of f required to move from one resolution approximation subspace to the next finer one. (5.1) is the wavelet decomposition of the function f. The *scaling* function ϕ satisfies the *two-scale* relation

$$\phi = \sum_n h_n \phi_{1,n} \, , \tag{5.2}$$

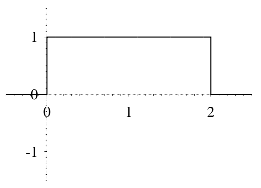
which defines a discrete *low-pass filter* operation with the filter $\{h_n\}_{n\in\mathbb{Z}}$.

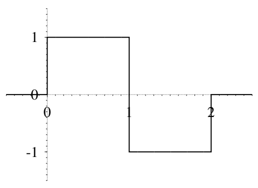
Now we start with a scale approximation $f^{j+1} = P_{j+1} f$ of a function f in V_{j+1} and decompose it into a coarser approximation in V_j. Due to the fact that $V_{j+1} = V_j \oplus W_j$, we have $f^{j+1} = f^j + \delta^j$, where $\delta^j = Q_j f$. In terms of the orthonormal bases $\{\phi_{j,n}\}_{n\in\mathbb{Z}}$ and $\{\psi_{j,n}\}_{n\in\mathbb{Z}}$, we have

$$f^j = \sum_n c_n^j \phi_{j,n} \ , \quad \delta^j = \sum_n d_n^j \psi_{j,n} \ ,$$

where the relation between the coefficients of the two levels of resolution is given by

$$c_n^{j-1} = \sum_k h_{k-2n} c_k^j \ , \quad d_n^{j-1} = \sum_k g_{k-2n} c_k^j \ , \quad \text{with} \quad g_n = (-1)^n h_{1-n} \ . \tag{5.3}$$





n		0	1	2	3
Haar	h_n	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$		
	g_n	$\frac{1}{\sqrt{2}}$	$-\frac{1}{\sqrt{2}}$		
Daub.	h_n	$\frac{1+\sqrt{3}}{4\sqrt{2}}$	$\frac{3+\sqrt{3}}{4\sqrt{2}}$	$\frac{3-\sqrt{3}}{4\sqrt{2}}$	$\frac{1-\sqrt{3}}{4\sqrt{2}}$
	g_n	$\frac{1-\sqrt{3}}{4\sqrt{2}}$	$\frac{-3+\sqrt{3}}{4\sqrt{2}}$	$\frac{3+\sqrt{3}}{4\sqrt{2}}$	$\frac{-1-\sqrt{3}}{4\sqrt{2}}$

Figure 5.1: The Haar scaling function, mother wavelet, and filter coefficients for Haar and Daubechies (4).

The coefficients h_n and g_n are the low-pass and high-pass filters, respectively. The decimation by a factor 2 corresponds to a down-sampling when going from one level to the next coarser one. This decomposition can be continued using the relation $V_{j+1} = V_j \oplus W_j$ and so on until a given level $J < j$, obtaining the following approximation for f:

$$f^{j+1} = \delta^j + \cdots + \delta^{J+1} + \delta^J + f^J$$

The inverse operation, the reconstruction of f^{j+1} from f^j and δ^j, is simply given by:

$$c_k^{j+1} = \sum_n (h_{k-2n}\, c_n^j + g_{k-2n}\, d_n^j) \tag{5.4}$$

Now let us take a look at an example. The simplest possible wavelet is the *Haar* wavelet. Figure 5.1 depicts its scaling function and the mother wavelet together with the filter coefficients.

We will now decompose a set of coefficients c_k^j into the c_k^{j-1} of the next coarser level. In Figure 5.2 the decomposition process is explained. The input data are convolved with the filter kernels h_n and g_n and down-sampled by a factor of 2. This process can be continued with the low-pass filtered coefficients c_k^{j-1}, until only one coefficient is left.

In order to reconstruct the original signal, the low- and high-pass filtered coefficients are processed as shown in Figure 5.3. The coefficients are up-sampled and then convolved with the reverted filter kernels according to (5.4).

So far we have only looked at one-dimensional data. For higher dimensions bases which are tensor products of the one-dimensional case are used. There exist other approaches for selecting orthogonal basis functions, but tensor product wavelets are easier to understand and faster to compute. The main disadvantage is the isotropic behavior of the filters.

Figure 5.4 reveals how a two-dimensional wavelet decomposition is performed. First, the data are decomposed line by line in the direction of the x-axis. In this example both the low-pass and the high-pass filtered data are stored side by side to each other, so they can be filtered in a second step column by column in the direction of the y-axis. Afterwards, the lower left part of the final figure reveals the two-dimensional low-pass filtered data, the upper left and lower right parts contain the perpendicular to one of the axes high-pass filtered data, and the upper right area shows the completely high-pass filtered coefficients.

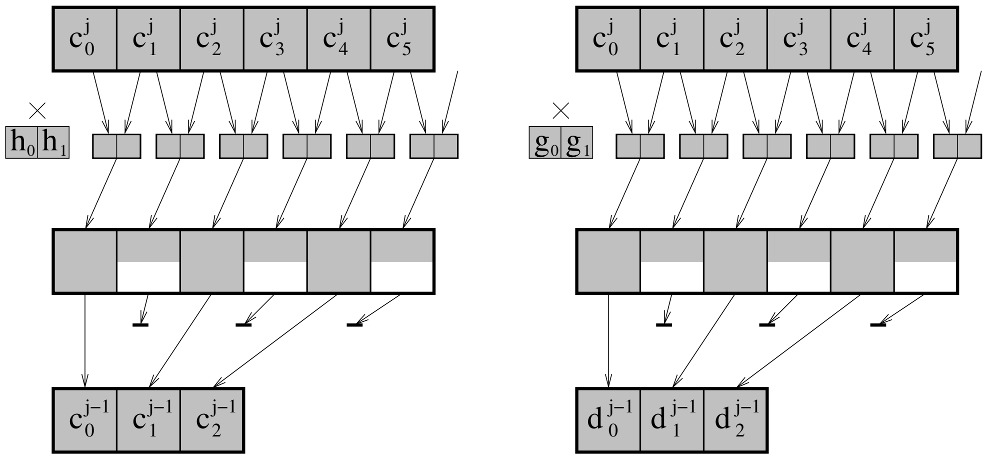

c_0^j c_1^j c_2^j c_3^j c_4^j c_5^j
×
h_0 h_1
g_0 g_1
c_0^{j-1} c_1^{j-1} c_2^{j-1}
d_0^{j-1} d_1^{j-1} d_2^{j-1}

Figure 5.2: Decomposition using Haar wavelets.

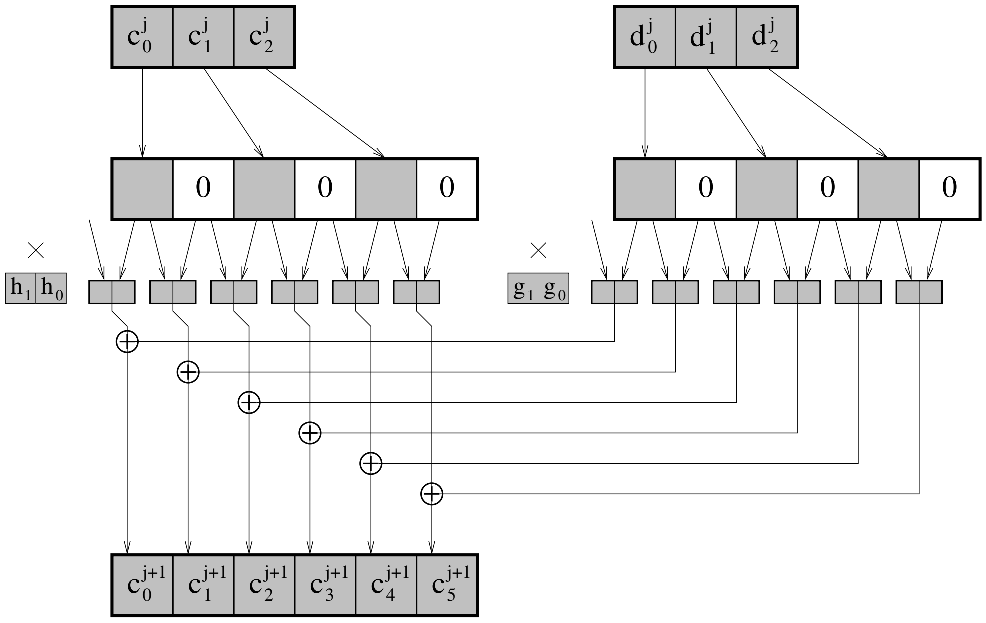


Figure 5.3: Reconstruction using Haar wavelets.

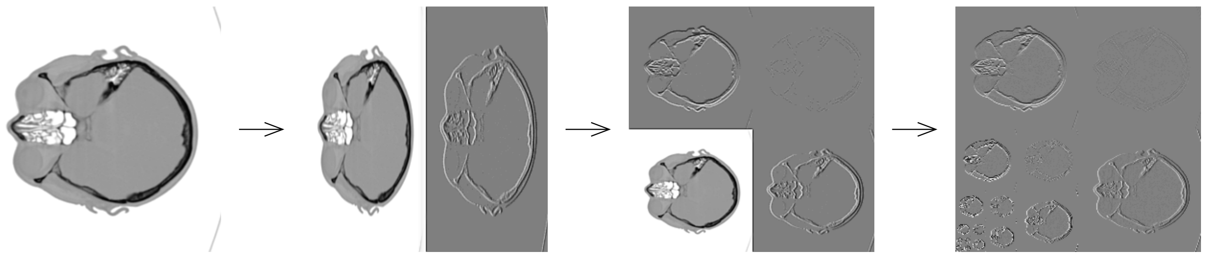


Figure 5.4: Two-dimensional wavelet decomposition using tensor product wavelets.

5.2 Related Work

Feature extraction has been proven to be a useful utility for segmentation and registration in volume visualization [Lürig et al. 1997a, Westermann and Ertl 1997]. Many edge detection algorithms used in this procedure employ wavelets or related basis functions for the internal representation of the volume. Additionally, wavelets can be used for fast volume visualization [Lippert et al. 1997] using the Fourier rendering approach [Malzbender 1993, Totsuka and Levoy 1993].

Many modern media compression algorithms are based on wavelet decompositions, for example JPEG2000 [JTC1/SC29 2002] for true color images. For scientific visualization, large volume data sets can be compressed with wavelets and rendered interactively [Guthe et al. 2002].

A new technique, called the lifting scheme, has been developed for the design of new wavelets [Sweldens 1997] and the acceleration of the fast decomposition and reconstruction steps [Daubechies and Sweldens 1998]. Incorporating this concept into future work will improve the software filtering times especially for large wavelet sizes, and it can be applied to the hardware-accelerated filters as well, reducing the number of necessary texture lookups.

Using lifting for the design of wavelets will usually lead to biorthogonal wavelets. This chapter, however, deals only with regular wavelets. Decomposition and reconstruction will work with biorthogonal wavelets as well, as only the filter specification has to be changed, but the adaption of data scaling (Section 5.5) remains future work.

Another interesting approach that could be useful for hardware-based implementations uses wavelets that map integers to integers [Calderbank et al. 1998], removing all accuracy problems. However, the theory is quite involved, and its implications for the SIMD programming model are unclear so far.

Using graphics hardware for the acceleration of hierarchical filters is not bound tightly to wavelets only, for instance see [Strzodka and Rumpf 2001] how level sets can be implemented with GPUs.

5.3 A New View on the Rendering Pipeline

As it can be directly derived from (5.3) and (5.4), wavelet decomposition is practically done by an input signal filtering and a down-sampling step. Reconstruction on the other hand is performed by first up-sampling and filtering afterwards. Graphics workstations support filtering and scaling (resampling) for image transfer operations, which will be utilized for hardware-based wavelet decomposition and reconstruction. The relevant part of the the OpenGL graphics pipeline is depicted in Figure 5.5. Modern PC graphics adapters have more flexibility with their programmable fragment pipelines, which will be used for a fast reconstruction scheme using textures for the input data. Figure 5.6 shows the relevant part of the OpenGL graphics pipeline for using programmable graphics hardware.

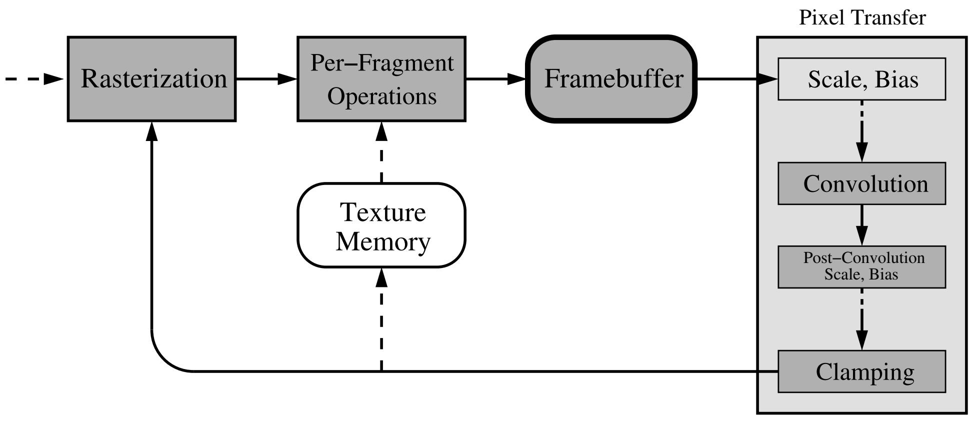


Figure 5.5: Relevant parts of the OpenGL graphics pipeline, using the imaging pipeline.

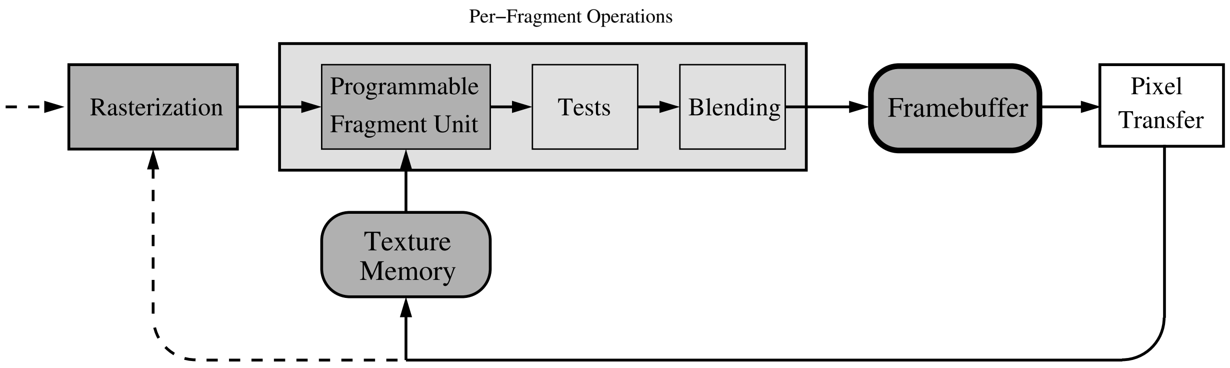


Figure 5.6: Relevant parts of the OpenGL graphics pipeline, using programmable GPUs.

First, I will concentrate on the implementation of wavelet transformations with the OpenGL imaging pipeline. In order to simplify the description of the process the process will be shown for a one-dimensional wavelet transformation. As we have seen in the previous chapter, tensor-product-based multi-dimensional wavelet transformations are a straight-forward extension to this approach. In particular, Chapter 3 covers the details of how to employ 3D texture hardware in order to perform three-dimensional convolution with separable filter kernels, which can be easily extended to cope with wavelet transformations.

Now let us consider how the graphics pipeline works on image data. When a rectangular part of the framebuffer is to be copied from a source area, its color values are piped through the pixel transfer system, the rasterizer and the per-fragment operation system before they are written to the destination area. Pixel transfer includes scaling and biasing of the color values, convolution with a prior defined filter kernel and clamping to the usual color value range $[0,1)$. The rasterizer transposes the input image to the designated destination area while zooming it with arbitrary zoom factors, in other words, it performs up- and down-sampling. In the final per-fragment operations step, the resulting pixel values are blended with the pixel values of the destination area using several pre-defined blending functions. This step includes a final clamping step as well.

In order to map the wavelet transformation onto the graphics hardware, a mathematical specification of the pixel operations of the graphics pipe is needed. Let p^{n+1} be the pixel data that results from a graphical operation on p^n. Again, for simplification we can assume that p^n is one-dimensional. A first approximation of the relevant part of the graphics pipeline can be written as a composition of a convolution (co), two clamping steps (cl), a transposition (tr), the scaling step (sc), and a blending operation (bl):

$$\begin{aligned}
p^{n+1} &= \mathrm{cl} \circ \mathrm{bl} \circ \mathrm{sc} \circ \mathrm{tr} \circ \mathrm{cl} \circ \mathrm{co}\,(p^n) && (5.5)\\
\mathrm{bl}(p_i) &= \Gamma(p_i, p_i^n) && (5.6)\\
\mathrm{sc}(p_i) &= p_{\lfloor zi \rfloor} && (5.7)\\
\mathrm{tr}(p_i) &= p_{i-x_s+x_d} && (5.8)\\
\mathrm{cl}(p_i) &= \max(0, \min(1, p_i)) && (5.9)\\
\mathrm{co}(p_i) &= s \cdot \sum_{j=0}^{m} k_j\, p_{i+j} \; + b\,, && (5.10)
\end{aligned}$$

with zoom z, source x_s and destination x_d position, scaling s, and bias b parameters, and with a convolution kernel k of size m. As explained above, (co) is performed in the pixel transfer system, (tr) and (sc) describe the task of the rasterizer, and (bl) illustrates the per-fragment operations. Clamping (cl) happens in both, pixel transfer and per-fragment operations.

These equations are applied to pixels p_i^{n+1} of the destination area $i \in [x_d, (x_d + w + 1 - m) \cdot z)$, with w being the image size. The remaining pixels stick to their old values, that is, they are equal to p_i^n. The blending function Γ can be chosen from a predefined set. For wavelet filter operations identity $\Gamma_{\mathrm{id}}(x,y) = x$, addition $\Gamma_{\mathrm{add}}(x,y) = x + y$ and subtraction $\Gamma_{\mathrm{sub}}(x,y) = y - x$ are used.

As we now have a mathematical model of the rendering pipeline, the problem of mapping wavelet transformations onto the hardware can be addressed as the next logical step.

5.4 Implementing Wavelets with OpenGL

Due to the individual abilities of graphics hardware, different strategies have to be used for hardware-based implementations of wavelet transformations. For GPUs without a programmable fragment pipeline, the convolution has to be performed with the imaging pipeline. For programmable GPUs, the convolution can be coded explicitly. As this simplifies implementation issues, only the more complicated reconstruction process will be examined in this chapter.

But first I will concentrate on the implementation of wavelet decomposition and reconstruction using the OpenGL imaging pipeline.

5.4.1 Decomposition using the Imaging Pipeline

Compared to the order of operations in the graphics pipeline, of which the relevant part is depicted in Figure 5.5, wavelet decomposition fits neatly into its scheme. Remembering that scaling is a part of the rasterization process, convolution is performed in the graphics pipe just before image scaling.

When we write the wavelet decomposition in (5.3) as

$$\breve{c}_n^{j-1} = \sum_i h_i c_{n+i}^j\,, \quad \breve{d}_n^{j-1} = \sum_i g_i c_{n+i}^j\,, \tag{5.11}$$

$$c_n^j = \breve{c}_{2n}^j\,, \qquad d_n^j = \breve{d}_{2n}^j \tag{5.12}$$

and compare it to (5.5) to (5.10), we see that each of the wavelet decomposition filter steps matches the calculations of the OpenGL graphics pipe perfectly, except for the clamping steps. (5.7) implements the down-scaling in (5.12) and (5.11) can be expressed with the convolution filters in (5.10). Clamping introduces several problems to these algorithms, that have to be addressed by using arbitrary scale and bias parameters. This aspect is discussed in detail in Section 5.5.

One thing to note is that the image data p_j^n as well as the filter kernel k_j are only defined for $j \geq 0$. The filter kernel size is further limited by hardware-specific constants, which are rather small. Thus it is necessary to displace the filter kernel and the input and output image specifications before invocation. Of course, the displacement has to be compensated in the final convolution step.

The input data have to be convolved using two different filters, so either the resulting images have to be written to another part of the framebuffer, or both filters have to be used together in one step. As we are usually dealing with multi-dimensional data, we can build a tensor product of two dimensions of the higher dimensional filter and filter the data set in half the number of passes that would be necessary if we would work with one-dimensional filters only. By combining both tensor product steps with the two different filters we get a total of four filters that have to be applied to the data.

The first implementation uses a framebuffer layout similar to Figure 5.4. Because the input data have to be convolved using two different filters, the resulting images have to be written to another part of the framebuffer so that the original data set is not overwritten. These two parts of the framebuffer can be used alternately when tensor product wavelet decompositions have to be computed.

Unfortunately, OpenGL is no pixel-exact specification. In particular, zooming is only well defined according to (5.7) for up-sampling, that is for zoom factors greater than one. When images are scaled down, it is up to the implementation which pixels to transfer. Even the implementations of one vendor — Silicon Graphics in this case — vary from architecture to architecture. In order to address this problem, a so-called *shift offset* δ is determined. When added to the specification of the source image's left edge, it corrects the internal pixel offset. Currently the only way to determine the shift offset is to draw a scaled-down version of a well-known image for several different shift values and to read it back afterwards for comparison with the desired result.

Additionally, care has to be taken at the borders of the input image. Several strategies have already been discussed, with blanking being the easiest and input mirroring being one of the best

```
Create convolution filters: $\tilde{h}_i = h_{i+\alpha_h}$ ,   $\tilde{g}_i = g_{i+\alpha_g}$ .
Set pixel zoom to 0.5 .
Set blending function to $\Gamma_{\mathrm{id}}$ .

Set post-convolution scaling to $s_h$ .
Set post-convolution bias to $b_h$ .
Copy area $[\delta+\alpha_h+i\,,\ \delta+\alpha_h+i+w+\Delta_h-1)$ to $[o_c\,,\ o_c+\frac{1}{2}w)$,
    using convolution filter $\tilde{h}$ (size $\Delta_h$).

Set post-convolution scaling to $s_g$ .
Set post-convolution bias to $b_g$ .
Copy area $[\delta+\alpha_g+i\,,\ \delta+\alpha_g+i+w+\Delta_g-1)$ to $[o_d\,,\ o_d+\frac{1}{2}w)$,
    using convolution filter $\tilde{g}$ (size $\Delta_g$).
```

h_j, g_j	Low- and high-pass filters, respectively
α_x	Index of first non-zero element of filter x
Δ_x	Size of filter x
δ	Shift offset (see text)
i, w	Input image offset and size
o_c, o_d	Output image offsets for low- and high-pass filtered coefficients
s_h, s_g	Scaling parameters for filters h and g
b_h, b_g	Bias parameters for filters h and g

Figure 5.7: Implementation sequence for OpenGL-based wavelet decomposition with luminance only convolution.

methods in order to suppress high frequencies that are not part of the image, but introduced by aliasing effects. Figure 5.7 shows the implementation sequence for the first algorithm implementing wavelet decomposition using graphics hardware. The calculation of the scaling and bias values, which is left out here for clarity, is discussed in detail for the one-dimensional case in Section 5.5.

As the graphics pipeline always works on RGBA images, another possibility to implement the filter arises by using RGBA convolution filters instead of luminance only filters to combine these four steps into one as depicted in Figure 5.8. This will speed up the decomposition significantly, as the raster manager needs to address only one fourth of the number of pixels of the previous mentioned approach, and the convolution pipeline is implemented for color filters anyway. Additionally, we do not have to copy the source image in order to save it for the second filter, which makes for another factor of two.

However, it turns out that we still have to copy the source image, because OpenGL does not provide a pre-convolution color matrix, which would be necessary to provide the same information to the four different filters. As only the low-pass filtered data of the previous step should be addressed, which is stored in the red component of the calculated image, this information has to be spread to all four color channels using SGI's color matrix OpenGL extension before invoking the convolution filter. Still, this approach has the advantage of better utilization of the graphics

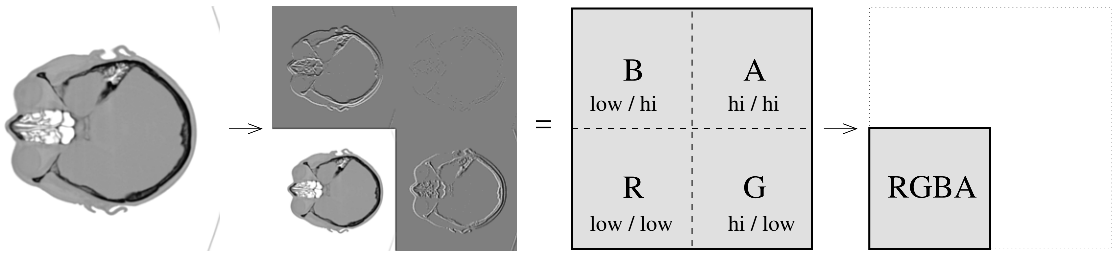


Figure 5.8: Using one RGBA convolution instead of four different luminance only convolutions.

```
Create convolution filter: $\tilde{h}_j = h_{j+\alpha}$ ,   $\tilde{g}_j = g_{j+\alpha}$ ,
$\mathbf{f}^R_{j,k} = \tilde{h}_j \cdot \tilde{h}_k$ ,   $\mathbf{f}^G_{j,k} = \tilde{g}_j \cdot \tilde{h}_k$ ,   $\mathbf{f}^B_{j,k} = \tilde{h}_j \cdot \tilde{g}_k$ ,   $\mathbf{f}^A_{j,k} = \tilde{g}_j \cdot \tilde{g}_k \quad \forall j,k$ .
Calculate scaling $\mathbf{s}$ and bias $\mathbf{b}$ .

Set post-convolution scaling to $\mathbf{s}$ .
Set post-convolution bias to $\mathbf{b}$ .
Set pixel zoom to $1.0 \times 1.0$ .   Set color matrix to $\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$ .

Copy area $[\delta_x + \alpha + i_x , \delta_x + \alpha + i_x + w_x + \Delta - 1) \times [\delta_y + \alpha + i_y , \delta_y + \alpha + i_y + w_y + \Delta - 1)$
  to $[o_x , o_x + w_x + \Delta - 1) \times [o_y , o_y + w_y + \Delta - 1)$.

Set pixel zoom to $0.5 \times 0.5$ .   Disable color matrix.

Copy area $[o_x , o_x + w_x + \Delta - 1) \times [o_y , o_y + w_y + \Delta - 1)$
  to $[o_x , o_x + \frac{1}{2}w_x) \times [o_y , o_y + \frac{1}{2}w_y)$, using convolution filter $\mathbf{f}$ (size $\Delta^2$).
```

h_j, g_j	Low- and high-pass filters, respectively
α	Index of first non-zero element of both filters
Δ	Size of filters
δ	Shift offset (see text)
i, w	Input image offset and size
o	Output image offsets
$\mathbf{s}$	Scaling parameters for filters $\mathbf{f}^*$
$\mathbf{b}$	Bias parameters for filters $\mathbf{f}^*$

Figure 5.9: Implementation sequence for OpenGL-based wavelet decomposition with RGBA convolution.

pipe.

Finally, Figure 5.9 shows the implementation sequence for the second algorithm implementing wavelet decomposition using graphics hardware. The calculation of the scaling and bias values have been left out here for clarity as well. Please note, that **s** and **b** are four-dimensional vectors in this case, that contain the values for the four different filters $\mathbf{f}^R$, $\mathbf{f}^G$, $\mathbf{f}^B$, and $\mathbf{f}^A$. All values have to be calculated independently from each other by the formulas discussed in Section 5.5.

5.4.2 Reconstruction using the Imaging Pipeline

In contrast to the decomposition algorithm, wavelet reconstruction is much more complicated, because according to (5.4) scaling and convolution is to be performed in inverse order compared to the rendering pipeline (Figure 5.5). Either scaling and convolution have to be performed in separate rendering steps, or the filters have to be split and special care has to be taken in order to render even and odd pixel positions separately. Either way, reconstruction is more complicated than decomposition.

We will discover in Section 5.5 that using separate rendering steps is not a feasible option. Therefore, we should concentrate on the second possibility of splitting the filters.

Consider the wavelet reconstruction in (5.4). In order to simplify the expression, we have to distinguish between k being even and odd. For even k we substitute h_{k-2n} using $h_n^{\text{ev}} = h_{-2n}$ (g accordingly) and get

$$\bar{c}_n^{j+1} = \sum_i (h_i^{\text{ev}} c_{i+n}^j + g_i^{\text{ev}} d_{i+n}^j) \ , \tag{5.13}$$

$$c_k^{j+1} = c_{2n}^{j+1} \ = \ \bar{c}_n^{j+1} \ . \tag{5.14}$$

For odd k we use $h_n^{\text{od}} = h_{1-2n}$, which results in

$$\hat{c}_n^{j+1} = \sum_i (h_i^{\text{od}} c_{i+n}^j + g_i^{\text{od}} d_{i+n}^j) \ , \tag{5.15}$$

$$c_k^{j+1} = c_{2n+1}^{j+1} \ = \ \hat{c}_n^{j+1} \ . \tag{5.16}$$

Again, we will concentrate on the low-pass filtered data first and simply neglect g in the terms above. We can see that (5.14) and (5.16) can be performed by setting according zoom factors in (5.7). (5.13) and (5.15) can be implemented in (5.10) by choosing h^{ev} and h^{od} as filter kernels, respectively. The blending function is set to Γ_{id} for this step, just as in the decomposition mechanism.

Of course, when rendering the odd coefficients, we have to make sure that we do not overwrite the previously rendered even coefficients. OpenGL knows about a so-called *stencil* buffer, which provides masking tests in the per-fragment operation part of the graphics pipeline. The stencil buffer has to be initialized with a striped pattern only once, after that the stencil test can be set to render even or odd pixels only. The test is activated for rendering odd pixels only due to speed reasons, as each activated test can slow down the rendering process.

Up to now we have only dealt with the low-pass filtered coefficients c_n^j. The next step is to add the convoluted d_n^j to the values that already reside in the framebuffer. Therefore, another rendering step is performed in which the high-pass filtered coefficients with the convolution kernels g^{ev} and g^{od} are copied just over the previously low-pass convolved coefficients. This time, Γ_{add} is selected as the blending function, by which the rendered data are added to the values in the framebuffer rather than overwriting them.

$$
\begin{aligned}
\tilde{h}_j^{\mathrm{ev}} &= h_{2\lfloor\frac{\alpha_h+\Delta_h}{2}\rfloor-2j}\,, & \tilde{h}_j^{\mathrm{od}} &= h_{2\lceil\frac{\alpha_h+\Delta_h}{2}\rceil-2j+1}\\
\tilde{g}_j^{\mathrm{ev}} &= g_{2\lfloor\frac{\alpha_g+\Delta_g}{2}\rfloor-2j}\,, & \tilde{g}_j^{\mathrm{od}} &= g_{2\lceil\frac{\alpha_g+\Delta_g}{2}\rceil-2j+1}\\
\delta_h^{\mathrm{ev}} &= -\lfloor\frac{\alpha_h+\Delta_h-1}{2}\rfloor\,, & \delta_h^{\mathrm{od}} &= 1-\lceil\frac{\alpha_h+\Delta_h-1}{2}\rceil\\
\delta_g^{\mathrm{ev}} &= -\lfloor\frac{\alpha_g+\Delta_g-1}{2}\rfloor\,, & \delta_g^{\mathrm{od}} &= 1-\lceil\frac{\alpha_g+\Delta_g-1}{2}\rceil\\
\Delta_h^{\mathrm{ev}} &= -\delta_h^{\mathrm{ev}}-\lceil\frac{\alpha_h}{2}\rceil+1\,, & \Delta_h^{\mathrm{od}} &= -\delta_h^{\mathrm{od}}-\lfloor\frac{\alpha_h}{2}\rfloor+1\\
\Delta_g^{\mathrm{ev}} &= -\delta_g^{\mathrm{ev}}-\lceil\frac{\alpha_g}{2}\rceil+1\,, & \Delta_g^{\mathrm{od}} &= -\delta_g^{\mathrm{od}}-\lfloor\frac{\alpha_g}{2}\rfloor+1
\end{aligned}
$$

h_j, g_j	Low- and high-pass filters, respectively
α_x	Index of first non-zero element of filter x
Δ_x	Size of filter x
δ_x^*	Input image offset for filter x

Figure 5.10: Filter specifications for OpenGL-based reconstruction.

Unfortunately, the second clamping step in (5.5) prohibits values < 0 to be correctly subtracted from the framebuffer. Therefore, the same convolution has to be rendered twice, one time using the scale and bias values discussed in the next section and Γ_{add} as blending function, one time using the negated scale and bias values, using Γ_{sub} for blending.

As we are up-sampling during reconstruction, we do not have to care about any shift offsets during zooming, as the OpenGL specification is pixel-exact in this case. However, hardware filter kernels h_k can only be specified for non-negative k. Together with the problem of odd sized filter kernels this leads to quite complex filter kernel specifications, which can be noted in Figure 5.10.

The implementation sequence of the filtering itself is depicted in Figure 5.11. The computation of scaling and bias values is discussed in detail in Section 5.5. Two bias values have to be calculated separately per filter, as the filters have to be split into one for even and one for odd filter coefficients. Again, care has to be taken about image borders as well. The policy here depends heavily on the policy taken during the decomposition step. Note that Haar wavelets are quite uncomplicated here, as the reconstruction filters have the size 1, which is a mere scaling.

If the low- and high-pass filtered coefficients of a two-dimensional wavelet transformation are stored in the framebuffer layout as used in the second decomposition algorithm (Figure 5.9), a faster version of the reconstruction algorithm can be used. As the necessary high-pass filtered coefficients d_n^j are stored as another component of the same pixels, SGI's color matrix extension can be used to combine them. Again, all four red, green, blue, and alpha components are used in order to work on 2D tensor product wavelets in one step. This is different to the first approach, where we treated the different coefficients in separate steps. This second approach is not only faster, but even more accurate, because color matrix operations are performed with higher precision than blending operations in the framebuffer, and we do not have to deal with

```
Set pixel zoom to 2.0 .
Initialize stencil buffer with $\begin{cases} 0 & \text{even pixels} \\ 1 & \text{odd pixels} \end{cases}$ .
Disable stencil test.

Set blending function to $\Gamma_{\text{id}}$ .
Set post-convolution scaling and bias to $\bar{s}_h$ and $\bar{b}_h^{\text{ev}}$ .
Copy area $[i_c + \delta_h^{\text{ev}}, i_c + \delta_h^{\text{ev}} + w + \Delta_h^{\text{ev}} - 1)$ to $[o, o + 2w)$, using filter $\tilde{h}^{\text{ev}}$ (size $\Delta_h^{\text{ev}}$) .

Set blending function to $\Gamma_{\text{add}}$ .
Set post-convolution scaling and bias to $\bar{s}_g$ and $\bar{b}_g^{\text{ev}}$ .
Copy area $[i_d + \delta_g^{\text{ev}}, i_d + \delta_g^{\text{ev}} + w + \Delta_g^{\text{ev}} - 1)$ to $[o, o + 2w)$, using filter $\tilde{g}^{\text{ev}}$ (size $\Delta_g^{\text{ev}}$) .

Set blending function to $\Gamma_{\text{sub}}$ .
Set post-convolution scaling and bias to $-\bar{s}_g$ and $-\bar{b}_g^{\text{ev}}$ .
Copy area $[i_d + \delta_g^{\text{ev}}, i_d + \delta_g^{\text{ev}} + w + \Delta_g^{\text{ev}} - 1)$ to $[o, o + 2w)$, using filter $\tilde{g}^{\text{ev}}$ (size $\Delta_g^{\text{ev}}$) .

Enable stencil test, render only pixels with stencil value 1.

Set blending function to $\Gamma_{\text{id}}$ .
Set post-convolution scaling and bias to $\bar{s}_h$ and $\bar{b}_h^{\text{od}}$ .
Copy area $[i_c + \delta_h^{\text{od}}, i_c + \delta_h^{\text{od}} + w + \Delta_h^{\text{od}} - 1)$ to $[o, o + 2w)$, using filter $\tilde{h}^{\text{od}}$ (size $\Delta_h^{\text{od}}$) .

Set blending function to $\Gamma_{\text{add}}$ .
Set post-convolution scaling and bias to $\bar{s}_g$ and $\bar{b}_g^{\text{od}}$ .
Copy area $[i_d + \delta_g^{\text{od}}, i_d + \delta_g^{\text{od}} + w + \Delta_g^{\text{od}} - 1)$ to $[o, o + 2w)$, using filter $\tilde{g}^{\text{od}}$ (size $\Delta_g^{\text{od}}$) .

Set blending function to $\Gamma_{\text{sub}}$ .
Set post-convolution scaling and bias to $-\bar{s}_g$ and $-\bar{b}_g^{\text{od}}$ .
Copy area $[i_d + \delta_g^{\text{od}}, i_d + \delta_g^{\text{od}} + w + \Delta_g^{\text{od}} - 1)$ to $[o, o + 2w)$, using filter $\tilde{g}^{\text{od}}$ (size $\Delta_g^{\text{od}}$) .
```

h^*, g^*	Low- and high-pass filters, respectively, as specified in Figure 5.10
δ_x^*	Input image offset for filter x
Δ_x^*	Sizes of filter x
i_c, i_d, w	Input image offsets for low- and high-pass filtered coefficients, and size of input image
o	Output image offset
$\bar{s}_h$, $\bar{s}_g$	Scaling parameters for filters h and g
$\bar{b}_h^{\text{ev}}$, $\bar{b}_h^{\text{od}}$, $\bar{b}_g^{\text{ev}}$, $\bar{b}_g^{\text{od}}$	Bias parameters for filters h and g

Figure 5.11: Implementation sequence for OpenGL-based wavelet reconstruction with luminance only convolution.

```
Create convolution filters:
$\mathbf{f}_{j,k}^{x,y,R} = \tilde{h}_j^x \cdot \tilde{h}_k^y\,,\quad \mathbf{f}_{j,k}^{x,y,G} = \tilde{g}_j^x \cdot \tilde{h}_k^y\,,\quad \mathbf{f}_{j,k}^{x,y,B} = \tilde{h}_j^x \cdot \tilde{g}_k^y\,,\quad \mathbf{f}_{j,k}^{x,y,A} = \tilde{g}_j^x \cdot \tilde{g}_k^y \quad \forall j,k\,,\ \forall x,y \in \{\mathrm{ev},\mathrm{od}\}\ .$
Calculate scaling $\mathbf{s}$ and bias $\mathbf{b}^{x,y}$, $x,y \in \{\mathrm{ev},\mathrm{od}\}$ .
Set pixel zoom to $2.0 \times 2.0$ .   Enable rendering to R only, disable rendering to G, B, and A.
Set color matrix to $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$ .   Initialize stencil buffer with $\begin{cases} 0 & x \text{ even}, y \text{ even} \\ 1 & x \text{ odd}, y \text{ even} \\ 2 & x \text{ even}, y \text{ odd} \\ 3 & x \text{ odd}, y \text{ odd} \end{cases}$ .
Disable stencil test.   Set post-convolution scaling and bias to $\bar{\mathbf{s}}$ and $\bar{\mathbf{b}}^{\mathrm{ev},\mathrm{ev}}$ .
Copy area $[i_x + \delta^{\mathrm{ev}}\,,\, i_x + \delta^{\mathrm{ev}} + w_x + \Delta^{\mathrm{ev}} - 1) \times [i_y + \delta^{\mathrm{ev}}\,,\, i_y + \delta^{\mathrm{ev}} + w_y + \Delta^{\mathrm{ev}} - 1)$
    to $[o_x\,,\, o_x + 2w_x) \times [o_y\,,\, o_y + 2w_y)$, using convolution filter $\mathbf{f}^{\,\mathrm{ev},\mathrm{ev}}$ (size $\Delta^{\mathrm{ev}} \times \Delta^{\mathrm{ev}}$) .

Do $\forall \{x,y,t\} \in \big\{\ \{\mathrm{od},\mathrm{ev},1\}\,,\ \{\mathrm{ev},\mathrm{od},2\}\,,\ \{\mathrm{od},\mathrm{od},3\}\ \big\}$ :
    Enable stencil test, render only pixels with stencil value $t$
    Set post-convolution bias to $\bar{\mathbf{b}}^{x,y}$ .
    Copy area $[i_x + \delta^x\,,\, i_x + \delta^x + w_x + \Delta^x - 1) \times [i_y + \delta^y\,,\, i_y + \delta^y + w_y + \Delta^y - 1)$
    to $[o_x\,,\, o_x + 2w_x) \times [o_y\,,\, o_y + 2w_y)$, using convolution filter $\mathbf{f}^{\,x,y}$ (size $\Delta^x \times \Delta^y$) .
```

h^*, g^*	Low- and high-pass filters, respectively, as specified in Figure 5.10
δ^*	Input image filter specific offsets
Δ^*	Sizes of filters
i, w	Input image offset and size
o	Output image offset
$\bar{\mathbf{s}}$	Scaling parameters for filters $\mathbf{f}^{\,*}$
$\bar{\mathbf{b}}_*^*$	Bias parameters for filters $\mathbf{f}^{\,*}$

Figure 5.12: Implementation sequence for OpenGL-based wavelet reconstruction with RGBA convolution.

clamping artifacts in this case either. Rendering to the green, blue, and alpha channels has to be disabled in order not to overwrite the high-pass filtered coefficients of the next hierarchy level in the framebuffer, which will be needed in further reconstruction steps.

Figure 5.12 reveals the implementation order of this approach. Again, scaling $\mathbf{s}$ and bias $\mathbf{b}^{x,y}$ with $x,y \in \{\mathrm{ev},\mathrm{od}\}$ are four-dimensional vectors in this case, that contain the values for the four different filters $\mathbf{f}^{\,R}$, $\mathbf{f}^{\,G}$, $\mathbf{f}^{\,B}$, and $\mathbf{f}^{\,A}$. The bias values have to be calculated for all four different combinations ev/ev, ev/od, od/ev, and od/od, as the filters have to be split for even and odd filter coefficients per dimension.

5.4.3 Reconstruction using Programmable Graphics Hardware

As we have seen in the previous sections, the order of operations had to be carefully fitted to the filtering equations. With modern GPUs, this process is no longer necessary due to the programmability of the fragment processors. As the reconstruction process is the more complicated part, it will be examined in detail in this section.

```
void main (in      float2      pos2    : TEXCOORD0,            image sample position
           in      float4      posf    : TEXCOORD1,            filter sample position
           out     float4      output  : COLOR,
           uniform sampler2D   data    : TEXTUREUNIT0,                    image data
           uniform samplerRECT filter  : TEXTUREUNIT1,          wavlet filter and bias
           uniform float4      offset  : C0     offset to high-pass filtered data (.w = 0)
{
  // look up data values
  float4 vll = tex2D (data, pos2);  // + offset.ww);
  float4 vhl = tex2D (data, pos2 + offset.xw);
  float4 vlh = tex2D (data, pos2 + offset.wy);
  float4 vhh = tex2D (data, pos2 + offset.xy);

  // calculate filter position and get filter data and bias
  float2 tpos = fmod (posf, float2 (2, 2));
  float4 filt = texRECT (filter, tpos);
  float  bias = texRECT (filter, tpos + float2 (2, 0)) .r;

  // now reconstruct
  output = (vll*filt.r + vhl*filt.g + vlh*filt.b + vhh*filt.a) + bias;
}
```

Figure 5.13: The fragment program for reconstructing one wavelet level with Haar wavelets, written in Cg.

For this case, the same memory layout as for the luminance-convolution-based algorithm has been chosen. In principle, the low- and high-pass filtered data could be stored in RGBA components as in the second algorithm, which would again result in a speed increase, but I wanted to keep the ability to filter RGBA data for image processing in a single step. Additionally, this demonstrates the flexibility of the new programming model more clearly.

In contrast to the former approaches, only a single pass is needed during reconstruction when using the fragment program depicted in Figure 5.13. For clarity, the convolution has been coded explicitly for Haar wavelets only, which have a reconstruction filter kernel of size 1. The program basically loads the four data values for low- and high-pass filtered 2D data from the input texture, gets the filter coefficients and bias values, and performs the convolution. Other, larger, wavelet filters would need more adjacent input data and additional filter coefficients, e.g. Daubecchies wavelets would require 16 data texture lookups, four filter lookups, and the single bias lookup. As programmable loops are not available in fragment programs so far, the size of the wavelet has to be a compile-time constant.

As discovered in Section 5.4.2, different filter coefficients are needed for odd and even pixels. Section 5.5 will show that we need different bias values as well. Both are looked up in a separate, very small texture of size 4×2. In the left half of the texture, the four sets of filter coefficients are stored, with all combinations of low- and high-pass filters for two-dimensional filtering in the four color components. In the right half the four different bias values are stored. For larger wavelet filters this texture would have to be expanded to hold the additional wavelet coefficients.

All wavelet coefficients are stored in a floating point texture, already scaled with the reconstruction scaling factor as specified in Section 5.5. Due to the small size of the texture, it will completely fit into the texture cache of the GPU, and texture stalls are very unlikely.

All texture coordinates have to be specified with extreme care, as round-off errors are difficult to track down, as they can lead to unexpected behavior even with nearest-neighbor lookup. For the program in Figure 5.13, `pos2` holds the input texture position $[0\dots\frac{1}{2})$ for addressing the filtered data, which is basically $\frac{1}{2}$ the output position. `posf` is the integer output position $[0\dots m)$, and `offset` holds the input texture offset for accessing the high-pass filtered data. `offset.w` has to be set to 0 for fast addressing of all four filtered parts of the input texture. When using larger wavelet filters, an additional offset $\frac{1}{m}$ is needed for accessing adjacent pixels in the input texture. All values imply that the input texture has the same size as the output region.

As the fragment program for reconstructing one wavelet step is rather small, it can be considered to reconstruct multiple or even all reconstruction steps with a single pass fragment program. However, this brute-force method needs $O(n^2)$ operations compared to $O(n\log n)$ for reconstructing a single value that has been decomposed n times, as the reconstruction of lower levels has to be repeated for every pixel.

The fragment program uses floats for the filter kernel and its internal computations, but the input data is stored in a regular single byte RGBA texture. Filtering can be accelerated by approximately 30% with the use of 16 bit floats for the internal computations, with some loss of accuracy. Table 5.1 in Section 5.6 will show this correlation more clearly.

5.5 Data Scaling

Up to now we dealt with the graphics pipe as if it could cope with floating point values. This is true for modern PC graphics cards with the availability of floating point P-buffers (as used in Chapter 7), but access to floating point textures and buffers is much more expensive than to textures with 8 bits per channel, and internal computations in the fragment pipeline in higher precision are slower as well. Other architectures as the SGI InfiniteReality graphics pipe do not support floating point framebuffers at all, and all fragments are clamped to the interval $[0,1)$. Luckily OpenGL provides the possibility to scale and bias pixel data after the convolution step, just before the clamping takes place. For fragment-program-based approaches scaling and biasing can be implemented explicitly.

In order to uphold consistency while performing the decomposition, scaling parameters s_h, s_g, and offset values b_h, b_g are introduced, that fit the resulting scaled wavelet coefficients $\tilde{c}$ and $\tilde{d}$, represented by the pixel values p_j^n, to the interval $[0,1)$. In the following only the low-pass filtering sequence will be addressed, because the high-pass filtering sequence is handled exactly the same way.

In particular we can define the *scaled* decomposition equation

$$\tilde{c}_n^{j-1} = s_h \cdot \sum_k h_{k-2n}\tilde{c}_k^j + b_h \tag{5.17}$$

and initialize the decomposition with $\tilde{c}^J_n = c^J_n$. We can see that for positive $\tilde{c}^j_n$ the sum $\sum_k h_{k-2n}\,\tilde{c}^j_k$ gets minimal for

$$\tilde{c}^j_k = \begin{cases} 0 & h_{k-2n} \geq 0 \\ 1 & h_{k-2n} < 0 \end{cases} .$$

The maximum of the sum can be determined equivalently. By imposing these extrema to the condition $\tilde{c}^j_n \in [0,1)$ which reflects the framebuffer clamping, we get the scaling factor s_h and the offset bias b_h as

$$\begin{aligned} \tilde{h}_{\text{neg}} &= \sum_k \min(0,h_k)\ , \quad \tilde{h}_{\text{pos}} = \sum_k \max(0,h_k)\ , \\ s_h &= \frac{1}{h_{\text{pos}} - h_{\text{neg}}}\ , \qquad b_h = -h_{\text{neg}}\, s_h\ . \end{aligned}$$

During reconstruction, the scaling and bias has to be compensated. In order to accomplish this, we first insert (5.3) into (5.4):

$$c^j_k = \Big(\sum_n h_{k-2n} \cdot \sum_i h_{i-2n}\, c^j_i\Big) + \Big(\sum_n g_{k-2n} \cdot \sum_i g_{i-2n}\, d^j_i\Big)\ . \tag{5.18}$$

Now we insert (5.17) into the equivalently scaled reconstruction equation and obtain

$$\begin{aligned} \tilde{c}^j_k = {} & \bar{s}_h \cdot \sum_n h_{k-2n} \cdot \Big(s_h \cdot \sum_i h_{i-2n}\,\tilde{c}^j_i + b_h\Big) \\ & + \bar{b}_h + \bar{s}_g \cdot \sum_n g_{k-2n} \cdot \Big(s_g \cdot \sum_i g_{i-2n}\,\tilde{d}^j_i + b_g\Big) + \bar{b}_g\ . \end{aligned} \tag{5.19}$$

The reconstruction process is performed in two steps, of which both are clamped to $[0,1)$. The condition $\tilde{c}^j_n \in [0,1)$ and the lossless wavelet reconstruction ensures that the final value $\tilde{c}^j_n$ does not exceed the clamping interval.

Now we decompose (5.19) and compare the coefficients with (5.18). This comparison yields

$$\begin{aligned} \bar{s}_h &= \frac{1}{s_h}\ , & \bar{s}_g &= \frac{1}{s_g}\ , \\ \bar{b}_h &= h_{\text{neg}} \cdot \begin{cases} \sum_i h_{2i} & k \text{ even} \\ \sum_i h_{2i+1} & k \text{ odd} \end{cases}\ , & \bar{b}_g &= g_{\text{neg}} \cdot \begin{cases} \sum_i g_{2i} & k \text{ even} \\ \sum_i g_{2i+1} & k \text{ odd} \end{cases}\ . \end{aligned}$$

Note that the bias parameters are different for odd and even input pixels. Therefore, it is not possible to implement wavelets using graphics hardware with a two-pass approach alone, separating scaling and convolution in different rendering steps. Even and odd pixels still have to be biased differently, and the technique using the stencil buffer for splitting the filters seems to be the natural solution.

Figures 5.14 and 5.15 sum up all calculations for the computation of the data scaling and bias parameters for both decomposition and reconstruction.

$$\begin{array}{llll}
\tilde{h}_{\text{neg}} & = \sum_k \min(0,h_k)\ , & \tilde{h}_{\text{pos}} & = \sum_k \max(0,h_k) \\
\tilde{g}_{\text{neg}} & = \sum_k \min(0,g_k)\ , & \tilde{g}_{\text{pos}} & = \sum_k \max(0,g_k) \\
s_h & = \frac{1}{\tilde{h}_{\text{pos}}-\tilde{h}_{\text{neg}}}\ , & s_g & = \frac{1}{\tilde{g}_{\text{pos}}-\tilde{g}_{\text{neg}}} \\
b_h & = -\tilde{h}_{\text{neg}} \cdot s_h\ , & b_g & = -\tilde{g}_{\text{neg}} \cdot s_g
\end{array}$$

h_j, g_j Low- and high-pass filter coefficients, respectively

Figure 5.14: Implementation sequence for the evaluation of scale and bias parameters for decomposition.

$$\begin{array}{llll}
\tilde{h}_{\text{neg}} & = \sum_k \min(0,h_k)\ , & \tilde{h}_{\text{pos}} & = \sum_k \max(0,h_k)\ , \\
\tilde{h}_{\text{ev}} & = \sum_k h_{2k}\ , & \tilde{h}_{\text{od}} & = \sum_k h_{2k+1}\ , \\
\tilde{g}_{\text{neg}} & = \sum_k \min(0,g_k)\ , & \tilde{g}_{\text{pos}} & = \sum_k \max(0,g_k)\ , \\
\tilde{g}_{\text{ev}} & = \sum_k g_{2k}\ , & \tilde{g}_{\text{od}} & = \sum_k g_{2k+1}\ , \\
\bar{b}_h^{\text{ev}} & = \tilde{h}_{\text{ev}} \cdot \tilde{h}_{\text{neg}}\ , & \bar{b}_h^{\text{od}} & = \tilde{h}_{\text{od}} \cdot \tilde{h}_{\text{neg}}\ , \\
\bar{b}_g^{\text{ev}} & = \tilde{g}_{\text{ev}} \cdot \tilde{g}_{\text{neg}}\ , & \bar{b}_g^{\text{od}} & = \tilde{g}_{\text{od}} \cdot \tilde{g}_{\text{neg}}\ , \\
\bar{s}_h & = \tilde{h}_{\text{pos}} - \tilde{h}_{\text{neg}}\ , & \bar{s}_g & = \tilde{g}_{\text{pos}} - \tilde{g}_{\text{neg}}\ .
\end{array}$$

h_j, g_j Low- and high-pass filter coefficients, respectively

Figure 5.15: Implementation sequence for the evaluation of scale and bias parameters for reconstruction.

5.6 Results and Comparison

Wavelet decomposition and reconstruction have in principle the same order of complexity. However, for the first presented algorithm hardware-based reconstruction is about two to three times slower than decomposition due to the limitations of the graphics pipe. This is not the case with the second algorithmic approach, at least for larger images. As the second approach is much faster than the first one as well, I have concentrated my timing measurements on the second approach. The drivers for PC-based graphics hardware usually do not have an optimized imaging pipeline. Thus only the fragment-program-based approach has been evaluated on this architecture.

In general the slower speed of the wavelet reconstruction is no major drawback, as wavelets are most often used for decomposition in order to accelerate volume rendering and feature detection. Currently, the expansion of compressed data, which would be a major application for wavelet reconstruction, does not map onto the described algorithms very well, because all wavelet coefficients have to be stored in the images regardless of their values. Run-length and entropy decoding — important steps of all image compression techniques yielding high compression ratios — cannot be implemented efficiently on current graphics hardware as well.

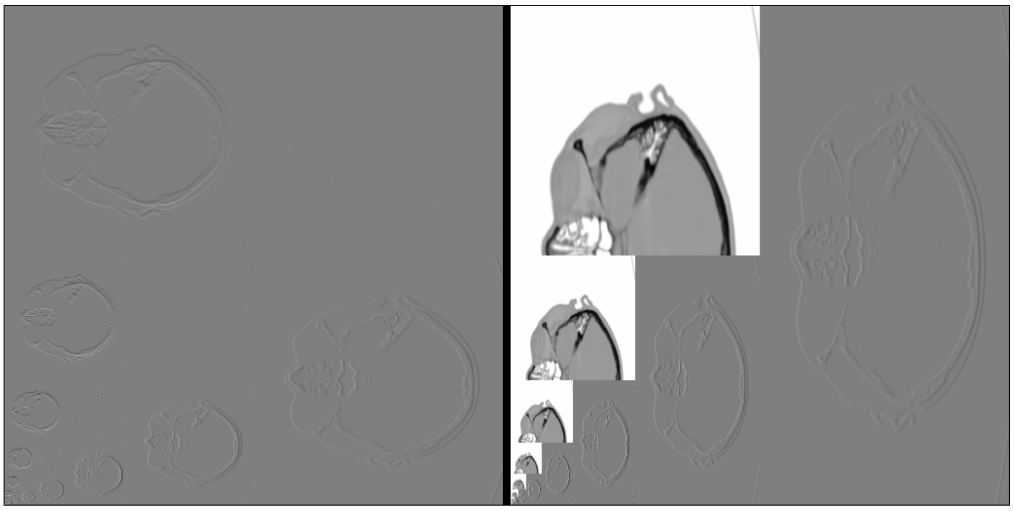


Figure 5.16: The head data set decomposed with Haar wavelets (algorithm 1).

Figure 5.16 shows a complete decomposition done with the first presented algorithm. It can be clearly seen, that the data set is first filtered along the *x* axis, while it is copied from the left half of the image to the right half. After that, it is copied back to the left half, while it is filtered along the *y* axis. As in the next step only the low-pass filtered coefficients in the lower left quarter of the left image are filtered, the upper and right half of the right image are left untouched.

Color Plate 1 shows a complete decomposition of the data set with the second algorithm. Please note that the high-pass filtered coefficients can be seen in the green and blue channel, while the low-pass only filtered coefficients are visible in the red channel. Of course, the alpha channel could not be visualized in this image at all.

Table 5.1 reveals that hardware-based wavelet filtering is much faster than a well tuned software implementation. Only for very small images the software system outperforms the OpenGL hardware on the SGI. Scaling and bias computation as well as filter kernel download adds an almost constant overhead which unsurprisingly leads to low execution speed for small images. As soon as the image does not fit into the processor cache any more, the advantage of using the high memory bandwidth to the GPU comes out more clearly. Additionally, a lot of time can be saved during a typical visualization cycle, as the data do not have to cross the graphics / host memory barrier. The times for fragment-program-based reconstruction show that the programmability of graphics hardware does not only ease the implementation of complex algorithms on GPUs, but it also improves their performance dramatically. Still, there are several possibilities to increase the rendering speed during wavelet filtering even more.

All imaging pipeline times have been measured on a Silicon Graphics Octane with an MXE graphics pipe. The BasicReality engine has been tested as well, but no performance numbers have been evaluated due to severe pipeline bugs that showed up on using post convolution biasing. All programmable fragment pipeline times have been measured on PC with an NVIDIA FX5950 Ultra graphics board, though the older FX5800 Ultra showed the same overall performance.

	Haar wavelet					Daubechies (4) wavelet				
Size	32^2	64^2	128^2	256^2	512^2	32^2	64^2	128^2	256^2	512^2
Soft. decomp.*	0.50	2.0	7.8	31	150	0.70	2.8	11	45	209
Hard. decomp.*	0.65	1.4	4.5	16	62	0.70	1.8	5.5	19	74
Speedup	**0.77**	**1.4**	**1.7**	**1.9**	**2.4**	**1.0**	**1.6**	**2.0**	**2.4**	**2.8**
Soft. recons.*	0.80	3.6	14	55	240	1.2	5.0	19	78	340
Hard. recons.*	1.4	2.0	5.0	18	66	1.4	2.0	5.1	18	66
Speedup	**0.57**	**1.8**	**2.8**	**3.1**	**3.6**	**0.86**	**2.5**	**3.7**	**4.3**	**5.2**
Soft. decomp.†	0.036	0.13	0.56	3.3	49.1	0.057	0.21	0.91	8.5	60
Soft. recons.†	0.053	0.20	0.85	4.11	46.8	0.077	0.29	1.14	9.0	60
Hard. recons.†	0.040	0.124	0.46	1.79	6.99					
Speedup†	**1.33**	**1.6**	**1.8**	**2.3**	**6.7**					
Hard. recons.‡	0.030	0.086	0.285	1.10	4.33					
Speedup‡	**1.77**	**2.3**	**3.0**	**3.7**	**10.8**					

* SGI Octane with R10000, 195MHz, MXE graphics, algorithm 2

† PC with P4, 2.8 GHz, Intel 7205 chipset, NVIDIA GeForce FX5950 Ultra graphics, algorithm 3, 32 bit registers

‡ Same PC, 16 bit registers

Table 5.1: Filter times in ms per 2D wavelet step.

Comparing the speedup factors between the SGI (algorithm 2) and the newer PC (algorithm 3), I can only come to the same conclusion as in Chapter 4 that the performance gap between CPU- and GPU-based algorithms is currently widening — though at a lower pace than in the previous chapter. Please note that fragment-program-based decomposition has not been analyzed and implemented yet.

As hardware-based wavelet filtering uses the framebuffer for its computations, which only has a limited depth, the accuracy of the computations cannot be as good as with software-based techniques, which in contrast only have to tolerate the typically small floating point errors. However, when using a framebuffer with a depth of 12 bits per base color, only single bit errors can be found in images of size 512^2 after complete wavelet decomposition and reconstruction. All accuracy problems could be overcome by using floating point P-buffers with the programmable fragment pipeline based algorithm, but this approach slows down computations, and one usually needs the decomposed image to be displayed anyway, which is not possible with these floating point buffers. We should now take a closer look at differences of software- and hardware-filtered images.

First, Figures 5.17 and 5.20 show the original `head` (size 512^2) and `lena` (size 256^2) data sets, which were used for this analysis. The contrast enhanced Figures 5.18, 5.19, 5.21, and 5.22 show the completely decomposed data sets for Haar and Daubechies wavelets of order 4. Figure 5.1 lists the filter kernel coefficients for these two wavelet types.

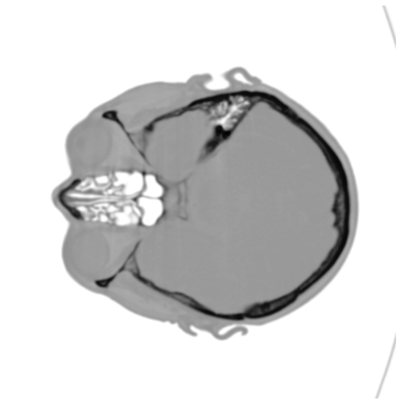


Figure 5.17: The `head` data set.

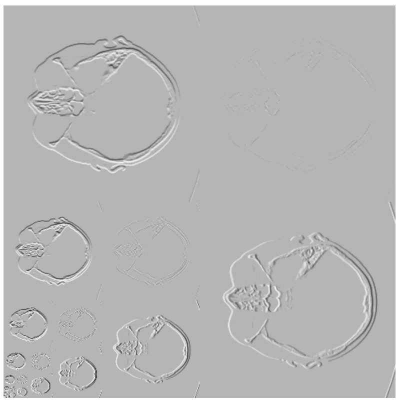


Figure 5.18: Haar wavelet decomposition.

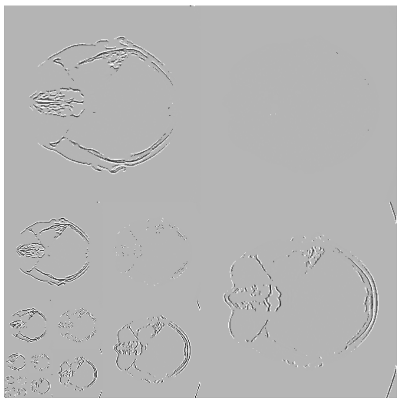


Figure 5.19: Daubechies wavelet decomposition.

Figure 5.20: The `lena` data set.

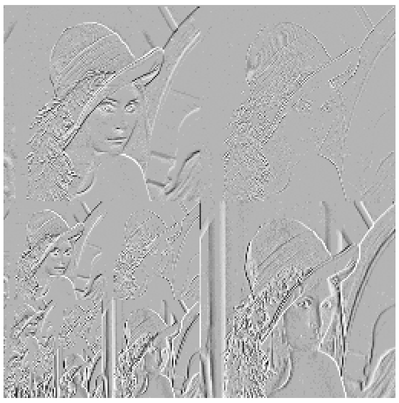


Figure 5.21: Haar wavelet decomposition.

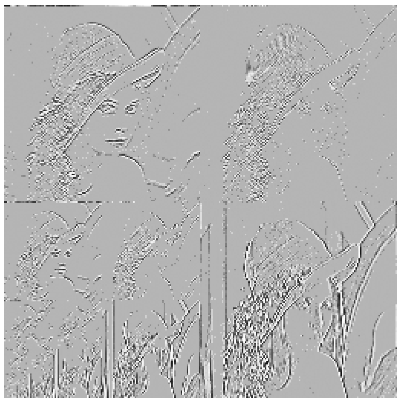


Figure 5.22: Daubechies wavelet decomposition.

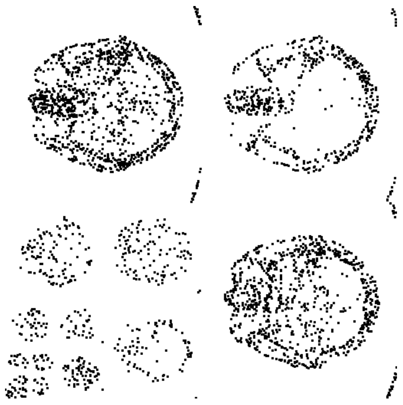


Figure 5.23: Least significant bit differences between software and hardware Haar decomposition (algorithm 1).

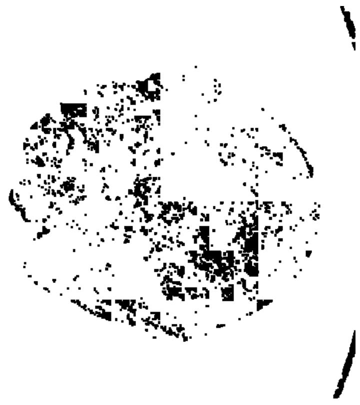


Figure 5.24: Least significant bit differences after full Haar decomposition and reconstruction (algorithm 1).

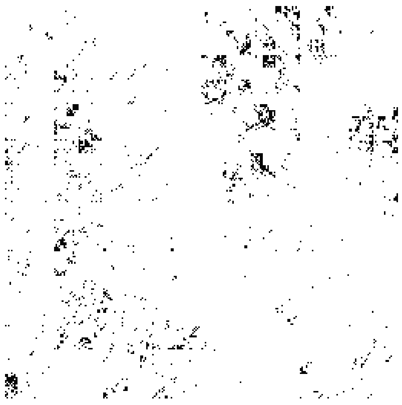


Figure 5.25: Least significant bit differences on the `lena` data set (algorithm 1).

The next images reveal the differences between software- and hardware-based Haar wavelet filtering in form of difference images of completely decomposed as well as decomposed and afterwards reconstructed data. The images had to be equalized in order to reveal *any* differences at all, as the hardware-filtered variants differ only in the least significant bit. Additionally, all erroneous positions have been increased in their size with morphological operators, otherwise printers would easily miss some positions during rasterization. Figure 5.23 reveals all 1-bit differences between hardware- and software-decomposed data sets in the case of Haar wavelets for the `head` data set, a 512^2 sized slice of a computer tomography. Figures 5.24 and 5.25 show the differences between completely Haar wavelet decomposed, then reconstructed data sets, and their originals. Note that software decomposition using floats is accurate enough to restore image data during reconstruction without any differences at all.

The algorithms seem to be pretty robust as can be seen in Color Plate 2. It shows the differences of a decomposition using the second algorithm, evaluated on two different graphic cards. Again, the image has been enhanced to show least significant bit differences.

As already mentioned in the previous sections, special care has to be taken at the borders. Without handling these special cases, artifacts will occur, as it can be seen in Figures 5.26 and 5.29. In order to verify that only border data are affected, difference images have been calculated, which can be investigated in Figures 5.27 and 5.30. As apparently no differences except for the borders can be detected, equalized versions are presented in Figures 5.28 and 5.31.

Framebuffers with only eight bits per base color yield less pleasing results. Figures 5.32 and 5.34 reveal the differences after complete decomposition and reconstruction. Again, the second image has been enhanced in order to reveal the differences.

As the second presented algorithm sums up the low-pass and high-pass reconstructed coefficients before storing to the framebuffer, it can benefit from higher internal precision in the OpenGL pipeline. Figures 5.33 and 5.35 show the differences after complete decomposition and reconstruction. It can be clearly seen that the occurring error is much more regular. It is assumed to be an unresolved rounding issue that could possibly be avoided with some additional research. It should be noted, however, that these rounding issues are most likely hardware-dependent. Note that the maximum error is decreased, while both mean error and peak signal noise ratio (PSNR) are worse.

The third algorithm seems to be much more robust for eight bit framebuffers, as it can be shown in Figure 5.36, compared to Figures 5.34 and 5.35. However, its quality really depends upon the register size, and it quickly degrades with 16 bit floats as it can be seen in Figure 5.37. All images on this page have been enhanced with the same scaling factor as Figure 5.34.

Finally, Table 5.2 gives an overview over all investigated algorithms and their quality. Using higher framebuffer depths is clearly the best approach for resolving inaccuracies of wavelet transformations, but the fragment-program-based approach shows that using high precision during a single step of the reconstruction process can improve the results significantly as well. It is noteworthy that this algorithm worked on input data with a resolution of only 8 bit as well, as compared to the 12 bit input data that was necessary for the two imaging-pipeline-based approaches in order to be competitive.

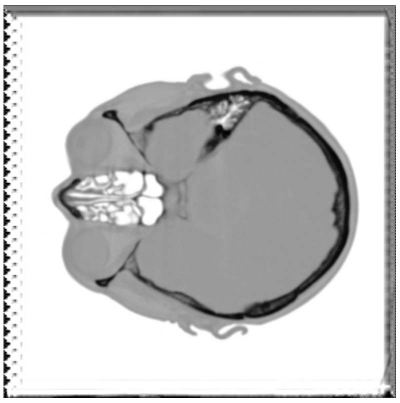


Figure 5.26: The `head` data set decomposed with Daubechies wavelets to level 4 and reconstructed afterwards.

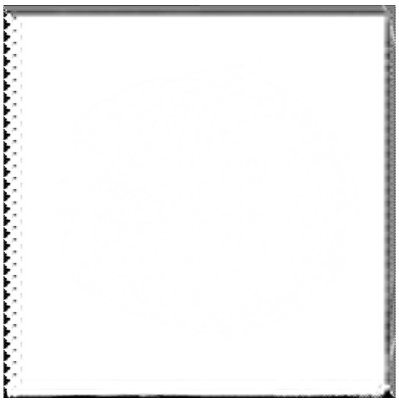


Figure 5.27: Differences between software and hardware Daubechies filtered data.

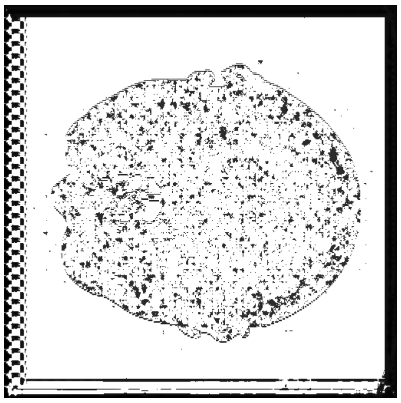


Figure 5.28: Enhanced version, making least significant bit differences visible.

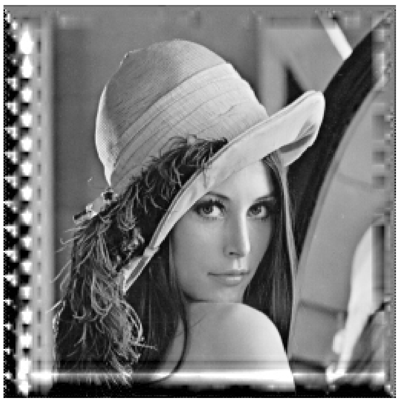


Figure 5.29: The `lena` data set decomposed with Daubechies wavelets to level 4 and reconstructed afterwards.

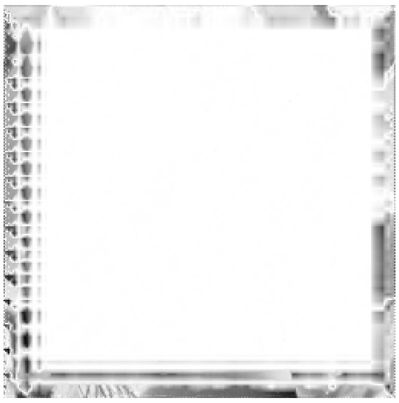


Figure 5.30: Differences between software and hardware Daubechies filtered data.

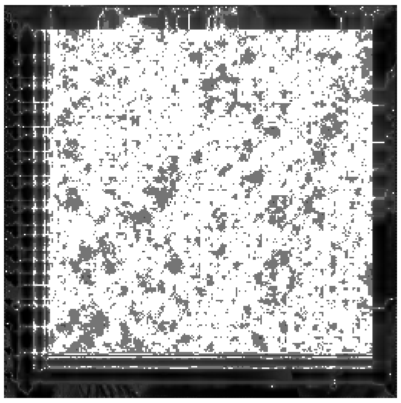


Figure 5.31: Enhanced version, making least significant bit differences visible.

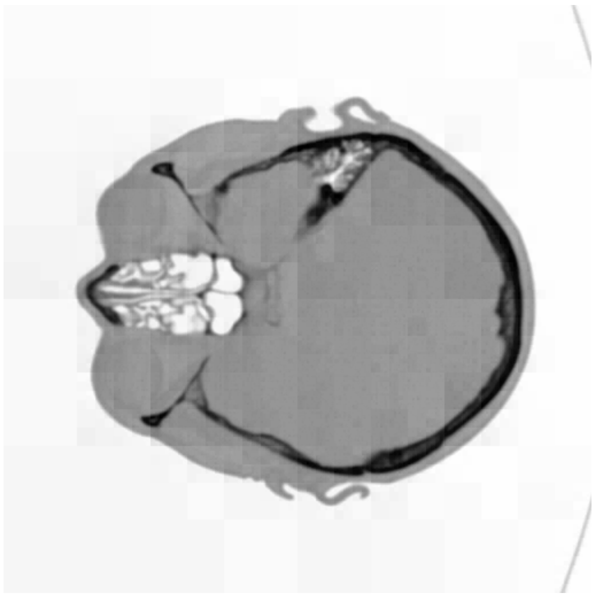


Figure 5.32: Reconstructed image using a framebuffer with 8 bits (algorithm 1).

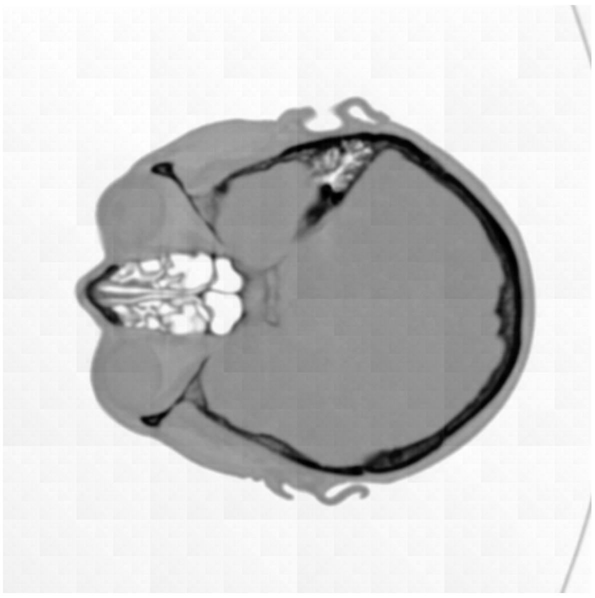


Figure 5.33: Reconstructed image using a framebuffer with 8 bits (algorithm 2).

Algorithm	Framebuffer	Registers	Max.	Mean	PSNR
Luminance convolution (#1)	8 bit	-	35	6.11	30.9dB
Luminance convolution (#1)	12 bit	-	1	0.027	63.8dB
RGBA convolution (#2)	8 bit	-	19	7.22	30.3dB
RGBA convolution (#2)	12 bit	-	1	0.079	59.1dB
Fragment program (#3)	8 bit	16 bit	13	3.26	35.9dB
Fragment program (#3)	8 bit	32 bit	8	1.03	42.7dB

Table 5.2: Maximum and mean screen space errors and the according peak signal noise ratio for different algorithms and framebuffer depths. The error is meassured for a complete decomposition and reconstruction of a 512^2 image with 8 bit color resolution.

The three presented algorithms for wavelet decomposition and reconstruction working directly on the graphics hardware of modern OpenGL capable workstations accelerate the time consuming filtering steps a lot. By using the convolution and color matrix extensions together with OpenGL's facilities to scale images during copy instructions, all necessary steps of 2D tensor product wavelet filtering can be performed on the GPU. Newer graphics hardware increases the filtering speed by an order of magnitude with the use of programmable fragment pipelines. This works without copying data from or to the machine's main memory, thus avoiding typical bottlenecks in the visualization cycle.

Using the framebuffer for mathematical operations is usually problematic in terms of accuracy [Teitzel et al. 1999, Strzodka 2002, Hadwiger et al. 2003] due to the limited depth of the framebuffer. However, wavelet decomposition and reconstruction have proven to be relatively robust. Only single-bit differences between software- and hardware-decomposed data can be detected when rendering intermediate images framebuffers with 12 bit precision. However, when using 8 bit only framebuffers for the decomposition, the algorithms based on the image pipeline can only be used for few decomposition and reconstruction steps in order to keep the final errors small. As soon as the computation during a single reconstruction step can be performed with 32 bit floats using fragment programs, even 8 bit input data is sufficiently accurate for getting good reconstructed results.

While the wavelet theory is the most well-known multiresolution analysis, there are other hierarchical bases as well. In the following two chapters, two other hierarchical basis systems will be examined.

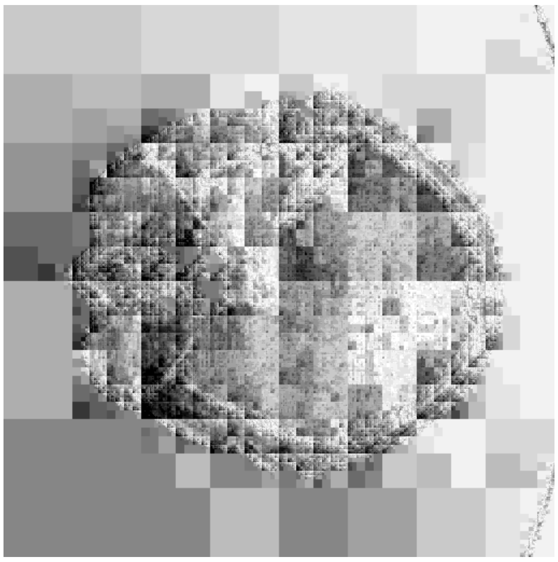


Figure 5.34: Enhanced differences after full Haar decomposition and reconstruction using a framebuffer with 8 bits per color (algorithm 1).

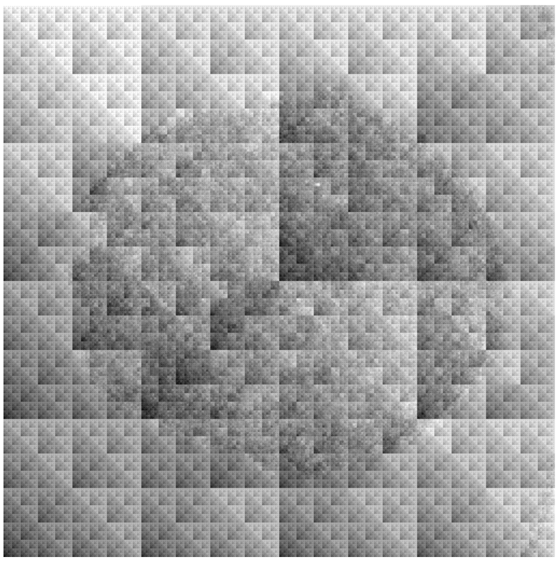


Figure 5.35: Enhanced differences after full Haar decomposition and reconstruction using a framebuffer with 8 bits per color (algorithm 2).

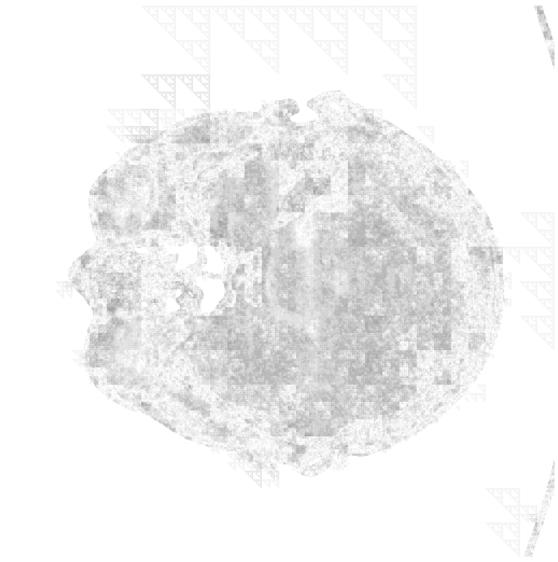


Figure 5.36: Enhanced differences of fragment-program-based reconstruction using a framebuffer with 8 bits per color (algorithm 3).

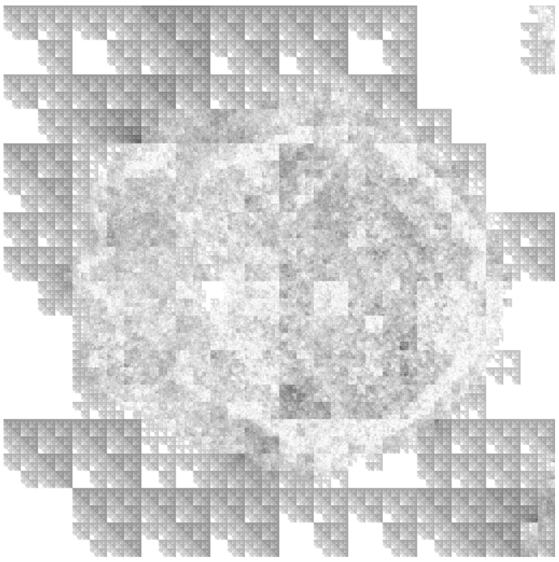


Figure 5.37: Enhanced differences of fragment-program-based reconstruction using a framebuffer with 8 bits per color and 16 bit floats during calculation (algorithm 3).

Ivanova: *An old Egyptian blessing?*
Marcus: *"May gods stand between you and harm*
in all the empty places where you must walk."

Babylon 5, Shadow Dancing

Chapter 6

Parallelized Sparse Grids

Compared to wavelets, which allow for lossless transformation of uniform data into wavelet space and back, the sparse grids basis is explicitely designed to be lossy in this case. The main advantage of sparse grids is that they are capable to represent a function in high spatial resolution with the lowest possible interpolation error for the used number of basis functions.

Sparse grids are generally not very good at approximating uniform data sets, as the theory only holds for functions that are C^2 in the direction of all axes. Algorithms that work entirely on sparse grids reflect this property, of course, and they can create data sets that cannot be handled on uniform full grid representations any more due to their size.

On the other hand, most visualization techniques are only capable of handling uniform grids. As the interpolation on sparse grids is a complicated and time consuming process, direct volume visualization is unthinkable for bigger data sets until the underlying interpolation is accelerated by some orders of magnitude. However, the use of improved algorithms like the combination technique is not always feasible with new sparse grid approaches, e.g. with Whitney forms.

Another possibility to reduce rendering times is the parallelization of the visualization algorithms. Nowadays, quite a number of supercomputers and PC clusters exist, providing MPI as the primary communication API. By streaming the data sets and the resulting images from and to the end user's workstation, their processing power can be utilized without leaving the office.

Parallelizing visualization techniques rises the necessity to balance the computational load, and for time consuming rendering methods previews are useful for the user. Both, generating preview images and load balancing, is performed explicitly in most cases.

A different approach, that will be discussed in this chapter, is to use special pixel rendering sequences to achieve preview generation and load balancing implicitly, which achieves superb results without generating any communication overhead. This work has first been published as a technical report [Hopf and Ertl 2001], and it has been presented as work in progress at the Visualization conference in 2001.

6.1 Sparse Grids

Based upon hierarchical tensor product bases, the sparse grid approach is a very efficient one improving the ratio of invested storage and computing time to the achieved accuracy for many problems in the area of numerical solution of partial differential equations, for instance in numerical fluid mechanics.

In this section a brief summary of the basic ideas of sparse grids is given. For a detailed survey of sparse grids please take a look at [Zenger 1990, Bungartz 1992].

When talking about volume visualization, the data is usually given on a uniform grid with trilinear basis functions. Interpolation on these grids is computationally cheap, as one has only to locate and evaluate the surrounding $2^3 = 8$ basis functions for one interpolation value. This number does not change with respect to the grid size.

Now let G_{i_1,i_2,i_3} be a uniform grid with respective mesh widths $h_{i_j} = 2^{-i_j}$, $j = 1,2,3$ and basis functions $b_k^{i_j}$. Let $\hat{L}_n$ be the function space of the piecewise trilinear functions defined on $G_{n,n,n}$ and vanishing on the boundary. Additionally, consider the subspaces S_{i_1,i_2,i_3} of $\hat{L}_n$ with $1 \le i_j \le n$, $j = 1,2,3$, which consist of the piecewise trilinear functions defined on G_{i_1,i_2,i_3} and vanishing on the grid points of all coarser grids, with

$$\hat{L}_n = \bigoplus_{i_1=1}^{n} \bigoplus_{i_2=1}^{n} \bigoplus_{i_3=1}^{n} S_{i_1,i_2,i_3} \quad . \tag{6.1}$$

This forms a hierarchical basis decomposition of the function space $\hat{L}_n$ where piecewise trilinear finite elements are used as basis functions in each subspace S_{i_1,i_2,i_3} (compare Figure 6.1 for one-dimensional examples). From now on we will deal with the interpolated function f_{i_1,i_2,i_3} on the grids of the above mentioned subspaces:

$$f_{i_1,i_2,i_3} = \sum_{k_1=1}^{2^{i_1-1}} \sum_{k_2=1}^{2^{i_2-1}} \sum_{k_3=1}^{2^{i_3-1}} c_{k_1,k_2,k_3}^{(i_1,i_2,i_3)} \cdot b_{k_1,k_2,k_3}^{(i_1,i_2,i_3)} \quad . \tag{6.2}$$

The values $c_{k_1,k_2,k_3}^{(i_1,i_2,i_3)}$ are called contribution coefficients. Please note that the basis functions of these subspaces do not overlap, compared to the basis functions of $\hat{L}_n$.

When looking at the interpolation error, one finds that $\| f_{i_1,i_2,i_3} \|$ has a contribution of the same order of magnitude, namely $O(2^{-2\,C})$ for all subspaces with $C = i_1 + i_2 + i_3 = \text{const}$:

$$\| f_{i_1,i_2,i_3} \| \;\le\; \left\| \frac{\partial^6 f}{\partial x_1^2 \partial x_2^2 \partial x_3^2} \right\| \cdot h_{i_1}^2 h_{i_2}^2 h_{i_3}^2 \ .$$

Additionally, these subspaces have the same number of basis functions, namely 2^{C-3}. Since the number of basis functions is equivalent to the number of stored grid points and because of the

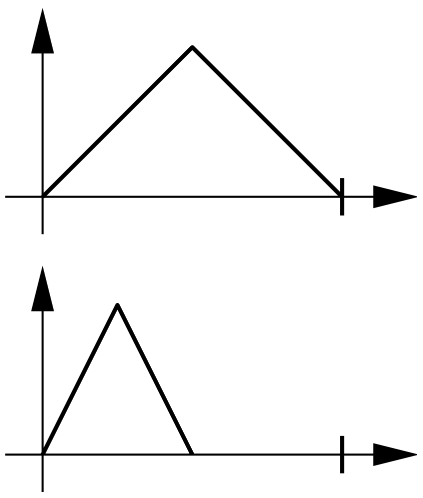


Figure 6.1: Examples of 1D basis functions b_1^1 and b_1^2.

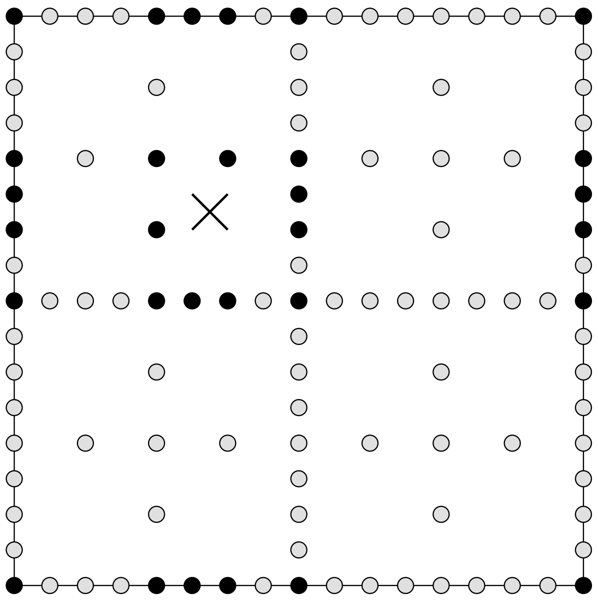


Figure 6.2: Interpolation on a two-dimensional sparse grid of level 4.

contribution argument as well, it seems to be straightforward to define a sparse grid space $\tilde{L}_n$ as follows (compare also Figure 6.3):

$$\tilde{L}_n := \bigoplus_{i_1+i_2+i_3 \leq n+2} S_{i_1,i_2,i_3}. \tag{6.3}$$

Then the interpolated function $\tilde{f}_n \in \tilde{L}_n$ is given by

$$\tilde{f}_n = \sum_{i_1+i_2+i_3 \leq n+2} f_{i_1,i_2,i_3} \tag{6.4}$$

Now we estimate the interpolation error with regard to the L^2 or L^∞ norm (compare [Bungartz 1992, pp. 23]):

$$\|f - \tilde{f}_n\| \leq O\left(h_n^2 \left(\log_2 \left(h_n^{-1}\right)\right)^2\right).$$

It shows that the sparse grid interpolated function $\tilde{f}_n$ is nearly as good as the full grid interpolated function $\hat{f}_n$:

$$\|f - \hat{f}_n\| \leq O\left(h_n^2\right).$$

Now we consider the dimensions of the function spaces $\hat{L}_n$ and $\tilde{L}_n$, which correspond to the number of nodes of the underlying grids. Obviously, the dimension of the full grid space is given by $\dim(\hat{L}_n) = O\left(2^{3n}\right) = O\left(h_n^{-3}\right)$. For the sparse grid the following relation holds: $\dim(\tilde{L}_n) =$

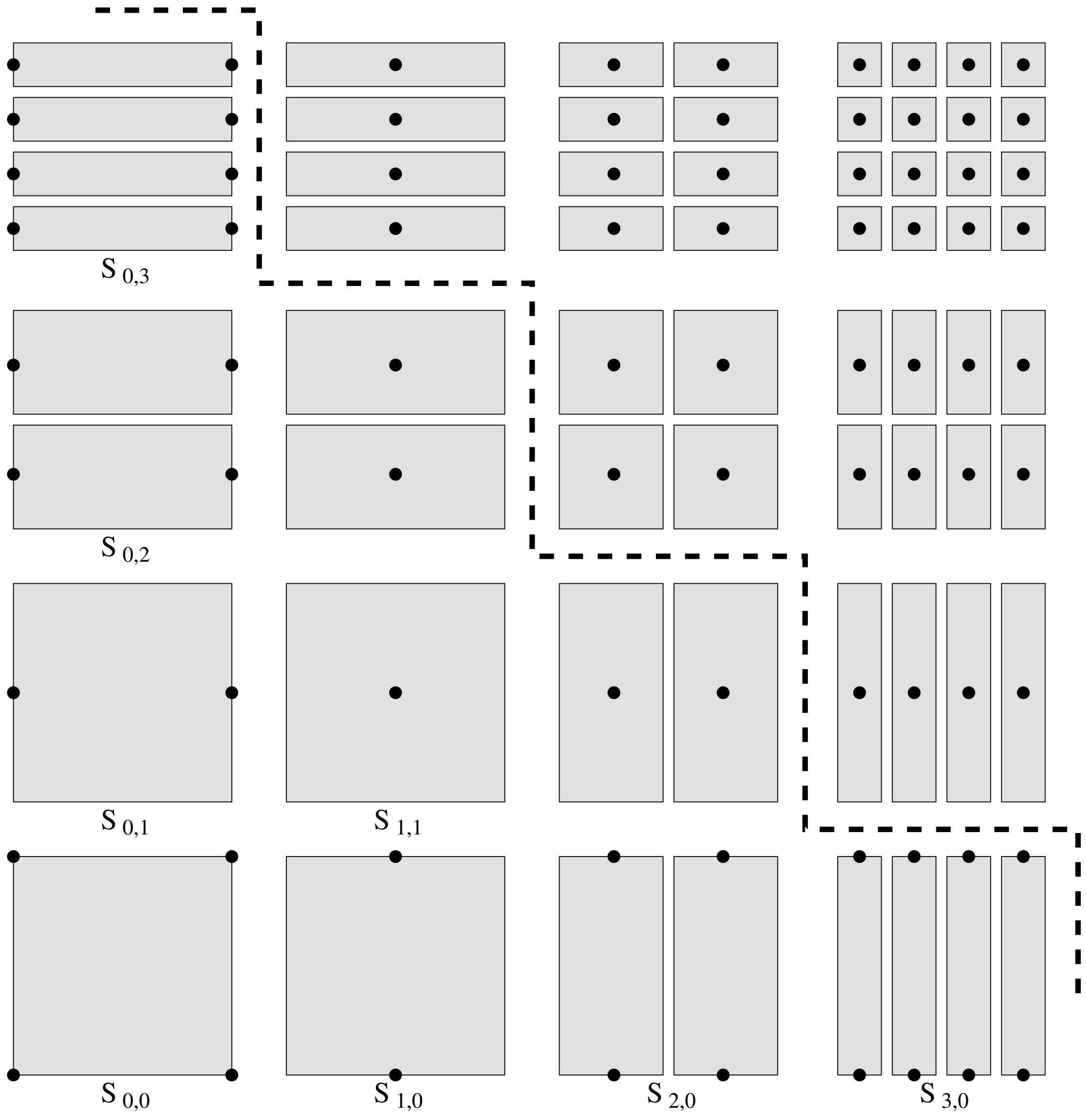


Figure 6.3: Two-dimensional hierarchical subspace decomposition. Note that this figure includes the necessary basis functions for functions not vanishing on the boundary.

$O\left(2^n n^2\right) = O\left(h_n^{-1}(\log_2\left(h_n^{-1}\right))^2\right)$. Therefore, a tremendous amount of memory can be saved if sparse grids are used instead of full grids.

Considering the number of basis functions that contribute to the interpolated function, sparse grids are much more computational intensive than uniform grids. Figure 6.2 gives an example which basis functions have to be evaluated to interpolate the sparse grid at the marked position. Note that the figure includes the basis functions necessary for functions that do not vanish on the boundary. They have not been considered so far in order to simplify the explanation.

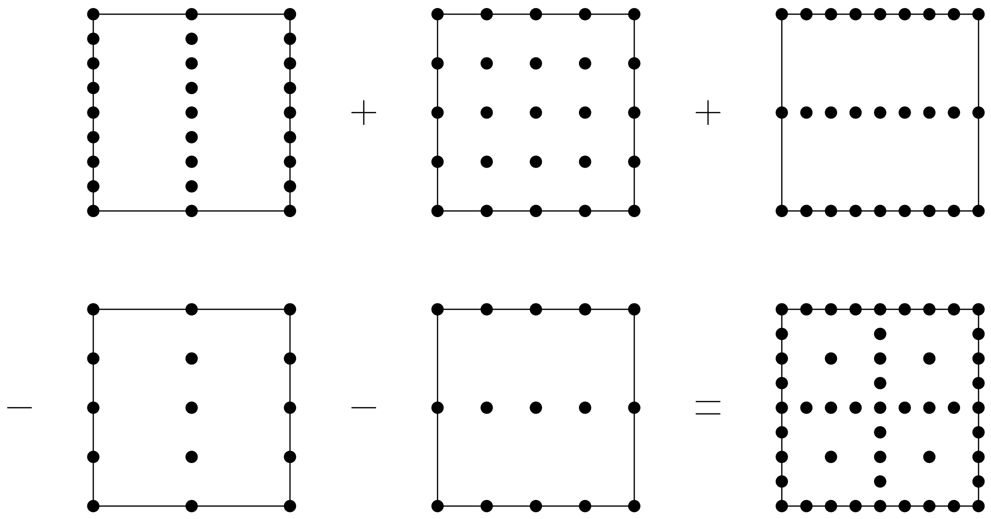


Figure 6.4: A two-dimensional sparse grid of level 3 can be reconstructed by linear combination of five full grids of low resolution.

6.1.1 Combination Technique

Since the described sparse grid interpolation of function values is quite complicated and rather time consuming, the so-called combination technique has been implemented. This method was introduced by Griebel, Schneider, and Zenger in [1992b]. Actually, the combination method has been used in numerical simulations in order to combine coarse solutions computed on smaller, suitable full grids to the desired sparse grid solution. However, we start with a data set given on a sparse grid and decompose the grid so that the data set is represented on certain uniform full grids of low resolution.

Using the trilinear interpolation on these uniform grids can improve the performance quite a lot compared to the regular sparse grid interpolation. Choosing the relevant basis function (i.e. cell location) is much faster for uniform grids, and the computation of the filter weights is less complicated as well.

The decomposition of a sparse grid into several uniform grids works, because it can be proven that the three-dimensional interpolating function $\tilde{f}_n \in \tilde{L}_n$ is given by

$$\tilde{f}_n = \sum_{i_1+i_2+i_3=n+2} f^c_{i_1,i_2,i_3} \quad - \quad 2 \quad \cdot \sum_{i_1+i_2+i_3=n+1} f^c_{i_1,i_2,i_3} \quad + \sum_{i_1+i_2+i_3=n} f^c_{i_1,i_2,i_3} \tag{6.5}$$

where $f^c_{i_1,i_2,i_3}$ denotes the trilinear interpolation of function values on the respective full grid. Figure 6.4 reveals the two-dimensional situation. Notice that the used full grids consist of the same nodes as the corresponding sparse grid.

Now let us turn to the benefit of the combination technique. The total number of summands of the standard sparse grid interpolation on a three-dimensional sparse grid of level n is given by

$$\sum_{i=1}^{n} \frac{i(i+1)}{2} = \frac{1}{6} n(n+1)(n+2) \tag{6.6}$$

(compare (6.4)), whereas the total number of trilinear interpolations of the combination method adds up to

$$\sum_{i=n-2}^{n} \frac{i(i+1)}{2} = \frac{3}{2}n(n-1)+1 \tag{6.7}$$

in the three-dimensional case (see (6.5)).

However, the lower complexity of the combination technique only pays off in terms of significant arithmetic operations for sparse grids of level 50 or above. As it is very unlikely, that such large grids will be used very soon, the main advantage of the combination technique is the fact that uniform full grids are used. Thus, the interpolation routine itself can be implemented in a tight loop, and selecting the correct basis functions is almost trivial compared to regular sparse grids.

6.1.2 Whitney Forms

So-called mixed finite element schemes — as defined e.g. in [Nédélec 1980] — cannot be used in traditional sparse grids, as these have been confined to Lagrangian finite elements. In recent research [Gradinaru and Hiptmair 2003] the sparse grid approach has been extended to deal with the lowest order discrete differential forms, the named Whitney-elements [Whitney 1957]. Due to the complexity of this topic the following description will focus on the structure and interpolation properties of these grids in three dimensions, and refer to the cited research papers for details. Please note that for the sake of simplicity no basis presented here has been normalized and, again, boundaries are not considered.

In Euclidean space, we do not need to distinguish between forms and their vector representatives, and we can use the same interpolation algorithms as in the scalar case. Therefore, we will stick to Cartesian coordinates in the remainder of the paper.

The local space of the components of Whitney l-forms is a tensor product of linear and constant functions. From the definition of the local space (see [Nédélec 1980]) we can derive the nodal representation of a function on a full grid with mesh width $h_n = 2^{-n}$ using the basis functions

$$b^{l,n}_{k_1,k_2,k_3,j}(x_1,x_2,x_3) \;=\; \prod_{i\in I^l_j} B\big(2^n x_i - k_i\big) \;\cdot\; \prod_{i\notin I^l_j} \varphi\big(2^n x_i - k_i\big) \quad .$$

with the following constant and linear basis templates

$$B(x) = \begin{cases} 1 & 0 < x \le 1 \\ 0 & \text{else} \end{cases} \quad , \quad \varphi(x) = \begin{cases} 1+x & -1 < x \le 0 \\ 1-x & 0 < x < 1 \\ 0 & \text{else} \end{cases}$$

and the multi-indices $I^l_j \subseteq \{1,2,3\}$, $|I^l_j| = l$ with l denoting the used form and

$$\begin{aligned} &I^0_j = \emptyset\ ,\ \ j\in\{1\} && , && I^1_j = \{j\}\ ,\ \ j\in\{1,2,3\} && , \\ &I^2_j = \{1,2,3\}\backslash\{j\}\ ,\ \ j\in\{1,2,3\} && , && I^3_j = \{1,2,3\}\ ,\ \ j\in\{1\} && . \end{aligned}$$

Parameter k_i describes the index and j the component of the basis function.

The basis functions show that a 0-form is equivalent to a regular linearly interpolated scalar sparse grid, while a 1-form has different properties: Its function values f in $\mathbb{R}^3$ consist of 3 components, which are piecewise linear perpendicular to their respective axis and piecewise constant alongside. I. e. f_1 is piecewise constant in x_1 and piecewise linear in x_2 and x_3.

Now we can use the same sparse grid decomposition scheme we used before and get a hierarchical basis. Here we can see, that the basis decomposition for the piecewise constant components is effectively described by a wavelet decomposition using Haar wavelets. Thus we get the hierarchical basis functions (again, without considering the boundary)

$$\tilde{b}^{l,n}_{k_1,k_2,k_3,j}(x_1,x_2,x_3) \;=\; \prod_{i\in I^l_j} \psi\big(2^n x_i - 2k_i - 1\big) \;\cdot\; \prod_{i\notin I^l_j} \varphi\big(2^n x_i - 2k_i - 1\big)$$

with ψ being the Haar mother wavelet

$$\psi(x) = \begin{cases} 1 & -1 < x \le 0 \\ -1 & 0 < x \le 1 \\ 0 & \text{else} \end{cases} \quad .$$

In Figure 6.5 the basis functions for the first component of 1-forms in two dimensions is shown. Similar to Figure 6.3 the basis functions needed for the boundary are included, and you can notice the different number of basis function components with respect to the axes.

6.2 Related Work

There are already several algorithms that work entirely on sparse grids [Zenger 1990, Bungartz 1992, Griebel et al. 1992a, Griebel et al. 1992b, Bungartz and Dornseifer 1998], creating data sets that cannot be handled on uniform grids in full resolution any more due to their size. Many of these systems are related to three-dimensional data.

Former publications about visualization toolkits working directly on sparse grids [Teitzel et al. 1998a, Teitzel et al. 2000] analyzed how the interpolation of functions given on sparse grids could be accelerated. By using special graphics hardware like described in [Hopf 1998, Teitzel et al. 1999] direct volume rendering could be performed interactively. However, the graphics hardware acceleration approach is limited to high end graphics systems with a high pixel depth and to sparse grids of level 10-11 (which resemble uniform grids of size 1025^3-2049^3) and below. Low end graphics systems had only a pixel depth of 8 bits per channel, which is far too less for sufficient accuracy. Sparse grids of level 12 and above have limited accuracy due to a high component scaling factor — for details see [Teitzel et al. 1999]. However, with recent advances like floating point framebuffers this limitation could be easily overcome.

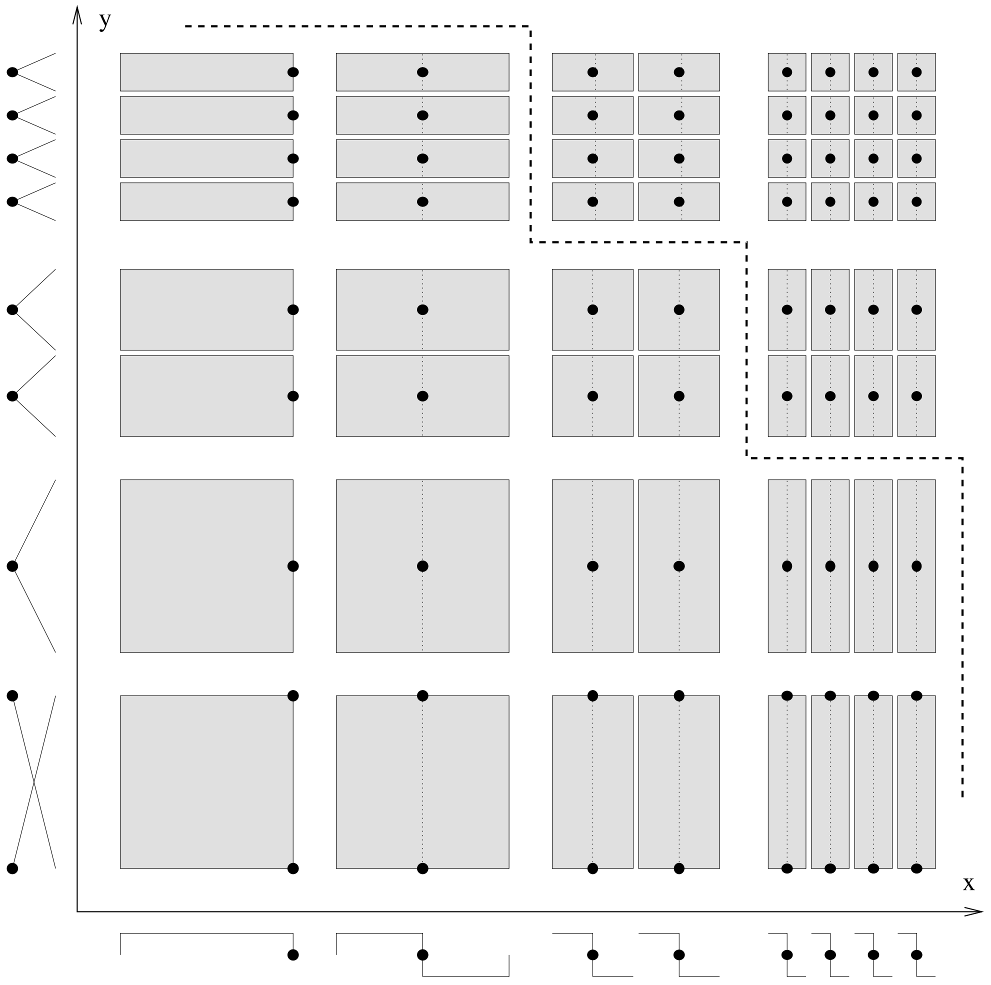


Figure 6.5: Components and support of basis functions for the first component of a 2D differential 1-form, sparse grid level 3.

For a long time, wavelets and sparse grids were disjunct worlds, and even researchers often mixed up the properties of these vastly different multiresolution analyses. Recently, the best of both worlds have been merged by using wavelet bases in the sparse grid representation of multiresolution data sets of mixed finite element schemes [Nédélec 1980], creating the so-called Whitney forms on sparse grids which have been introduced in the previous section.

Velocity information and 1-forms can be processed on-the-fly so that direct volume visualization can be used for this data type as well, e.g. with three-dimensional Line Integral Convolution (LIC) [Rezk-Salama et al. 1999]. The visualization of 2-forms is still in its infancy, only few experimental techniques exist, for example hue-balls and lit-tensors [Kindlmann and Weinstein 1999]. As the underlying interpolation is the dominating algorithmic part concerning computation time, we cannot benefit from hardware-accelerated volume visualization techniques. Thus, raycasting is used, which guarantees for the best image quality at little or no additional cost, while maintaining the greater flexibility. Furthermore, the way graphics hardware interprets textures prohibits the use of mixed basis functions, which is the case for Whitney forms.

In contrast to mathematical algorithms that work directly on sparse grids, visualization techniques have to address the interpolated function at arbitrary positions. Especially in the case of volume visualization an immense number of function evaluations has to be performed. In contrast to trilinear full grid interpolation, sparse grid interpolation does not operate locally, because one basis function in every subspace contributes to the function value. Thus, interpolation is the most time consuming task in the visualization process.

The hierarchical coefficients of the sparse grid are usually stored in a binary tree [Bungartz 1992, Bungartz and Dornseifer 1998, Heußer and Rumpf 1998]. Then a recursive tree traversal has to be performed in order to interpolate the function value. This tree traversal is very slow. Although caching strategies can increase the efficiency of the traversal [Heußer and Rumpf 1998], the computation of the values remains rather time consuming.

Another technique that seems to be used quite often is to store the hierarchical coefficients using linked hash tables. The hash function introduced by Griebel [Griebel 1998, Schiekofer 1998] turned out to be quite good for maintaining a short mean link length, and the algorithm is very straight forward to implement and optimize.

Due to the regular structure of the sparse grids another possibility exists to keep the hierarchical coefficients more easily addressable. By dividing the sparse grid into its different levels and the levels into subspaces, we can save all coefficients in arrays inside the subspaces. During interpolation, a loop over all subspaces in all levels is performed, and the contributions of all subspaces are summed up. Recall that only one basis function per subspace is unequal to zero at a certain position because all basis functions have disjunct supports. This approach turned out to be the fastest variant of all systems working directly with sparse grid hierarchical coefficients. The major drawback of the method is the complexity of the related classes.

However, these techniques are still too slow for interactive volume rendering. Using the combination technique, interpolation can be sped up by almost an order of magnitude. While it needs more arithmetic operations for sparse grids of level 4 up to about 50, it can be implemented with a tight loop over all uniform full grids. Cell location is almost trivial, and interpolation on the full grid is well understood and thus very fast. The major drawback of this method is that the decomposition of a sparse grid using Whitney forms into the respective grids for the combination technique is unknown so far, if at all possible.

6.3 Parallelization

A lot of work has been done to implement parallel volume raycasting on PC clusters, but no approach has considered sparse grids up to now. For sparse grid visualization the parallelization process itself is relatively straight-forward, spreading the rays across the available processors in a domain decomposition scheme. Memory management is not really an issue, and as sparse grids need only very little data space, they can be replicated throughout the cluster.

6.3.1 Remote Rendering

A key problem that is noteworthy is that scientists are often unable to work at the front-end nodes of the cluster directly. Thus, the data set has to be streamed to the computing cluster, where a communication node accepts the data (typically not all nodes have direct internet connection) and distributes it to the rendering nodes. In return, the rendered data has to be streamed to the users' workstation. Again this is done by the front-end communication node that collects incoming ray data and serves the TCP stream. Figure 6.6 depicts the top-level architecture of this scheme.

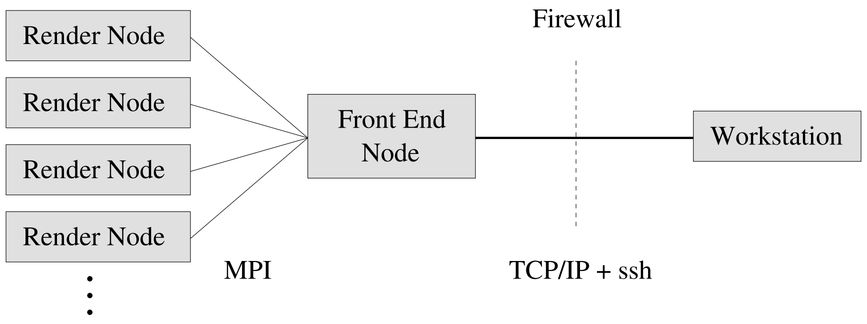


Figure 6.6: Top-level architecture of the parallelized visualization system.

As the pixels delivered by the render nodes may arrive in any order, the communication node sends both pixel position and RGBA values to the workstation, making a total package size of 8 bytes per pixel. With this information the visualization process can continuously generate preview images from early rendered rays. The front-end node adaptively decides for each incoming pixel, which image resolution the pixel contributes to, and updates the image accordingly. Color Plate 3 shows in an example, how the image resolution is changed adaptively. Note that for each incoming pixel the image is updated on the next hierarchy level that has not been used for the pixel so far.

On most PC-based computing clusters MPI is used as the primary communication API, while the wide area network usually employs TCP/IP. The communication node has to adapt for different data types (host / network byte order) and APIs.

Very often the clusters are shielded by firewalls, thus secure shell tunneling may be required. This seems to be a horrible bottleneck, but in fact the interpolation process on sparse grids is so computational intensive that slow communication is not hindering the visualization process.

With replicated data sets the distribution of rays among the nodes can be chosen freely. Usually, a ”master” node selects by some scheme which node shall render which ray and sends new orders, when a job has finished. However, when several nodes finish their job at the same time, the lag between delivering rays and getting new job data can reduce the rendering speed significantly.

6.3.2 Implicit Preview and Load Balancing

Implicit assignment of rays as a function of the images size $s = s_x \cdot s_y$, the number of processors n, and the rank r (the index of the current processor) prevents any additional communication overhead and reduces the lag between rendered rays to the time needed to calculate the next ray assignment.

The quality of the ray assignment function has great impact on equalizing the rendering time of the processors as well as on the possibility to generate previews from early rendered rays. By providing previews the user can very often make decisions about the significance of the rendered images, when only a very small fraction of the rays have been computed. So the quality of the ray assignment is reflected in three properties:

a) The distribution of rays for one processor should be evenly spread in space.
b) The distribution of rays should be evenly spread in time.
c) Rays that fall in slots of coarser grids should be rendered first.

Property a) ensures that the load between the processors is stochastically balanced implicitly, while b) and c) ensure that early rendered rays can be combined to form a preview image. The hierarchical adaptive image display routine shown in the last section relies on a scheme with property c) so that the hierarchy is traversed smoothly. Of course, due to the parallel nature of the system b) and c) cannot be guaranteed for the total ray order. However, with load balancing we can get very close to the ideal pixel order.

A very simple scheme assigns the rays $\left\{p : r\frac{s}{n} \le p < (r+1)\frac{s}{n}\right\}$ to rank r. In order to be able to render early previews, one could index the rays in both image dimensions. However, the processors will typically be active for very different times, and the process is only finished after the last processor is done.

By assigning every n$^{\text{th}}$ ray with $\{p : p \bmod n = r\}$ in an interleaving pattern to the nodes one can overcome this problem. However, with this pattern no previews can be generated from early rendered rays.

When we use a specific pattern for a total ordering (see Figure 6.7) of all rendered rays, we can ensure properties b) and c), which enables the generation of previews from early rendered

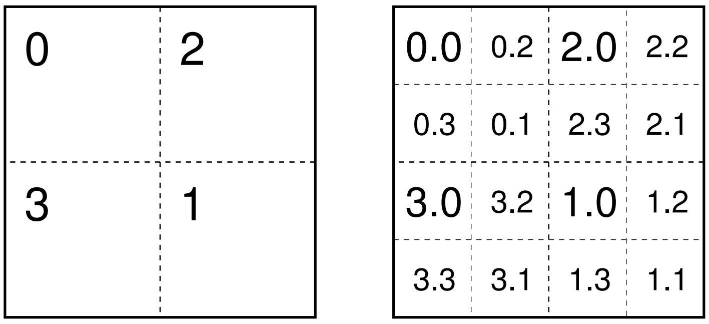


Figure 6.7: Recursive Pattern for total ordering scheme.

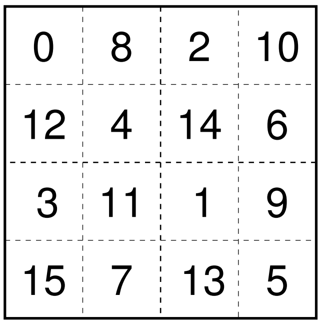
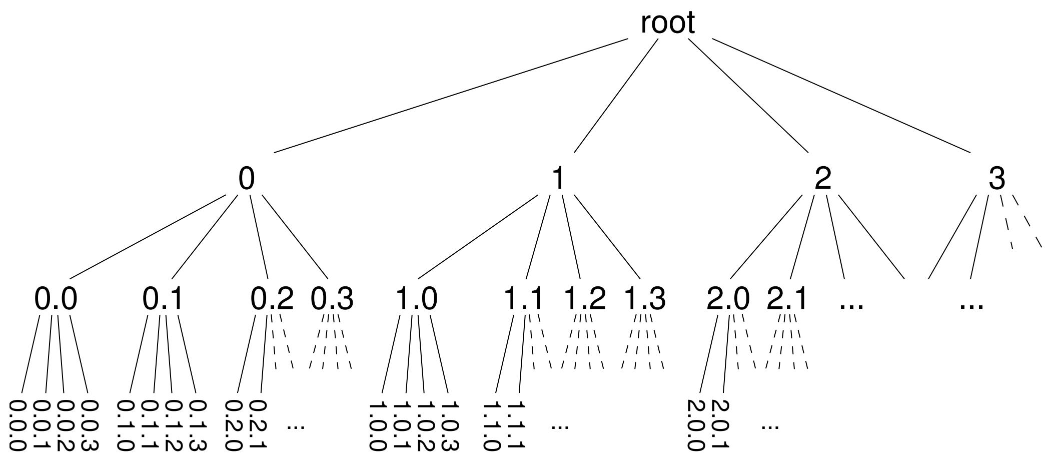


Figure 6.8: Pattern for 4×4 rays.



Transposed Indices:

0.0.0 1.0.0 2.0.0 3.0.0 0.1.0 1.1.0 2.1.0 3.1.0 0.2.0 ... 0.0.1 1.0.1 2.0.1 ...

Figure 6.9: Transposed indexing of the index tree.

pixels, and we distribute the rays perfectly in space. The pattern itself is created by generating a quadtree with pixel index leafs (Figure 6.9). The indices are now transposed so that the highest node index runs fastest. Then continuous numbers are assigned to the leafs, which results in the pixel ordering scheme that can be seen in Figure 6.8. In practical implementations the pixel order indices can be generated by a recursive function without explicitly building the tree. For arbitrary image sizes $s_{x,y} \neq 2^m$ the pattern has to be cropped and the indices have to be reordered.

We can now divide the index list into $\frac{s}{n}$ slots S_t of size n

$$S_t := \{x : tn \leq x < (t+1)n\}$$

We will assign the rays of each slot individually to the processor nodes, so that each processor gets exactly one ray from each slot, which it will render in the order of increasing t.

The pattern is organized so that the first 2^{2i} rays fill exactly the pixel slots of the coarser grid of size $2^i \times 2^i$ for all i, as can be seen in Figure 6.10.

This way we get preview images that can be computed without additional cost from the rays rendered on these coarser grids. As the fastest running index in this scheme corresponds to the lowest resolution grid, adjacent indices are usually not close-by in image space. Thus the temporal distribution of rays is perfectly balanced as well.

The selection of ray indices for one processor

$$x_t := tn + i(t,r) \quad \text{with} \quad x_t \in S_t \ , \ \ 0 \leq i(t,r) < n$$

has one more freedom to investigate, the index selection function $i(t,r)$. As the index selection

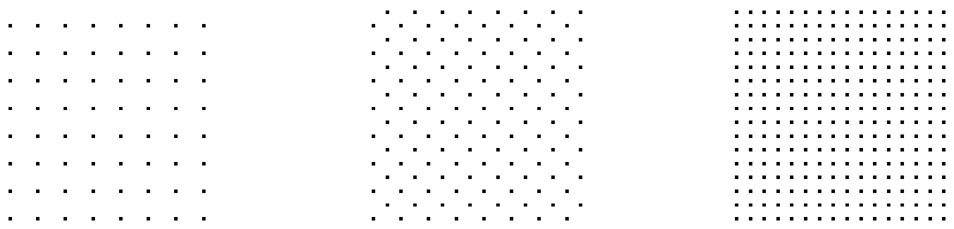


Figure 6.10: The first 64, 128, 256 rendered rays on an 64^2 image.

should make no differences for the individual processors, we can set

$$i(t,r) := (\tilde{\imath}(t) + r) \bmod n \quad .$$

This way we can easily ensure that the sets of rays do not intersect.

In order to spread the rays for one processor evenly in space, $\tilde{\imath}$ has to be selected carefully. The very first thought would be to use $\tilde{\imath}(t) = \text{const}$ or $\tilde{\imath}(t) = t$. But both trivial functions do not spread well for all combinations of s and n. Especially when n divides s_x or s_y, the rays cluster on one part of the image. For values that are prime, however, good results can be achieved (see Color Plate 4).

Heuristically, we have found that

$$\begin{aligned} h &:= (s_x s_y + 1) \bmod n \\ d &:= \begin{cases} h & h < 2 \\ \min_l \{n \bmod l \neq 0 \land (n+1) \bmod l \neq 0 \land l \geq h\} & h \geq 2 \end{cases} \\ \tilde{\imath}(t) &:= \left\lfloor \frac{tn}{s_x s_y} \right\rfloor \cdot d \end{aligned}$$

creates very evenly spaced ray selections for almost all combinations of s and n, and the worst cases encountered so far are not as problematic as the ones described above.

Although it is currently unknown whether the quality of this index selection function can be proven somehow, it is still a very usable approach in practical implementations as it can be computed iteratively with very low computational costs. Color Plate 5 shows several distributions created with this approach.

By subdividing the image plane into several tiles, each one with a size of at least n pixels, one can use an iterative algorithm, that counts how many times a ray has already been assigned to the processor on a particular tile, thus ensuring that every tile is at least addressed once by each processor. The most important aspect of this idea is that the algorithm can, again, run separately on each processor without additional communication.

The suggested ray selection method has the one drawback that cache coherency will not be employed at all. Memory bound problems may be slower using this approach than with tile-based techniques. But this is a problem shared by most non-explicit load balancing algorithms. For the case of sparse grid volume visualization we get almost perfect linear speedup, even for high processor counts.

6.4 Results and Comparison

Using the combination technique turned out to be the best performing algorithm, when this method was applicable. However, for Whitney forms it is still unknown whether there exists a decomposition into a set of according uniform grids, which is necessary for applying the combination technique. In Table 6.1 a short overview over the different system speeds is given. It turns out that both hierarchical techniques tend to get limited by available memory bandwidth for the high levels. Especially the hash-table-based method uses quite a lot of RAM for the tables, which seems to completely spoil any second level cache.

Grid size	8	10	12	14	16
Hash-Table-Based	5280	2780	1560	815	350
Class-Based Interpolation	7240	3730	1740	885	520
Combination Technique	52500	22000	7250	3930	2440

Table 6.1: Interpolation speed in samples/s per processor for different techniques and grid sizes for 0-forms.

The parallelized version has been tested both on a set of workstations with a TCP/IP implementation of MPI (LAM) and on the PC cluster ”Kepler”[1] of the University of Tübingen. This cluster consists of 96 dual PIII nodes connected with Myrinet and two additional front-end nodes. The results were streamed to the University of Stuttgart. All rendering times presented here include the communication lag, which off course affects the rendering speedup significantly. The visualization of the incoming ray data is performed in a sparse grid visualization toolkit that effectively hides the parallelization technique from the user.

First, we were interested in the scalability and load balancing quality of the presented approach. As one can see in Figure 6.11, the system scales almost perfectly with the number of processors, as long as the problem is computational bound, and the final TCP streaming is not hindering the rendering process. Load balancing works also extremely well for a system that does not require any additional communication at all. The load balancing presented in Figure 6.12 is expressed as the quotient of the rendering time of the fastest and the slowest processor. Note that a bad load balancing has immediate influence on the scaling properties as well.

Being able to generate previews completely eliminates the need to reduce the image resolution e.g. for finding good views of the volume. As soon as one is satisfied with image precision, the rendering process is interrupted and a new view can be set. Color Plate 6 shows different stages of this process.

Color Plates 7 and 8 show views of a 0-form data set that have been rendered in interactive rates for the very first time. This data set and the other following data sets have been computed directly on sparse grids and cannot be expanded to uniform grids due to their size.

[1] `http://kepler.sfb382-zdv.uni-tuebingen.de/`

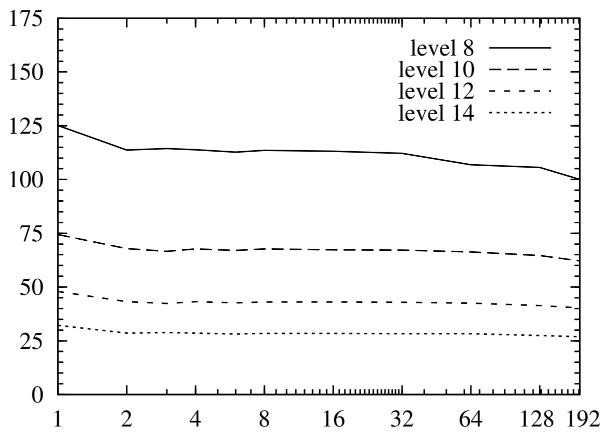


Figure 6.11: Rendering speed in rays per second and processor vs. number of processors.

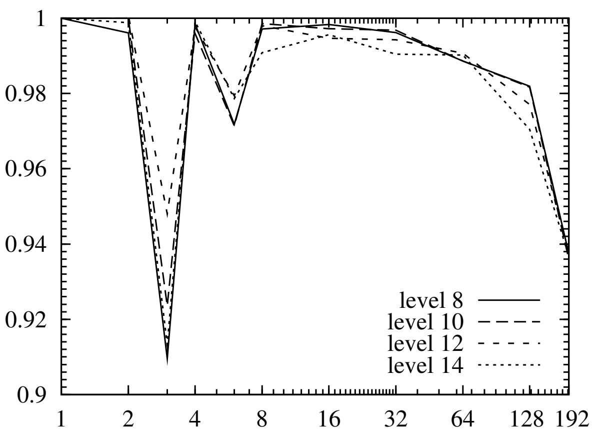


Figure 6.12: Load balancing quality.

A sparse grid 1-form of level 12 is depicted in Figure 6.13, by using line integral convolution (Volume LIC, see e.g. [Rezk-Salama et al. 1999] for additional information). Color Plate 9 shows a view of the same data set by rendering field lines for some seed points on a given starting plate, using the new interpolation classes together with a different, previously published, visualization toolkit [Teitzel et al. 1998a, Teitzel et al. 2000]. Note that the usability of the field line visualization strongly depends on the choice of the seed points. LIC does not have this restriction and shows features all over the data set. On the other hand, the resulting images are much harder to read.

Parallelization is *the* key feature to create high quality volume visualization images of multiresolution data sets at interactive rates. Due to the nature of sparse grids (small data size, high computational complexity), the parallelization itself is relatively straight-forward, and neither memory consumption nor access times are problematic. In this context implicit load balancing works very well, and does not imply any additional communication overhead. By using a specialized ray distribution pattern early preview images can be created at no additional cost. Visualizing sparse grids with Whitney forms enables scientists to work with this very promising kind of grids interactively. However, interpolation on these grids cannot be performed with the combination technique right now, which could accelerate the process even more.

As sparse grids are not very good at approximating any given uniform data set, other approaches have to be taken to represent data with as few basis functions as possible. A type of basis function that is very well capable of representing sparse, but irregular data with high local resolution are the radial basis functions, which are discussed in the next chapter.

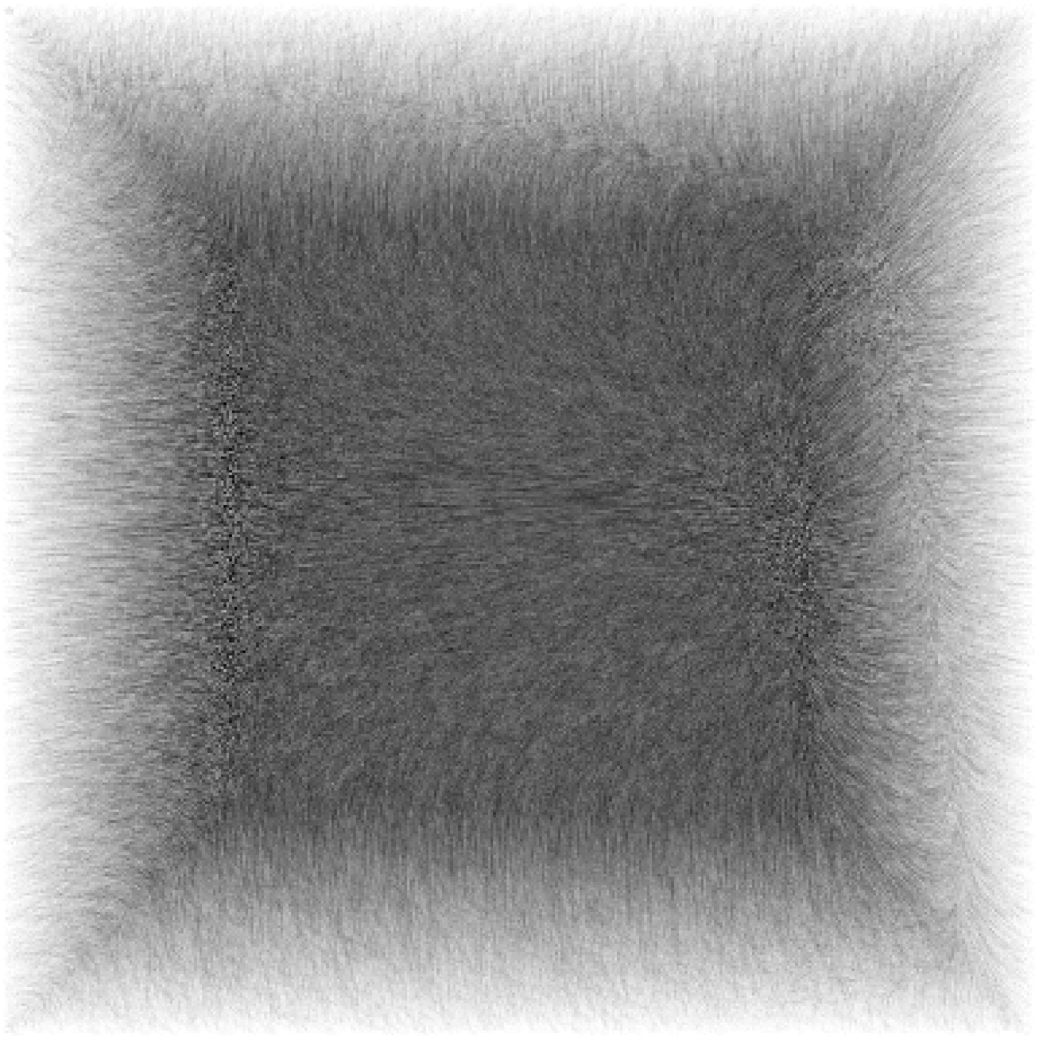


Figure 6.13: Visualizing a sparse grid 1-form of level 12 with Volume LIC.

Talia: *They're not happy about you knowing about the center either.*
Garibaldi: *Well, it's an imperfect universe.*

Babylon 5, A Voice in the Wilderness, Part 1

Chapter 7

Hierarchical Radial Basis Functions

Despite all advances in cell projection and raycasting techniques interactive rendering and exploration of large scattered or unstructured data sets are still challenging problems compared to volume rendering of uniform grids. One possibility to deal with these issues is to encode the data with radial basis functions (RBFs), which allow for a compact representation with only few coefficients. This basis representation can then be used for volume visualization with modern graphics hardware, which can reconstruct the function on a per-fragment level.

Apparently, adapting the structure of the data to the capabilities of the GPU is one of the most important steps in the process of mapping algorithms onto the graphics hardware. In this particular case, we are evaluating many basis functions per fragment, which means that the coefficients have to be accessible from the fragment pipeline. This will lead to a new concept of data set representations residing in graphics memory. Eventually, this reduces the amount of information that has to be transfered to the graphics card during visualization to only few attributes per rendered cell.

In order to reduce the number of necessary basis function evaluations, an octree-like hierarchy is developed, where the individual octree cells only store the minimum number of basis functions that are necessary to represent the given function within a certain error tolerance.

Traditionally, RBF representations use non-compact basis functions, or even bases that have increasing influence with increasing distance, because of better function approximation characteristics [Carr et al. 2001]. Of course, this contradicts the idea of using only a subset of the basis functions for evaluation per octree cell, thus a different basis with compact support or at least exponential falloff is used. Figure 7.1 shows an overview of the presented algorithms, which has been joined work with Yun Jang[1], Manfred Weiler[2], Jingshu Huang[1], David S. Ebert[1], Kelly P. Gaither[3], and Thomas Ertl[2]. This work has been accepted for publication at the Eurographics Symposium on Visualization in [Jang et al. 2004].

[1]Purdue University
[2]University of Stuttgart
[3]University of Texas

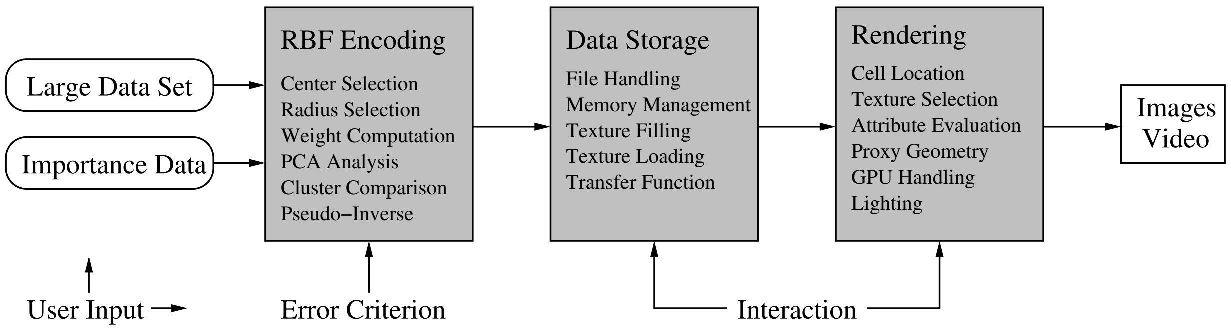


Figure 7.1: Data flow for interactive visualization of RBF encoded data sets.

7.1 Radial Basis Functions

The RBF representation of a data field is grid-less, in contrast to conventional data encodings. Thus, the positions of the basis functions are no longer implicitly given by the grid, but part of the stored information. Ideally, the basis functions should be quite simple so that they can be evaluated relatively fast. The gradient of the data set can also be calculated efficiently, if the derivative of the basis functions can be represented analytically.

Radial basis functions are circularly-symmetric functions centered at a single point [Ghosh and Nag 2001]. Possible basis functions include thin-plate splines, multi-quadrics, and Gaussians. In a nutshell, the main advantages of RBFs are their compact description, ability to interpolate and approximate sparse, non-uniformly spaced data, and analytical gradient calculation.

A function $f(\mathbf{x})$, that is represented with N radial basis functions ϕ_i, can be expressed as

$$f(x) = w_0 + \sum_{i=1}^{N} w_i \phi_i (\|\mathbf{x} - \mu_i\|) \quad , \tag{7.1}$$

where w_i is the weight and μ_i the center of RBF i. The vector norm depends on the approximation properties of the chosen basis function. An additional offset term w_0 is introduced to account for a global offset that cannot be encoded in the w_i because of the compact basis functions.

In order to interpolate a function with N points, the simple form of an RBF places basis function centers at each of the N points and then solves the linear equation system for the weights of each RBF. For data compression and smoothing, a reduction of the number of basis functions is usually performed based on optimization criteria, providing a compact functional description of the input data. Some reduction schemes introduce a constant error term, that can be represented by the global offset w_0 in (7.1). Other schemes have linear error terms that have to be included as additional terms. Often, the ϕ_i only differ by their width $\phi_i(x) = \Phi(s_i\, x)$, or they are equal $\phi_i \equiv \Phi$.

7.2 Related Work

RBFs are widely used in many fields (e.g. image processing [Fornefett et al. 1999] and medical applications [Zhang et al. 2002]). Within computer graphics, RBFs are most commonly used for compactly representing surface models and for mesh reduction [Savchenko et al. 1995, Turk and O'Brien 1999, Carr et al. 2001, Morse et al. 2001, Turk and O'Brien 2002]. RBFs have also been used for surface construction and rendering of large scattered data sets [Goshtasby 2000, Carr et al. 2001, Haber et al. 2001]. The main advantages of RBFs include their compact description, ability to interpolate and approximate sparse, non-uniformly spaced data, and analytical gradient calculation [Zhang et al. 2002].

Interactive rendering for unstructured volumes is almost entirely based on the Projected Tetrahedra (PT) algorithm [Shirley and Tuchman 1991], which is the most known cell projection algorithm. This algorithm exploits hardware-accelerated triangle scan conversion by decomposing projected tetrahedra into triangles and rasterizing these triangles with the correct color and opacity computed at the triangle vertices by ray integration. Improvements of the basic PT algorithm include better rendering quality [Stein et al. 1994, Roettger et al. 2000] and exploiting today's programmable vertex and fragment units by mapping the tetrahedra decomposition to standard graphics hardware [Weiler et al. 2002, Wylie et al. 2002, Weiler et al. 2003a], thus freeing CPU resources. Other approaches to render unstructured volumes include resampling the unstructured grid on-the-fly during volume rendering [Weiler and Ertl 2001] and using graphics-hardware-based raycasting [Weiler et al. 2003b].

Unstructured grids can provide an adaptive representation of the volume data. However, the rendering performance for unstructured grids is still inferior to that of texture-based volume rendering of structured grids. The main bottleneck is processing the tetrahedra in the correct visibility order [Max et al. 1990, Williams 1992]. Yet, the visualization of structured grids can benefit from RBF encoding as well, as texture memory is a scarce resource.

The transformation of a given function into an RBF representation is not a simple basis transformation, as the basis functions do not reside at fixed positions. Typically, this leads to a nonlinear parameter optimization problem. Many different algorithms have been used in the literature to solve this problem [Orr 1996], which can be interpreted as representing the input data density with only few basis functions with localized responses [Ghosh and Nag 2001].

7.3 RBF-Based Visualization

The RBF approach provides a superior uniform solution for the visualization of structured and unstructured volume data, especially for large data sets. Compared to a multiresolution hierarchy of structured volumes [LaMar et al. 1999, Weiler et al. 2000b], a radial basis function representation of the same data can achieve higher compression ratios since no topological information is required. The relatively small number of basis functions required to reconstruct a single fragment leads to local memory access schemes that can benefit from texture caching compared to the large bandwidth required by rendering algorithms based on uniform grids.

By reconstructing radial basis functions via per-fragment operations during rasterization we can combine a slice-based rendering approach with a compact volume representation and apply all rendering and blending techniques that are well established for texture-based volume visualization.

Grid-less encodings like radial basis functions have the ability to store details in much higher resolution than structured grid-based techniques. Effectively, the new approach combines the advantages of structured and unstructured grids. Details are usually modeled with RBFs that have a small extent, and the evaluation of these basis functions is useless for positions that are far away from the center of the basis. Space is divided hierarchically into cells to reduce the number of RBFs that have to be evaluated for any given position. For each cell a list of influencing (active) RBFs is determined and used during the rendering process.

As the encoding approach only uses a small number of basis functions, splatting [Huang et al. 2000] could be a valid alternative to per-fragment reconstruction. However, splatting with footprints of different sizes does not work well with hierarchical decomposition. Since the influence regions of different RBFs will overlap significantly, a global ordering of the RBFs is not possible. Therefore, a slice-based approach has to be taken.

7.3.1 Basis Functions

There are many basis functions that may be used in RBF encoding, such as Gaussian functions, thin-plate splines, multiquadrics, inverse multiquadrics, biharmonic splines, and triharmonic splines. Biharmonic and triharmonic splines are well-suited for surface representation and can provide better results than compactly supported RBFs [Carr et al. 2001].

When scalar data sets are encoded using compactly supported RBFs, their limited spatial support results in a small set of functions that must be evaluated to reconstruct the scalar data values at any given point. Any compact RBF will work with the realtime reconstruction method described in the following sections. The Gaussian function has been chosen as a compact basis function because the functional value exponentially converges to zero, as compared to other basis functions whose values converge polynomially. This does not exactly match the definition of compact basis functions, but the approximation is sufficient for all examined types of data. Moreover, by specifying the width for each of the Gaussian RBFs, spatially isolated functions can be modeled that accurately represent local features. With Gaussian basis functions, the RBF representation with N basis functions can be expressed as

$$f(x) = w_0 + \sum_{i=1}^{N} w_i \, e^{-\frac{\|x-\mu_i\|^2}{2\sigma_i^2}} \quad .$$

Therefore, to effectively encode a scalar data set, we need to compute the center location μ_i, weight w_i, and width σ_i^2 of each basis function.

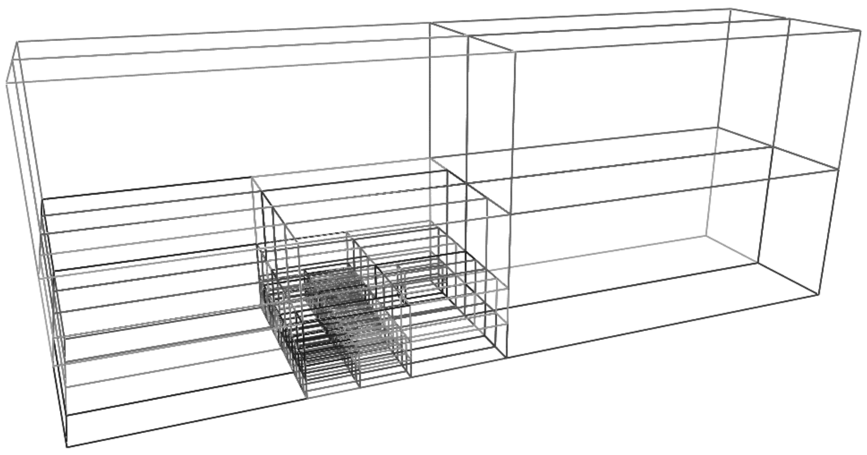


Figure 7.2: Example for a typical grid structure (blunt fin data set, see also Color Plate 13).

7.3.2 Determination of RBF Parameters

The optimal determination of basis function center locations, widths, and weights is a challenging nonlinear optimization problem. For the center selection, several approaches exist, including random subset selection, clustering algorithms, and Gaussian mixture models [Ghosh and Nag 2001]). The random subset selection and k-means clustering algorithms are trained with a Gaussian mixture model, which is prohibitive for large data sets. Therefore, the simple principal component analysis (PCA) [Jolliffe 1986] can be applied as a clustering algorithm for center determination. For regularly spaced data and for discontinuous or noisy scattered data a four-dimensional PCA (3D spatial + scalar value) is found to provide better clustering results than three-dimensional (spatial) clustering and k-means clustering techniques. For scalar scattered fields with smoothly varying data, 4D PCA is almost as accurate as 3D k-means with Gaussian mixture model training (approximately 0.5% higher average error), but it is ten times faster. Finally, we end up with a clustering method similar to [Co et al. 2003]. Figure 7.2 shows a typical grid structure that can be achieved with this clustering technique.

The basis function width should cover the local data spread, as it affects the smoothness of the functional interpolation. Although there are optimal solutions and training algorithms for the selection of width [Orr 1996], a multiple, typically 1.5 to 2, of the average distance to some number of nearest neighbors is usually sufficient for a good approximation [Ghosh and Nag 2001].

As a last step, the weight of each basis function is determined, usually by minimizing the summed square error. There are several solution methods for the least square problem, such as gradient descent, Cholesky decomposition, singular value decomposition (SVD), and orthogonal search [Ahmed 1994], and each method has its advantages and disadvantages. In our case, since we have already determined the best centers and widths of the basis functions, we can use the pseudo-inverse method to find the weights, employing singular value decomposition. Traditionally, orthogonal search simultaneously determines the centers, weights, and widths one by one.

7.3.3 Encoding Error Minimization

After the initial set of RBFs is determined, we can measure the error of the encoding compared to the actual scalar value at each point in the input data set. Since the summed square error has been minimized, good mean errors and RMS error values are achieved. However, a maximum error cannot be guaranteed, though experiments with smooth data sets have shown that this approximation results in typically less than 1% data points having significant errors. If the input data set is noisy and has outliers, this might be the desired result. However, if the goal is to preserve the fine detail variation in the data distribution, further processing of the data is required.

The maximum error can be reduced by adding new RBF centers at the positions containing significant errors. The user can then either use the error value as the weight for an additional RBF with a narrow width for best preservation of the fine features, or add these points as new centers with small widths and resolve the pseudo-inverse to globally adjust the functional approximation. The first method produces the best local error minimization, while the second method reduces the error while producing smoother functional approximation. Using the first method must be considered harmful, though, if the errors exceed certain safety margins. As error differences are typically only measured on former grid points, this technique does not provide any control over the functional approximation between the chosen RBF centers.

The accuracy of the RBF Gaussian encoding for several data sets can be seen in Table 7.1. These figures show that, with accuracy criteria of a few percent, very good compression of unstructured data sets can be achieved using RBF encoding. As expected, compression is not as good for discontinuous scalar values, such as computational fluid dynamics (CFD) shock values.

Once the original volume data is encoded as a weighted sum of RBFs, the connectivity information and the original grid can be discarded. The set of RBFs will reproduce the original scalar field within the accuracy tolerance specified during encoding.

Data set	# data points	# RBFs	Avg. Abs. Error
X38 shock	1,943,483	2,932	0.05
X38 density	1,943,483	1611	0.014
Oil reservoir	156,642	458	0.007
Stanford bunny	69,451	5,199	0.03

Table 7.1: Accuracy and compression for RBF encoding of several data sets.

7.3.4 Spatial Data Structure

The RBF parameters are stored in an adaptive octree that must be traversed to reconstruct the data value for a given region in space. The octree itself is created top-down in a hierarchical process, subdividing cells containing more RBFs than a user-defined limit. For each cell all basis functions are evaluated at the point of the cell that is closest to the center of the RBF (for most cases, this will be one of the corner points), and it is added to the list of contributing basis functions of the cell if its influence is larger than a given threshold ε. The subdivision

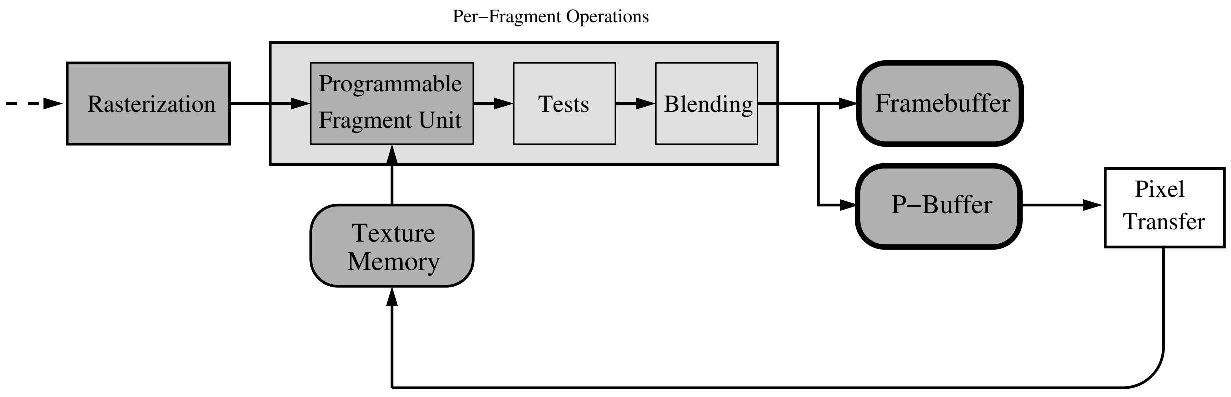


Figure 7.3: Relevant pipeline part for reconstructing RBF encoded scalar data fields.

terminates when the number of basis functions per cell is less than n with n being a user-specified upper bound, typically representing the maximum number of RBFs that can be rendered in one pass (see Subsection 7.4.1). The process terminates as well when further subdivision does not significantly reduce the number of basis functions for the eight children cells.

If very high accuracy is needed, the user may choose to still render all the basis functions at the cost of interactivity. To account for errors while supporting more interactive rendering, the error introduced by skipping several RBFs per cell is evaluated and stored with the corners of each cell of the octree structure. During rasterization, these error values are interpolated in order to reduce discontinuities at cell boundaries. From initial experiments, this linear approximation provides good results. However, for high ε discontinuities are replaced by linear interpolation artifacts.

7.4 Interactive Reconstruction

As stated previously, the programmability of the GPU fragment processor can be used to perform an on-the-fly reconstruction of the RBF encoded volumetric data during rasterization. Since the RBFs are evaluated by the GPU for each rendered fragment, the reconstructed volumetric data can be mapped onto any rendered geometric property, such as color or opacity. Figure 7.3 shows the relevant part of the OpenGL pipeline for interactive reconstruction.

With this approach the encoding of the data is hidden from the rendering and a variety of visualization algorithms can be used, such as arbitrarily oriented cutting planes, hardware-accelerated texture-based volume rendering, and volume-rendered non-polygonal isosurfaces. Since current graphics hardware such as the GeForce FX has the ability to run fragment programs with up to 1000 operations in a single pass, high-level shader languages have been used for the reconstruction process, as long assembler programs are hard to code and to debug. The implementation is based on NVIDIA's Cg [Mark et al. 2003], which supports both graphics APIs, DirectX and OpenGL, by providing different compiler profiles. However, during final tests we noticed that using hand-optimized assembler code instead of generated code can accelerate the rendering by up to 100%. Still, the optimized code of the loop does not look much different from the compiled code (10 vs. 12 assembler instructions for evaluating a Gaussian RBF without gradient). In contrast to modern CPUs, there is no documentation about internal pipeline design and no optimization guidelines for the current GPUs, thus it can only be guessed that this speedup mostly stems from the lower number of temporary registers needed.

7.4.1 Data Storage

During rendering, the fragment processor has to be able to access the basis functions that are selected for function value reconstruction inside the current cell. The high memory bandwidth of the graphics adapter is exploited by storing the RBF data at full precision in a set of two floating point texture maps. The first map is an RGBA map holding the positions of the RBF centers and the weights of the RBF functions in the RGB and alpha components, respectively. The second map consists of only one color component storing the widths of the RBF functions. In order to reduce the number of fragment operations required for the reconstruction, we do not store the actual widths, but instead store $(2\sigma_i^2)^{-1}$. The required textures reside within the local memory of the graphics adapter, since the total amount of RBF data is small. Thus, the bottleneck of transferring data from the CPU to the GPU is avoided.

Since all the parameters of a single cell are stored consecutively in the texture maps, the fragment processor can access several RBFs in a single pass by applying an increasing offset to the texture coordinates. Thus, the number of RBF centers that can be processed is only limited by the number of fragment operations that can be performed within a single pass, e.g. 1024 on the GeForce FX chip.

Figure 7.4 shows how several sets of RBFs for different cells are stored in a single texture. Additional texture sets can be used if the RBF data exceeds the maximum size of one texture map, though we did not encounter a data set that needs more RBFs for its representation than can be stored in a single texture.

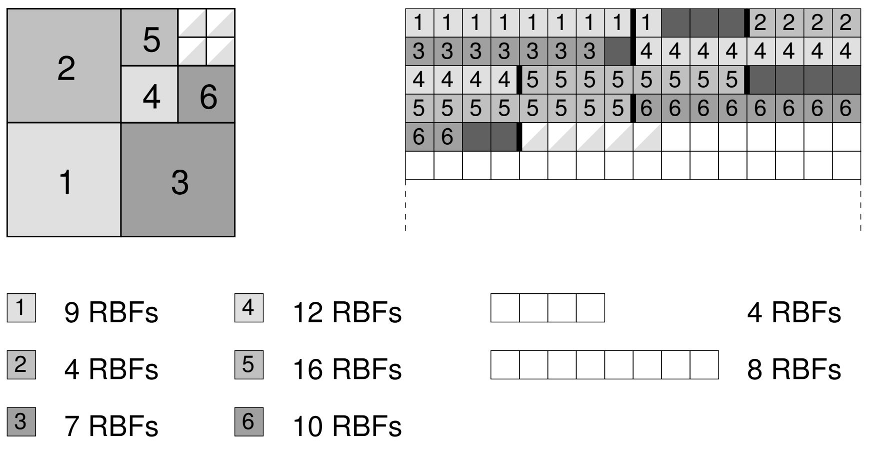


Figure 7.4: The RBF data for all cells is tightly packed into a single set of texture maps. In this example, two different fragment programs for 4 and 8 RBF evaluations are available.

Since current graphics hardware does not support dynamic loops in fragment processing, the number of RBF functions to be evaluated in a single program is fixed. This requires specialized programs for each possible number of RBFs, as the number of RBFs to be evaluated per cell may vary throughout the spatial decomposition. However, extensive program switching can be avoided in order to reduce performance penalties by the restriction to a rather small set of different programs for 10, 20, 30, ... RBFs. Cells that require an intermediate number of RBFs pad their RBF data in the texture maps with zero values up to the next available fragment program size.

7.4.2 High Level Rendering

All used rendering techniques utilize slice polygons, which are computed by intersecting a plane with the bounding box of the desired volume domain. For volumetric rendering, a set of these slices oriented orthogonal to the viewing direction are placed equidistantly within the volume domain, rendered, and finally composited back-to-front.

As mentioned previously, a spatial decomposition of the data domain is used in order to reduce the number of RBF centers that have to be considered for a single fragment. Since each cell of the decomposition potentially has a different set of RBF centers, different rendering states have to be set for each cell. This situation is quite similar to the bricking approach taken in texture-based volume rendering when the size of the data set exceeds the physical texture memory [Grzeszczuk et al. 1998]. There, the data set is decomposed into a set of blocks or bricks, and each brick is rendered with a separate three-dimensional texture. The bricks are sliced independently in back-to-front order to minimize state changes. Thus, all slice planes are clipped at the boundaries of a given cell and the resulting polygons are rendered within the cell before proceeding to the next cell.

For semi-transparent volume rendering, we also have to apply a visibility sorting of the cells to achieve proper rendering. For uniform grid spatial decompositions, correct ordering can easily be determined by sorting the grid cells by the distance of their centers from the viewer's position. For the octree structure, a recursive algorithm is applied at each level that sorts the eight children using the same distance criterion and descends in a depth-first manner based on the level-wise ordering of the children.

If a cell contains more RBFs than can be handled by the fragment processor in one step, multiple passes have to be rendered, using a set of two hardware-accelerated floating point P-buffers. Two P-buffers are needed since the GeForce FX does not support simultaneous read and write operations on the same buffer. The fragment program partially evaluates the RBF sum and writes the intermediate result into one of the buffers, which is used as an input for the next rendering pass by binding the P-buffer to a texture map. In order to minimize the number of expensive rendering context switches, the inner loop of the rendering routine is changed when multiple passes are needed. We render a complete slice for all cells first, then proceed to the next slice. Thus, sorting of the cells is no longer necessary. Figure 7.5 shows the corresponding pseudocode.

```
for (all slices)
{
  activeCellList      = createActiveCellList();
  intersectedCellList = copy(activeCellList);

  // Phase I
  while (cells in activeCellList)
  {
    setupRenderingPass();

    for (each cell c in activeCellList)
    {
      if (c contains only ONE chunk of unrendered RBFs)
        removeFromActiveCells(c);
      else
        renderIntersectionPolygon(c);
    }
  }

  // Phase II
  setupFinalRenderingPass();

  for (each cell c in intersectedCellList)
    renderIntersectionPolygon(c);
}
```

Figure 7.5: Traversal algorithm for slice-based RBF-rendering.

Multi-pass rendering is performed in two phases: In the first phase the fragment program partially evaluates the RBF sum and writes the intermediate result into one of the buffers. This result is then used as an input for the next rendering pass by binding the P-buffer to a texture map. The final pass (Phase II) directly writes to the graphics context of the program window.

An active cell list is utilized to minimize the cell traversal costs. For each slice, the list is initialized with all cells intersected by the the current slice. For each cell we render all but one of the multiple passes during the first phase. The last rendering pass is performed in the second phase, guaranteeing that the intersected slice area for each cell is finally rasterized into the framebuffer, using the provided color lookup table as a transfer function. Therefore, the cell is removed from the active cell list as soon as only one additional pass would be required. A second list is maintained to store all cells which must be traversed during the last rendering pass. It is initialized with all intersected cells as well. Note that the multi-pass rendering allows for a tight packing of the cells' data within several texture sets, since we only have to guarantee that the RBF parameters needed for one pass are stored consecutively.

```
// Maximum number of basis functions for loop unrolling
#define CONST_NUMFUNCS 36

void main ( // Current world coordinates and interpolated error
           in      float4      inpos    : TEXCOORD0,
           in      float       error    : TEXCOORD1,
           // Resulting color
           out     float4      output   : COLOR,
           // Color table
           uniform sampler1D   map,
           // RBF textures
           uniform samplerRECT rbfcenter,
           uniform samplerRECT rbfwidth,
           // Texture  addressing: offset + increment
           uniform float4      texstart,
           uniform float4      texinc,
           // Bias for RBF reconstruction
           uniform float       bias,
           // Color table scale + bias, alpha scale
           uniform float4      mapSBA)
{
  float  val    = 0.0;
  float4 texpos = texstart, output;

  // Lookup and evaluate RBFs
  for (float i = 0; i < CONST_NUMFUNCS; i++) {
    float4 tmp    = texRECT (rbfcenter, texpos.xy);
    float  w_inv  = texRECT (rbfwidth, texpos.xy);
    float3 vec    = tmp.rgb - inpos.xyz;
    float  expval = - dot (vec, vec) * w_inv;
    val    += tmp.a * exp (expval);
    // Advance to next RBF parameter location
    texpos += texinc;
  }
  // Add bias and interpolated error
  val += bias + error;
  // Color table lookup after scale + bias
  output.rbga = tex1D (map, (val + mapSBA.r) * mapSBA.g);
  // Transparency correction for volume slicing
  output.a   *= mapSBA.a;
  return output;
}
```

Figure 7.6: The fragment program for reconstructing Gaussian radial basis functions.

7.4.3 Per-Fragment Reconstruction

The RBF visualization system loads pre-compiled fragment programs instead of using Cg's on-the-fly compilation in order to reduce start-up overhead. Additionally, this approach easily allows for adding support for other basis function types and writing hand optimized assembler code.

Based on the RBF encoding in the texture maps, the fragment program presented in Figure 7.6 is applied for the reconstruction. All multi-pass related parts have been removed in the presented program for clarity. The extension to multiple passes is straightforward, but introduces several additional state variables that are not necessary for understanding the basic principles. The program uses the Gaussian radial basis function presented in Section 7.3.1. Other fragment programs implement inverse multiquadric RBFs and different visualization modes, in particular isosurface rendering. The latter additionally performs lighting calculations based on the data gradient that is analytically evaluated in parallel to the data value as:

$$\nabla f(x) = -\sum_{i=1}^{N} \frac{x-\mu_i}{\sigma_i^2} w_i \, e^{-\frac{\|x-\mu_i\|^2}{2\sigma_i^2}}$$

The programs accumulate the contributions of the RBF functions in a local variable. The iteration over the number of RBFs include the lookup of the RBF center coordinates, the RBF weight and width from the texture maps, the computation of the center's distance to the current fragment position, and finally the evaluation of the RBF function. The model space coordinates needed for the distance calculation are provided by the interplated texture coordinates `inpos`. Due to performance issues the base-two exponential is used instead of the standard `exp`, which is compensated with a correction factor multiplied to σ_i-entries in the texture maps. Two texture lookups are required per fragment since the five RBF parameters cannot be stored in only one texel. By combining four RBF widths in one RGBA texel and performing the width lookup only every four RBFs, the total number of lookups could been reduced by up to 37.5%. However, the more complicated data handling and the computation of the different texture coordinates resulted in very little performance increase.

After evaluating all RBF functions, the constant bias is added. If the spatial data structure includes error values at the corner of each cell, the linearly interpolated error provided through the secondary texture coordinates is added. An additional scale and bias operation accounts for the relevant data range. The resulting scalar value is finally mapped to an output color by a 1D texture lookup in the transfer function table.

7.5 Results and Comparison

The algorithm has been implemented and tested on a Pentium 4 1800MHz processor with an NVIDIA GeForce FX5800 Ultra graphics processor on a variety of data sets. These data sets include a computationally simulated X38 configuration, a natural convection simulation, a black oil reservoir simulation, the Stanford bunny, and the blunt fin data set. Unless stated differently all timings are measured on a GeForce FX5900 Ultra with a viewport of 400×400 pixels using a set of fragment programs for 20, 40, 60, 80, and 100 basis functions. In the following, the results obtained from encoding each of these data sets using the previously described radial basis functions are discussed.

The X38 data set that has been used to test the accuracy of the RBFs is based on a tetrahedral finite element viscous calculation computed on geometry configured to emulate the X38 Crew Return Vehicle. This data set represents a single time step in the reentry process into the atmosphere. The simulation was computed on an unstructured grid containing 1,943,483 tetrahedra at a 30 degree angle of attack.

The procedural encoding has been merged with domain specific data culling in order to test the ability of RBFs to encode sparsely selected high-resolution features of a data set. We computed the normal Mach number and extracted data values greater than 0.6, as the shock is created due to the transition from sub-sonic speeds ($<$ Mach 1) to super-sonic speeds ($>$ Mach 1). The actual shock volume has a normal Mach number very close to 1.0. This clamped data set was then encoded with $2,932$ Gaussian RBFs. The images in Color Plate 10 show volume renderings of a tight bound on the shock volume, with data between 0.9 and 1.1. Color Plate 11 shows a comparison of encoding the shock data with 855 and 1,147 RBFs, respectively, for the less limited range of values 0.7 to 1.7. For the important narrow shock data range $0.8 - 1.02$ shown in the color plate, the overall structure of the shock is the same, with details of the bow shock missing in the left image.

The density data from this simulation has also been encoded. In this data set the most interesting values are density values less than 0.5. The data set was encoded using only 1,611 RBFs since the density variation doesn’t have the sharp discontinuities of the shock data. A volumetric isosurface rendering of the low density region of the data can be seen in Figure 7.7.

Another isosurface rendering in Figure 7.8 shows the Stanford bunny, which was created from a polygon model with $69,451$ vertices. This data set was encoded using 5,199 basis functions with a maximum of 100 RBFs per octree cell. Please note, that the image — while showing a surface — has been created using the described volume rendering approach.

So far, only a single cell has been used for encoding, which results in poor performance since evaluating all RBFs for each fragment is extremely time consuming. Interactive volume rendering of these data sets is only possible, when hierarchical encoded volumes are used.

Color Plate 12 shows volume and isosurface rendering using a hierarchical RBF composition generated from a natural convection simulation. The volume rendering shows the underlying cell decomposition of the RBF data set and runs at approximately 1.8 fps using 32 slices. In isosurface mode the performance drops to 0.4 fps, since the isosurface fragment program is more expensive, and the most optimized program version could not be used here due to accuracy issues. The original data set contains 48,000 tetrahedras from the 80th time step of temperature generated from a natural convection simulation of a non-Newtonian fluid in a cube. The domain is heated from below, cooled from above, and has a fixed linear temperature profile imposed on the sidewalls.

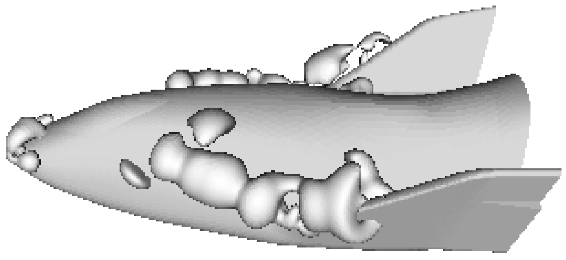


Figure 7.7: Volume isosurface rendering of the X38 density data reconstructed with 1,611 RBFs.

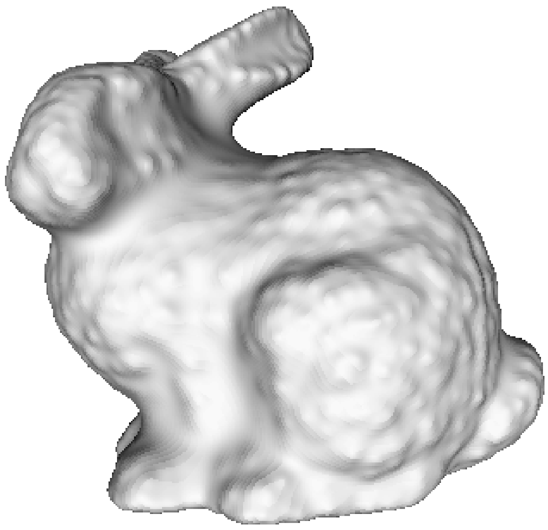


Figure 7.8: Volumetric isosurface rendering of the Stanford bynny, using 5,199 basis functions.

The blunt fin data set displayed in Color Plate 13 has been encoded hierarchically with 695 RBFs, distributed into 238 cells (see Figure 7.2) with a maximum of 60 RBFs per cell. A set of fragment programs with up to 60 basis functions has been used for rendering the data set at interactive rates. Using 64 slices the data set renders at approximately 3.7 fps.

Color Plate 14 shows volume renderings of the reconstructed oil reservoir data set computed by the Center for Subsurface Modeling at the University of Texas at Austin. The data set is a simulation of a black-oil reservoir model used to predict placement of water injection wells to maximize oil from production wells. The data set has 195,102 tetrahedra containing water pressure values for the injection well. The data set renders at approximately 1.8 fps using 64 slices.

All of the presented figures were generated with the hardware-accelerated reconstruction algorithm. For slice plane rendering, we achieve a performance of 7 to 75 fps on a completely filled 400^2 viewport due to a comparatively high amount of RBFs per cell. Exploiting the half-float register type of Cg led to a performance improvement between 30% and 300%, depending on the rendering mode. However, this program could not be applied to all tested data sets due to the limited 16 bit precision.

The 1024 fragment program instruction limit of the GeForce FX allow to evaluate 59 to 126 RBFs per pass, depending on data encoding and the rendering mode. When multi-pass rendering is needed, larger fragment programs show higher performance, because they need to write intermediate results to the P-buffer less often. With adaptive octree encoding, the addition of cells in the hierarchy causes a measurable overhead in the rendering process. However, the reduction of wasted Gaussian RBF evaluations clearly outweighs this overhead and significantly increases the overall performance.

The demonstrated technique is a novel, unified approach for the interactive reconstruction and visualization of arbitrary 3D scalar fields, including voxel data, unstructured data, and polygonal data. By combining compact RBF functional encoding, hardware-accelerated functional reconstruction, and domain knowledge of data importance, a system has been developed that avoids the traditional data transfer bottleneck of hardware-accelerated rendering of large scalar fields. The flexibility and extensibility of functional encoding and interactive reconstruction allows the interactive exploration of very large data sets from a variety of sources. The presented algorithm is capable of visualizing data sets with a few million tetrahedra at interactive rates using cutting planes. For smaller data sets, interactive rates can be achieved for volume rendering as well, using the hierarchical decomposition approach. For large number of slices or high resolutions however, volume rendering of the RBF data with current graphics hardware may take up to seconds per frame. Image quality could also be improved by incorporating pre-integrated volume rendering, and due to the nature of the slice-based volume rendering algorithm this seems to be straight-forward. Additionally, the RBF encodings could greatly benefit from clustering techniques. When basis functions are far away from the currently processed cell, encoding a set of these functions with a single RBF could provide a far better approximation of the original data than neglecting the contribution of these bases altogether.

RBFs are a good means to encode scalar data that is mainly smooth, but has a high resolution in small regions of the data set only. However, they are not useful for encoding types of data that have the main purpose of comprising a lot of dedicated high-resolution positions in the data set. All volume rendering techniques based on slice rendering will either fail to visualize the positions with sufficient resolution, or they will be extremely slow while needing extraordinary amounts of memory due to the size of the resampled data set. For the visualization of these uncorrelated data sets splatting based approaches seem to be more appropriate, which will be analyzed in the next chapter.

Sheridan: (to Garibaldi)
The Universe doesn't give you any points for doing things that are easy.

Babylon 5, The Geometry of Shadows

Chapter 8

Splatting of Uncorrelated Data

Numerical particle simulations and astronomical observations create huge data sets containing uncorrelated 3D points of varying size. These data sets cannot be visualized interactively by simply rendering millions of colored points for each frame. Therefore, in many visualization applications a scalar density corresponding to the point distribution is resampled on a regular grid for direct volume rendering. However, many fine details are usually lost for voxel resolutions which still allow interactive visualization on standard workstations. Since no surface geometry is associated with these data sets, the recently introduced point-based rendering algorithms described in the related work section below cannot be applied as well.

In this chapter a method to accelerate the visualization of scattered point data by a hierarchical data structure based on a PCA clustering procedure is presented. By traversing this structure for each frame we can trade-off rendering speed vs. image quality. This scheme also reduces memory consumption by using quantized relative coordinates and it allows for fast sorting of semi-transparent clusters. Various software and hardware implementations of the renderer are analyzed, and it is demonstrated that it is now possible to visualize data sets with tens of millions of points interactively with sub-pixel screen space error on current PC graphics hardware employing advanced vertex shader functionality. This work has first been published at the Visualization conference in [Hopf and Ertl 2003]. Work about splatting of time-dependent data has been accepted for publication in Computer Graphics and Applications' special issue about Point Rendering 2004.

8.1 Rendering Point Clouds

Quite a number of physical simulations create large point-based data sets, for example Smoothed Particle Hydrodynamics (SPH) [Monaghan 1992] and n-body simulations [Jenkins et al. 1998] in astrophysics. Other sources of scattered point data are astronomical observations where new techniques for measuring three-dimensional positions of stars as in the GAIA project[1] will create

[1] `http://astro.estec.esa.nl/GAIA/gaia.html`

huge real-world data sets in the near future as well. These data sets contain up to hundreds of millions of points each with information about position $\mathbf{x}_i$, diameter s_i, and intensity c_i at various wavelengths.

These data sets are too large to be rendered in their entirety at interactive frame rates and the memory requirements are quite problematic for standard PCs as well. An alternative approach [Kähler et al. 2002, Park et al. 2002] is to resample the data sets and use standard texture-based volume visualization. However, this technique imposes a low-pass filter on the data set, and for reasonable frame rates and memory usage the filter domain is so large that almost no subtle details within the data will be visible any more. To some extent this can be avoided for off-line rendering of animations. In this case hierarchical volume scene graphs can be used e.g. for visualizing stellar nebula [Nadeau et al. 2000].

In order to allow scientists to view these data sets at high resolution interactively on desktop workstations or PCs, we want to visualize the scattered data directly without resampling them to a density volume. We can achieve significant speedup by applying clustering techniques to create a hierarchical representation of the data set. The hierarchy can then be rendered adaptively according to screen resolution and focus points, and a lower hierarchy level can be chosen for the visualization during interaction. Of course, hierarchical data structures generate additional memory overhead imposing even greater restrictions on the maximum data size, but storage requirements can be reduced using relative position coding, while still maintaining high accuracy with respect to the particle positions.

In order to visualize scattered data interactively the point coordinates have to be transformed into image space and rasterized into the framebuffer. Current graphics hardware is highly optimized for this task and frees up the CPU for concurrent hierarchy selection and traversal. As triangles are the dominating primitive in computer games, rasterization throughput may be higher for polygons than for points. However, this will have no major effect, since the presented approach is more likely to be geometry limited rather than rasterization limited, because large numbers of points can only be visually perceived well as long as they do not overlap too much. For certain types of data — e.g. with widely varying point sizes or semitransparent appearance — blending may be necessary in order to enable visual depth perception. This requires the points to be sorted according to their projected z coordinates. Due to the high number of points this is nontrivial to do in real-time, but can be efficiently implemented based on the hierarchical data structures.

8.2 Related Work

There has been quite a lot of work in the area of using footprints as rendering primitives for sampled data, as indicated in Section 2.3. Regarding hierarchical algorithms, splatting has been introduced by Laur and Hanrahan [1991], using Gouraud-shaded polygons for volume rendering. Most research in the splatting community was about the improvement of the visual quality of texture splatting; however, the techniques described in these papers only apply to the reconstruction of continuous functions e.g. for volume rendering of regular grid data, and they do not address adaptive rendering or data size reduction. Additionally, there exist a number of non-real-time rendering systems for large point-based data sets, e.g. for rendering film sequences [Cox 1996]. Another promising approach uses points for enhancing a low-pass filtered regular data set in regions with high frequencies [Wilson et al. 2002]. Again, this does not match the needs for rendering data sets with extremely detailed positional information.

Using points as rendering primitives is a topic of ongoing research. However, almost all publications in this area deal with the rendering of geometric surfaces. Alexa et al. [2001], Pfister et al. [2000], Rusinkiewicz and Levoy [2000], Wand et al. [2001], and Zwicker et al. [2001b] showed different methods to create and efficiently render data hierarchies of surfaces represented by sample points. As the intrinsic model of points describing a surface is fundamentally different from the model used for scattered data, their clustering techniques cannot be applied here. Pauly et al. [2002] used principal component analysis for clustering, but with a different hierarchy concept compared to the described approach. Some systems [Rusinkiewicz and Levoy 2000, Botsch et al. 2002] use quantized relative coordinates for storing the points in a hierarchical data structure, but these approaches were not optimized for fast GPU access because the data structures had to be interpreted by the CPU. Additionally, the presented rendering techniques have been designed to create smooth surfaces without holes and they allow no or only few layers of transparency. Again, this does not meet all requirements for volumetric visualization.

First steps for rendering uncorrelated samples for SPH data have been presented by Rau and Straßer [1995]. Meredith and Ma [2001] introduced multiresolution splatting for rendering irregular volume data. Most of the techniques they developed deal with the handling of the unstructured data, but the adaptive rendering based on octrees could be the basis for an extended algorithm working with grid-less data as well. Their renderer is able to render approximately 400,000 splats per second on a high-end machine. Jang et al. [2002] presented a multiresolution splatting approach for non-uniform data. However, in their solution the higher hierarchy levels are always stored in uniform grids, and they cannot render more than approximately 135,000 splats per second. This technique seems to be more appropriate for almost flat and regular data.

For rendering large quantities of splats a simple brute force approach would store the complete data set on the graphics card and use point array rendering for displaying the data set. As soon as the data set does not fit into graphics memory, rendering speed can drop by an order of magnitude.

In order to allow scientists to view these data sets at high resolution interactively on desktop workstations or PCs, we want to visualize the scattered data directly without resampling them to a density volume. In this chapter a hierarchical data structure based on a principal component analysis (PCA) clustering procedure is proposed for accelerating the visualization of scattered point data. By traversing this structure for each frame trade rendering speed can be traded for image quality, and lower hierarchy levels can be used during interaction. The presented scheme also allows interpolating the given point positions using cubic splines, and it reduces memory consumption by using quantized relative coordinates and Lempel-Ziv compression of delta encoded control points. Additionally, it supports fast approximative sorting of semi-transparent clusters. It will be demonstrated that it is now possible to visualize data sets with tens of millions of points interactively with sub-pixel screen space error on current PC graphics hardware

- Select cluster j (point indices I_j) with largest distortion Δ_j
- Calculate auto-covariance matrix from centroid X_j:
  $A = \sum_{i \in I_j} (\mathbf{x}_i - \mathbf{X}_j)(\mathbf{x}_i - \mathbf{X}_j)^T$
- Find Eigenvector $e_{\max}$ of A corresponding to the largest Eigenvalue $\lambda_{\max}$
- Split cluster j into two new clusters:
  $I_{\mathrm{n1}} = \{i \in I_j : \langle \mathbf{x}_i - \mathbf{X}_j, e_{\max} \rangle \geq 0\}$
  $I_{\mathrm{n2}} = \{i \in I_j : \langle \mathbf{x}_i - \mathbf{X}_j, e_{\max} \rangle < 0\}$
- Calculate centroids and distortions for the new clusters

Figure 8.1: The PCA split algorithm.

employing advanced vertex shader functionality. For time-dependent data sets, we are still able to visualize data sets with millions of points and several time steps interactively.

As PCA is a standard technique, only a short summary of the PCA split algorithm is presented in the following section, more details can be found e.g. in [Jolliffe 1986]. More information about interpolation using cubic splines can be found for example in [Farin et al. 2002]. The system uses Lempel-Ziv for compression, a standard technique that is covered e.g. by [Sayood 2000].

8.3 Creating the Hierarchy

In order to create one level of the hierarchy the input data points have to be sorted into bins. For each bin a point on the next lower hierarchy level is created, representing all points that fell into that bin. The properties of the newly created point are chosen so that its visual representation matches that of the substituted points best.

To obtain the set of bins several clustering schemes can be used. The most common solution is to subdivide the data set into an octree, which can be used efficiently for sorting as well (Section 8.7).

Another approach that has much better spatial adaptation properties is the principal component analysis. It can be used to find a splitting plane for a set of points that divides the set into two clusters, so that the distortion of the individual sets as defined below gets minimal. Basically, this plane is perpendicular to the Eigenvector corresponding to the smallest Eigenvalue of the inertial tensor. After some simplifications, you finally get the PCA split algorithm as seen in Figure 8.1.

8.3.1 Static Data

In the following I_j denotes the set of indices of the points of cluster j. That is, cluster j consists of all points $\mathbf{x}_i$, $i \in I_j$ with diameters s_i, and has the weighted centroid $\mathbf{X}_j$ and distortion Δ_j with

$$\mathbf{X}_j = \frac{\sum_{i \in I_j} s_i \cdot \mathbf{x}_i}{\sum_{i \in I_j} s_i} \quad , \qquad \Delta_j^2 = \sum_{i \in I_j} \left\| \mathbf{x}_i - \mathbf{X}_j \right\|_2^2 \quad .$$

As this split operation has to be performed several million times, fast cluster selection is of uttermost importance. Therefore, the clusters are kept in a skip-list [Pugh 1990], sorted by decreasing Δ_j^2. A skip-list is essentially a sorted linked list with randomized link depth, with $O(\log n)$ complexity in the average case for search, insert, and delete operations. Its properties are similar to balanced trees, with the advantage of faster insert and delete, $O(1)$ largest value search, very small memory footprint, and almost trivial implementation.

This splitting process is continued until the maximum distortion or the average cluster size fall below pre-defined minima. After the visual properties of the new points have been obtained, these points undergo another series of PCA splits in order to create the next hierarchy level. This can be seen as a bottom-up hierarchy creation process, clustering each level top-down.

For most applications like the virgo data set a hierarchy depth of more than about 6 levels is usually not appropriate. For this data set with its 16.8 million points the hierarchy creation process takes only a few minutes.

8.3.2 Time-Varying Data

For dynamic data, the PCA split is not performed in the standard euclidian space, but in $3T$-dimensional space, with T being the number of time steps. Each vector $\mathbf{x}_i \in \mathbb{R}^{3T}$ represents the position of a single point in all time steps simultaneously. This way, points that are close to each other in one time step, but get separated during time, will be put into different bins. Therefore, the typical bin size will be much smaller than in the static case. The analysis of measures that can be taken in order to improve the clustering in the time-dependent case remains future work.

As time-dependent data sets are usually larger than the available main memory, the clustering process has to be implemented out-of-core. In order to reduce hard disk accesses, only a subset of all time steps is used for the first few clustering steps. As soon as all time steps of the currently investigated cluster fit into main memory, the full resolution is used. As the first levels of the hierarchy are never used for rendering, any errors introduced due to the low temporal resolution can be easily compensated in the following levels.

In order to implement this out-of-core approach efficiently, the hierarchy creation process has to be reversed, compared to the static case. Finally, we end up with a top-down hierarchy creation process, clustering each level top-down.

8.3.3 Creating Representatives

For creating a visually approximative representation of the cluster j compared to its children the most important aspect is that the radiant flux Φ has to be the same. For the irradiance c_j of the new centroid point representing the cluster this means for each of the representative wave lengths

$$\Phi_j = A_j \cdot c_j = \frac{\pi}{4} s_j^2 \, c_j = \frac{\pi}{4} \sum_{i \in I_j} s_i^2 \, c_i \quad . \tag{8.1}$$

The cluster representative should be larger than the largest of its children in order to keep some visual continuity. Additionally, small cluster points would have very high local intensities, which could saturate the covered pixels in the blending step during rendering. Sparse clusters — that is clusters with a large average distance of their children to the centroid compared to the children's point sizes — should have larger and dimmer representatives than locally agglomerated ones. On the other hand, they must not be too large, as the human eye is very sensitive to edges, and enlarging a point implies reducing its intensity, diminishing the visibility of the edge.

After comparing several different functions, I found a trade-off that creates acceptable results for almost all point distributions. It tries to combine point sizes and their distances to the centroid, and ensures that the final size does not fall below the size of the largest point of the cluster:

$$\begin{aligned} m_j &= \operatorname*{argmax}_{i\in I_j} s_i \ , \\ s_j &= \frac{0.5}{|I_j|-1}\sqrt{\sum_{i\in I_j\setminus\{m_j\}} s_i \left\|\mathbf{x}_i - \mathbf{X}_j\right\|_2} \ + s_{m_j} \ . \end{aligned} \tag{8.2}$$

The scaling factor $\frac{1}{2}$ in (8.2) of the weighted average point size of all points except the largest one before adding to the largest point size s_{m_j} has been determined empirically.

This calculated point size is subject to further restrictions, if intensities are stored in main memory as unsigned bytes in order to save memory. The system has to assure that the calculated point size does not overflow the intensity range, and it has to increase the point size in case of saturation.

(8.1) and (8.2) are highly dependent on the blending function, and the presented definition only holds for cumulative blending ($C = c_1 + c_2$). For other blending functions, like the over operator ($C = \alpha_1 c_1 + (1-\alpha_1)c_2$), c_j may be view-dependent, as the total flux of overlapping points is no longer necessarily the sum of the individual fluxes of the points. With the current approach view-dependent intensities cannot be modeled. However, with adaptive rendering we will use coarser hierarchy levels only for clusters that are projected to areas on the screen that are small or outside some region of interest, and it is very unlikely that view dependencies will be noticed in such small regions.

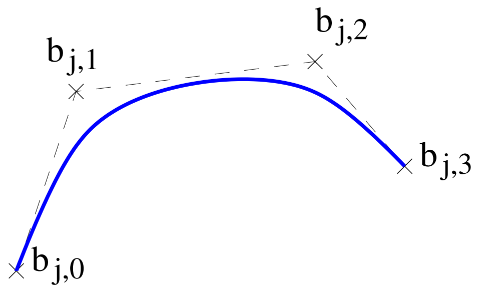
8.4 Interpolating Coordinates

In order to have a smooth visualization of time-varying data, the positions of the points have to be interpolated between the given time steps. Cubic splines are used for the evaluation of particle positions, which will help with compressing the data as well (see Subsection 8.5.2).

Cubic splines are defined by piecewise cubic Bézier segments with some additional constraints. Bézier curves can be created using the so-called Bernstein polynomials B_i^n by

$$\mathbf{x}_j(t) = \sum_{i=0}^{3} \mathbf{b}_{j,i}\, B_i^3(t) \ , \quad B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i \ .$$

The segments are stitched together so that the given points $\mathbf{x}_j$ are interpolated and the transition between the segments is smooth (C^2 continuous):



$$\mathbf{x}_j(0) = \mathbf{b}_{j,0} = \mathbf{x}_j \ , \quad \mathbf{x}_j(1) = \mathbf{b}_{j,3} = \mathbf{x}_{j+1} \ ,$$

$$\frac{d\mathbf{x}_j(1)}{dt} = \frac{d\mathbf{x}_{j+1}(0)}{dt} \ , \quad \frac{d^2\mathbf{x}_j(1)}{dt^2} = \frac{d^2\mathbf{x}_{j+1}(0)}{dt^2} \ .$$

By inserting the derivatives into these formulas we get the additional conditions

$$\mathbf{x}_j = \frac{\mathbf{b}_{j-1,2} + \mathbf{b}_{j,1}}{2} \ ,$$

$$\mathbf{b}_{j-1,1} + 2(\mathbf{b}_{j-1,2} - \mathbf{b}_{j-1,1}) = \mathbf{b}_{j,2} + 2(\mathbf{b}_{j,1} - \mathbf{b}_{j,2}) \ .$$

Together with one additional constraint at each boundary this leads to a set of equations, which have to be solved for each point of the hierarchy.

During clustering, the equations for evaluating the spline coefficients are set up and solved for each data point. For n points this process creates $3n-2$ different coefficients that need to be stored in order to evaluate the splines at arbitrary positions. As the coefficients are not independent from each other, we can convert the Bézier splines to B-Splines and store the control points $\mathbf{D}_j = \mathbf{b}_{j,2} + 2(\mathbf{b}_{j,1} - \mathbf{b}_{j,2})$ instead. The spline coefficients can be easily regenerated during rendering:

$$\mathbf{b}_{j,0} = \mathbf{b}_{j-1,3} = \frac{\mathbf{D}_{j-1} + 2\mathbf{D}_j + \mathbf{D}_{j+1}}{6} \ ,$$

$$\mathbf{b}_{j,1} = \frac{2\mathbf{D}_j + \mathbf{D}_{j+1}}{3} \ , \quad \mathbf{b}_{j,2} = \frac{\mathbf{D}_j + 2\mathbf{D}_{j+1}}{3} \ .$$

Thus, only $n+2$ control points have to be stored.

During rendering the splines are evaluated in order to reveal the positions of the particles at the rendering time step t. For this process, the 4 nearest control points are needed. When t crosses spline segment boundaries, only one control point has to be loaded as the other three control points can be reused from the last segment by means of a ring buffer.

8.5 Data Storage and Compression

Using hierarchical structures imposes higher memory requirements than storing the same data in flat arrays. A trivial implementation can easily exhaust main memory on regular workstations for large data sets, even in the static case. Memory bandwidth is limited, and traversing the hierarchy for rendering adds overhead for recursive function calls and pointer dereferencing. Additionally, with current graphics APIs there is no means to hand this process over to the GPU.

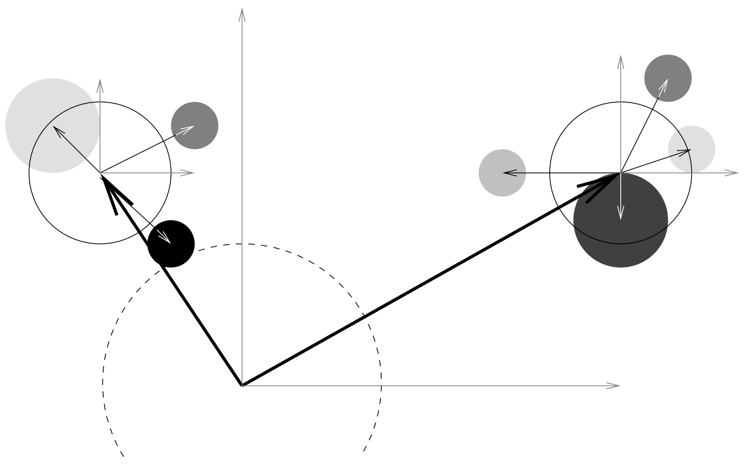


Figure 8.2: Point coordinates are scaled and quantized relatively to the position of the cluster centroid for storage.

Therefore, hierarchy structures (clusters) have been decoupled from data structures (points). The clusters contain a pointer to the next hierarchy level, a pointer to offspring point data, and the number of children. The point data itself only contains the point position, size, and color values. In principle one would like to store raw data values and use runtime classification for point size and color selection, but the cluster hierarchy itself and especially the pre-processed cluster representatives highly depend on point sizes and colors.

8.5.1 Static Data

Figure 8.3 shows the highest three levels of a typical data hierarchy. The finest level n does not contain any hierarchy information at all, thus no cluster nodes are needed. In level $n-1$ an array of point data structures contains the centroids of the cluster nodes, which are stored in an array parallel to the point data. Cluster nodes and point data are connected on the previous level. The diagram shows which points are rendered for a typical cluster node for rendering levels $n-2$, $n-1$, or n.

The decoupled data structure enables us to store point data for any given combination of rendering level and cluster node in a continuous array. This reduces the number of necessary recursive function calls and helps us with accelerating rendering by using graphics hardware. Additionally, it ensures that the data is contiguous for efficient cache usage.

Point data sets tend to get really large, and they need high positional resolution. Memory requirements can be reduced to one half or even one quarter by storing coordinates relatively to the centroids of the inspected clusters as depicted in Figure 8.2. As the necessary positional resolution is much lower for encoding relative coordinates, they can be quantized using bytes or shorts instead of floats. Bytes have been sufficiently accurate for all analyzed data sets, but cannot be used for rendering using vertex arrays on ATI's graphics hardware due to missing driver capabilities.

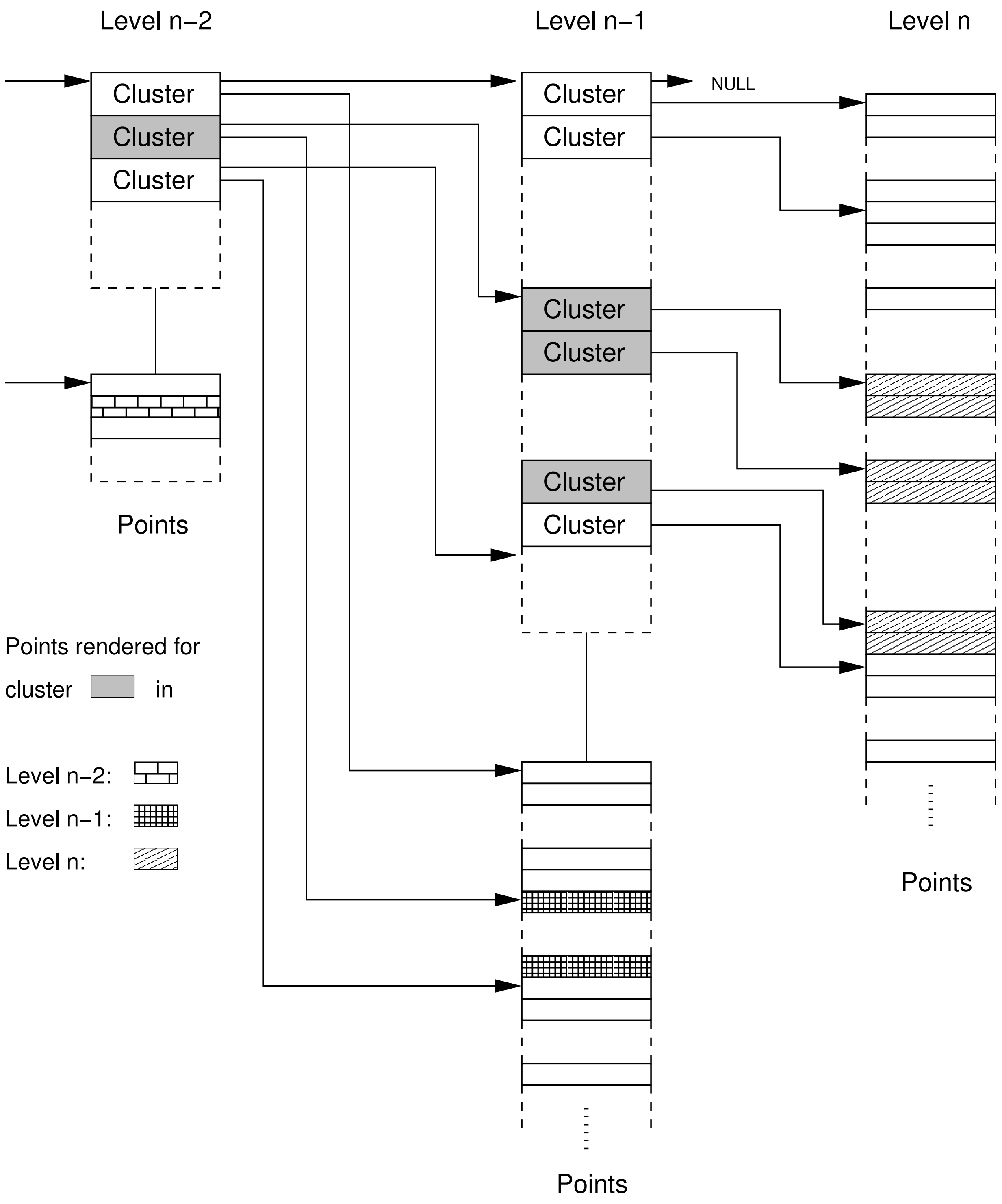


Figure 8.3: The last three levels of a typical data hierarchy. The gray clusters can be rendered at their highest hierarchy level in a single loop without recursively descending the data structure. The point data correlated to the cluster centroids is not embedded in the clusters but stored in point structures parallel to the clusters.

For a typical data set like the virgo n-body simulation (Color Plate 16) with 16.8 million points in level 6 we need 160 Mbytes for point data and 32 Mbytes for the cluster hierarchy when storing point coordinates in bytes only. Points on the highest level exhibit a positional mean error of $9.3 \cdot 10^{-6}$ and a maximum positional error of $3.9 \cdot 10^{-5}$, which is invisible compared to the typical point size of $1.25 \cdot 10^{-3}$. This point size on the highest level has an average error of 0.7%, which can be neglected as well for typical projected point diameters of one or two pixels.

8.5.2 Time-Varying Data

For non-static data we do not need to store coordinates, but the control points of the splines. Memory requirements are even higher in this case, as four control points are necessary to evaluate one spline. The splines do no longer encode particle positions directly as described in Section 8.4, but the difference vectors of the points to the parent centroids. As these vectors tend to get smaller with each level of the hierarchy, the spline coefficients can be encoded with smaller quantization factors for higher levels, and still get a high positional resolution. Figure 8.4 shows this relation. Note that during clustering different levels may have different time resolutions for the spline coefficients due to memory constraints. The control points in between can then be calculated using subdivision.

Particles in time-dependent data sets usually show high positional coherence with respect to their neighbors. In order to exploit this regularity, we are not encoding the relative vectors themselves by splines, but their differences to the previous time step. Ideally, for smooth particle flows this delta coding would create a sequence of zeros on the higher levels, with only few exceptions. This sequence can be further compressed using Lempel-Ziv with Huffman coding. This combination of techniques has been chosen because of its high decompression speed. The compression ratio, of course, depends very much on the data properties. Data sets like the diesel injection in Figure 20 that contain only very small particle movements due to computational noise in large parts of the volume can be compressed to less than 15% of the size of the uncompressed quantized data. Delta coding proved to be the most efficient part of the compression system here.

Finally, the encoded spline control points are stored into a file together with hierarchy information and index pointers for fast out-of-core access during the rendering process.

8.6 Hierarchy Traversal

During rendering the hierarchy is traversed recursively. For each cluster the system may decide to descend further down into the hierarchy, render the centroid at the current level, or skip the cluster altogether when it is not visible. The decision can be based upon some maximum screen error metrics, the distance to the viewer, or some given region of interest. These rules should be computationally cheap. As a rule of thumb, evaluating the rule should be cheaper than interpolating, transforming and rendering one point of the cluster.

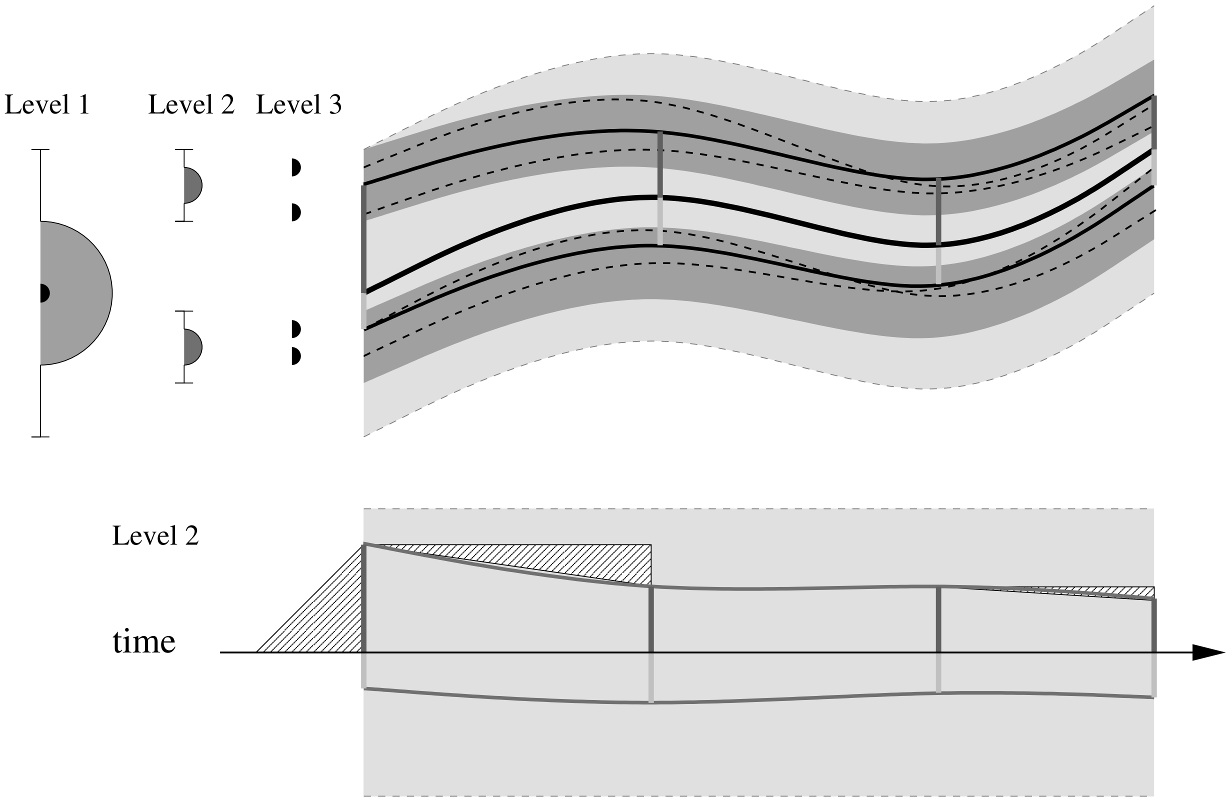


Figure 8.4: For time-varying data spline coefficients are encoded instead of particle positions. Encoding points in the hierarchy relative to their parent centroids creates smaller extension tubes for higher levels. Thus we can quantize the spline coefficients with a few bits only, and still get a high positional resolution. Additionally, all control points are delta encoded in time.

8.6.1 Static Data

For more complex rules and for accelerating the traversal process, the system may already decide on a lower level n, that it will render all offsprings of level $n+\delta_n$ (see Figure 8.5). Then the children do not have to be traversed. Even for simple adaptivity rules this has a strong effect on rendering performance. As described in Section 8.5, the point data of all children is stored linearly in memory, thus they can be addressed in a single loop, or even by a single OpenGL array rendering call. Figure. 8.6 shows a pseudo code fragment for traversing a static hierarchy.

Remember that relative coordinates are used for storing the point locations. In this context children of different clusters can only be rendered in a single loop when the base centroid and the scaling factor for the relative coordinates is the same for all considered children. We can use the coordinates of a centroid of level n for the calculation of the relative coordinates of all descendants of level $n+\delta_n$. In order to use this arrangement efficiently, δ_n has to be constant for the data set. A speedup of about 50 percent can be achieved for an average cluster size of 5 points and $\delta_n = 2$, more for larger clusters. For adaptive rendering higher δ_n are less efficient as the

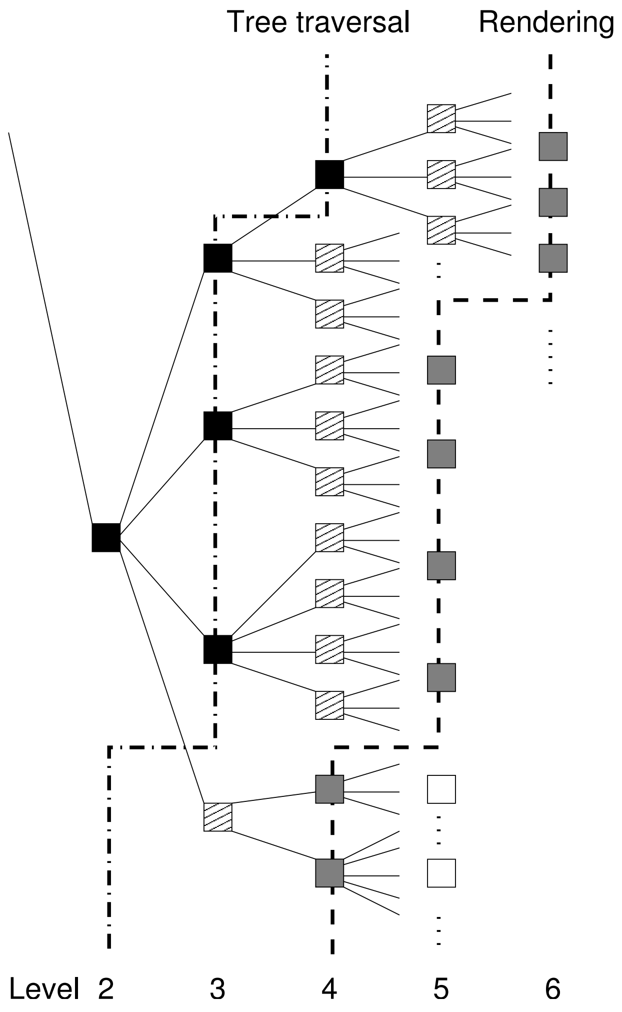


Figure 8.5: During traversal, the final rendering level should be selected at some lower level of the hierarchy for speedup reasons ($\delta_n = 2$ in this case).

traversal routine has to select the clusters to be rendered on a higher level. Another drawback is that being able to render a set of clusters in one run comes at the cost of higher discretization errors.

8.6.2 Time-Varying Data

Again, prerequisites change when we have to deal with dynamic data. $\delta_n > 1$ complicates the interpolation process, as several levels of spline coefficients have to be evaluated. For time-dependent data sets the complex interpolation easily nullifies any advantages in these cases.

Additional complexity is introduced to the rendering process, as out-of-core data has to be loaded on the fly during rendering. Figure. 8.7 shows a pseudo code fragment for traversing a hierarchy of time-varying data.

```
void render_cluster (cluster_t *c, point_t *p) {
  if (cluster_visible (c)) {                               /* trivial reject */
    if (descend_cluster (c, p, $\delta_n$)) {
      for i = 0...c→len[0]                                 /* recursion */
        render_cluster (&c→children[i], &c→points[i])
    } else {
      len = c→len[$\delta_n - 1$]
      for i = 0...$\delta_n - 2$                           /* find first point of hierarchy depth $\delta_n$ */
        c = c→children[0]                                  /* not executed for $\delta_n = 1$ */
      render_points (c→points, len)
} } }
```

Figure 8.6: Pseudo code for traversing the hierarchy in the static case.

```
void render_cluster (cluster_t *c,
                     time_t t, int level) {
  if (cluster_visible (c)) {                               // trivial reject
     if (spline_segment_changed (level, t)) {
      shift       (c, c→len)                               // shift ring buffer of spline control points
      read        (c, c→len, level, t)                     // load out-of-core data
      decompress  (c, c→len)                               // lz decompression + delta coding
    }
    interpolate (c, c→len, t)                              // spline evaluation
    if (descend_cluster (c)) {
      cluster_t *cn = c→children
      for i = 0...c→len                                    // recursion
        render_cluster (&cn[i], t, level+1)
    } else {
      render_points (c→pts, c→len)                         // render $\delta_n \equiv 1$
} } }
```

Figure 8.7: Pseudo code for traversing the hierarchy for time-dependent data. Cluster and point data are combined into a single structure in this code fragment for clarity.

8.7 Sorting

For many of the investigated data types cumulative blending is an effective way of visualizing both global and local structures in the data sets. However, with other data sets, for instance reversible Apollonian packings (Color Plate 22), the over operator is necessary to visualize the visual depth. Non-commutative blending operators require the data points to be sorted according to view distance.

The implemented hierarchy can be used to efficiently sort the cluster centroids using quicksort or bucketsort. The cluster points themselves have to be sorted before rendering as well. Bucketsort only creates an approximative sorting order, but has the advantage of lower computational complexity ($O(n)$ vs. $O(n \log n)$) and its implementation is much simpler and thus faster. The

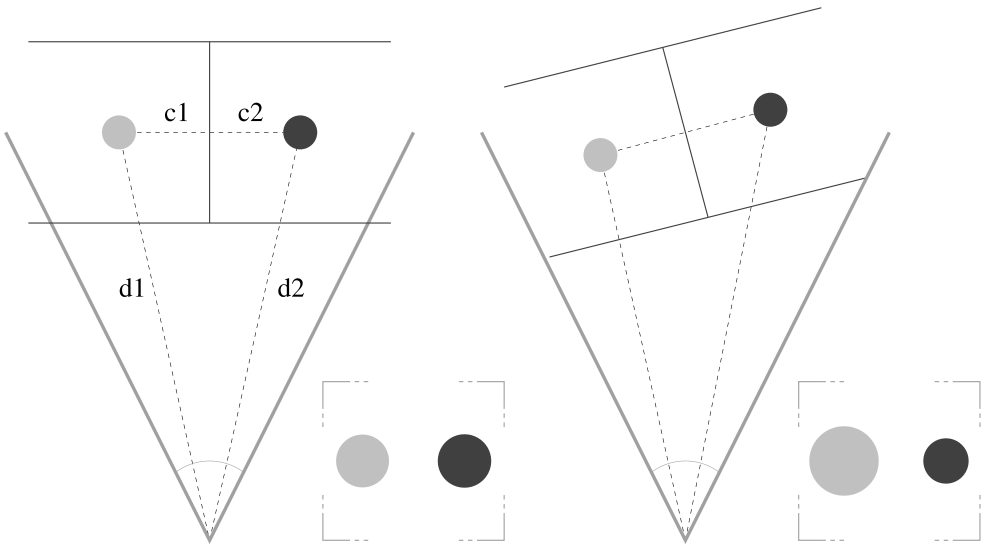


Figure 8.8: Distance sorting according to $d1$, $d2$ is equivalent to BSP sorting for $c1 = c2$. Note that the sorting order of $d1$, $d2$ changes exactly at the same time the visibility order of the two cells changes.

rendered images are almost indistinguishable when using a relatively large number of buckets. Still, sorting has a major performance impact on rendering, and it can destroy cache coherence.

Sorting the cluster centroids is equivalent to the typical BSP-tree sorting, as long as the distances of any two neighboring centroids to their common splitting plane are equal (Figure 8.8). The octree partitioning approach has this property, but its spatial adaption to the local point density is much worse than the proposed PCA split approach. Still, we have no choice but to use octrees if we really care about the correct sorting order.

With time-dependent data sets, things become even worse. Clusters do no longer subdivide the space into cells in 3D, as they may even overlap, if their children are grouped tightly together in one time step, and further apart in another. Figure 8.9 depicts an example. Even if we have a well-behaved data set that does not show this property, we can have particles crossing cluster boundaries due to spline interpolation, as shown in Figure 8.10. So far, these issues are unresolved.

Even with correctly sorted clusters, there is a chance that overlapping points are rendered in the wrong order, as can be seen in Figure 8.11. For many data sets the points can be thought to be infinitely small, in that case the points are rendered correctly. Other data sets are more sensitive to their sorting order, and require larger average cluster sizes by combining several octree levels to a single level or using large PCA cluster sizes. This helps reducing the chance of sorting errors, as the points of a single cluster are always rendered in the correct order, except for overlapping clusters in the time-dependent case.

For the static case in order to sort the cells produced by the PCA splits correctly, additional connectivity and splitting plane information is needed for MPVO [Max et al. 1990, Williams 1992]

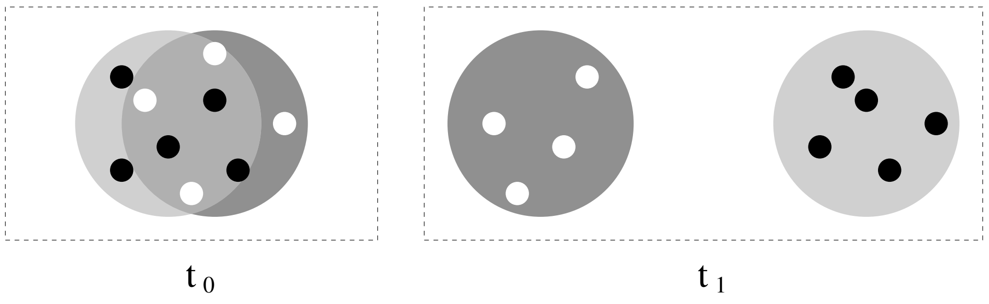


Figure 8.9: Points that are nearby in one time step may be split apart in the future, and may thus belong to different clusters. These clusters overlap in the former time step and cannot be sorted correctly.

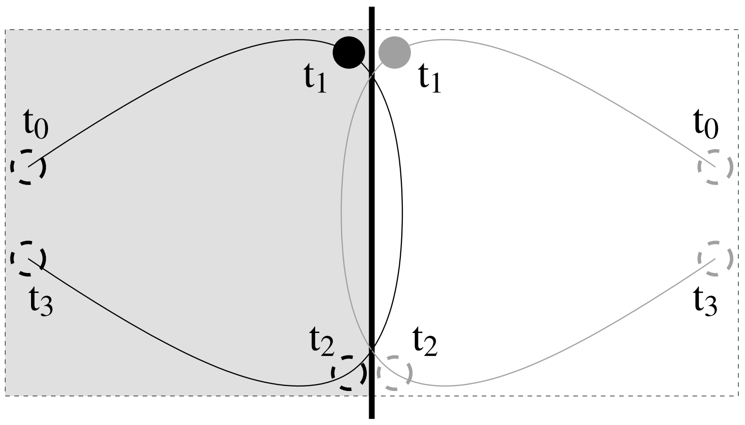


Figure 8.10: Even points of non-overlapping clusters can cross cluster boundaries between time steps due to interpolation.

or equivalent algorithms. This implies a huge additional memory overhead, and it will not help with overlapping clusters and points that drop out of their cluster boundaries. It is ongoing research how this approach can be combined with per-pixel clipping or z-test dependent blending to render correctly sorted points even for cases like in Figures 8.9 to 8.11.

8.8 Rasterization

Since using only one vertex per primitive can accelerate the rendering process significantly, the splats will usually be approximated using OpenGL anti-aliased points. Other footprints can be used by rendering point sprites without additional cost (see Color Plate 22), but they are only available on NVIDIA hardware right now. On ATI's Radeon series, fragment programs can be used in order to emulate sprites, though.

For rendering large quantities of points the generally fastest approach is to use vertex coordinates and attributes that are given by vertex arrays or display lists. However, display lists have to be

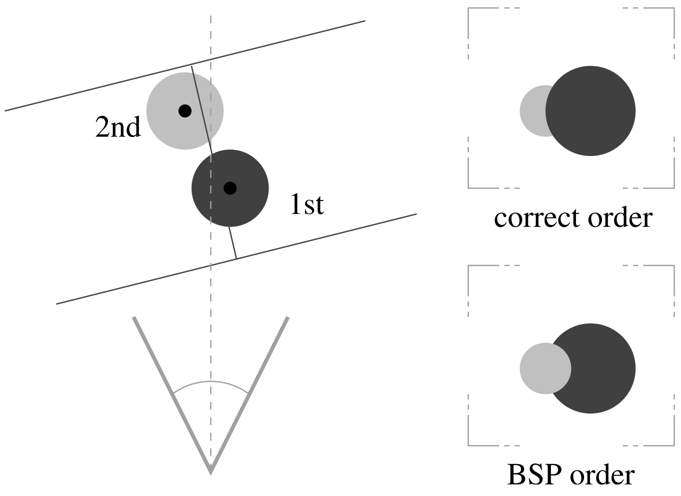


Figure 8.11: Back-to-front distance sorting according to BSP fails for non-split overlapping points. In this example the right cluster is rendered before the left one due to BSP order.

stored in precious graphics memory and are more likely to be larger in size, as the graphics card has to store additional information about contents and format. For time-dependent data, the vertex positions are created on the fly from out-of-core data, thus they cannot be stored in GPU memory. Still, it is much faster to fill render buffer caches with the calculated vertex positions and render these buffers in one go instead of sending each and every vertex with an individual operation to the graphics hardware.

When a point projects to an area with diameter $\tilde{s}$ smaller than a single pixel on the screen, its brightness has to be attenuated. The new alpha value is

$$\tilde{\alpha} = \alpha \cdot \tilde{s}^2 \quad , \tag{8.3}$$

assuming that point color is multiplied with alpha during blending. Note that attenuation increases quantization artifacts due to the limited framebuffer depth. Therefore, adaptive rendering can even improve the image quality by choosing lower levels for parts of the cluster that tend to project to very small screen areas.

For drawing points with varying sizes vertex programs can be used on programmable graphics hardware, introduced by NVIDIA with the GeForce3. Besides changing the point size on a per vertex level and adding the last contribution of the relative coordinates for static data, the vertex shader is responsible for correct alpha attenuation as indicated in 8.3, which is not possible without using vertex programs at all when we want to employ vertex arrays. Figure 8.12 shows the program parameter configuration and the actual vertex program written in Cg [Mark et al. 2003] that contains all of the above. Additionally, point size and alpha values are multiplied by two global scaling factors. It compiles into 29 program statements for both `NV_vertex_program` and `ARB_vertex_program`.

The latter extension, which is supported by ATI's Radeon 9700, enables us to evaluate the algorithm on this card as well. The previous `EXT_vertex_shader` extension did not allow to

	x	y	z	w
posoffset	-	-	-	-coord quant. offset
basepos	rel. base coords			coords scale
scale	point sprite scale	-	alpha scale	size scale
posin	rel. point coords			point size

```
void main (in      float4 posin   : POSITION,
           in      float4 colin   : COLOR0,
           out     float4 posout  : POSITION,
           out     float4 colout  : COLOR0,
           out     float4 sizeout : PSIZE,
           uniform float  atten,
           uniform float4 posoffset,
           uniform float4 basepos,
           uniform float4 scale)
{
  uniform float4x4 model = glstate.matrix.modelview[0];
  uniform float4x4 proj  = glstate.matrix.projection;
  float4  vec, homeye, eye;
  float   tmp;

  // relative coords → absolute coords
  vec.xyz   = (posin.xyz + posoffset.www)
              * basepos.www + basepos.xyz;
  vec.w     = 1.0;
  // modelview transformation + projection
  homeye    = mul (model, vec);
  posout    = mul (proj, homeye);
  eye       = homeye / homeye.w;
  // effective point size calculation
  tmp       = posin.w * basepos.w * scale.w / (atten * -eye.z);
  // clamping minimum point size to 1
  sizeout.x = scale.x * max (tmp, 1.0);
  // alpha calculation and attenuation for point sizes < 1
  tmp       = min (tmp, 1.0);
  colout    = colin;
  tmp       = colin.w * scale.z * tmp * tmp;
  // clamping minimum alpha value to keep extremely small points visible
  colout.w  = max (tmp, 4.0/256);
}
```

Figure 8.12: The vertex program written in Cg.

change the point size on a per-vertex basis. As most of the performance gain comes from this last step, we could not really benefit from the Radeon's high performance geometry engine with the old extension. Unfortunately, both anti-aliased point image quality and execution speed is clearly below all expectations with the current drivers (see Section 8.10 for a comparison).

For dynamic data, the rendering process is dominated by the evaluation of the cubic splines for finding the current point positions. As long as the points do not need to be sorted, this could be handed over to the vertex program as well by sending the control points to the GPU.

ATI's DirectX drivers are more mature than their OpenGL drivers, thus it had to be investigated whether this API would be an option. However, the so-called flexible vertex format of DirectX up to version 8 only supports vertices specified as floats. As we do not want to store the points' vertices in this format due to its memory requirements, we would have to convert them on the fly, which would make the use of vertex buffers extremely expensive as they would have to be converted by the CPU.

With the availability of vertex shaders we can now use vertex arrays to send a large part of the hierarchy to the graphics hardware. When sorting is enabled, index arrays have to be used to select the points in the correct order. These calls are highly optimized, and the CPU can already continue to select the next cluster to be rendered in parallel to the rasterization process. As pointed out in Section 8.6 we have to take care that we only send down that part of the hierarchy in one piece that is related to the same base centroid for the calculation of the relative coordinates.

8.9 Alternative Rendering Approaches

For comparison, several other techniques have been developed and integrated into the rendering framework. The different rendering backends can be selected during runtime at almost no cost.

As hierarchy traversal, interpolation, and coordinate transformation seem to be the limiting factors for the visualization of scattered data, a software rasterizer is a valid option to be considered. Most points of a low hierarchy level project to a very small area on the screen, so the rasterizer should be optimized for single pixel points. This implementation can also function as a reference for the OpenGL-based render backends, as it draws the points to a floating point framebuffer. With this feature the chance of missing contributions of very small or dim points is reduced. However, the CPU is completely responsible for vertex transformation and rasterization, thus this solution is most likely to be the least efficient of the presented methods, and modern graphics hardware like the Radeon 9700 or the GeForce FX is able to render into floating point framebuffers as well.

In contrast to regular PC workstations used by typical end users, virtual reality environments are still often based on Silicon Graphics systems. As the InfiniteReality hardware does not have a programmable graphics pipe, a regular billboard renderer has been implemented additionally. Using billboards is less efficient compared to OpenGL anti-aliased points or point sprites, as four vertices have to be calculated and sent down the pipeline for a single data point.

Note that rendering points without vertex programs is not an option, as with the regular OpenGL pipeline one can only set the current point size outside an `glBegin()` / `glEnd()` pair, which reduces the overall speed considerably due to the state changes.

8.10 Results and Comparison

The images in Color Plates 15 to 19 show visualizations of two n-body simulations carried out by the Virgo Supercomputing Consortium. All images show redshift $z = 0$ for the τCDM model. The velocities of the galaxies relative to the simulated base cube have been color coded. In Color Plates 17 to 19 one can see different levels of the first data set. Note that level 3 would usually not be used for rendering, but it is a potential level for deciding on the rendering depth, as shown in Figure 8.5.

Color Plates 20 to 22 show other data sets and rendering modes. Color Plate 23 shows another dark matter n-body and smoothed particle hydrodynamics simulation carried out by the Texas Advanced Computing Center. The visualization of a molecular dynamics simulation from the Institute of Theoretical and Applied Physics in Stuttgart can be seen in Color Plate 24. The dual shock front in the quasi crystal is clearly visible. Please note that the clearly visible aliasing in Color Plates 20 and 24 is inherent to the according data sets and not an artefact of the presented rendering technique. In most areas the data sets contain an almost regular grid and the splats are used for visualizing the grid structure and not for approximating any underlying function.

The data sets from the Virgo Supercomputing Consortium are available for several time steps, which can be rendered using the presented approach for time-dependent data. Color Plate 25 shows several time steps of one of their data sets. Color Plate 26 shows another extremely large data set, created using the LCDM model with 134 million particles per time step.

Despite the speed of modern processors, the OpenGL-accelerated version is still superior to the software approach, which employed a very crude rasterizer that renders large points in poor quality only. One major drawback of the software-based system is that the floating point framebuffer has to be sent down the AGP bus to the graphics card, though with latest AGP 8x graphics hardware and current drivers this only imposes an upper limit of 40 fps for a 1000^2 viewport on a GeForce FX, not including the time for clearing and rendering the software buffer. However, software-based rendering still seems to be one of the slowest approaches. Using a 24 bit framebuffer could accelerate this process, but then we lose the major advantage of the software solution.

Table 8.1 lists some performance measurements for the different algorithms and levels for $\delta_n = 2$, except where noted, together with the number of points, and the average projected size. It can be noticed that using billboards is rather slow, as the CPU has additional work to do for setting up four times the amount of vertices to be sent to the graphics pipe. The system used for the evaluation was a Pentium4 2800 MHz with an Intel 7205 chipset with 4 GB dual channel DDR 333 memory and a GeForce FX5800 Ultra graphics pipe on a Linux system, except where noted. The Windows XP drivers showed similar but slightly lower performance figures for the

Level	# Points	Av. pt. size	Software	Billboards	Vertexprogs	V.p.a. $\delta_n = 1$	V.p.a.	V.p.a. adapt. $\delta_n = 1$	V.p.a. adapt.	ATI V.p.a.†
6	16.8M	0.5	847	1724	495	1389	427	104	153	1490
5	3.3M	0.6	433	752	229	262	85	93	79	287
4	671K	0.9	120	161	44	50	17	17	47	57
3	123K	1.5	48	30	8.8	10	3.7	3.7	10	12
2	24K	2.9	26	7.6	2.6	2.7	1.8	1.8	2.7	2.6

$\dagger$	`ARB_vertex_program` with vertex arrays, evaluated on ATI's Radeon 9700, WindowsXP, $\delta_n = 2$
V.p.a.	Vertex program with array rendering

Table 8.1: Rendering times in ms for different rendering techniques and levels for a 500^2 viewport, $\delta_n = 2$ except where noted.

Level	# Points	Soft* $\delta_n = 1$	Soft† $\delta_n = 1$	Soft†	Soft‡	V.p.a.* $\delta_n = 1$	V.p.a.† $\delta_n = 1$	V.p.a.†	V.p.a.‡	V.p.a. adapt.‡ $\delta_n = 1$	V.p.a. adapt.‡
6	16.8M	6670	5880	4760	3570	5880	5260	3125	1890	252	690
5	3.3M	1320	1250	1040	752	1100	1040	658	379	233	356
4	671K	298	282	238	182	220	204	134	78	126	78
3	123K	85	81	71	61	44	40	26	16	28	16
2	24K	41	40	28	29	10	8.2	3.2	3.2	6.1	3.2

		$*$	quicksort
Soft	Software rendering	$\dagger$	bucketsort, # of buckets $= \max(16 \cdot \text{# of points per cluster}, 1024)$
V.p.a.	Vertex program with array rendering	$\ddagger$	bucketsort, # of buckets $= \max(\text{# of points per cluster}, 128)$

Table 8.2: Rendering times in ms for rendering sorted points with correct blending for a 500^2 viewport, $\delta_n = 2$ except where noted.

Viewport	Rendering time	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6
160^2	9.5	0	4K	51K	1.7K	0	0
400^2	75	0	2	53K	348K	224K	23K
700^2	348	0	0	418	343K	1.4M	1.8M
1000^2	769	0	0	0	82K	1.9M	6.2M

Table 8.3: Number of rendered points per level and rendering times in ms for adaptive rendering vs. viewport size, $\delta_n = 1$.

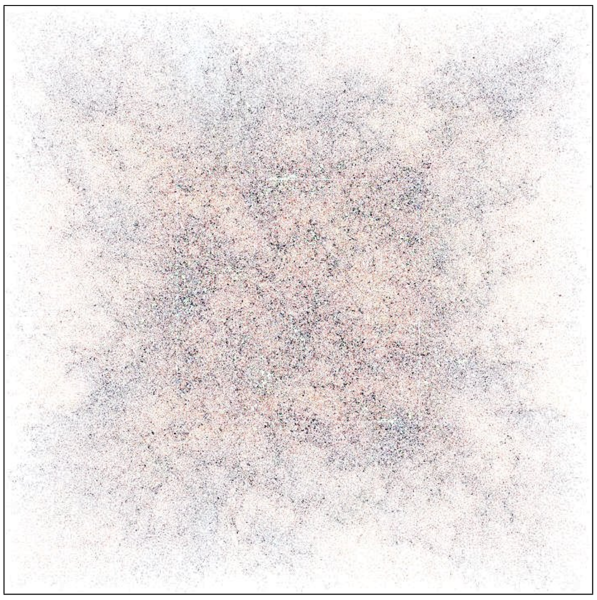


Figure 8.13: Differences of adaptive vs. full data rendering
(contrast enhanced by 400%).

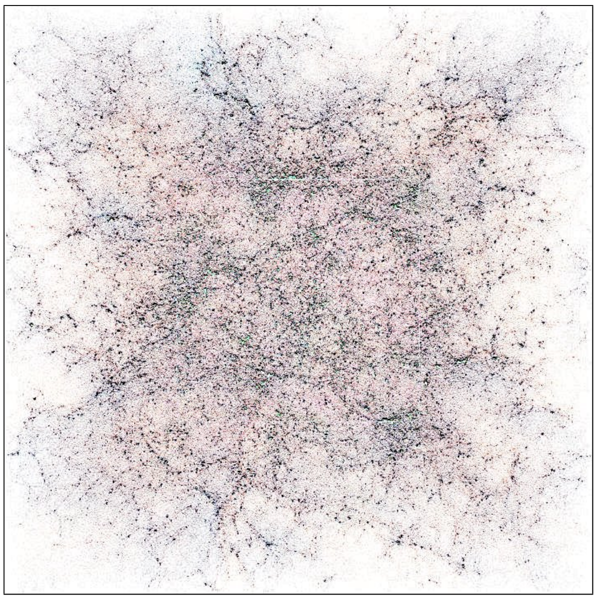


Figure 8.14: Differences of software vs. OpenGL rasterization
(contrast enhanced by 400%).

GeForce. Please note that the high performance memory setup has a much larger impact on the software rasterizer than on the vertex array renderer.

The adaptive algorithm shown in the table uses a simple adaptive scheme with vertex programs and vertex arrays, selecting the clusters that should be traversed on the CPU by the maximum screen projection size of the cluster children and the given maximum traversal depth. If the projected size exceeds 2 pixels, the cluster is traversed further, otherwise its children are rendered to screen for $\delta_n = 1$, for $\delta_n = 2$ the same criterion is applied to its grandchildren. Using these settings, there is almost no visual difference between the data set rendered in full resolution compared to the adaptive rendering. With $\delta_n = 1$ we get a finer hierarchy selection, but we also reduce the average array size that can be used for rendering, which explains the performance loss in some levels. The difference image in Figure 8.13 shows quite some changes in the visualization, however, they have the same visual impact as additional noise and do not disturb the appearance. Most of the screen space difference comes from points that happen to be rendered one pixel off to their original positions. While the human visual system is not able to notice these differences, they have a rather large impact on difference images. It can also be noticed that the quantization and floating point roundoff errors introduced by using graphics hardware for rendering (Figure 8.14) are larger than the ones created by adaptive rendering. The contrast of both difference images has been enhanced to 400 percent in order to show their properties more clearly.

Table 8.2 lists some times for combinations of different sorting and rendering techniques. Please note that the qsort-based sorting algorithm will slow down significantly for large cluster sizes, as it is $O(n \log n)$ compared to $O(n)$ for the bucketsort. The two bucketsort variants use different

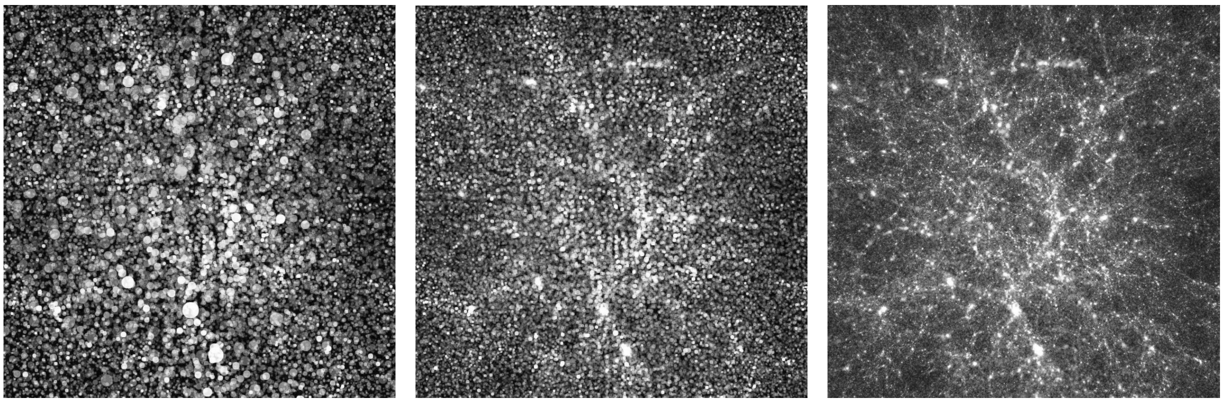


Figure 8.15: A close-up of the virgo data set, rendered at the really coarse level 3 (123K pts. = 0.74%, left), adaptively with approximately the same number of points and high potential projection error (130K pts., middle), and with all points (16.8M pts., right), respectively.

bucket sizes, trading speed for quality. The algorithm using larger buckets has almost the same visual appearance as the qsort algorithm, but exhibits some flickering during rotation on critical data sets containing large and almost overlapping points.

The cluster selection scheme has about the same performance impact on the rendering system as the flexible rendering backend (less than 2 percent each), which allows to switch the rendering technique on the fly. Please note that for large viewports like 1000^2 the effect of adaptive hierarchy traversal is not noticeable for low maximum traversal depths, as all clusters are traversed due to their large projected size.

Things change when the viewport size is reduced. Table 8.3 lists rendering times and the number of rendered points in the levels 1 to 6 for this scheme with no maximum traversal depth. We get early view frustum culling at almost no cost for the adaptive rendering algorithm, as this can be incorporated in the point projection size calculation process. However, all tables show rendering times and point numbers for viewing the full data set.

Compared to static data, the rendering of time-depended data sets is much more involved and thus slower. We still get up to 38% of the speed of the static algorithm for $\delta_n = 1$, or approximately 4.6 million points per second, which is quite remarkable for the large overhead from the hierarchical spline interpolation. However, the visualization of static data can be accelerated by deciding the rendering depth on a higher level ($\delta_n > 1$), which is not manageable with time-dependent data. Rendering sorted points using bucket sort reduces the performance to approximately 2.9 million points per second. Loading the spline parameters for a single time step from the local hard disk takes less than one third of a second. Please note that time-varying data sets need at least four times the main memory for storing the spline control points. With these constraints, the interactive visualization of extremely large data sets like the one depicted in Color Plate 26 is hardly possible on 32 bit systems for the non-static case.

Figure 8.15 shows a close up region, rendered intentionally with a very high projection size error of 14 in the left image in order to reveal the differences. The next two images show the same region rendered without adaption with approximately the same number of points and all points, respectively. Note that the projected screen size is only an approximation for the maximum screen space error, as the centroid size used for evaluating the screen space error is not directly coupled with the maximum distribution width of the children which influences the error as well.

The presented technique accelerates the visualization of scattered point data compared to rendering flat point arrays. Principal component analysis was employed for creating a hierarchy of point clusters, stored with quantized relative coordinates or spline control points in a data structure that separates cluster from point data. With this data representation visualization quality can be traded for speed with an adaptive rendering algorithm. The rendering process itself was accelerated using vertex programs on current PC graphics hardware. Finally, it is now possible to visualize data sets with tens of millions of points in the static case interactively on standard workstations.

For time-dependent data sets, a lot of work is still done by the CPU that could be off-loaded to the graphics card. For instance, the evaluation of the particle positions using cubic Bézier splines could be performed by the GPU, as long as commutative blending modes are used. Sorting the points for non-commutative blending modes has still several unresolved issues, though it works astonishingly well with most data sets.

One of the most promising — but also most challenging — extensions to the algorithm is the handling of time-varying clustering. This process will have to handle cluster transitions of single particles in a smooth way, such that popping artifacts will not occur.

Additionally, there are some issues with the overestimation of the radiant flux during rendering with cumulative blending in areas of high saturation. The system should detect these areas and reduce the brightness of the generated clusters accordingly. Alternatively, high dynamic range rendering to floating point framebuffers could be used.

An open question is how to handle multivariate data and how to change the visualized data during runtime. Storing several color values per point structure is one possibility, but this increases memory usage again. Color quantization and table lookup in the rendering step could help with regard to this aspect.

There are still several issues with the rendering of sorted points, especially for time-dependent data sets. Additionally, there might be a chance to implement bucket sorting by rendering to off-screen textures with the next generation graphics hardware, and radix sorting could be an interesting alternative as well.

Ivanova: *Ambassador.*
Do you really want to know
what's going on down there right now?
Londo: *Yes, absolutely.*
Ivanova: *Boom. Boom, boom, boom. Boom, boom.*
BOOM!
Have a nice day.

Babylon 5, A Voice in the Wilderness, Part 2

Chapter 9

Conclusion

Interactive visualization of large data sets is only possible employing efficient algorithms throughout all parts of the visualization pipeline. In this work, the opposite ends of the pipeline have been analyzed for several fundamentally different data types. It has been shown that with the use of hierarchical data structures and adaptive rendering techniques higher performance and/or quality can be achieved for filtering (e.g. using wavelets) *and* rendering (e.g. the splatting of uncorrelated data). Additionally, many physical and mathematical models already create data with a hierarchical basis (e.g. sparse grids), which cannot be converted to a non-hierarchical basis without losing detail information.

The processing power of modern graphics hardware is a resource available in all modern PCs that cannot be neglected for the acceleration of visualization algorithms. In this work it has been shown that both filtering (e.g. with linear filters) *and* rendering (e.g. of radial basis functions) can benefit from GPU-based implementations. The gap in arithmetic power between general purpose processors and graphics processors is actually widening (e.g. for nonlinear filters), with graphics-hardware-based algorithms being up to two order of magnitudes faster than standard software approaches.

9.1 Contributions to the World of Visualization

One of the most important aspects of visualization is that interactivity is often the key to understanding complex features and details of extremely large data sets. Animated views are especially useful for the analysis of three-dimensional structures, and interaction with the data sets improves comprehension even more.

In this work three basic strategies have been followed for the filtering and rendering steps of the visualization pipeline in order to improve the presentation of scientific data:

- increase of processing power
- reduction of data size
- adaptive reduction of visual quality

9.1.1 Acceleration Using Graphics Hardware

The SIMD processing power of modern graphics hardware is much higher than the performance of the main CPU. Additionally, memory paths are better optimized for regular data access, and lower parasitic capacities and data signal lengths allow for higher clock rates. In this work it has been shown that these attributes help with the acceleration of a variety of algorithms compared to software-based solutions. It has been shown that the gap between implementations on CPUs and GPUs is still widening.

Graphics-hardware-based algorithms are still prone to numerical inaccuracies of the graphics pipeline. The interpolation of sparse grids is very sensitive to accumulating errors, and more complex basis functions cannot easily be represented by textures anymore, therefore a parallelized volume renderer has been developed for this grid type to investigate its uses.

9.1.2 Reducing Memory Consumption

In order to reduce the data size, one can change the function representation, e.g. by using radial basis functions instead of unstructured grids. Additionally, by embedding the function representation in a hierarchy basis functions that have a significant contribution only have to be evaluated in the according regions of the hierarchy cells. The combination of these two solutions leads to interactive frame rates for the visualization of these gridless data types.

Another approach to reduce the amount of data is to compress it using quantization. In order to maintain the positional resolution inherent to point-based data, relative coordinates have been used to store the data in a hierarchical data structure. If memory access is extremely slow, as it is the case with out-of-core data, more advanced compression techniques can accelerate the visualization of data that can no longer be stored in main memory.

9.1.3 Trading Speed for Quality

The introduced data hierarchy is used to adapt the rendering algorithm to the screen resolution. Details that are smaller than a single pixel can be merged, which accelerates rendering and decouples rendering times from data set resolutions, as larger data sizes imply smaller details. Additionally, it is a straight-forward extension to the presented error criteria to make them sensitive to regions of interest. The error threshold could also be increased during interaction in order to create smoother animations.

Another approach to create smoother animations, or at least provide faster feedback is to reduce the resolution during interaction. For raytracing rendering times scale linearly with the number of pixels. It has been shown in this work that it is possible to select a ray distribution implicitly without inter-processor communication that both creates early previews and balances the work load evenly between the involved processors.

9.2 Data-Dependent Visualization

Data representation has to be selected carefully, depending on the type of data and the visualization objective. In Chapters 3 to 5 uniform data sets were the canonical grid type for applying filters, while in Chapter 6 sparse grids were used in order to reduce data storage size. Chapters 7 and 8 dealt with grid-less data with different properties.

In the individual chapters, the presented algorithms have been demonstrated on data sets that were perfectly fitting to the used representation. In this section the suitability of the applied data representations for different types of data is analyzed.

For the following examination I will distinguish between regular (uniform grids) and irregular (unstructured or grid-less) data, as well as between data with low and high information density. The latter term shall denote the occurrence of features, whatever they are defined to be in the current context. There is no explicit definition for information density, however, we are dealing with ambiguous problems here anyway. Very often there does not exist a single best-suited data type, but only the distinction between reasonable and unreasonable representation methods.

In the following sections data sets with characteristics of the four possible combinations of regularity and information density are used to show the properties of the used data representations. Some discrepancies in the images may come from the mapping process, as different visualization systems had to be used to create the images, but the main characteristics of the data models still predominate. Additionally, each image shows a close-up region magnified to 300% for better comparison.

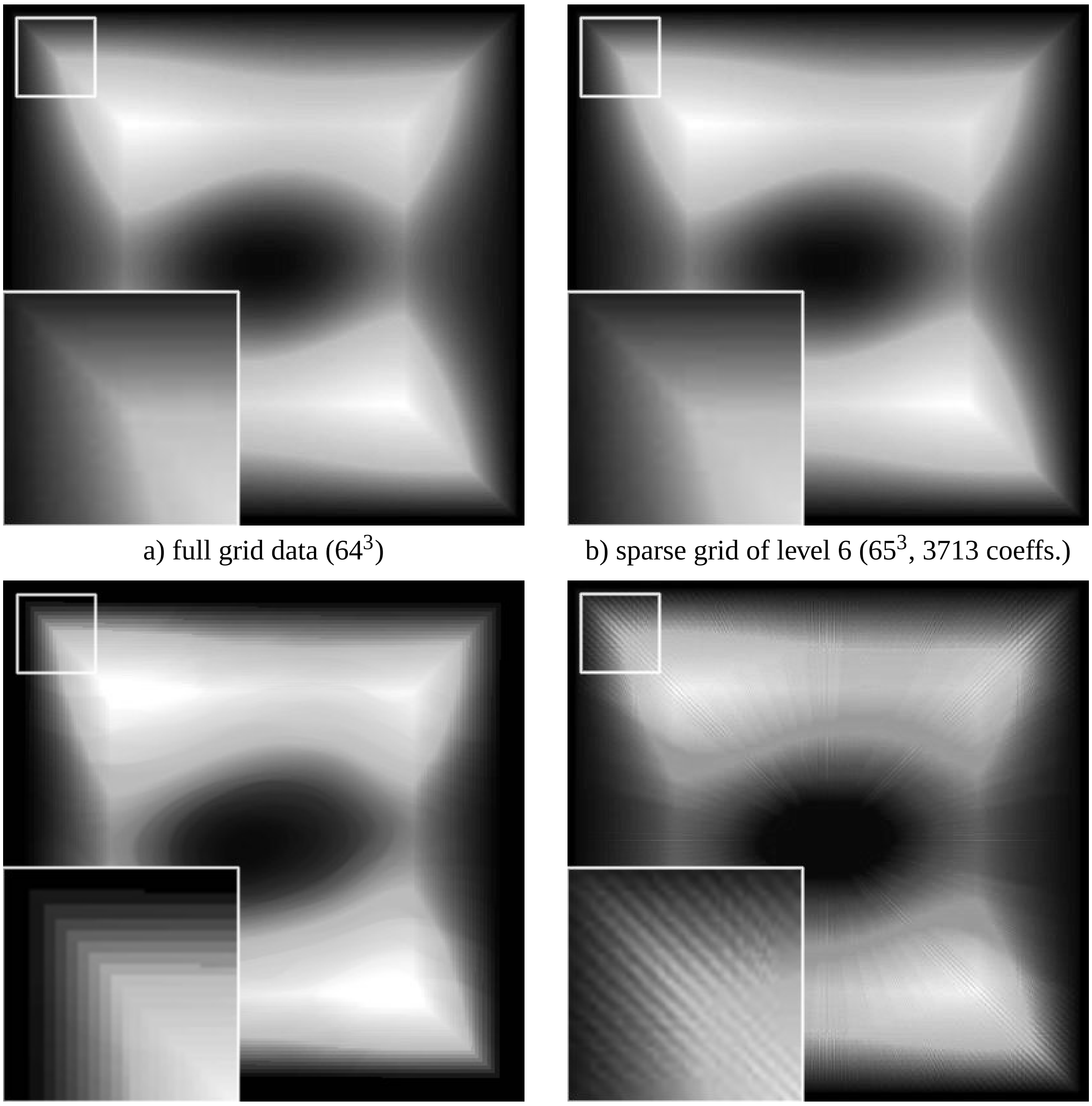
9.2.1 Regular Volume Data with Low Information Density

Figure 9.1 shows a 64^3 uniform volume data set that contains the pressure of a simulated cavity flow, heated at the left side and cooled at the right side. It is a very low frequency data set and clearly satisfies the conditions for having low information density.

Sparse grids are able to reproduce the data set with astonishing quality, though they only need to store 3,713 coefficients for a level 6 sparse grid, compared to 262,144 coefficients of the uniform grid. This type of data is also a perfect candidate for any wavelet compression approach.

Larger data sets of this type can be easily represented with sparse grids where uniform grids may fail due to memory constraints. It also shows that this type of data can be represented with a relatively small number of RBFs (260 RBFs with a maximum of 27 RBFs per octree cell), though the reconstructed data set shows some local variations not present in the original data, because RBFs cannot approximate constant or nearly constant regions very well. The image in Subfigure c) exhibits some rendering artifacts as well which result from the smaller number of slices compared to the former examples.

Using hierarchical splatting for uniform data shows a lot of aliasing artifacts, as the presented approach was used to visualize the positions of basis functions and not for the smooth interpola-



a) full grid data (64^3)

b) sparse grid of level 6 (65^3, 3713 coeffs.)

c) radial basis functions (260 RBFs)

d) hierarchical splatting (262K pts.)

Figure 9.1: Regular volume data with low information density: Pressure of a simulated cavity flow.

tion between basis functions. There exist several approaches for antialiased splatting of uniform data, see Section 8.2 for details.

Please note that rendering uniform grids is several orders of magnitude faster than rendering sparse grids, and at least an order of magnitude faster than rendering RBF-based representations and hierarchical splatting.

9.2.2 Regular Volume Data with High Information Density

Figure 9.2 shows a $128 \times 128 \times 54$ uniform volume data set created by an magnetic resonance tomography (MRT) scan of a human head. It contains a lot of high frequencies and particularly noise. As shown with a comparable data set in Figures 3.14 and 3.15, filters are an effective means of reducing the noise.

Sparse grids are not capable of representing the data set at all, as can be seen in Subfigure b). Even a sparse grid of level 9 with 49,665 coefficients that resembles a uniform grid of size 513^3 cannot create a rough approximation of the original data set. A more detailed analysis why sparse grids are obviously a bad decision for this type of data can be found in [Hopf 1998]. Wavelets should be capable of compressing this data set, but the compression ratio will be lower than for the cavity data set.

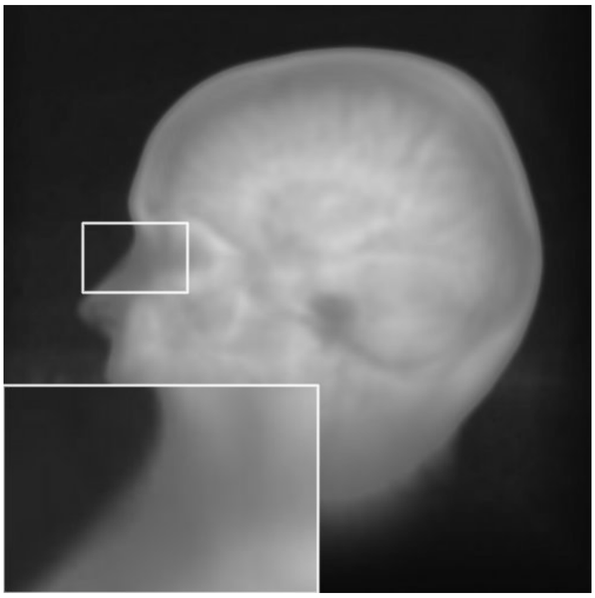
RBFs are not well suited in this case, either. The noise in the data set makes the encoding process more difficult, and the high number of details in the data can only be represented by a lot of RBF coefficients, which makes the rendering of this representation type either slow or inaccurate. Due to the extraordinary size of 328,441 RBFs for the globally encoded data set I have desisted of creating a volume rendering. The hierarchical encoding had worse approximation properties, but still needed 5,354 RBFs, which is still way too much for interactive volume rendering.

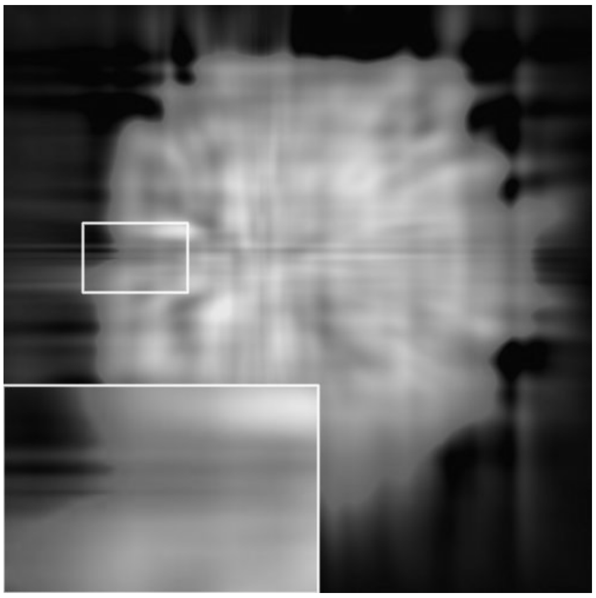
Using hierarchical splatting for uniform data shows the same type of aliasing artifacts as for the cavity data set, distracting the viewer from the real features that are actually present in the data set.

9.2.3 Irregular Volume Data with Low Information Density

In order to reveal interpolation differences for irregular data with low information density more clearly, a cutting plane has been chosen for Figure 9.3. It shows Mach numbers of the reentry shock of the X38 crew vehicle data set in the range $[0.8, 1.2]$. Subfigure a) shows the data set resampled to a full grid, which misses some of the smaller features of the RBF encoding in Subfigure c), which uses the reference data set described in Chapter 7. Note that the RBF encoding needs only 817 basis functions for storing the complete volume!

Compared to full grids, sparse grids are not capable of encoding the data set with reasonable accuracy. Even a sparse grid of level 9 shows major artifacts throughout the cutting plane, though it does a remarkably better job compared to the results achieved with the MRT data set in Figure 9.2.


a) full grid data ($128^2 \times 54$)


b) sparse grid of level 9 (513^3, 50K coeffs.)

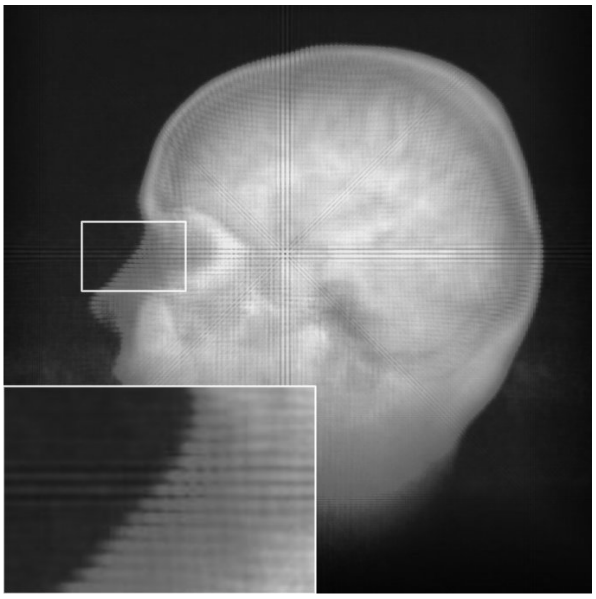

d) hierarchical splatting (884K pts.)

Figure 9.2: Regular Volume Data with High Information Density: MRT data set of a head.

Resampling RBF functions to representations valid for point-based rendering is nontrivial due to the overlapping nature of the RBF basis functions, and it is unknown so far how unstructured grids like the one this data set is derived from can be converted to point-based data sets as well. Thus no point-based rendering could be created for this data set. As the presented point rendering technique is not capable of interpolating adjacent data values, the observed aliasing effects would most likely be even more severe in this case.

9.2.4 Irregular Volume Data with High Information Density

Color Plate 27 shows the hierarchical splatting of the virgo data set, containing 16 million particles. In order to compare it with regular volume rendering, it has been resampled to a uniform grid of size 128^3, which is shown in Color Plate 28. It can be clearly seen that the resampled data shows the same features of the original data set, but at a much lower spatial resolution. With current graphics hardware, the maximum volumes size that can be used for texture-based volume rendering is clearly below 512^3 with all current PC graphics cards. A lower volume resolution has been chosen for the comparison image in order to reveal the differences more clearly.

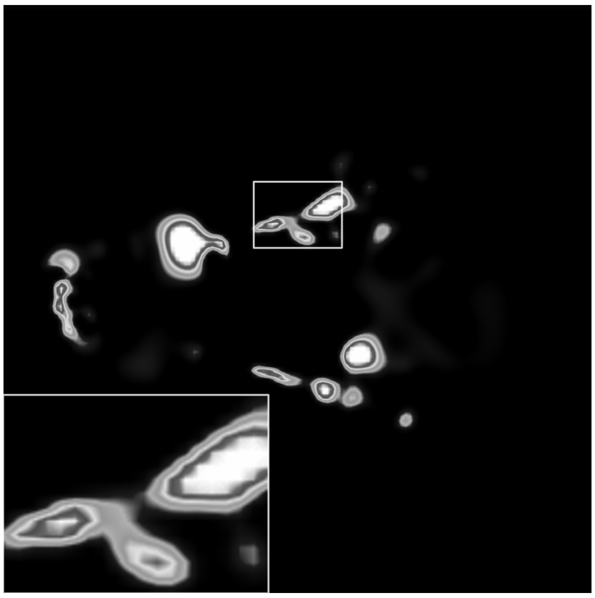
As the resampled volume is already of lower quality, and sparse grids have been proven to be a bad representation for data with a lot of features, I have desisted from creating a sparse grid approximation of the data set.

The main information in this data set is not the scalar and vectorial values that are associated with every particle, but the particle positions themselves. Radial basis functions are also not capable of representing sharp particle positions. Additionally, the optimization of the RBF representation is a computationally very expensive process, and the amount of 16 million particles is well above the manageable RBF center limit.

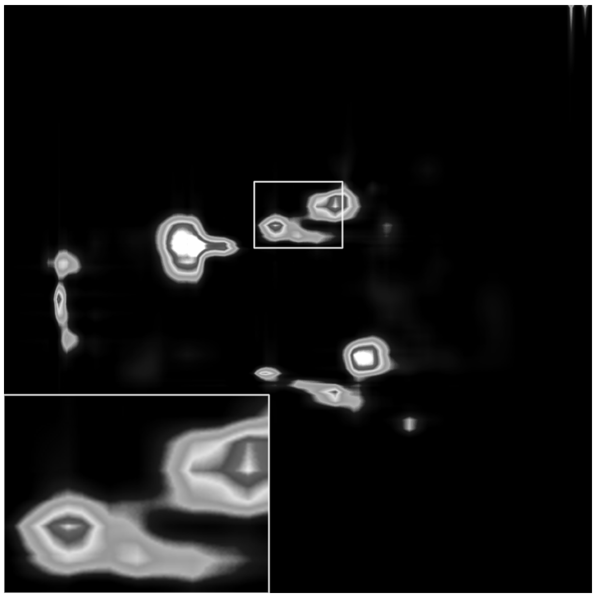
Note that hierarchical splatting is still slower than volume rendering when we want to visualize the data set at full resolution. However, keep in mind that volume rendering does not have the same spatial resolution as splatting. Additionally, we can easily trade speed for quality with the splatting approach, which is not possible to the same extent with texture-based volume rendering.

9.3 Summary

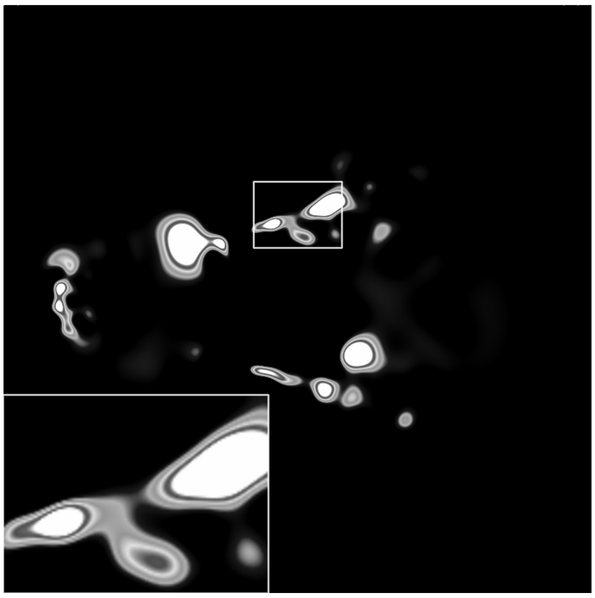
For the acceleration of filtering techniques, the problem of performing operations on three-dimensional data had to be solved first. Graphics hardware renders to two-dimensional frame-buffers only, Chapter 3 deals with the extension of tensor product type linear filters into three dimensions. In order to increase the number of possible filters, Chapter 4 shows how nonlinear filters can be implemented on GPUs. These filters play an important role in noise reduction and segmentation processes. As precision of graphics-hardware-based algorithms is not an issue for these morphological operations, they are even applicable for medical image processing.



a) full grid data (128^3)



b) sparse grid of level 9 (513^3, 50K coeffs.)



c) radial basis functions (817 RBFs)

Figure 9.3: Irregular Volume Data with Low Information Density: Shock volume of the X38 crew return vehicle.

Many modern filtering techniques are based on the wavelet analysis, thus it was the next logical step to extend the proposed acceleration techniques for wavelets in Chapter 5. Several implementations are discussed, and their accuracy is analyzed.

Wavelets are often used for the compression of images and volume data. However, the compression techniques are all based on run-length and entropy coding of quantized coefficients, which does not allow for the direct processing and filtering of compressed data. Another multiresolution analysis that can be used especially for the compact storage of large three-dimensional data sets uses sparse grids. Many numerical algorithms exist nowadays that work directly in sparse grid space. One drawback of this technique is the expensive interpolation, which effectively renders trivial visualization approaches impossible. As graphics hardware is usually not able to interpolate large data sets accurately enough, a parallelized volume renderer is presented in Chapter 6.

The algorithms presented so far only work with data on grids, represented by basis functions with implicitly given positions. For more unstructured data, grid-less approaches are more appropriate to represent small features and details of the data sets. One general approach uses radial basis functions as representatives, which can additionally benefit from hierarchical clustering approaches. The GPU-accelerated visualization of data sets encoded with this basis is presented in Chapter 7.

Not all grid-less data is meant to represent some scalar field like in the approaches before. In many cases of uncorrelated data sets the data consists of particle positions and properties, which have to be visualized directly. Therefore, a splatting approach has been presented in Chapter 8 that allows for the interactive visualization of extremely large data sets. In order to allow the visualization of time-dependent data, advanced compression and out-of-core access techniques have been employed to enable the use of data sets that do not fit into main memory any more.

Finally, the suitability of the different algorithms depending on the properties of the chosen data sets is analyzed and depicted with some examples in Section 9.2.

9.4 Future Challenges

For a wider acceptance of graphics-hardware-based algorithms the most important aspect is the visual quality, depending on the accuracy of the internal pipelines. As newer graphics hardware is able to perform calculations with floating point resolution, precision problems will impose less restrictions in the future. Accessing floating point textures and P-buffers is slower than using standard single byte textures, thus care has to be taken to only use floating point data where necessary. This could be an option for sparse grid visualization, as former graphics-hardware-based approaches were severely limited by framebuffer resolution.

With respect to the presented algorithms of this work, there are some issues with data range overflows for linear filters that have to be solved. Wavelet decomposition and reconstruction still produces artifacts at image edges when used with larger wavelet filters. Even more important —

but also much more involved — for exact hierarchical filtering techniques would be the implementation of integer-based wavelets and the lifting scheme on graphics hardware. Hierarchical splatting has still some nontrivial issues with sorting and the overestimation of radiant flux. So far it is unclear whether these problems can be fixed without larger performance penalties.

The efficiency of some of the presented GPU-based algorithms can be improved with some additional research. Wavelet decomposition has not been implemented using the programmable fragment pipeline of modern graphics hardware, in contrast to wavelet reconstruction. The huge potential performance increase comes along with higher framebuffer or texture depths, which will lead to higher accuracy as well.

So far, all algorithms have only been tested thoroughly on NVIDIA's graphics hardware due to some issues with ATI's drivers and fragment program model. The Radeon series should be an interesting alternative at least for radial basis functions and hierarchical splatting as soon as these issues are solved. Additionally, other parts of the presented splatting system could be implemented using the GPU as well, for instance spline evaluation for time-dependent data sets and approximative sorting for blending using the over operator.

The presented algorithms may have great influence on continuative research. With the development of GPU-based wavelet decomposition and reconstruction the basis for a variety of algorithms has been laid, as for instance hierarchical feature detection and tracking systems, automatic segmentation, and compression techniques. The integration of the proposed algorithms into these complex areas of research seems to be equally challenging as promising. Another delicate task will be the development of a combination technique for Whitney sparse forms.

Subject of further investigations are also splatting based approaches for the visualization of radial basis functions, together with clustering approaches for the efficient merging of basis functions. For hierarchical splatting future research will have to analyze time-dependent clustering approaches and the handling of multivariate and vectorial data.

Delenn: *Do you not have files on the Vorlons?*
Sinclair: *Absolutely. Very large files.*
There's nothing in them, of course.
*How much do **YOU** have?*
Delenn: *More than you it would seem.*
Naturally it's all classified.
Sinclair: *Naturally.*

Babylon 5, The Gathering

Appendix A

Lists

A.1 List of Figures

A.2 List of Tables

A.3 List of Color Plates

A.4 List of Abbreviations and Acronyms

AGP	Advanced Graphics Port
API	Application Programming Interface
ARB	Architecture Review Board
BSP	Binary Space Partitioning
CFD	Computational Fluid Dynamics
COTS	Commodity Of The Shelf
CPU	Central Processing Unit, Processor
DDR	Double Data Rate
DFG	German Research Foundation (Deutsche Forschungsgesellschaft)
DSP	Digital Signal Processor
GPU	Graphics Processing Unit
HLSL	High Level Shading Language
ICA	Institute of Computer Applications / Stuttgart
ICES	Institute of Computational Engineering and Sciences / UT-Austin
IP	Internet Protocol
ITAP	Institute of Theoretical and Applied Physics / Stuttgart
LCDM	(Computational model for astrophysical n-body problems)
LIC	Line Integral Convolution
MD	Molecular Dynamics
MIP	Maximum Intensity Projection
MPI	Multi-Processor Interconnect
MPVO	Mesh Polyhedra Visibility Ordering
MRT	Magnetic Resonance Tomography
PC	Personal Computer
PCA	Principal Component Analysis
PCI	Peripheral Component Interconnect
PIC-MC	Particle-In-Cell Monte Carlo
PSNR	Peak Signal Noise Ratio
PT	Projected Tetrahedra
RAM	Random Access Memory
RBF	Radial Basis Function

RGB	Red, Green, Blue
RGBA	Red, Green, Blue, Alpha
SFB	Center of Excellence (Sonderforschungsbereich)
SIMD	Single Instruction, Multiple Data
SPH	Smoothed Particle Hydrodynamics
SSE	Streaming SIMD Extensions
SVD	Singular Value Decomposition
UT	University of Texas / Austin
TAT	Theoretical Astrophysics / Tübingen
τCDM	(Computational model for astrophysical n-body problems)
TCP	Transport Control Protocol
TI	Technical Computer Science (Technische Informatik) / Tübingen
VIS	Visualization and Interactive Systems / Stuttgart
VMV	Workshop on Vision, Modeling, and Visualization

A.5 List of Hardware, Software, and Vendor Acronyms

AMD	PC processor vendor
Athlon	Processor family from AMD
ATI	PC graphics hardware vendor
BasicReality	Graphics system for SGI workstations
Cg	High-level shading language
Cosmo	High-level graphics API
CRM	Graphics hardware for SGI workstations
DirectX	Low-level graphics API
GeForce	PC graphics chip family
InfinteReality	Graphics system for SGI workstations
Intel	PC processor vendor
Inventor	High-level graphics API
JPEG	Lossy image format
LAM	MPI implementation
MXE	Graphics system for SGI workstations
NVIDIA	PC graphics hardware vendor
O_2	Graphics workstation from SGI
Octane	Graphics workstation series from SGI
Onyx	Graphics workstation series from SGI
OpenGL	Low-level graphics API
OpenSceneGraph	High-level graphics API
OpenSG	High-level graphics API
Optimizer	High-level graphics API
P4	Pentium 4, processor from Intel
PIII	Pentium 3, processor from Intel

Pentium	Processor family from Intel
Performer	High-level graphics API
PowerVR	PC graphics hardware vendor and graphics chip family
Radeon	PC graphics chip family
SGI	Silicon Graphics Incorporated
TiVOR	Volume rendering system
V8	Graphics system for SGI workstations
V12	Graphics system for SGI workstations

Ivanova: (to Sinclair)
I know, I know. It's a Russian thing.
When we're about to do something stupid,
we like to catalog the full extent of our stupidity
for future reference.

Babylon 5, A Voice in the Wilderness, Part 1

Appendix B

Bibliography

[Ahmed 1994] W. Ahmed. 1994. *Fast Orthogonal Search For Training Radial Basis Function Neural Network*. Master's thesis, University of Maine.

[Alexa et al. 2001] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. 2001. Point Set Surfaces. In *Proc. Visualization '01,* pp. 21–28, IEEE.

[Botsch et al. 2002] M. Botsch, A. Wiratanaya, and L. Kobbelt. 2002. Efficient High Quality Rendering of Point Sampled Geometry. In *Proc. Workshop on Rendering '02*, pp. 53–64, EG.

[Bro-Nielson 1996] M. Bro-Nielson. 1996. *Medical Image Registration and Surgery Simulation*. PhD thesis, IMM, Technical University of Denmark.

[Bungartz and Dornseifer 1998] H.-J. Bungartz and T. Dornseifer. 1998. Sparse Grids: Recent Developments for Elliptic Partial Differential Equations. In *Multigrid Methods V*, vol. 3 of *Lecture Notes in Computational Science and Engineering*. pp. 45–70, Springer.

[Bungartz 1992] H.-J. Bungartz. 1992. *Dünne Gitter und deren Anwendung bei der adaptiven Lösung der dreidimensionalen Poisson-Gleichung*. PhD thesis, Technische Universität München, Germany.

[Cabral et al. 1994] B. Cabral, N. Cam, and J. Foran. 1994. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proc. Symposium on Volume Visualization '94*, pp. 91–98, ACM.

[Calderbank et al. 1998] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo. 1998. Wavelet Transforms that Map Integers to Integers. *Appl. Comput. Harmon. Anal. 5:3,* pp. 332–369.

[Carr et al. 2001] J. Carr, R. Beatson, J. Cherrie, T. Mitchell, W. Fright, B. McCallum, and T. Evans. 2001. Reconstruction and Representation of 3D Objects With Radial Basis Functions. In *Proc. SIGGRAPH '01*, pp. 67–76, ACM.

[Chui 1992] C. K. Chui. 1992. *An Introduction to Wavelets*. Academic Press, Inc., San Diego.

[Co et al. 2003] C. S. Co, B. Heckel, H. Hagen, B. Hamann, and K. I. Joy. 2003. Hierarchical Clustering for Unstructured Volumetric Scalar Fields. In *Proc. Visualization '03*, pp. 325–332, IEEE.

[Cox 1996] D. J. Cox. 1996. Cosmic Voyage: Scientific Visualization for IMAX film. In *SIGGRAPH '96 Visual Proceedings*, p. 129 and 147, ACM.

[Daubechies and Sweldens 1998] I. Daubechies and W. Sweldens. 1998. Factoring Wavelet Transforms into Lifting Steps. *J. Fourier Anal. Appl. 4:3*, pp. 245–267.

[Daubechies 1992] I. Daubechies. 1992. *Ten Lectures on Wavelets*. No. 61 in CBMS-NSF Series in Applied Mathematics. SIAM, Philadelphia.

[Diewald et al. 2001] U. Diewald, T. Preusser, M. Rumpf, and R. Strzodka. 2001. Diffusion Models and their Accelerated Solution in Computer Vision Applications. *Acta Mathematica Universitatis Comenianae AMUC:70*, pp. 15–31.

[Engel et al. 2001] K. Engel, M. Kraus, and T. Ertl. 2001. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Proc. Workshop on Graphics Hardware '01*, pp. 9–16.

[Farin et al. 2002] G. Farin, J. Hoschek, and M. S. Kim. 2002. *Handbook of Computer Aided Geometric Design*. Academic Press.

[Foley et al. 1993] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. 1993. *Computer Graphics — Principles and Practice*, second ed. Addison-Wesley.

[Fornefett et al. 1999] M. Fornefett, K. Rohr, and H. S. Stiehl. 1999. Elastic Registration of Medical Images Using Radial Basis Functions with Compact Support. In *Proc. Computer Vision and Pattern Recognition '99*, pp. 402–409.

[Ghosh and Nag 2001] J. Ghosh and A. Nag. 2001. An Overview of Radial Basis Function Networks. In *Radial Basis Function Networks 2*. pp. 1–36, Physica.

[Goshtasby 2000] A. A. Goshtasby. 2000. Grouping and parameterizing irregularly spaced points for curve fitting. *Transactions on Graphics 19:3*, pp. 185–203, ACM.

[Gradinaru and Hiptmair 2003] V. Gradinaru and R. Hiptmair. 2003. Mixed Finite Elements on Sparse Grids. *Numer. Math. 93:3 (January)*, pp. 471–495.

[Griebel et al. 1992a] M. Griebel, W. Huber, U. Rüde, and T. Störtkuhl. 1992. The combination technique for parallel sparse-grid-preconditioning or -solution of PDE's on multiprocessor machines and workstation networks. In *Proc. Joint International Conference on Vector and Parallel Processing '92*, pp. 217–228, Springer, CONPAR/VAPP.

[Griebel et al. 1992b] M. Griebel, M. Schneider, and C. Zenger. 1992. A combination technique for the solution of sparse grid problems. In *Proc. International Symposium on Iterative Methods in Linear Algebra '92*, pp. 263–281, Elsevier, IMACS.

[Griebel 1998] M. Griebel. 1998. Adaptive Sparse Frid Multilevel Methods for Elliptic PDEs Based on Finite Differences. *Computing 61:2*, pp. 151–179, Springer.

[Grzeszczuk et al. 1998] R. Grzeszczuk, C. Henn, and R. Yagel. 1998. Advanced Geometric Techniques for Ray Casting Volumes. In *SIGGRAPH '98 Course Notes 4*, ACM.

[Guthe et al. 2002] S. Guthe, M. Wand, J. Gonser, and W. Straßer. 2002. Interactive Rendering of Large Volume Data Sets. In *Proc. Visualization '02*, pp. 53–60, IEEE.

[Haber et al. 2001] J. Haber, F. Zeilfelder, O. Davydov, and H. P. Seidel. 2001. Smooth approximation and rendering of large scattered data sets. In *Proc. Visualization '01*, pp. 341–348, IEEE.

[Hadwiger et al. 2001] M. Hadwiger, T. Theußl, H. Hauser, and E. Gröller. 2001. Hardware-Accelerated High-Quality Filtering on PC Hardware. In *Proc. Vison, Modeling and Visualization VMV '01*, pp. 155–162, IEEE.

[Hadwiger et al. 2002] M. Hadwiger, I. Viola, T. Theußl, and H. Hauser. 2002. Fast and Flexible High-Quality Texture Filtering With Tiled High-Resolution Filters. In *Proc. Vison, Modeling and Visualization '02*, pp. 155–162, infix, IEEE.

[Hadwiger et al. 2003] M. Hadwiger, H. Hauser, and T. Möller. 2003. Quality Issues of Hardware-Accelerated High-Quality Filtering on PC Graphics

Hardware. In *Proc. Visualization in Scientific Computing WSCG '03*, pp. 213–220, EG.

[Hanrahan and Lawson 1990] P. Hanrahan and J. Lawson. 1990. A Language for Shading and Lighting Calculations. In *Proc. SIGGRAPH '90*, pp. 289–298, ACM.

[Hege et al. 1993] H.-C. Hege, T. Höllerer, and S. D. 1993. Volume Rendering. Tech. Rep. 93-7, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Germany.

[Heußer and Rumpf 1998] N. Heußer and M. Rumpf. 1998. Efficient Visualization of Data on Sparse Grids. In *Proc. Visualization and Mathematics '98*, pp. 31–44, Springer.

[Hoehne and Hanson 1992] K.-H. Hoehne and W. A. Hanson. 1992. Interactive 3D Segmentation of MRI and CT Volumes using Morphological Operations. *Journal of Computer Assisted Tomography 16:2*, pp. 285–294.

[Hopf and Ertl 1999a] M. Hopf and T. Ertl. 1999. Accelerating 3D Convolution using Graphics Hardware. In *Proc. Visualization '99*, pp. 471–474, IEEE.

[Hopf and Ertl 1999b] M. Hopf and T. Ertl. 1999. Hardware Based Wavelet Transformations. In *Proc. Vision, Modeling, and Visualization '99*, pp. 317–328, infix, IEEE.

[Hopf and Ertl 2000a] M. Hopf and T. Ertl. 2000. Accelerating Morphological Analysis with Graphics Hardware. In *Proc. Vision, Modeling, and Visualization '00*, pp. 337–345, infix, IEEE.

[Hopf and Ertl 2000b] M. Hopf and T. Ertl. 2000. Hardware Accelerated Wavelet Transformations. In *Proc. VisSym '00*, pp. 93–103, EG/IEEE.

[Hopf and Ertl 2001] M. Hopf and T. Ertl. 2001. Parallelizing Sparse Grid Volume Visualization with Implicit Preview and Load Balancing. Tech. Rep. 8/2001, Visualization and Interactive Systems Group, University of Stuttgart, Germany.

[Hopf and Ertl 2003] M. Hopf and T. Ertl. 2003. Hierarchical Splatting of Scattered Data. In *Proc. Visualization '03*, pp. 443–440, IEEE.

[Hopf 1998] M. Hopf. 1998. *Volumenvisualisierung auf dünnen Gittern*. Diplomarbeit, Computer Graphics Group, University of Erlangen-Nürnberg, Germany.

[Huang et al. 2000] J. Huang, N. Shareef, K. Mueller, and R. Crawfis. 2000. Fast-Splats: Optimized Splatting on Rectilinear Grids. In *Proc. Visualization '00*, pp. 219–226, IEEE.

[Iserhardt-Bauer et al. 2001] S. Iserhardt-Bauer, P. Hastreiter, T. Ertl, K. Eberhardt, and B. Tomandl. 2001. Case Study: Medical Web Service For the Automatic 3D Documentation For Neuroradiological Diagnosis. In *Proc. Visualization '01*, pp. 425–428, IEEE.

[Jang et al. 2002] J. Jang, W. Ribarsky, C. D. Shaw, and N. Faust. 2002. View-Dependent Multiresolution Splatting of Non-Uniform Data. In *Proc. VisSym '02*, pp. 125–132, EG/IEEE.

[Jang et al. 2004] Y. Jang, M. Weiler, M. Hopf, J. Huang, D. S. Ebert, K. P. Gaither, and T. Ertl. 2004. Interactively Visualizing Procedurally Encoded Scalar Fields. In *Proc. VisSym '04*, EG/IEEE. Accepted for publication.

[Jenkins et al. 1998] A. Jenkins, C. S. Frenk, F. R. Pearce, P. A. Thomas, J. M. Colberg, S. D. M. White, H. M. P. Couchman, J. A. Peacock, G. P. Efstathiou, and A. H. Nelson. 1998. Evolution of Structure in Cold Dark Matter Universes. *ApJ 499:1*, pp. 20–40.

[Jolliffe 1986] I. T. Jolliffe. 1986. *Principal Component Analysis*. Springer, New York.

[JTC1/SC29 2002] JTC1/SC29. 2002. JPEG 2000 image coding system — Part 1: Core coding system. Tech. Rep. 15444-1:2000, ISO/IEC.

[Kähler et al. 2002] R. Kähler, D. Cox, R. Patterson, S. Levy, H.-C. Hege, and T. Abel. 2002. Rendering The First Star In The Universe - A Case Study. In *Proc. Visualization 02*, pp. 537–540, IEEE.

[Kindlmann and Weinstein 1999] G. Kindlmann and D. Weinstein. 1999. Hue-Balls and Lit-Tensors for Direct Volume Rendering of Diffusion Tensor Fields. In *Proc. Visualization '99*, pp. 183–189, IEEE.

[Kniss et al. 2001] J. Kniss, G. Kindlmann, and C. Hansen. 2001. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. In *Proc. Visualization '01*, pp. 255–262, IEEE.

[Lacroute and Levoy 1994] P. Lacroute and M. Levoy. 1994. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Proc. SIGGRAPH '94*, pp. 451–457, ACM.

[LaMar et al. 1999] E. LaMar, B. Hamann, and K. Joy. 1999. Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *Proc. Visualization '99*, pp. 355–361, IEEE.

[Laur and Hanrahan 1991] D. Laur and P. Hanrahan. 1991. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. In *Proc. SIGGRAPH '91*, pp. 285–288, ACM.

[Levoy 1988] M. Levoy. 1988. Display of Surfaces from Volume Data. *Computer Graphics & Applications 8:3 (May)*, pp. 29–37, ACM.

[Lippert et al. 1997] L. Lippert, M. H. Gross, and C. Kurmann. 1997. Compression Domain Volume Rendering for Distributed Environments. In *Proc. EUROGRAPHICS '97*, pp. C95–C107, EG.

[Louis et al. 1994] A. K. Louis, P. Maass, and A. Rieder. 1994. *Wavelets*. B. G. Teubner, Stuttgart.

[Lürig and Ertl 1998] C. Lürig and T. Ertl. 1998. Hierarchical Volume Analysis and Visualization Based on Morphological Operators. In *Proc. Visualization '98*, pp. 335–341, IEEE.

[Lürig et al. 1997a] C. Lürig, R. Grosso, and T. Ertl. 1997. Combining Wavelet Transform and Graph Theory for Feature Extraction and Visualization. In *Proc. Visualization in Scientific Computing WSCG '97*, pp. 137–144, EG.

[Lürig et al. 1997b] C. Lürig, R. Grosso, and T. Ertl. 1997. Implicit Adaptive Volume Ray-Casting. In *GraphiCon '97*, pp. 114–120.

[Luttenberger 2003] M. Luttenberger. 2003. *Visualisierung zeitabhängiger gitterloser Daten*. Studienarbeit, Visualization and Interactive Systems Group, University of Stuttgart, Germany.

[Magallón et al. 2001] M. Magallón, M. Hopf, and T. Ertl. 2001. Parallel Volume Rendering using PC Graphics Hardware. In *Proc. Pacific Graphics '01*, pp. 384–389, IEEE.

[Malzbender 1993] T. Malzbender. 1993. Fourier-Volume-Rendering. *Transactions on Graphics 12:3 (July)*, pp. 233–250, ACM.

[Mark et al. 2003] W. R. Mark, S. Glanville, and K. Akeley. 2003. Cg: A System for Programming Graphics Hardware in a C-like Language. In *Proc. SIGGRAPH '03*, pp. 896–907, ACM.

[Max et al. 1990] N. Max, P. Hanrahan, and R. Crawfis. 1990. Area And Volume Coherence For Efficient Visualization Of 3D Scalar Functions. *Computer Graphics 24:5*, pp. 27–33, ACM.

[Meredith and Ma 2001] J. Meredith and K.-L. Ma. 2001. Multiresolution View-Dependend Splat Based Volume Rendering of Large Irregular Data. In *Proc. Symposium on Parallel and Large-Data Visualization and Graphics '01*, pp. 92–99, IEEE.

[Monaghan 1992] J. J. Monaghan. 1992. Smoothed Particle Hydrodynamics. *Ann. Rev. Astron. Astrophys. 30*, pp. 543–574.

[Morse et al. 2001] B. S. Morse, T. S. Yoo, P. Rheingans, D. T. Chen, and K. R. Subramanian. 2001. Interpolating Implicit Surfaces From Scattered Surface Data Using Compactly Supported Radial Basis Functions. In *Proc. Shape Modeling International*, pp. 89–98, IEEE.

[Mueller et al. 1999] K. Mueller, T. Möller, and R. Crawfis. 1999. Splatting without the Blur. In *Proc. Visualization '99*, pp. 363–370, IEEE.

[Nadeau et al. 2000] D. R. Nadeau, J. D. Genetti, S. Napear, B. Pailthorpe, C. Emmart, E. Wesselak, and D. Davidson. 2000. Visualizing Stars and Emission Nebulas. *Computer Graphics Forum 20:1*, pp. 27–33, EG.

[Nédélec 1980] J. Nédélec. 1980. Mixed Finite Elements in R^3. *Numer. Math. 35*, pp. 315–341.

[Orr 1996] M. J. L. Orr. 1996. Introduction to Radial Basis Function Networks. Tech. rep., Center for Cognitive Science, University of Edinburgh, April.

[Park et al. 2002] S. Park, C. Bajaj, and V. Siddavanahalli. 2002. Case Study: Interactive Rendering of Adaptive Mesh Refinement Data. In *Proc. Visualization '02*, pp. 521–524, IEEE.

[Pauly et al. 2002] M. Pauly, M. Gross, and L. Kobbelt. 2002. Efficient Simplification of Point-Sampled Surfaces. In *Proc. Visualization '02*, pp. 163–170, IEEE.

[Pfister et al. 2000] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. 2000. Surfels: Surface Elements as Rendering Primitives. In *Proc. SIGGRAPH '00*, pp. 335–342, ACM.

[Pugh 1990] W. Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM 33:6*, pp. 668–676.

[Rau and Straßer 1995] R. Rau and W. Straßer. 1995. Direct Volume Rendering of Irregular Samples. In *Proc. Visualization in Scientific Computing WSCG ’95*, pp. 72–80, EG.

[Rezk-Salama et al. 1999] C. Rezk-Salama, P. Hastreiter, C. Teitzel, and T. Ertl. 1999. Interactive Exploration of Volume Line Integral Convolution Based on 3D–Texture Mapping. In *Proc. Visualization ’99*, pp. 233–240, IEEE.

[Rezk-Salama et al. 2000] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. 2000. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *Proc. Workshop on Graphics Hardware ’00*, pp. 109–118,147, EG/ACM SIGGRAPH.

[Rezk-Salama 2002] C. Rezk-Salama. 2002. *Volume Rendering Techniques for General Purpose Graphics Hardware*. PhD thesis, Computer Graphics Group, University of Erlangen-Nürnberg, Germany.

[Roettger et al. 2000] S. Roettger, M. Kraus, and T. Ertl. 2000. Hardware-Accelerated Volume and Isosurface Rendering Based On Cell-Projection. In *Proc. Visualization ’00*, pp. 109–116, IEEE.

[Rumpf and Strzodka 2001] M. Rumpf and R. Strzodka. 2001. Nonlinear Diffusion in Graphics Hardware. In *Proc. Visualization ’01*, pp. 75–84, IEEE.

[Rusinkiewicz and Levoy 2000] S. Rusinkiewicz and M. Levoy. 2000. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Proc. SIGGRAPH ’00*, pp. 343–352, ACM.

[Savchenko et al. 1995] V. V. Savchenko, A. A. Pasko, O. G. Okunev, and T. L. Kunii. 1995. Function Representation of Solids Reconstructed from Scattered Surface Points and Contours. *Computer Graphics Forum 14:4*, pp. 181–188, ACM.

[Sayood 2000] K. Sayood. 2000. *Introduction to Data Compression*. Morgan Kaufmann.

[Schiekofer 1998] T. Schiekofer. 1998. *Die Methode der Finiten Differenzen auf Dünnen Gittern zur adaptiven Multilevel-Lösung partieller Differentialgleichungen*. PhD thesis, Universität Bonn, Institut für Angewandte Mathematik.

[Schröder and Stoll 1992] P. Schröder and G. Stoll. 1992. Data Parallel Volume Rendering as Line Drawing. In *Proc. Workshop on Volume Visualization ’92*, pp. 25–32, ACM SIGGRAPH.

[SGI 1996] SGI. 1996. *OpenGL on Silicon Graphics Systems*. Silicon Graphics Inc., Mountain View, California.

[Shirley and Tuchman 1991] P. Shirley and A. Tuchman. 1991. A Polygonal Approximation to Direct Volume Rendering. In *Proc. Workshop on Volume Visualization '91*, pp. 63–70, ACM SIGGRAPH.

[Shreiner et al. 2004] D. Shreiner, M. Woo, J. Neider, and T. Davis. 2004. *OpenGL Programming Guide*, 1.4 ed. Addison-Wesley.

[Sommer et al. 1998] O. Sommer, A. Dietz, R. Westermann, and T. Ertl. 1998. An Interactive Visualization and Navigation Tool for Medical Volume Data. In *Proc. Visualization in Scientific Computing WSCG '98*, pp. 362–371, EG.

[Stein et al. 1994] C. M. Stein, B. G. Becker, and N. L. Max. 1994. Sorting and Hardware Assisted Rendering for Volume Visualization. In *Proc. Symposium on Volume Visualization '94*, pp. 83–89, ACM.

[Steinberg 1986] S. Steinberg. 1986. Grayscale Morphology. In *Proc. Computer Vision, Graphics and Image Processing '86*, pp. 333–355.

[Strang and Nguyen 1996] G. Strang and T. Nguyen. 1996. *Wavelets and Filter Banks*. Wellesley-Cambridge, Wellesley, Massachusetts.

[Strzodka and Rumpf 2001] R. Strzodka and M. Rumpf. 2001. Level Set Segmentation in Graphics Hardware. In *Proc. International Conference on Image Processing ICIP '01*, pp. 1103–1106.

[Strzodka 2002] R. Strzodka. 2002. Virtual 16 Bit Precise Operations on RGBA8 Textures. In *Proc. Vison, Modeling and Visualization '02*, pp. 171–178, 521, infix, IEEE.

[Swan et al. 1997] J. E. Swan, K. Mueller, T. Möller, N. Shareef, R. Crawfis, and R. Yagel. 1997. An Anti-Aliasing Technique for Splatting. In *Proc. Visualization '97*, pp. 197–204, IEEE.

[Sweldens 1997] W. Sweldens. 1997. The Lifting Scheme: A Construction of Second Generation Wavelets. *SIAM J. Math. Anal. 29:2*, pp. 511–546.

[Teitzel and Hopf 2000] C. Teitzel and M. Hopf. 2000. Visualization of Vector Fields. In *Principles of 3D Image Analysis and Synthesis*. pp. 270–278, Kluwer Academic Publishers.

[Teitzel et al. 1998a] C. Teitzel, R. Grosso, and T. Ertl. 1998. Particle Tracing on Sparse Grids. In *Proc. Visualization in Scientific Computing WSCG '98*, pp. 132–142, EG.

[Teitzel et al. 1998b] C. Teitzel, M. Hopf, R. Grosso, and T. Ertl. 1998. Volume Ray Casting on Sparse Grids. Tech. Rep. 5/1998, Universität Erlangen-Nürnberg, Lehrstuhl für Graphische Datenverarbeitung (IMMD IX), Erlangen, March.

[Teitzel et al. 1999] C. Teitzel, M. Hopf, and T. Ertl. 1999. Volume Visualization on Sparse Grids. *Computing and Visualization in Science 2:1*, pp. 47–59, Springer.

[Teitzel et al. 2000] C. Teitzel, M. Hopf, and T. Ertl. 2000. Scientific Visualization on Sparse Grids. In *Proc. Scientific Visualization - Dagstuhl '97*, pp. 284–295, IEEE.

[Totsuka and Levoy 1993] T. Totsuka and M. Levoy. 1993. Frequency Domain Volume Rendering. *Computer Graphics 27:4 (August)*, pp. 271–78, ACM.

[Turk and O'Brien 1999] G. Turk and J. O'Brien. 1999. Shape Transformation Using Variational Implicit Functions. In *Proc. SIGGRAPH '99*, pp. 335–342, ACM.

[Turk and O'Brien 2002] G. Turk and J. F. O'Brien. 2002. Modelling with implicit surfaces that interpolate. *Transactions on Graphics 21:4*, pp. 855–873, ACM.

[Viola et al. 2003] I. Viola, A. Kanitsar, and E. Gröller. 2003. Hardware-Based Nonlinear Filtering and Segmentation using High-Level Shading Languages. In *Proc. Visualization '03*, pp. 309–316, IEEE.

[Wand et al. 2001] M. Wand, M. Fischer, I. Peter, F. Meyer auf der Heide, and W. Straßer. 2001. The Randomized z-Buffer Algorithm. In *Proc. SIGGRAPH '01*, pp. 361–370, ACM.

[Weiler and Ertl 2001] M. Weiler and T. Ertl. 2001. Hardware-Software-Balanced Resampling for the Interactive Visualization of Unstructured Grids. In *Proc. Visualization '01*, pp. 199–206, IEEE.

[Weiler et al. 2000a] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. 2000. Level-Of-Detail Volume Rendering via 3D Textures. In *Proc. Volume Visualization and Graphics Symposium '00*, pp. 7–13, IEEE.

[Weiler et al. 2000b] M. Weiler, R. Westermann, C. Hansen, K. Zimmerman, and T. Ertl. 2000. Level-Of-Detail Volume Rendering via 3D Textures. In *Proc. Volume Visualization and Graphics Sympsium '00*, pp. 7–13, IEEE.

[Weiler et al. 2002]	M. Weiler, M. Kraus, and T. Ertl. 2002. Hardware-Based View-Independent Cell Projection. In *Proc. Symposium on Volume Visualization '02*, pp. 13–22, IEEE.
[Weiler et al. 2003a]	M. Weiler, M. Kraus, M. Merz, and T. Ertl. 2003. Hardware-Based View-Independent Cell Projection. *Transactions on Visualization and Computer Graphics 9:2*, pp. 163–175, IEEE.
[Weiler et al. 2003b]	M. Weiler, M. Kraus, M. Merz, and T. Ertl. 2003. Hardware-Based Ray Casting for Tetrahedral Meshes. In *Proc. Visualization '03*, pp. 333–340, IEEE.
[Weiskopf and Hopf 2002]	D. Weiskopf and M. Hopf. 2002. *Direct3D Shaderx: Vertex and Pixel Shader Tips and Tricks*. ch. Real-Time Simulation and Rendering of Particle Flows, pp. 414–425, Wordworth.
[Weiskopf et al. 2001]	D. Weiskopf, M. Hopf, and T. Ertl. 2001. Hardware-Accelerated Visualization of Time-Varying 2D and 3D Vector Fields by Texture Advection via Programmable Per-Pixel Operations. In *Proc. Vision, Modeling, and Visualization '01*, pp. 439–446, infix, IEEE.
[Weiskopf et al. 2002]	D. Weiskopf, G. Erlebacher, M. Hopf, and T. Ertl. 2002. Hardware-Accelerated Lagrangian-Eulerian Texture Advection for 2D Flow Visualization. In *Proc. Vision, Modeling, and Visualization '02*, pp. 74–84, 516, infix, IEEE.
[Westermann and Ertl 1997]	R. Westermann and T. Ertl. 1997. A Multiscale Approach to Integrated Volume Segmentation and Rendering. In *Proc. EUROGRAPHICS '97*, pp. 96–107, EG.
[Westermann and Ertl 1998]	R. Westermann and T. Ertl. 1998. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proc. SIGGRAPH '98*, pp. 169–179, ACM.
[Westover 1990]	L. Westover. 1990. Footprint Evaluation for Volume Rendering. In *Proc. SIGGRAPH '90*, pp. 367–376, ACM.
[Whitney 1957]	H. Whitney. 1957. *Geometric Integration Theory*. Pinceton Univ. Press, Princeton.
[Wickerhauser 1994]	M. V. Wickerhauser. 1994. *Adapted Wavelet Analysis from Theory to Software*. IEEE, New York.
[Williams 1992]	P. L. Williams. 1992. Visibility Ordering Meshed Polyhedra. *Transactions on Graphics 11:2*, pp. 103–126, ACM.

[Wilson et al. 2002] B. Wilson, K.-L. Ma, and P. S. Mc Cormick. 2002. A Hardware-Assisted Hybrid Rendering Technique for Interactive Volume Visualization. In *Proc. Symposium on Volume Visualization and Graphics '02*, pp. 123–130, IEEE.

[Wylie et al. 2002] B. Wylie, M. Kenneth, L. A. Fisk, and P. Crossno. 2002. Tetrahedral Projection using Vertex Shaders. In *Proc. Symposium on Volume Visualization '02*, pp. 7–12, IEEE.

[Zenger 1990] C. Zenger. 1990. Sparse grids. In *Proc. Seminar on Parallel Algorithms for Partial Differential Equations GAMM '90*.

[Zhang et al. 2002] Y. Zhang, R. Rohling, and D. K. Pai. 2002. Direct surface extraction from 3D freehand ultrasound images. In *Proc. Visualization '02*, pp. 45–52, IEEE.

[Zhuang and Haralick 1986] X. Zhuang and R. Haralick. 1986. Morphological Structuring Element Decomposition. In *Proc. Computer Vision, Graphics and Image Processing '86*, pp. 370–382.

[Zwicker et al. 2001a] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. 2001. EWA Volume Splatting. In *Proc. Visualization '01*, pp. 29–36, IEEE.

[Zwicker et al. 2001b] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. 2001. Surface Splatting. In *Proc. SIGGRAPH '01*, pp. 371–378, ACM.

Sheridan: *Why? What's inside there?*
Kosh: *One moment of perfect beauty.*

Babylon 5, There All the Honor Lies

Appendix C

Color Plates

Chapter 5: Hierarchical Filters

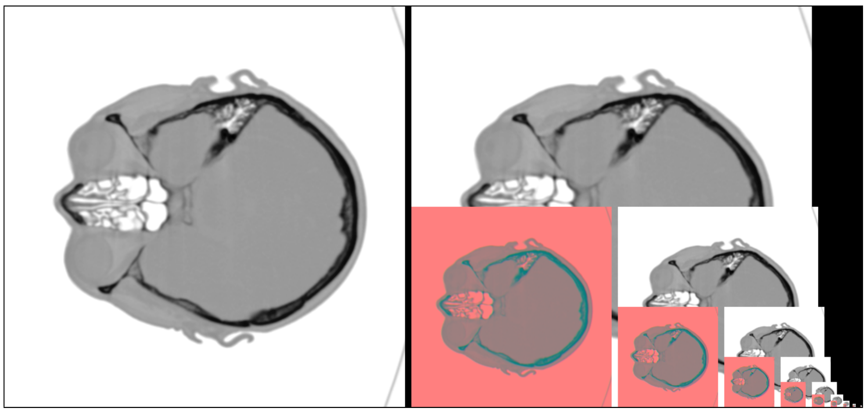


Plate 1: The head data set decomposed with Haar wavelets with algorithm 2.

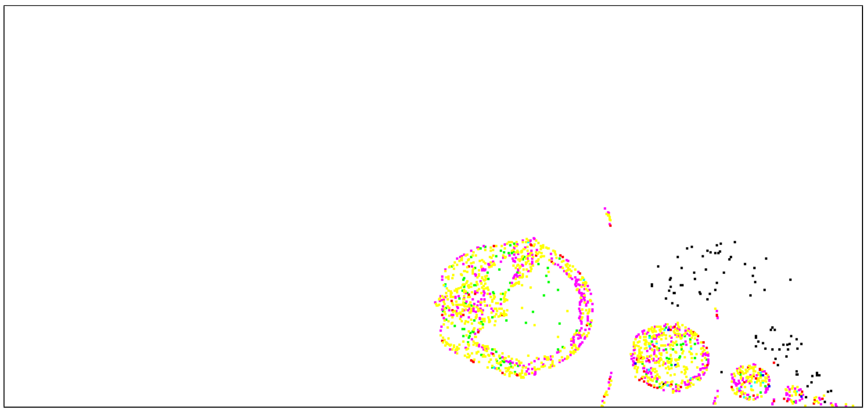
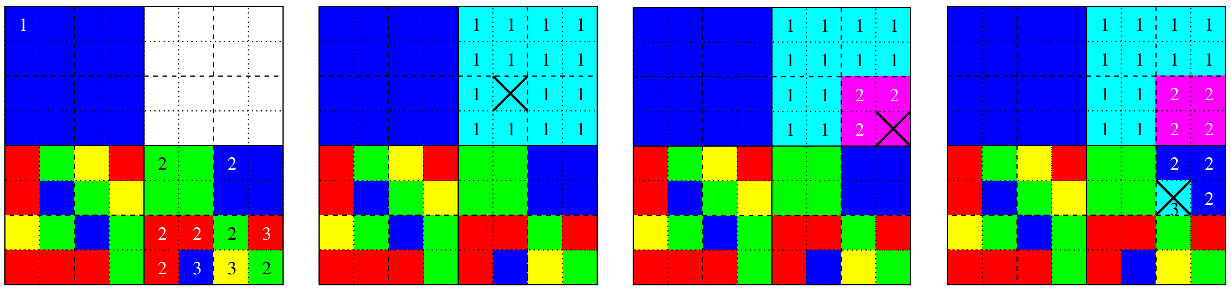


Plate 2: Least significant bit differences of the decomposition between an SGI MXE and an Intergraph Wildcat with algorithm 2.

Chapter 6: Parallelized Sparse Grids



Hierarchy Level ——— 1 – – – 2 ······· 3

Plate 3: Adaptive image update shown for the the ray at the marked (x) pixel position on the front end workstation. Note that pixel levels are not displayed for all pixels.

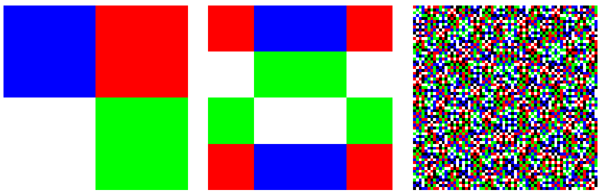


Plate 4: Ray distribution for trivial index selection functions, different processors are encoded with different colors. 64^2 rays, $\tilde{\imath}(t) = const$ on 4 processors, $\tilde{\imath}(t) = t$ on 4 and 5 processors.

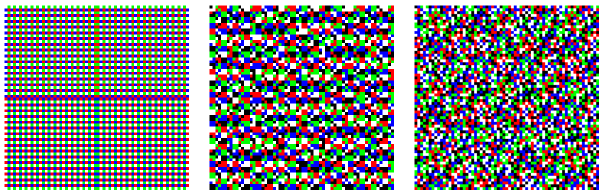


Plate 5: Ray distribution for the heuristic index selection function, 64^2 rays on 4 and 5 processors, 65^2 rays on 5 processors.

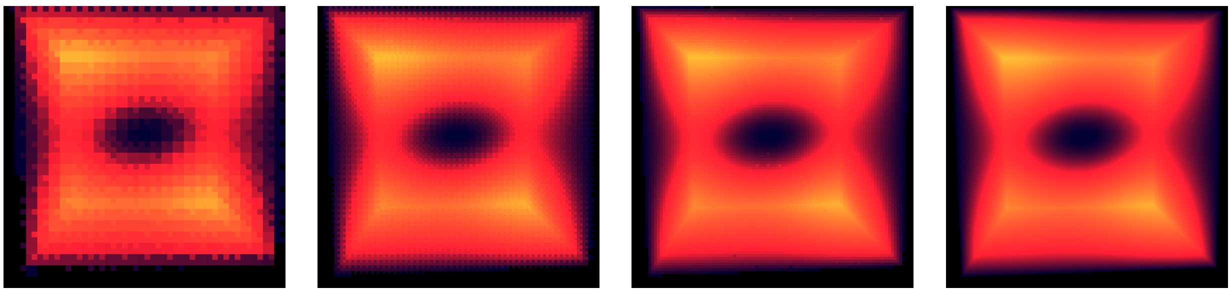


Plate 6: Preview images after 0.625%, 3.125%, 6.25%, and 100% of 160000 rays have been rendered.

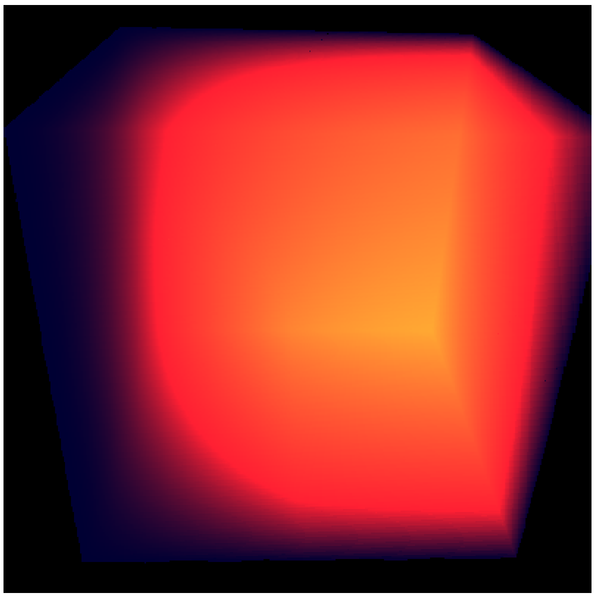


Plate 7: A sparse grid data set of level 12 (corresponding to a full grid of size 2047^3) rendered with an X-ray shading method.

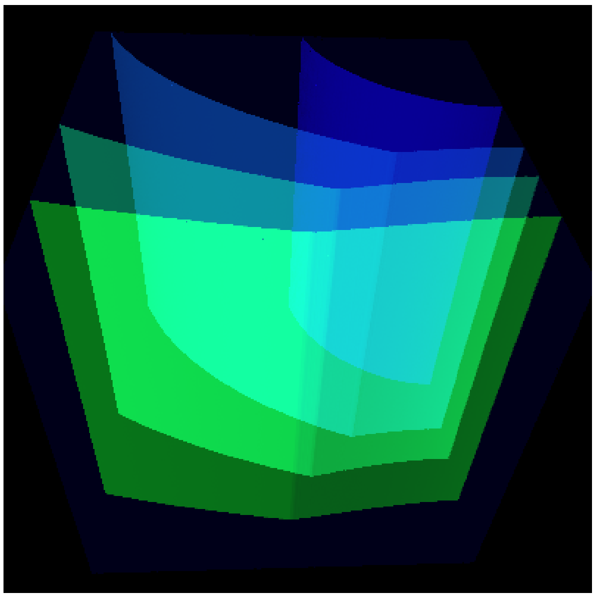


Plate 8: The same data set rendered with multiple semitransparent shaded isosurfaces.

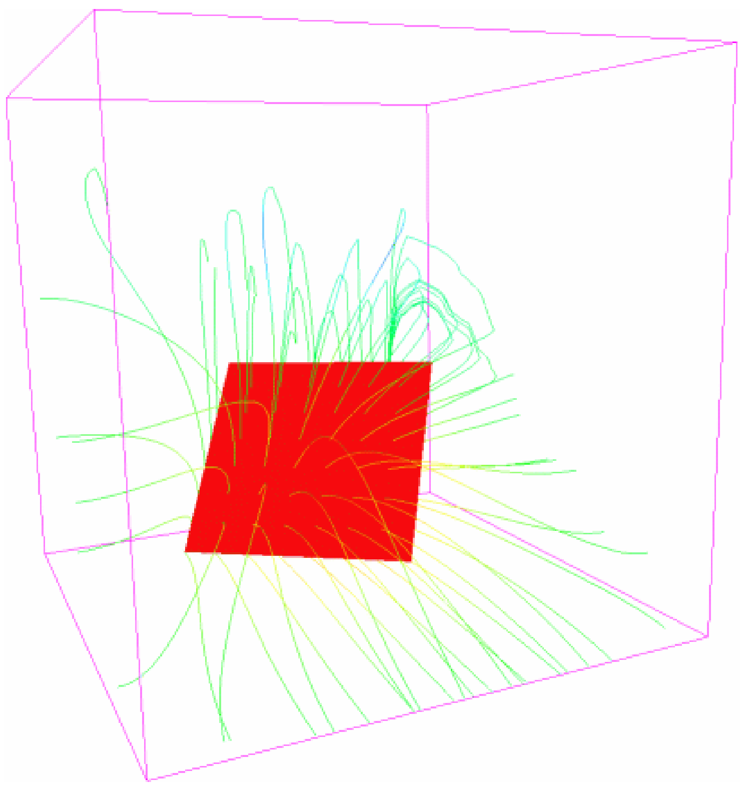


Plate 9: Some field lines of a sparse grid 1-form of level 12.

Chapter 7: Hierarchical Radial Basis Functions

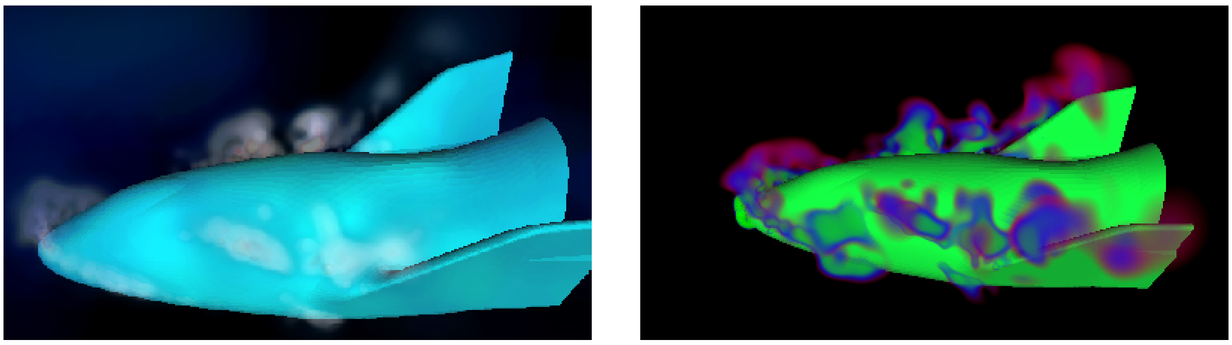


Plate 10: Volume renderings of the unstructured X38 shock CFD data. The original data set contained 1,943,383 tetrahedras and was encoded with 2,932 RBFs.

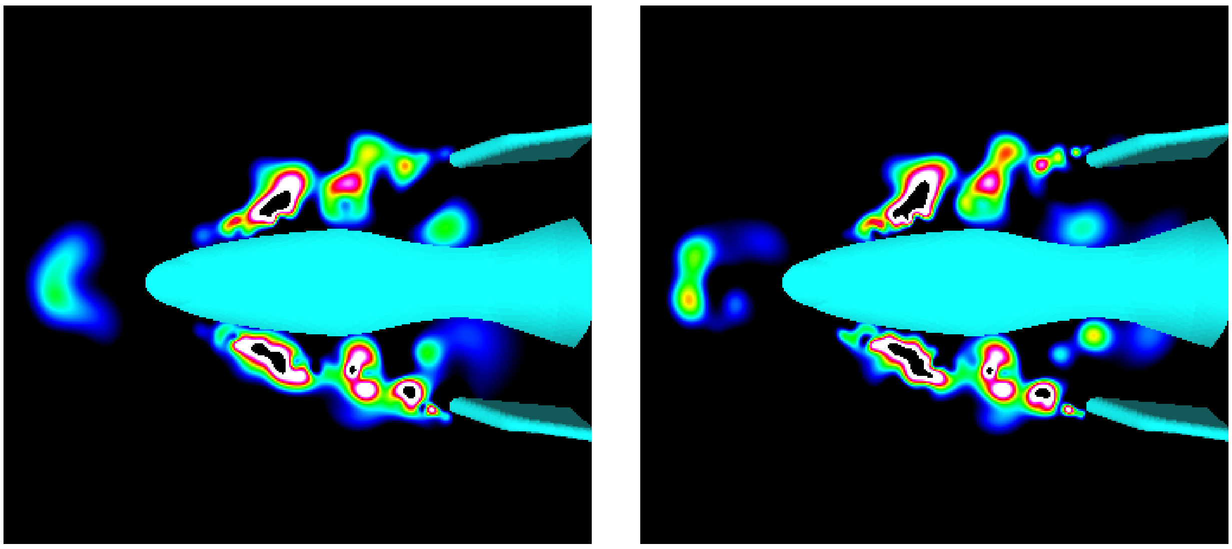


Plate 11: Slice planes at $z = 40$ in the X 38 shock data set. 855 RBFs are used for reconstruction in the left image, 1,147 RBFs for the right image.

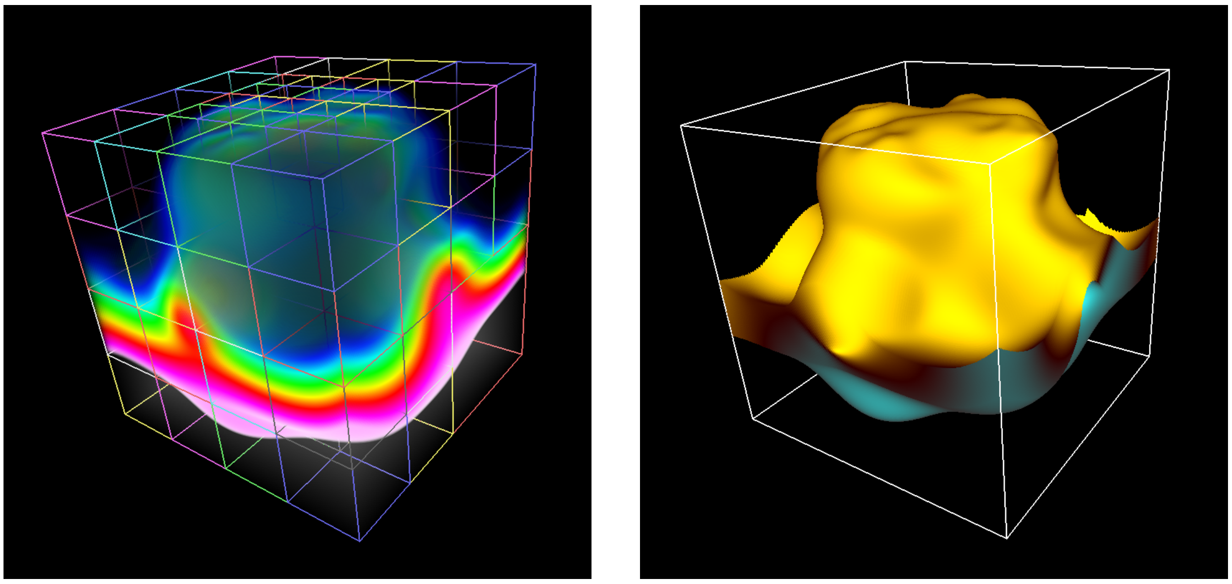


Plate 12: Volume and isosurface rendering of temperature generated from a natural convection simulation (48,000 tetrahedras). The data set is encoded with 435 RBFs in 85 cells, with a maximum of 100 RBFs per cell.

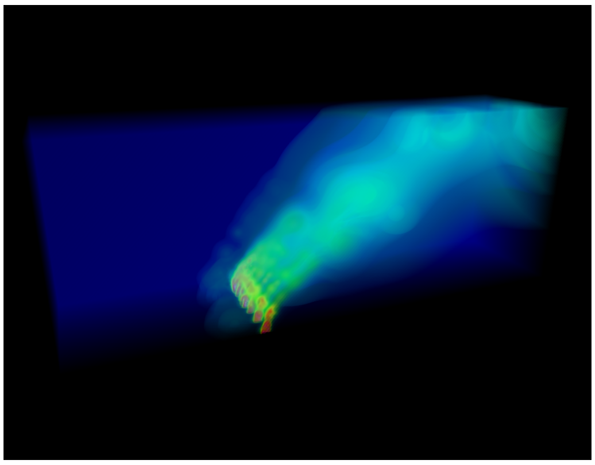


Plate 13: Volume rendering of the blunt fin data set. It has been encoded using 695 RBFs in 238 cells, with a maximum of 60 RBFs per cell.

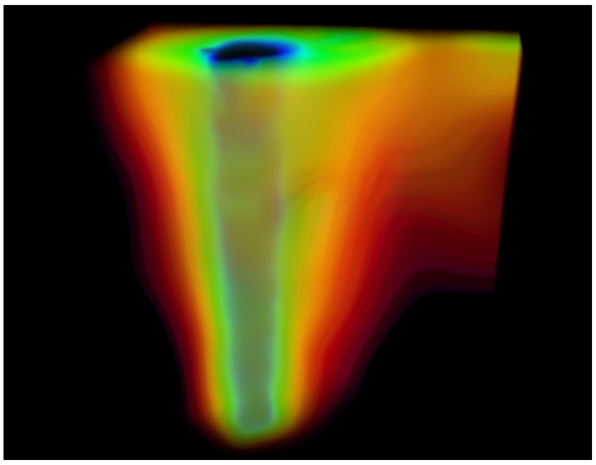


Plate 14: Volume rendering of water pressure for an injection well. The 156,642 tetrahedra data set is encoded using 222 RBFs in 49 cells.

Chapter 8: Splatting of Uncorrelated Data

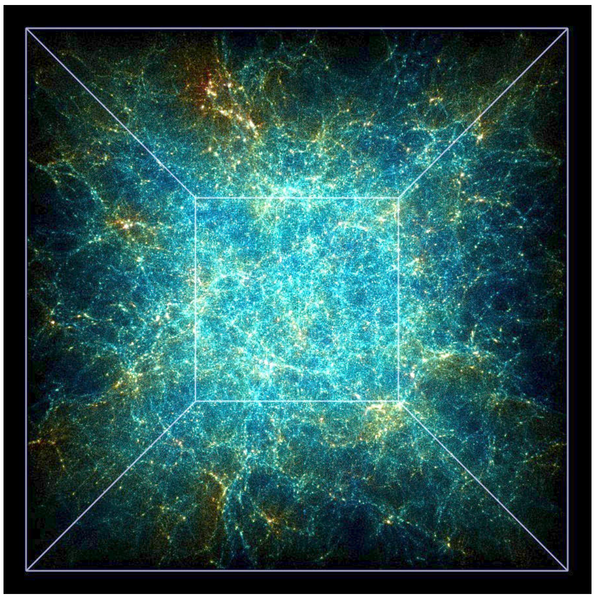


Plate 15: A total view of one of the virgo n-body simulations rendered adaptively with a maximum screen space error of 2 pixels, indistinguishable from the complete data set with 16.8 million particles.

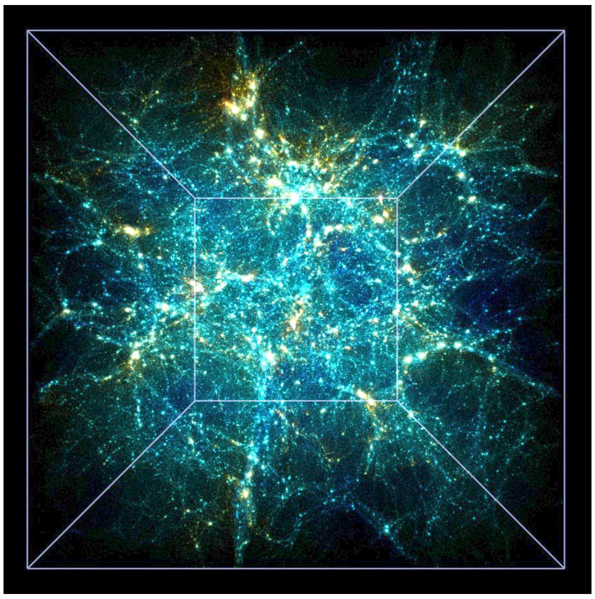


Plate 16: A total view of one of the close up simulations with 16.8 million particles as well.

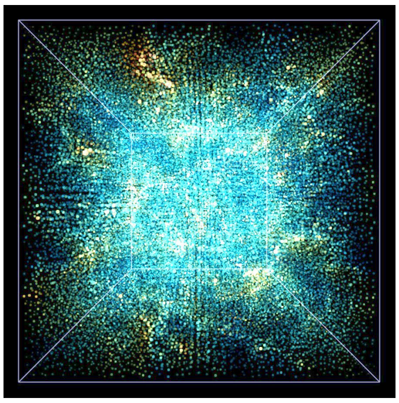


Plate 17: Virgo at level 3.
(123,000 clusters = 0.74%)
Rendered at 270 fps.

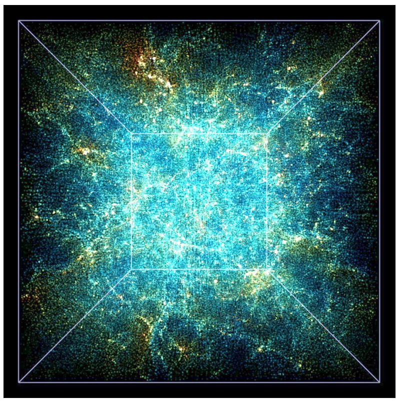


Plate 18: Level 4.
(671,000 clusters = 4%)
Rendered at 59 fps.

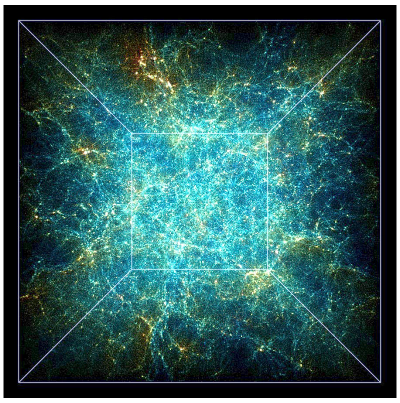


Plate 19: Level 5.
(3.3 million clusters = 19.7%)
Rendered at 12 fps.

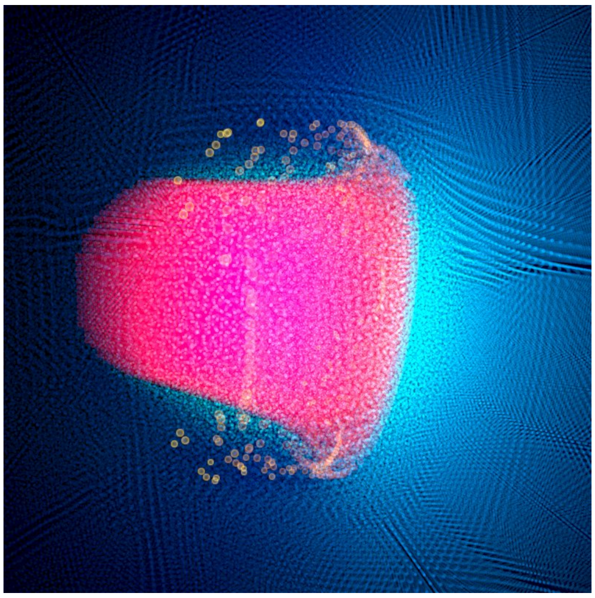


Plate 20: Visualization of a shock front, simulated with SPH (2.5 million points per time step).

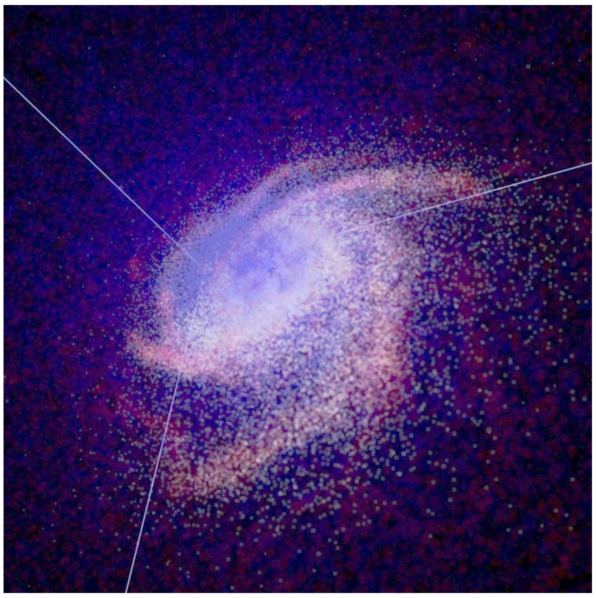


Plate 21: SPH and dark matter galaxy formation simulation rendered with sorted anti-aliased points (540,000 points).

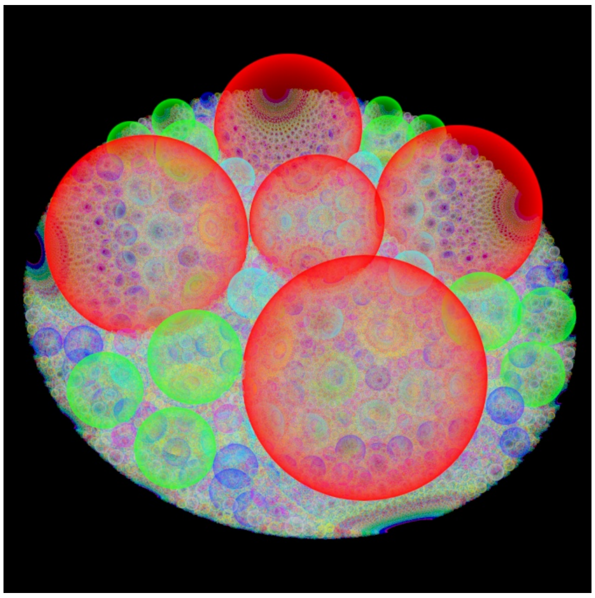


Plate 22: Reversible Apollonian packing rendered with sorted point sprites (6 million points).

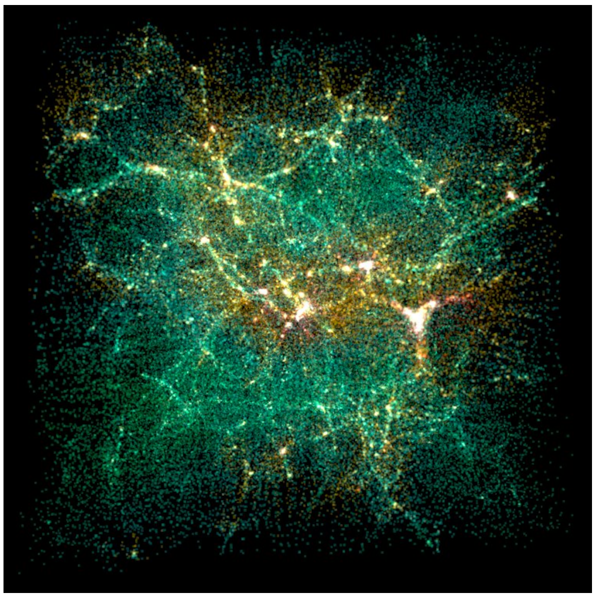


Plate 23: A dark matter SPH simulation from the Texas Advanced Computing Center (262,000 points per time step).

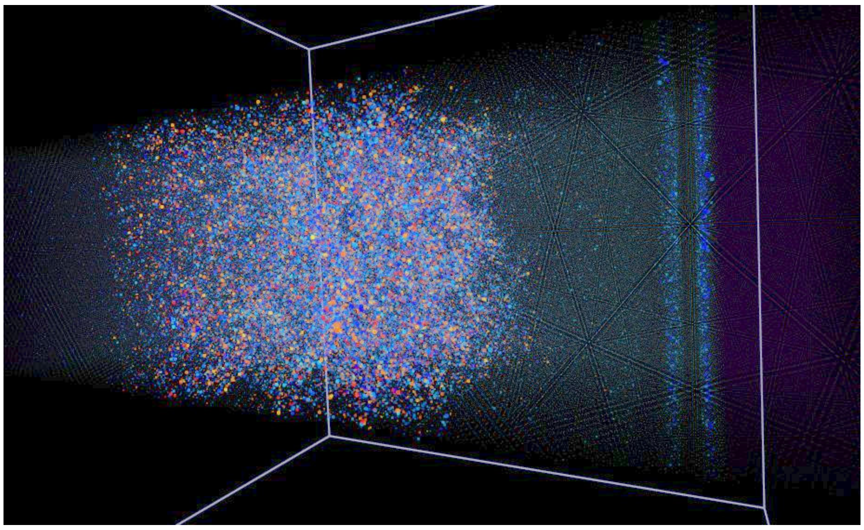


Plate 24: Molecular dynamics simulation of a shock in a quasi crystal using one million particles.

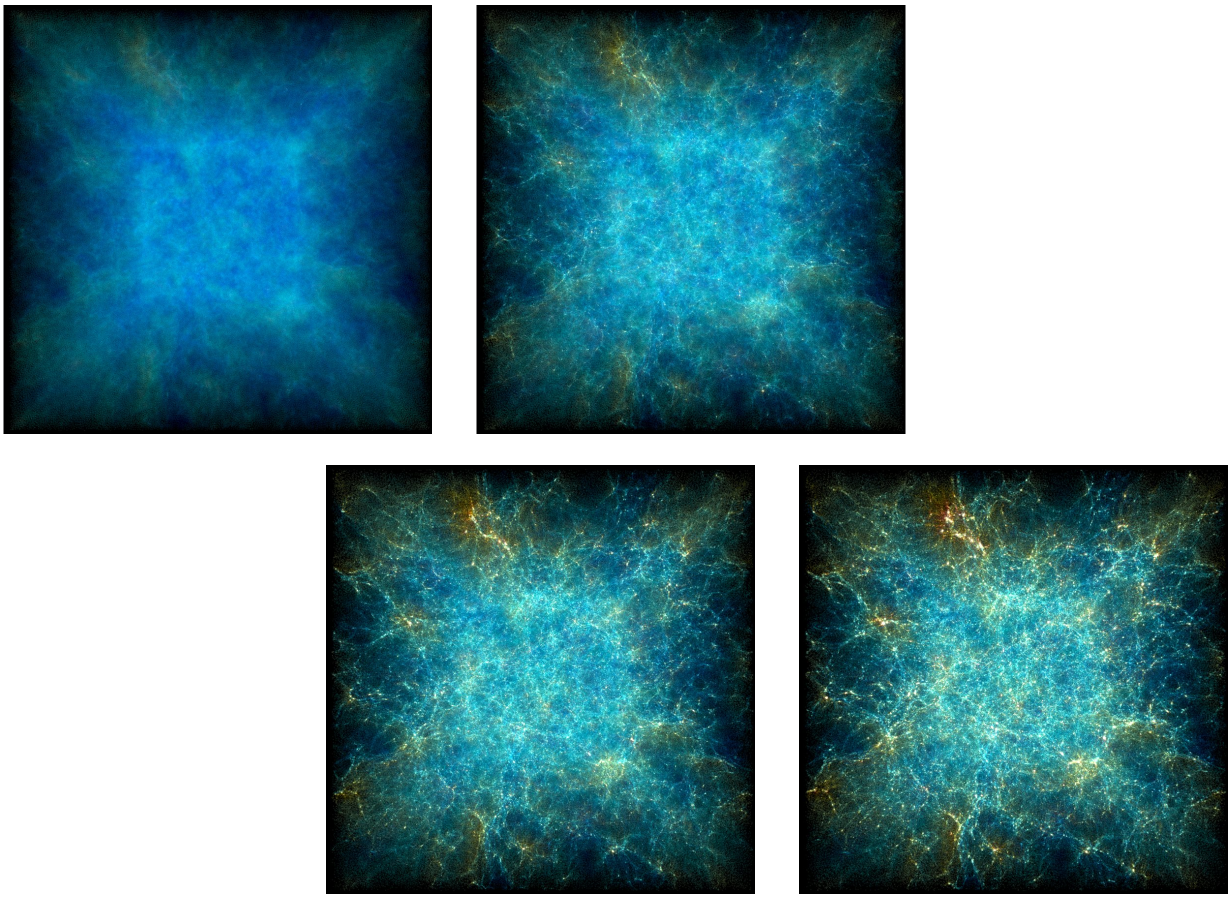


Plate 25: Some frames of an animated view of one of the virgo n-body simulations consisting of 10 time steps with 16.8 million points each.

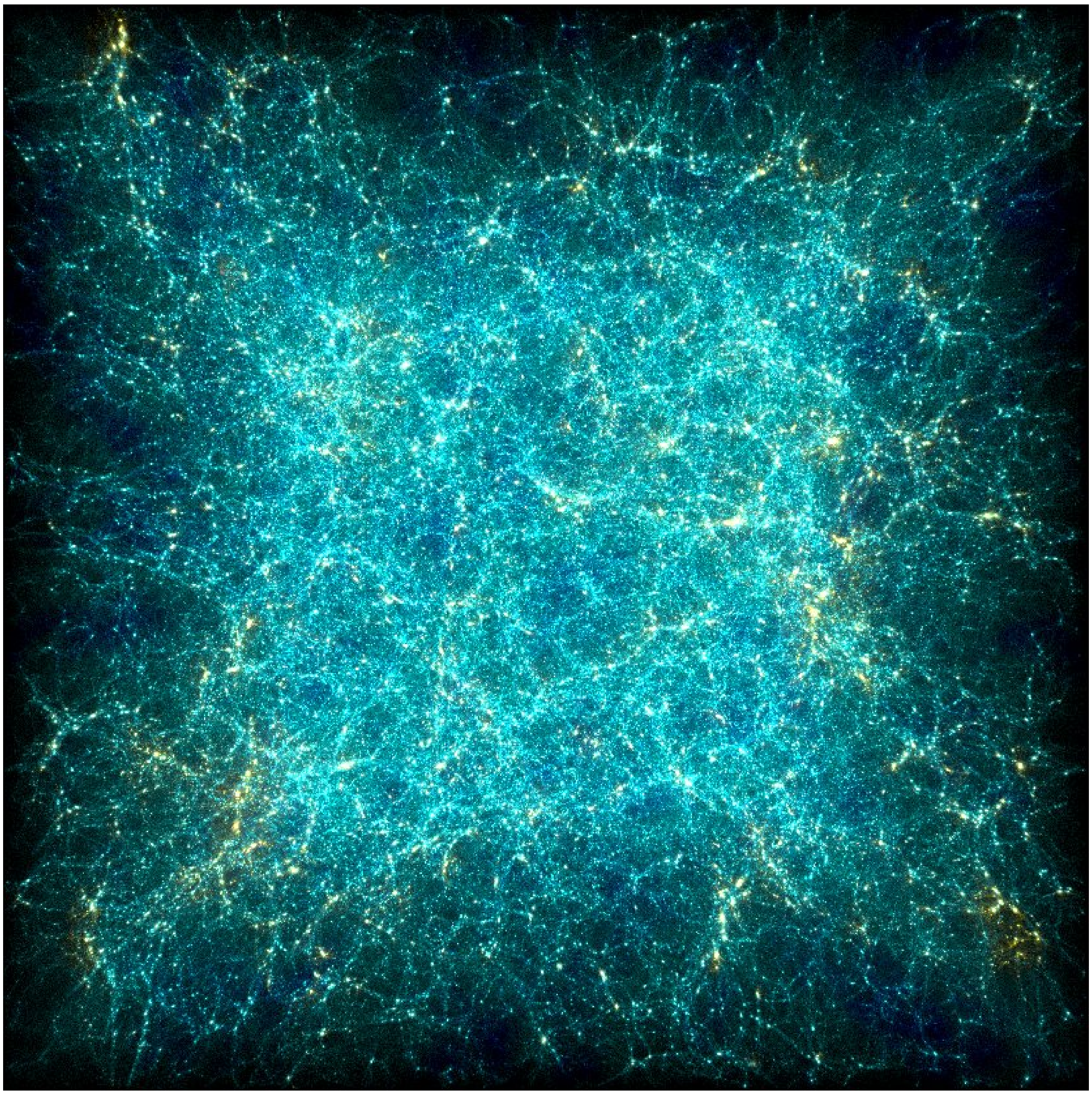


Plate 26: A total view of an n-body simulation from the Virgo consortium with 134 million particles per time step.

Chapter 9: Conclusion

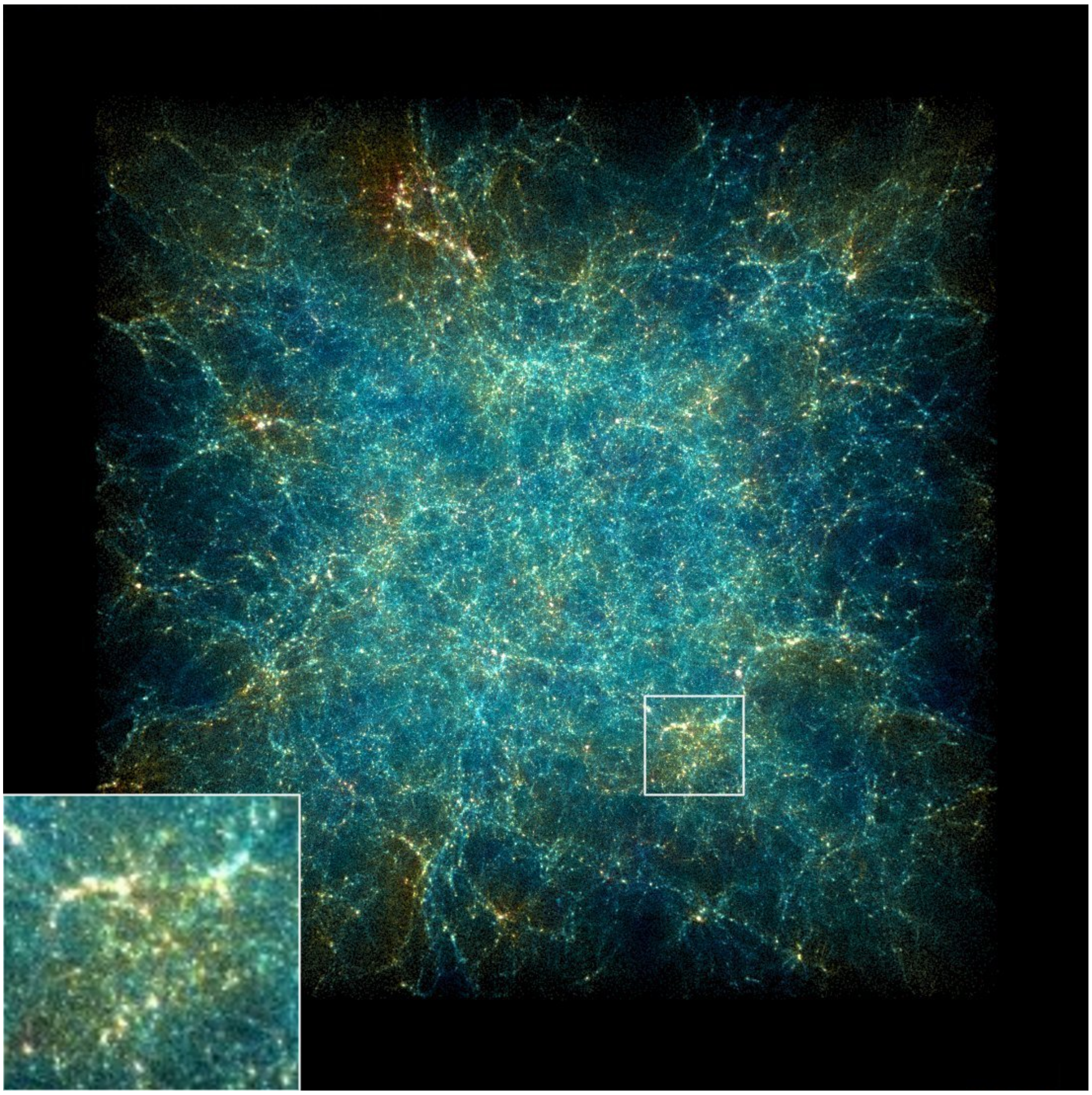


Plate 27: Irregular Volume Data with High Information Density: The virgo data set, rendered with hierarchical splatting (16.8 million points).

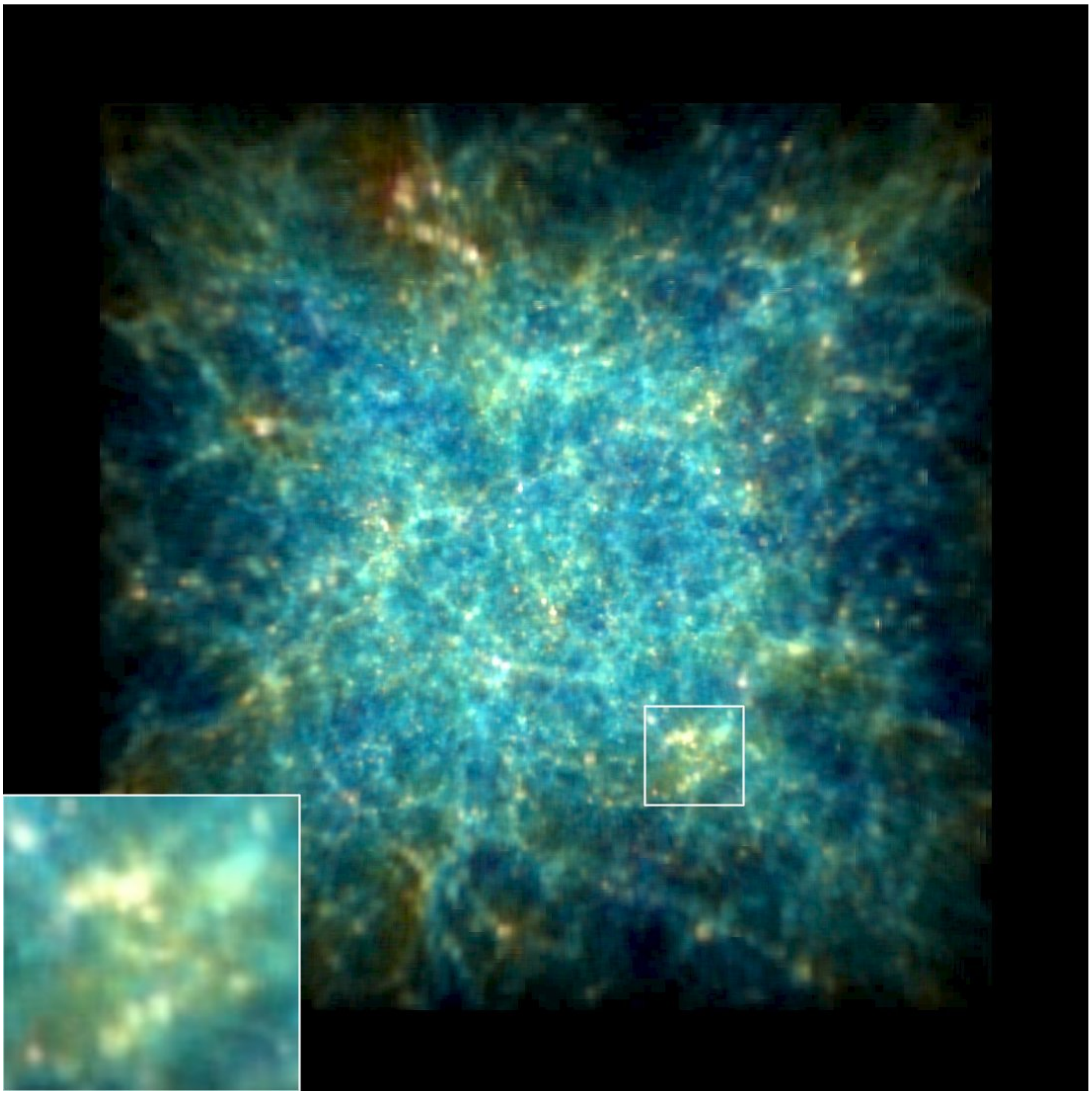


Plate 28: Irregular Volume Data with High Information Density: The virgo data set, rendered with texture-based volume rendering (resampled to a 128^3 RGB volume).

The Babylon 5 Mantra

Ivanova is always right.
I will listen to Ivanova.
I will not ignore Ivanova’s recommendations.
Ivanova is God.
And if this ever happens again,
Ivanova will personally rip your lungs out.

Ivanova in Babylon 5, A Voice in the Wilderness, Part 1