Accelerated Volume Rendering on
Structured Adaptive Meshes

vorgelegt von
Ralf Kähler

Am Fachbereich Mathematik und Informatik
der Freien Universität Berlin
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
eingereichte Dissertation

Februar 2005
Betreuer:
Prof. Dr. Dr. h. c. Peter Deuflhard
Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)
Takustr. 7
14195 Berlin
Acknowledgments

The work described in this thesis has been carried out at the department of Scientific Visualization at the Zuse Institute Berlin (ZIB) in cooperation with the Max-Planck-Institute for Gravitational Physics (Albert-Einstein-Institute, AEI). First of all I would like to thank Prof. Dr. Dr. h. c. Peter Deuflhard and Hon. Prof. Hans-Christian Hege for the opportunity to graduate in the exciting and innovative field of scientific visualization. Without their support and encouragement this work would not have been possible.

I also thank Prof. Dr. Ed Seidel for his constant interest and motivation and for giving me the possibility to collaborate with several international research groups.

I would like to thank all members of the Scientific Visualization department at ZIB and the Numerical Relativity group at AEI for many fruitful and inspiring discussions. Special thanks deserves Dr. Detlev Stalling for sharing his knowledge and insight. Finally I thank Prof. Dr. Tom Abel for the inspiring cooperation and for providing his exciting datasets.
Volume Rendering of a cosmological adaptive mesh refinement (AMR) simulation. The semi-transparent shells depict areas of constant gas density inside a proto-galaxy. Since many length scales had to be considered to correctly model the physical phenomena, ranging form 18,000 light years to cover the galaxy down to several light hours, to resolve the evolving proto-star in the central region, adaptive techniques were indispensable. The simulation was carried out on a time-dependent structured AMR grid with up to 30 levels of refinement. (dataset courtesy of T. Abel, Stanford University)
## Contents

1 Introduction .................................................. 1  
   1.1 Problem Formulation ................................... 1  
   1.2 Outline ................................................. 2  

2 Function Approximation on Discrete Grids ................. 5  
   2.1 Computational Grids .................................. 5  
      2.1.1 Structured Grids .................................. 6  
      2.1.2 Unstructured Grids ............................... 8  
      2.1.3 Block-Structured, Overlaid and Hybrid Grids ....... 8  
      2.1.4 Grid Adaption and Local Refinement ............... 9  
   2.2 Adaptive Mesh Refinement (AMR) ........................ 10  
      2.2.1 Structured Adaptive Mesh Refinement (SAMR) ....... 11  
      2.2.2 Notations .................................... 11  
      2.2.3 A Clustering Algorithm ........................... 13  
      2.2.4 Temporal Refinement Scheme ..................... 14  
   2.3 Interpolation .......................................... 17  
   2.4 Spatial Access Methods ................................ 21  

3 Visualization Methods for Scalar Data ..................... 25  
   3.1 The Visualization Pipeline ............................ 25  
   3.2 Visualization Methods for 3D Scalar Data .............. 27  
   3.3 Indirect Volume Rendering ............................. 28  
      3.3.1 Slice-Based Techniques ......................... 28  
      3.3.2 Isosurface Extraction ............................ 29  
   3.4 Direct Volume Rendering ................................ 33  
      3.4.1 Transfer Functions ............................... 35  
      3.4.2 Emission-Absorption Models ...................... 36  
      3.4.3 Raycasting .................................. 38  
   3.5 Texture-Based Volume Rendering ....................... 39  
      3.5.1 2D Texture-Based Volume Rendering ............. 40  
      3.5.2 3D Texture-Based Volume Rendering ............. 41  

V
Chapter 1

Introduction

1.1 Problem Formulation

Multi-scale phenomena are abundant in many application fields like material science, fluid dynamics, geophysics, meteorology and astrophysics. Representing and numerically simulating such processes is a challenging task since quite different scales have to be resolved, which often requires enormous amounts of storage and computational power. An important strategy in this context is adaptivity, i.e. local adjustment of the spatio-temporal resolution to the details to be resolved. A standard representation therefore are hierarchical, locally refined grids.

A specific adaptive approach for solving partial differential equations, usually called AMR (Adaptive Mesh Refinement), was introduced by Berger et al. in 1984 [9]. The basic idea is to combine the simplicity of structured grids and the advantages of local refinement. In this numerical scheme the computations are started on a set of coarse, potentially overlapping structured grids, that cover the computational domain. Local error criteria are applied to detect regions that require higher resolution. These are covered by subgrids with decreasing mesh spacing, which do not replace, but rather overlap the refined regions of the coarser patches. The equations are advanced on the finer subgrids and the refinement procedure recursively continues until all cells fulfill the considered error criteria, giving rise to a hierarchy of nested levels of refinement.

An advantage of this approach is that each subgrid is treated as a separate grid with its own storage space. This allows to process them almost independently during integration and hence AMR is well suited for parallel processing. Since the computations are carried out on structured grids, AMR further allows the reuse of many existing finite difference codes with only minor modifications.

In 1989 Berger et al. [8] proposed a variant of this scheme, called Structured Adaptive Mesh Refinement (SAMR), which reduces some of the complexity of the original approach. While the separate subgrids in the AMR scheme could be rotated against each other, in SAMR they are aligned with the major axes of the coordinate system, which for example simplifies the computation of fluxes of (conserved) quantities through the cell faces. SAMR has become more and more popular in the last decade, and nowadays it is
applied in many domains like hydrodynamics, meteorology and in particular in cosmology and relativistic astrophysics.

Due to this growing popularity, an increasing number of scientists is in need of appropriate interactive visualization techniques to interpret and analyze AMR simulation data. Tools for both, 2D analysis to quantitatively convey the information within single slices and 3D representations to apprehend the overall structure are required.

In this thesis we develop direct and indirect volume visualization algorithms for scalar fields that are defined on SAMR grids. Additionally we investigate the applicability of SAMR data structures for visualizing large datasets. The developed algorithms were designed to meet the followings goals:

- **Accuracy**: Artifact-free visualization is a prerequisite for a faithful representation of scientific data. For AMR data problems typically arise at the interfaces between different levels of resolution. The existence of dangling nodes in these regions necessitates special attention to ensure globally continuous spatial interpolation. Also the special temporal refinement scheme of AMR can cause problems if dense output is required in the visualization phase.

- **Performance**: The visualization methods should allow interactive exploration even for large datasets. In order to achieve this, we designed algorithms that operate directly on the given SAMR grid, exploiting both, the inherent hierarchical structure of the data and the fact that the grid itself is composed of blocks with regular topology. An important design aspect thereby was that the resulting algorithms should allow the utilization of dedicated graphics hardware.

- **Usability**: The algorithms had to be provided to several research groups in the fields of cosmology and numerical relativity. Therefore the integration into an existing visualization system was another important design criterion. In particular the visualization routines had to allow an intuitive usage and the amount of user interaction had to be kept small.

1.2 Outline

This thesis is divided into three chapters; the next two cover basic concepts in the field of scientific visualization, while the last two focus on the new contribution of this thesis.

Chapter 2 starts with an overview about commonly used computational grids and data structures (Section 2.1). In particular the spatial and temporal refinement scheme of the AMR approach is discussed and notations that are used throughout this work are introduced in Section 2.2. The chapter ends with two sections on interpolation of discrete grid functions and spatial access methods (Section 2.4 and 2.3).

Chapter 3 addresses the most popular visualization methods for scalar data. While indirect methods usually involve some form of dimensionality reduction and/or conversion of the data to auxiliary graphics primitives, the underlying idea of direct volume rendering is to map the data field to physical quantities like absorption and emission coefficients
and to compute the intensity distribution of light traveling through this semi-transparent medium in the image plane. Starting with a brief discussion of the rendering pipeline in Section 3.1, we review the most important indirect and direct volume rendering algorithms in Section 3.3 and 3.4.

Chapter 4 first gives an overview on related work in the field of accelerated volume rendering. Next we discuss a raycasting approach that is based on the utilization of an adaptive integration scheme (Section 4.3). In Section 4.4 we investigate approaches that accelerate texture-based volume rendering for the frequently occurring case of large, sparse data, i.e. highly resolved data where only a small fraction of the voxels contains relevant information.

In Chapter 5 our work on visualization of structured adaptive mesh refinement data is presented, starting with an overview about related work in this field in Section 5.1. We propose a domain decomposition scheme for SAMR hierarchies in Section 5.2 and discuss interpolation issues for discrete SAMR data in Section 5.3. In Section 5.4 we present indirect volume rendering methods; in particular an algorithm for the extraction of $C^0$-continuous isosurfaces is discussed. Software-, and hardware-based direct volume rendering approaches are presented in Section 5.5. In Section 5.6 we propose an approach for temporal interpolation of AMR data.

Key ideas described in this work have been presented at international conferences and in scientific journals over the last three years. The hardware-accelerated volume rendering algorithm for SAMR data appeared in [37]. An application of this approach in a cooperation with several scientists at the NATIONAL CENTER FOR SUPERCOMPUTING APPLICATIONS (NCSA) and the UNIVERSITY OF PENNSYLVANIA, for rendering one of the most complex AMR simulation ever carried out in numerical astrophysics, was presented at the IEEE VISUALIZATION 2002 conference [36].

All algorithms have been implemented in the framework of an extension to the 3D-visualization system AMIRA [80, 1], developed at the ZUSE-INSTITUTE BERLIN.
Chapter 2

Function Approximation on Discrete Grids

Two main sources of scientific data can be distinguished: the first group consists of measured data, for example acquired by 3D imaging devices like computer tomographs or microscopes. In this case the original, continuous signal is sampled at certain positions in space and/or time. The second main source of scientific data are numerical simulations, which for example compute the solution of partial differential equations. The most popular approaches are finite difference and finite element schemes. Whereas in the first case the continuous solution is approximated at discrete points, lines, area or volume elements, in the latter case it is represented by a finite set of locally defined (polynomial) functions, so called shape functions.

So in the vast majority of cases scientific data is not given as an analytical expression $f(x)$ that can be evaluated at any position within the data domain $\Omega$, but rather as a finite number of data samples $f_i$ defined at discrete locations $x_i \in \Omega$.

Computational grids are employed to represent the geometrical and topological structure of these discrete approximations of the continuous data. In order to faithfully reconstruct the original signal (ideal reconstruction), respectively the continuous function, the discretization has to fulfill certain constraints. In particular the sampling theorem states that the spatial/temporal distance of the sampling locations has to correspond to the highest frequency components contained in the Fourier spectrum of the signal.

We will sketch the topics of computational grids and reconstruction, respectively interpolation in the next two sections.

2.1 Computational Grids

It is often advantageous to distinguish the topological structure (abstract complex) of a grid from its geometrical embedding (realization). Following the discussion in [12], these concepts can be defined as
Definition 1 (Abstract complex): An abstract finite complex \( C \) of dimension \( d \) is a finite set of elements \( e \), together with a mapping \( \dim : C \rightarrow \{0, \ldots, d\} \subseteq \mathbb{N} \) and a partial order relation \(<\), such that \((e_1 < e_2) \implies (\dim(e_1) < \dim(e_2))\). \( \dim(e) \) is called the dimension of \( e \). Elements of dimension 0 are called vertices, dimension 1-elements are called edges, dimension \((d-1)\)-elements are called faces and \( d \)-dimensional elements are called cells.

Definition 2 (Geometric realization): A geometric realization of an abstract complex \( C \) is a Hausdorff space \( H \) together with a mapping

\[
\Phi : C \rightarrow \Phi(C) = \bigcup_{e \in C} \Phi(e) \subseteq H
\]

that fulfills the following requirements:

\[
\begin{align*}
(i) \quad & e_1 < e_2 \iff \Phi(e_1) \subset \partial \Phi(e_2) \quad \text{and} \\
(ii) \quad & \partial \Phi(e_2) = \bigcup_{e_1 < e_2} \Phi(e_1),
\end{align*}
\]

for all elements \( e_1, e_2 \in C \).

This definition is relatively general and allows for example cells that contain holes. In the special case that each cell in the complex is homeomorphic to open balls in \( \mathbb{R}^d \), which is still general enough to cover more or less all types of cells that are employed in numerics and geometrical modeling, the complex is also called a CW-complex\(^1\). A detailed discussion of these topics is for example given in [25, 11].

According to their topological structure computational grids can be classified into two main categories, namely structured and unstructured grids, as well as mixtures of these two types.

2.1.1 Structured Grids

Structured grids are logically rectangular in the sense that their vertices can be arranged on a rectangular lattice in an appropriate geometric realization. Hence the vertices can be addressed by sets of integer indices, such that sequential indices refer to vertices that are connected by an edge, and so this type of grids does not require the storage of explicit connectivity. Structured grids consist of quadrilateral, respectively hexahedral cells. Their implicit connectivity relation allows for fast and efficient access of adjacent data samples and hence structured grids are popular in finite difference approaches.

\(^1\)CW stands for closure-finite weak topology.
Figure 2.1: Examples of structured grids with uniform (a), rectilinear (b) and curvilinear coordinates (c).

The simplest but nevertheless very important example, particularly for measured data, are uniform grids. In this case the computational domain is discretized by rectangular, axis-aligned cells. Neglecting a potential offset vector, the coordinates of the vertices $x_{ijk}$ can be computed from the index triples $(i, j, k)$ via

$$x_{ijk} := (ih_0, jh_1, kh_2).$$

Here $h$ are the edge lengths of the cells, which are constant along each coordinate axis, but might differ between the three main directions. In the following the cell that contains the vertices

$$\{ x_{lmn} \mid l = i, i + 1; m = j, j + 1; n = k, k + 1 \}$$

will be denoted by $\Omega_{ijk}$.

Rectilinear grids are a generalization of uniform grids in the sense that the edge lengths $h_i$ might vary along each coordinate axis, compare Figure 2.1 (b). Although the connectivity between the cells is still defined implicitly, the coordinates of the vertices have to be provided explicitly for each axis, usually in form of three separate lists.

Curvilinear grids have a geometric realization such that the cells are not axis-aligned to an Euclidean coordinate system, as shown in Figure 2.1 (c). Often an analytic mapping from the uniform parameter space to the actual coordinate system is given, which allows to compute the vertex coordinates from their index triples. Otherwise the coordinates have to be stored explicitly. This separation allows to benefit from the memory efficient regular topology, while at the same time curvilinear grids are flexible enough to model a vast range of complex geometries.
2.1.2 Unstructured Grids

The second main category of computational grids are *unstructured* grids. They often consist of triangles or quadrilateral cells in two dimensions, respectively tetrahedral and hexahedral cells in the three dimensional case, but also prisms or pyramid cells are employed. Due to their flexibility, unstructured grids are well suited for modeling highly complex geometries. They further allow for easy grid adaption and local refinement, as discussed below. A disadvantage are their high memory requirements, since vertex coordinates as well as cell connectivity information have to be stored explicitly. Unstructured grids are primarily applied in finite element schemes.

2.1.3 Block-Structured, Overlaid and Hybrid Grids

A third category are grid types that combine aspects of both, structured and unstructured grids. *Block-structured* grids, introduced in the 1980s, further increase the flexibility of curvilinear grids. The computational domain is covered by a set of structured grids, which are pieced together at their boundary interfaces, ensuring coinciding vertices in these regions. Figure 2.3 (a) shows an example.

![Figure 2.2: Unstructured grid that models the flow field inside a turbine.](image)

![Figure 2.3: Examples of block-structured (a), overlaid (b) and hybrid (c) grids.](image)
A generalization of block-structured grids are so-called chimera or overlaid grids, compare Figure 2.3 (b). In this case vertices on the overlapping patch region do not have to coincide, which facilitates the grid generation process substantially. A drawback is the increased complexity for interpolation due to partially overlapping cells in the boundary regions of adjacent patches and communication, especially in case of parallel computing.

Hybrid grids are grids that consist of different types of cells. This allows for example to cover homogenous regions by hexahedral cells, while complex shaped geometries at the boundaries of computational domains might be modeled with tetrahedra. Hybrid grids usually require complex data structures and increase the complexity of the numerical solvers, in order to handle the different types of cells. Figure 2.3 (c) shows an example.

In the remainder of this sections we will deal with the special case, that the discretized data is a numerical solution approximation of some equation, for example a system of partial differential equations.

2.1.4 Grid Adaption and Local Refinement

A powerful strategy to increase the accuracy of numerical solutions is to adapt the underlying grid structure, in order to better adjust to the physical behavior of the given problem.

There exist various grid adaption methods. In the r-method the topological structure of the grid remains unchanged. Instead the geometric location of the nodes is altered, based on an analysis of the current solution. Hereby it is crucial to avoid degenerated and overlapping elements. In contrast to this p-methods adjust the degree of the approximation by employing higher order shape functions (in the finite element approach). Therefore additional nodes have to be added to existing elements. In h-methods the grid adaption is carried out by refining (and coarsening) grid elements. Approaches that follow both of the latter two branches are called hp-methods.

In principle h-refinement can be carried out by refining the whole computational grid (global refinement), for example by simply replacing every cell by a number of smaller cells. However, for realistic grid sizes this usually results in too high computational efforts, in terms of memory and computational requirements. A much more efficient way is to refine only those cells $\Omega_i$ that cover regions where the local error of the solution $e_{\text{loc}}(\Omega_i)$ is above a certain threshold, aiming at an equal distribution of the error over the whole computational domain (local refinement). This usually involves the application of some kind of error estimator $\bar{e}_{\text{loc}}(\Omega_i)$ for the unknown local error $e_{\text{loc}}(\Omega_i)$, for example by comparing the solutions obtained by applying shape functions of different order or by different grid spacing.

Local refinement of tetrahedral grids is usually carried out by replacing the cells that need refinement by sets of smaller tetrahedra, generating new vertices in the interior of the cells. Special refinement strategies (for example red and green refinement rules), are necessary to avoid the creation of degenerated cells that can cause numerical problems, compare for example [69, 6].

Local refinement is more problematic for structured grids, since it interferes with the regular grid topology, and thus requires more sophisticated data structures to store the
Figure 2.4: Refinement process for AMR schemes: Cells that require refinement are determined using local error criteria (a) and clustered into separate subgrids (b), which cover the regions with higher resolution. This process is recursively continued until each cell fulfills the error criteria (c).

resulting grid structures. A related problem is the introduction of so-called *hanging* or *dangling* nodes. These are nodes in the interior of the domain, that have a smaller number of emerging edges, compare Figure 5.4. These nodes require special treatment by the numerical solver in order to ensure the desired continuity properties of the solutions. One method is to restrict the solution at these nodes to the solution obtained at the location within the adjacent coarse cell (*dependent nodes*). Alternatively a so-called *conforming closure* might be performed, in which cells that contain dangling nodes are replaced by cells of a different types, resulting in a hybrid grid.

### 2.2 Adaptive Mesh Refinement (AMR)

A special adaptive method for solving hyperbolic partial differential equations was introduced by Berger et al. in 1984 [9]. The basic idea of AMR is to combine the simplicity of structured grids with the advantages of local grid adaption.

In this approach the computational domain is covered by a set of coarse, potentially overlapping structured subgrids. During the computation local error estimators are utilized to detect cells that require higher resolution. These cells are covered by a set of rectangular subgrids which may have arbitrary orientations. Unlike in finite element approaches these subgrids do not replace, but rather overlay the refined regions of the coarse base grid. The equations are advanced on the finer subgrids and this refinement procedure recursively continues until all cells fulfill the considered error criterion, giving rise to a hierarchy of nested refinement levels, as shown in Figure 2.4.

A further advantage of AMR is that each subgrid can be viewed as an separate, independent grid with a separate storage space. This allows to process subgrids almost independently during integration and hence the approach is well suited for parallel processing.
2.2.1 Structured Adaptive Mesh Refinement (SAMR)

In 1989 Berger and Collela [8] proposed a variant of the approach above, called structured adaptive mesh refinement (SAMR), dedicated to simplify the application of the AMR scheme to hyperbolic conservation laws.

The main difference of SAMR is that the subgrids do not have arbitrary orientations anymore, but are rather aligned with the major axes of the Euclidean coordinate system. In particular this facilitates the computation of fluxes of conserved quantities like mass or energy through the cell faces. In principle the base grid and the subgrids can be rectilinear grids, but usually uniform patches are employed, compare Figure 2.5.

In the last decade SAMR has gained more and more popularity and nowadays it is applied in many domains like computational fluid dynamics [4], meteorology [3], relativistic astrophysics [73, 48] and in particular in cosmology [15, 2]. In the following we will describe the spatial and temporal refinement scheme in more detail and introduce basic notations for SAMR, which are used in the remainder of this thesis.

2.2.2 Notations

Let \( \Omega \subset \mathbb{R}^3 \) denote the data domain, which is discretized by a hierarchy of axis-aligned, uniform grids \( (\Omega^l)_{l=0,1,...,l_{\text{max}}} \) with decreasing mesh spacings. The index \( l \) numbers the refinement level, starting with 0 for the coarsest level. Let the mesh spacing of the coarsest grid be given by \( h^0 = (h^0_0, h^0_1, h^0_2) \). The mesh spacings of the finer grids are recursively defined by \( h^l := (h^{l-1}_0 / r, h^{l-1}_1 / r, h^{l-1}_2 / r) \), where the positive integer \( r \) denotes the so-called refinement factor. In principle this factor can differ for each direction and each level, but in order to simplify the notation we assume that it is constant.

Further let \( n_i \) be the number of vertices along the \( i \)-th coordinate axis of the base grid \( \Omega^0 \). Assuming that the origin of the coordinate system is located at \((0,0,0)\), the coordinates of the vertices of \( \Omega^l \) are given by

\[
x^l_{ijk} := (ih^l_0, jh^l_1, kh^l_2); \quad i = 0, 1, ..., (n_0 - 1)(r)^l, \quad j, k = ..., (2.2)
\]

and thus the coordinate \( x^l_{ijk} \in \Omega^l \) coincides with \( x^{l+1}_{r_1,r_2,r_3} \in \Omega^{l+1} \).
Figure 2.6: Two-dimensional example of a structured AMR grid. The root level \( \Gamma_0 \) is refined by three subgrids \( \Gamma_{1,0}, \ldots, \Gamma_{1,2} \) that generate the refinement level \( \Lambda^1 \). \( \Lambda^1 \) itself is refined by one subgrid \( \Gamma_2^0 \). All three subgrids on \( \Lambda^1 \) are siblings and each is a parent of \( \Gamma_2^0 \).

The grid cell \( \Omega_{ij}^l \subseteq \Omega^l \) is given by

\[
\Omega_{ij}^l := \text{conv}\{ x \in \Omega^l \mid x = x_{ijk}^l + \sum_{m=0}^{2} \alpha_m e_m, \alpha_m \in [0, h^l_m] \},
\]

where \((e^0, e^1, e^2)\) denote the standard basis in \(\mathbb{R}^3\). Each coarse cell can be decomposed into a set of \(r^3\) cells of the next finer discretization

\[
\Omega_{ijk}^l = \bigcup_{i,j,k} \Omega_{i,j,k}^{l+1} \text{ with } \hat{i} = ri, ri + 1, \ldots, ri + r; \hat{j}, \hat{k} = \ldots,
\]

so the cells \(\Omega_{i,j,k}^{l+1}\) provide a refinement of the coarse cell \(\Omega_{ijk}^l\). Since in AMR grids can be represented as a tree of nested levels, the coarse base grid \(\Omega^0\) is also called root level in this context. In general it may be composed of a set of non-overlapping, axis-aligned uniform patches, but in order to simplify the notation we assume that \(\Omega^0\) consists of just one patch.

As mentioned above, the solution of the equations are initially approximated on this root grid and the coarse solution is inspected utilizing the error estimator that detects cells that require higher resolution. These cells are clustered into disjoint, axis-aligned rectangular regions, which define new subgrids, consisting of cells of the next finer discretization \(\Omega^1\). We will sketch a clustering algorithm for this purpose in Subsection 2.2.3.

Notice that cells are either completely refined by cells of the next finer grid according to Equation (2.4), or remain completely unrefined. Let the \(m\)-th subgrid of \(\Omega^l\) be denoted
by
\[
\Gamma^l_m = \{ \Omega^l_{ijk} \subseteq \Omega^l \mid i = p^m_0, ..., p^m_0 + n^m_0; \ j, k = ... \},
\]
where \( p^m \) is the integer offset vector of this subgrid, and \( n^m_0 + 1 \) is the number of cells per i-th coordinate axis.

**Definition 3**:

(i) The union of all level \( l \) subgrids \( \Lambda^l \) is called refinement level \( l \) or just level \( l \). By construction these levels are nested, i.e. \( \Lambda^{l+1} \subseteq \Lambda^l \subseteq \Omega^l \). We will denote the whole grid hierarchy, i.e. the union of all refinement levels by \( \Lambda^\infty := \bigcup_{l=0}^{\infty} \Lambda^l \).

(ii) The level \( l \) subgrid \( \Gamma^l_m \) is called a child or descendant of \( \Gamma^l_{m-1} \), denoted by \( \Gamma^l_m = C(\Gamma^l_{m-1}) \), if \( \Omega^l_{ijk} \cap \Omega^l_{m-1,ijm-k} \neq \emptyset \). In this case \( \Gamma^l_{m-1} \) is a parent of \( \Gamma^l_m \), denoted by \( \Gamma^l_{m-1} = P(\Gamma^l_m) \).

(iii) Two subgrids \( \Gamma^l_m \) and \( \Gamma^l_{m'} \) on level \( l \) are called siblings, denoted by \( \Gamma^l_m = S(\Gamma^l_{m'}) \iff \Gamma^l_m = S(\Gamma^l_{m'}) \), if \( \Omega^l_{ijk} \cap \Omega^l_{m'} \neq \emptyset \). In the original AMR scheme described in [8] each refinement level had to be enclosed by at least one layer of cells from the next coarser level of resolution, such that adjacent cells differ by at most one level. This constraint was relaxed by others, see for example [2, 63]. In the following we will refer to AMR grids that contain adjacent cells which differ by at most one level as restricted AMR grids, and to the more general case, like for example the one depicted in Figure 2.6, as unrestricted AMR grids.

### 2.2.3 A Clustering Algorithm

A crucial part in the AMR algorithm is the generation of the structured subgrids, which cover the cells that require higher resolution. An efficient and fast algorithm for clustering collections of cells into axis-aligned regions was suggested by Berger et al. [10]. It adopts signature based methods used in computer vision and pattern recognition. We will briefly describe the basic ideas in this section.

Assume that the information about which cells of a subgrid \( \Gamma^l_m \) are selected for clustering is encoded by the binary function defined on the index domain of \( \Gamma^l_m \):
\[
S : [p^m_0, ..., p^m_0 + n^m_0] \times [p^m_1, ...] \times [p^m_2, ...] \rightarrow \{0, 1\}
\]
with
\[
S(i, j, k) = \begin{cases} 
1, & \text{if } \Omega^l_{ijk} \text{ is marked for clustering} \\
0, & \text{otherwise.}
\end{cases}
\]

In a first step the number of cells that need refinement is computed for each slice perpendicular to the three major coordinate planes and stored in so called signature lists. For example the entry for slice number \( i \) parallel to the \( yz \)-plane is given by
\[
S_{yz}(i) = \sum_{j=p^m_1}^{(p^m_1+n^m_1)} \sum_{k=p^m_2}^{(p^m_2+n^m_2)} S(i, j, k)
\]
Figure 2.7: 2D example of the clustering procedure: (a) In a first step the signature lists are computed. (b) Exterior rows and columns with zero entries are pruned off. (c) Interior zero entries and inflection points indicate splitting edges.

and similarly for the two other orientations.

A two dimensional example is shown in Figure 2.7 (a). In the next step exterior zero-entries in these lists are detected and pruned off in order to place a minimal bounding box around the marked cells, as shown in Figure 2.7 (b). Any interior zero entry in these lists indicates a potential splitting index, i.e. a position at which the given volume is subdivided into two smaller subregions. If all signatures are non-zero, the second derivative

$$\Delta_{yz}(i) = S_{yz}(i + 1) - 2S_{yz}(i) - S_{yz}(i - 1), \quad (2.9)$$

and similar for $\Delta_{xy}(k)$, $\Delta_{xz}(j)$, of each signature list is computed and the largest inflection point is chosen as the splitting plane, compare Figure 2.7 (c). This procedure is repeated recursively on the newly created subregions until one of the following halting criteria is satisfied:

- The subregion exceeds the efficiency ratio, i.e. the ratio of the number of cells tagged for clustering to its total number of cells is greater than a preselected threshold.

- Further subdivision of the region would result in grid dimensions smaller than some minimal extension.

Notice that according to these criteria the clusters usually contain a number of cells that are not marked, in order to keep the number of created subgrids low. Usually an efficiency ratio of 85% and a minimal extension of 8 yields good results in terms of the number of grids and the additional memory overhead for the additional cells.

2.2.4 Temporal Refinement Scheme

For numerical solvers of partial differential equations with explicit time-integration, stability conditions demand that the time step size $\Delta t$ of the scheme corresponds to the mesh size $\Delta x$, in the sense that the time step decreases as the mesh spacings decreases. As an
example consider the Courant-Friedrichs-Levy (CFL) condition, which for hyperbolic and parabolic systems implies that

$$\frac{\Delta x}{\Delta t} \geq k,$$

respectively

$$\frac{(\Delta x)^2}{\Delta t} \geq k,$$

have to be fulfilled, where $k$ is a system dependent constant. Hence a global time step for all subgrids in an AMR hierarchy would be determined by the cell size of the highest resolved level present in the hierarchy, resulting in a large computational overhead for the coarser levels.

This is the reason for the fact that besides the spatial refinement, AMR schemes for solving partial differential equations additionally perform a refinement in time. That means the spatially refined levels are updated more frequently than the coarser ones. The order in which the levels are advanced in time is demonstrated in the following pseudo code:

```
IntegrateLevel (level, parentTime) {
    dt = getTimeStep (level);
    while (time < parentTime) {
        time = time + dt;
        SetBoundaryValues();
        AdvanceEquations (dt);
        if (level < maxLevel) {
            IntegrateLevel (level + 1, time);
            ProjectSolution (level + 1, level);
            RegridHierarchy (level + 1);
        }
    }
}
```

Figure 2.8: Pseudo code for the recursive AMR time-integration scheme.

First the coarse level $\Lambda^l$ is advanced for a large time step ($\text{AdvanceEquation}$). Boundary values for the subgrids on levels are provided by interpolation on the next coarser level $\Lambda^{l-1}$ or by copying from sibling subgrids on $\Lambda^l$ ($\text{SetBoundaryValues}$).

Next the integration routine is recursively called for the refined levels $\Lambda^{l+1}, ..., \Lambda^{l_{\text{max}}}$, and these subgrids are advanced with a decreasing time step size. The integration of the finer levels is followed by the so-called restriction step ($\text{ProjectSolution}$), that updates the coarse grid function by the more accurate values of the finer ones.

In the last step the solution is inspected and the grid structure is adapted based on the local error criterion ($\text{RebuildHierarchy}$). This implies that the topology of $\Lambda^{l+1}, ..., \Lambda^{l_{\text{max}}}$ might change after each integration step on a level $l$. In general the structure of the whole
hierarchy (except for the root level) is modified at time steps at which the root level grid is updated.

The time steps of the refined levels do not necessarily have to be equally distant, but it has to be ensured that after an integer number of updates the times of all levels in the hierarchy match up again.

A temporal refinement factor $r_t \in \mathbb{N}$ between a pair of two consecutive refinement levels $(\Lambda^l, \Lambda^{l+1})$ indicates that $\Lambda^l$ is evolved one large step $\Delta t_l$, and next $\Lambda^{l+1}$ is evolved $r_t$ times with a step sizes of $\Delta t / r_t$. Figure 2.9 depicts the AMR integration order for a temporal refinement factor of 2. In the following we denote the union of level $l$ subgrids at a certain time by $\Lambda^l(t)$ and the grid hierarchy by $\mathcal{H}(t)$.

![Figure 2.9: Order of temporal integration of a grid hierarchy with an overall temporal refinement factor of $r_t = 2$.](image-url)
2.3 Interpolation

In the following we will restrict the discussion to interpolation schemes that are employed in the remainder of this thesis. For the sake of simplicity we will assume real-valued, scalar grid functions. For more detailed information the reader might refer to textbooks like [97].

Let the grid function \( f : G \rightarrow \mathbb{R} \) and possibly also some of its derivatives \( f', f'', \ldots \) be defined at the \( n \) discrete locations \( p_i, i = 0, \ldots, n - 1 \) on the computational grid \( G \). Common examples are grid functions sampled at the vertices of the grid (vertex-centered grid functions), as well as at cell or face centers (cell-, respectively face-centered grid functions). In order to obtain data values at arbitrary positions inside the data volume \( \Omega \), an interpolant

\[
I(f, f', f'', \ldots | p_0, \ldots, p_{n-1}) : \Omega \rightarrow \mathbb{R} \quad \text{with} \quad p \mapsto I(p)
\]

(2.10)

needs to be specified. Interpolation functions are often expressed as linear combinations of a set of weighting functions \( \omega_i, \tilde{\omega}_i, \ldots : G \rightarrow \mathbb{R} \), that are also called shape functions in the context of finite element analysis

\[
I(p) = \sum_{i=0}^{n-1} \omega_i(p) f(p_i) + \tilde{\omega}_i(p) f'(p_i) + \ldots
\]

(2.11)

According to the Shannon sampling theorem [62, 75], a continuous signal \( \hat{f} \) can be reconstructed from its discrete data samples (ideal reconstruction), if the sampling rate was larger than the highest frequency contained in the spectrum of the signal:

**Theorem 1** Let \( \hat{f} \) be a function that is band-limited to \( [-\sigma, \sigma] \), i.e.

\[
\hat{f}(t) = \frac{1}{\sqrt{2\pi}} \int_{-\sigma}^{\sigma} g(\kappa)e^{-i\kappa t}d\kappa \quad \text{for} \quad t \in \mathbb{R},
\]

with \( g \in L^2(-\sigma, \sigma) \). Then \( \hat{f} \) can be reconstructed from its samples \( \hat{f}(\frac{k\pi}{\sigma}) = \hat{f}_k \), taken at equally spaced nodes \( \frac{k\pi}{\sigma} \in \mathbb{R} \), via convolution with the sinc function

\[
\hat{f}(t) = d \sum_{k=-\infty}^{\infty} \frac{\sin(\sigma t - k\pi)}{\sigma t - k\pi} \hat{f}(\frac{k\pi}{\sigma}), \quad \text{i.e.} \quad \omega_i := \text{sinc}(\sigma t - k\pi).
\]

Since the support of the sinc function is infinite, the values of the interpolant \( I \) at each location depend on all data samples present in the volume. Thus ideal reconstruction is computationally expensive, especially for highly resolved data. For this reason usually approximations of the sinc function by weighting functions with finite support are used. The simplest example are box functions, that are centered at the data locations

\[
\omega_i(x) = \begin{cases} 
1, & \text{if } |x - x_i| < \frac{h}{2} \\
0, & \text{otherwise}
\end{cases}
\]

(2.12)
respectively the products \( \omega_i(x) := \omega_i(x_0)\omega_i(x_1) \) ... for higher dimensions. This results in so-called constant or nearest-neighbor interpolation

\[
I(x) = f_i \quad \text{for } x \in \text{supp}(\omega_i).
\]  

(2.13)

This interpolation is usually applied to cell-centered grid functions, which for example result from finite volume simulations. In this case the data often represents the average of some (conserved) quantity \( q(x) \) over the domain of cell \( \Omega_i \), i.e.

\[
f_i = \bar{q} := \frac{\int_{\Omega_i} q_i(x) dx}{\int_{\Omega_i} dx},
\]  

(2.14)

like for example mass density in hydrodynamic simulations. Of course the resulting global interpolant is discontinuous. Higher order interpolation for hexahedral cells is usually realized by shape functions that are (tensor-)products of the Lagrange polynomials

\[
L_i(x|x_0, x_1, \ldots, x_m) := \prod_{j=0; j \neq i}^{m} \frac{x - x_j}{x_i - x_j}.
\]  

(2.15)

Consider the one dimensional interval \([x_0, x_1]\) with data located at the boundaries, and set \( \omega_i(x) := L_i(x|x_0, x_1) \). Introducing the local coordinates \( \xi = \frac{x - x_0}{x_1 - x_0} \) with \( \xi \in [0, 1] \), one obtains \( w_0(\xi) = 1 - \xi \) and \( w_1(\xi) = \xi \). Hence Equation (2.11) yields the linear interpolant \( I(\xi) = f_0 + \xi(f_1 - f_0) \) for the interval.

For quadrilateral cells, the shape functions in local coordinates are analogously defined by

\[
\omega_{ij}(\xi, \eta) := L_i(\xi|0, 1) L_j(\eta|0, 1),
\]  

(2.16)

with vertex numbering according to Figure 2.10.

---

**Figure 2.10:** Local coordinates and numbering of vertices for hexahedral cells.
Hence the bilinear interpolant reads as follows

\[
I(\xi, \eta|f_{00}, \ldots, f_{11}) = \sum_{i=0}^{1} \sum_{j=0}^{1} \omega_{ij}(\xi, \eta) f_{ij}
\]  

(2.17)

\[
= L_0(\eta) (L_0(\xi)f_{00} + L_1(\xi)f_{10}) +
L_1(\eta) (L_0(\xi)f_{01} + L_1(\xi)f_{11})
\]

\[
= I(\xi|f_{00}, f_{10}) + \eta(I(\xi|f_{01}, f_{11}) + I(\xi|f_{00}, f_{10})).
\]

(2.18)

This shows that bilinear interpolation is equivalent to two linear interpolations along the \(x\)-axis followed by a linear interpolation along the \(y\)-axis. This is computationally more efficient than a direct evaluation of (2.17), since the number of multiplications is reduced from 12 to 4.

Completely analogously the shape functions for hexahedral elements and trilinear interpolation are defined by

\[
\omega_{ijk}(\xi, \eta, \rho) := L_i(\xi|0, 1) L_j(\eta|0, 1) L_k(\rho|0, 1),
\]

(2.19)

and the resulting trilinear interpolant for hexahedral cells reads

\[
I(\xi, \eta, \chi) = \sum_{i,j,k=0}^{1} \omega_{ijk}(\xi, \eta, \chi) f_{ijk} = ...
\]

(2.20)

\[
= I(\xi, \eta|f_{0,0,0}\ldots f_{1,1,0}) +
\chi (I(\xi, \eta|f_{0,0,1}, \ldots, f_{0,1,1}) - I(\xi, \eta|f_{0,0,0}, \ldots, f_{1,1,0})).
\]

(2.21)

(2.22)

Hence trilinear interpolation is equivalent to two bilinear interpolations parallel to the \(xy\)-coordinate plane, followed by a linear interpolation along the \(z\)-direction.

If in addition to the function values also information about the first derivative is available, \(C^1\)-continuous Hermite interpolation might be applied. Let us again consider the case of a one-dimensional interval \([x_0, x_1]\) with the function values \(f_0, f_1\) and first derivatives \(f'_0, f'_1\) given at the interval boundaries. Equation 2.11 reads as

\[
I(x|f_0, f_1, f'_0, f'_1) = f_0 H^3_0(x) + f'_0 H^3_1(x) + f_1 H^3_2(x) + f'_1 H^3_3(x),
\]

(2.23)

where the shape functions \(\{H^3_0, \ldots, H^3_3\}\) are given by the cubic Hermite polynomials. They can be defined by

\[
H^3_0(x_0) = 1, \quad \frac{d}{dx} H^3_0(x_0) = 0, \quad H^3_0(x_1) = 0, \quad \frac{d}{dx} H^3_0(x_1) = 0,
\]

\[
H^3_1(x_0) = 0, \quad \frac{d}{dx} H^3_1(x_0) = 1, \quad H^3_1(x_1) = 0, \quad \frac{d}{dx} H^3_1(x_1) = 0,
\]

\[
H^3_2(x_0) = 0, \quad \frac{d}{dx} H^3_2(x_0) = 0, \quad H^3_2(x_1) = 1, \quad \frac{d}{dx} H^3_2(x_1) = 0,
\]

\[
H^3_3(x_0) = 0, \quad \frac{d}{dx} H^3_3(x_0) = 0, \quad H^3_3(x_1) = 0, \quad \frac{d}{dx} H^3_3(x_1) = 1.
\]

Using local coordinates \(\xi \in [0, 1]\) they have the explicit form

\[
H^3_0(\xi) = 2\xi^3 - 3\xi^2 + 1,
\]

\[
H^3_1(\xi) = -2\xi^2 + 3\xi^3,
\]

\[
H^3_2(\xi) = \xi^3 - 2\xi^2 + \xi,
\]

\[
H^3_3(\xi) = \xi^3 - \xi^2.
\]

(2.24)
Piecewisely connected, cubic Hermite polynomials result in globally $C_1$-continuous interpolation for a one-dimensional grid $[\xi_0, \xi_1, \ldots, \xi_n]$. Often no information about the derivatives of the function is available. In this case the first derivative at a vertex $\xi_i$ might be approximated by the slope of two quadratic polynomials fitted through $(\xi_{i-1}, f_{i-1}), (\xi_i, f_i), (\xi_{i+1}, f_{i+1})$, respectively $(\xi_i, f_i), (\xi_{i+1}, f_{i+1}), (\xi_{i+2}, f_{i+2})$. For equidistant intervals, the resulting Catmull-Rom spline, reads as follows

$$I(\xi|\xi_0, \xi_1) = \frac{1}{2} \left( 2f_i + (f_{i+1} - f_{i-1}) \xi \right) + (2f_{i-1} - 5f_i + 4f_{i+1} - f_{i+2}) \xi^2 + (3f_i - f_{i-1} - 3f_{i+1} + f_{i+2}) \xi^3.$$

Similar to the bi-, and trilinear interpolation discussed above, interpolants for quadrilateral and hexahedral cells are generated via products of the one-dimensional cubic Hermite polynomials.
2.4 Spatial Access Methods

Figure 2.11: Decomposition of the AMR grid example given in Figure 2.6 into disjoint blocks that consist of cells from the same level of resolution, see also Section 5.2. Image (a) shows a result for a point-quadtree, whereas in (b) a region-quadtree is employed. The numbers indicate the order in which the first quadrants are inserted. In the region-octree example quadrants that lie outside the grid domain have been omitted.

For many visualization algorithms it is necessary to locate cells that enclose a particular spatial position, for example during the interpolation operation for volume rendering via raycasting. This point location operation has a complexity of $O(1)$ for uniform grids, since according to Equation (2.1), the index triple $(i, j, k)$ of the cell containing the location $x = (x_0, x_1, x_2)$ can be computed via

$$i = \left\lfloor \frac{x_0 - k_0}{h_0} \right\rfloor, \quad j = \left\lfloor \frac{x_1 - k_1}{h_1} \right\rfloor, \quad k = \left\lfloor \frac{x_2 - k_2}{h_2} \right\rfloor,$$

where $k$ denotes the grid offset vector and $h$ is the grid spacing. However, for unstructured grids no such simple relation holds and the simple $O(n)$ approach of inspecting each of the $n$ grid cells is unfeasible even for moderately sized grids.

In order to accelerate the performance of the point location and other spatial operations like intersection-, or adjacency-queries, dedicated spatial data structures are employed. They have in common that they decompose the search domain into a set of smaller, usually polyhedral subregions of simple geometry, which index into the set of spatial objects contained within the search domain. The decomposition is often organized in a hierarchical manner, which results in a logarithmic search complexity.

In the following we will sketch the most popular spatial data structures that are employed for these purposes. For more detailed information the reader might refer to textbooks, like [67, 71].
Figure 2.12: Decomposition of the hierarchy of Figure 2.6 using a kD-tree (a) and an adaptive kD-tree (b) data structure. The numbers indicate the order of the first partition axes.

Developed in 1974, quadtrees are among the first data structures investigated for spatial data access. Quadtrees and their three dimensional analog, octrees, are rooted trees, which recursively decompose the data domain into four axis-aligned rectangles, respectively eight subvolumes. Two types of quadtrees are distinguished in the literature: region- and point-quadtrees. Point-quadtrees are primarily used to store collections of points. For each point that is inserted in the tree, the leaf node that contains the point is subdivided into four disjoint axis-aligned subnodes, such that the newly inserted point coincides with the intersection of the partition hyperplanes. In contrast to this, nodes of region quadtrees (octrees) are always subdivided into subregions of the same size. Figure 2.11 shows two examples.

Besides to accelerate the spatial operations mentioned above, (region) octrees are often employed to generate multi-resolution representations of volumetric datasets. Here the root node represents the coarsest resolution of the data, which is recursively refined, until the original data resolution is reached. In this context a full octree, i.e. an octree where all leaf nodes are located on the same level, is called a pyramidal representation of the original data.

Bin-trees are similar to region-quadtrees, with the difference that each internal node is recursively split into two equal sized subregions along one axis-aligned hyperplane. The splitting direction alternates from one level of the tree to the next. kD-trees, which operate on k-dimensional domains, are a generalization of bin-trees in the sense that each non-leaf node is subdivided into two subregions of potentially different size. As for bin-trees the (k-1)-dimensional hyperplane is aligned with one of the major coordinate axes, with a cyclic change of orientation from level to level, compare Figure 2.12 (a).

The adaptive kD-tree variant relaxes the restriction of the alternating orientations of the dividing planes. Here the direction of the next axis-aligned plane at an internal node is
unrelated to the depth of the node and might rather be changed in an flexible way, compare Figure 2.12 (b). *Binary-Space-Partition trees* or just *BSP-trees* are even more general, in the sense that the division planes do not have to be axis-aligned, but may rather have an arbitrary orientation.

*R-trees* are well suited for representing hierarchies of overlapping d-dimensional intervals. Each internal node stores the minimal axis-aligned rectangle or bounding box that encloses all the object indexed by its children.

The R-tree is a height-balanced tree, with a maximal height of \( \log_m(n) \) for \( n \) objects in the index set. A R-tree has a degree of \((m, M)\), where \( m, M \in \mathbb{N} \) and \( m \leq \frac{M}{2} \), if each interior node has a number of children that ranges between \( m \) and \( M \). The root node has to contain at least two children. In contrast to R-trees, sibling nodes of \( R^+ \)-trees do not overlap. This might be achieved by clipping objects that intersect more than one interval on the same level. This results in an increased search efficiency, since point queries require traversing only one path through the tree, but on the other hand increase the storage requirements due to a potentially higher number of nodes.

The examples given in Figures 2.11 to 2.13 suggest that adaptive kd-trees are well suited for decomposing AMR hierarchies into non-overlapping blocks of constant resolution, due to the small number of resulting regions. Compared to the \( R^+ \)-tree example that also requires a relatively small number of blocks, they have the further advantage to guarantee that the regions can be traversed in a view-consistent order, which is advantageous for volume rendering approaches, as discussed in Section 4.4.9. In Section 5.2 we will propose an algorithm that utilizes an adaptive kd-tree for such a decomposition.

Figure 2.13: Decomposition of the hierarchy of Figure 2.6 using a \( R^+ \)-tree. To avoid visual clutter only the first two levels are shown.
Chapter 3

Visualization Methods for Scalar Data

The aim of data visualization is to support cognition of and enable insight into given input data by means of visual representations [81, 86]. Historically two major areas are distinguished in this context, namely scientific visualization and information visualization, depending on the type of data that is processed. Though not strictly defined, the term scientific visualization usually refers to the visualization of data that has a more or less inherent representation in space and time, like for example discrete data defined on spatial, possibly time-dependent, computational grids. In contrast to this, information visualization addresses data that does not possess a natural spatial representation. However, there are numerous examples that show that this strict distinction is not always the best choice and there is still research on defining better taxonomies for this field [19, 29]. One alternative approach is to base the classification on the employed visualization algorithms, rather than on the data itself, compare [85].

Nowadays mainly computers are employed for generating and displaying the resulting output representations, which tightly connects the field of data visualization to computer science, in particular to computer graphics. In the next section we will briefly discuss some technical issues of the visualization process and graphics hardware in general, whereas Section 3.2 will address the most popular visualization methods for volumetric scalar data.

3.1 The Visualization Pipeline

The visualization process can be formally characterized by the so-called visualization pipeline, which is subdivided into three main stages, namely filtering, mapping and rendering, compare Figure 3.1.

- During the filtering stage the raw input data is converted into a format which is better suited for the later stages. This might involve tasks like interpolation in order to generate missing data samples, data reduction, for example by subset and component selection and/or dimensionality reduction as well as the extraction of topological features of the data.
• In the *mapping stage* the filtered data is mapped onto graphical primitives, which nowadays usually are directly supported by graphics hardware. Examples are the approximation of surface data by triangular meshes or volumetric data represented as dense ensembles of points. The primitives may be equipped with additional attributes like color and transparency information.

• In the last stage, the *rendering stage*, the two dimensional image is generated from the graphical primitives. This involves tasks like the rasterization of the primitives into a pixel representation, the culling of occluded subregions as well as shading and blending operations.

Nowadays the rendering stage is often accelerated utilizing dedicated graphics hardware, which allows to perform sophisticated visualization tasks at interactive frame rates even for a large number of geometric primitives.

In particular for these hardware-accelerated graphics architectures the rendering stage itself is usually subdivided into three stages, namely *vertex operations*, *rasterization* and *fragment operations*, see Figure 3.2.

In the first stage the illumination computations are performed and linear transformations like rotations, translations and scaling are applied to the vertices of the graphics primitives in order to place the geometry according to the actual viewpoint settings. Further parts of the geometry that are outside the actual viewing volume are removed and the projection into screen space is carried out in this stage.

In the *rasterization stage* the primitives are converted into so-called *fragments*, which correspond to pixels in the output images (*scan conversion*). Visual attributes like color and texture coordinates are determined for each fragment. Before this information is actually written into the frame buffer, each fragment has to pass several tests performed in the last stage of the rendering pipeline. In particular z-buffer-based depth sorting is carried out at this point and fragments may be combined with the pixel data in the frame buffer, for example to realize blending effects for semi-transparent objects. Further the so-called stencil buffer can be used to mask out portions of the frame buffer, in order to prevent that these pixels are being replaced by other fragments.
In order to abstract and facilitate the access to different graphics hardware architectures, the hardware is usually not programmed directly but rather via an API (application programming interface), which acts as a layer between the application software and graphics hardware. Nowadays two main graphics APIs are supported by hardware manufacturers, namely OPENGL [74], introduced in 1992 by SILICON GRAPHICS, which is available on almost every platform and supported by many programming languages and DIRECTX [7], developed by MICROSOFT and available only for WINDOWS platforms.

The implementation of the algorithms presented in this thesis was done within the framework of AMIRA [80, 1], which is based upon OPENGL.

### 3.2 Visualization Methods for 3D Scalar Data

In the following we will review the most important rendering methods for volumetric scalar data, i.e. functions $f : \Omega \subset \mathbb{R}^3 \rightarrow \mathbb{R}$. These are usually categorized as indirect or direct volume rendering. Indirect methods convert the data into some auxiliary, usually polygonal representation in the mapping stage of the rendering pipeline. This often involves some form of dimensionality reduction. A standard example is the 'marching cubes' algorithm [52] for isosurface extraction, which will be discussed in more detail in Section 3.3. A drawback of indirect methods is that they usually allow to visualize only a small subset of the data at once.

In contrast to this, for direct volume rendering approaches, which are discussed in Section 3.4, in principle every data sample may contribute to the final image. This is achieved by assigning physical quantities like absorption and emission coefficients to each data sample and by modeling the transport of light traveling through the resulting participating medium. In particular the intensity distribution of the light in the image plane is computed and displayed. An example are raycasting methods, as discussed in Subsection 3.4.3.
3.3 Indirect Volume Rendering

3.3.1 Slice-Based Techniques

Slicing is a simple, but nevertheless popular indirect technique, which displays the data values $f(x(\lambda, \mu))$ within the intersection of the data volume $\Omega$ and an arbitrarily oriented plane

$$\mathcal{P} = \{ x(\lambda, \mu) \mid x(\lambda, \mu) = a + \lambda v_1 + \mu v_2 \}.$$  

Often a colormap $C$ is employed to assign color and possibly transparency information to each scalar value. In computer graphics it is usually represented by a quadruple of three color scalars $C_i \in [0, 1], i = r, g, b$ for the red, green and blue color components, as well as opacity value $C_\alpha \in [0, 1]$.

$$C : \text{Im}(f) \mapsto \mathbb{R}^4, \text{ with } f(x) \mapsto (C_r, C_g, C_b, C_\alpha).$$

Figure 3.3: Scalar field visualization using combined height field and slicing techniques. (dataset courtesy of M. Norman, National University of California)

Nowadays implementations often utilize 2D texture capabilities, supported on almost every modern graphics hardware, for rendering the resulting slices, which allows real-time interaction even if a large number of highly resolved slices has to be displayed simultaneously, compare Section 3.5.
In order to further enhance the perception of small differences of data samples, so-called *height fields* or *carpet plots*, are well suited. The data within the slice is rendered as a curved surface, which is computed as follows

\[ s(\lambda, \mu) = x(\lambda, \mu) + k \cdot f(x(\lambda, \mu)) \frac{v_1 \times v_2}{|v_1||v_2|}, \]

where \(k\) is a scaling constant. In addition the surface points might be color-coded to allow a better comparison of the height of distant locations on the surface, compare Figure 3.3.

### 3.3.2 Isosurface Extraction

Another very popular visualization method for scalar data is the rendering of the function’s *level sets* for a certain level \(v_{iso}\):

\[ \mathcal{L}(v_{iso}, f) := f^{-1}(v_{iso}) = \{ x \in \Omega \mid f(x) = v_{iso} \}. \]

It is important to distinguish between the isosurface \(S_{orig}\) of the original, sampled function \(f\), the isosurface \(S_{int}\) of the interpolant that is employed for the reconstruction and the approximation \(S_{app}\) of the latter by graphical primitives.

There exist a number of natural requirements for this approximation \(S_{app}\). It should at least be \(C^0\)-continuous for continuous interpolants and topologically consistent with \(S_{int}\), i.e. the same discrete grid points are separated by both surfaces. Further the approximating surface should be invariant under the sign inversion operation

\[ f_{ijk} \rightarrow (-f_{ijk}), v_{iso} \rightarrow (-v_{iso}). \]

In addition the surface should yield a ‘good’ approximation, in the sense that the difference between \(S_{int}\) and \(S_{app}\) is small, it should allow for efficient computation and require as few polygons as possible.

The standard algorithm for extracting isosurfaces of grid functions defined on hexahedral (quadrilateral) cells is the so-called *Marching Cubes* (*Marching Squares*) algorithm, first proposed in 1987 by Lorensen et al. [52]. In this approach the isosurface of the piecewise trilinear interpolant is approximated by a triangular mesh, compare the Figure 3.4.

An advantage of the marching cubes method is that it requires only local information about the grid function for the surface construction. Each grid cell is inspected for intersection with the surface, based on a classification of the cells vertices. Vertices are classified as inner and outer ones, depending on their scalar value being below or above the isovalue. Since trilinear interpolation is employed, a cell is intersected by the isosurface \(S_{int}\) if and only if the cell contains inner and outer vertices.

Identifying configurations that can be mapped onto each other by rotation and mirroring operations, the 256 possible cases for hexahedral cells can be grouped into 15 equivalence classes. For each of these classes the topology of the isosurface is approximated by up to four triangles, with nodes located at the edges of the cell, as shown in Figure 3.5.

---

1 We will use the term *isosurfaces* for the two-dimensional case, respectively *isolines* or *isocontours* for the one-dimensional case as a synonym for the level sets in the following.
In order to accelerate the surface extraction process, the triangulation for each case is precomputed and stored in a lookup table. Once the topological configuration of the triangle patch is determined, the positions of the triangle vertices are computed by linear interpolation between the values at the vertices on the intersected cell edges.

The local triangulations sketched in Figure 3.5 can violate the requirement of global continuity listed above, since they might yield inconsistent surface patches for adjacent cells. These become visible as artificial holes that are not present in the exact isosurface $S_{\text{int}}$ of the piecewise trilinear interpolant. This problem arises for faces that have the same classification for opposite vertices but different classifications at edge ends, namely the cases 3, 6, 7, 10, 12 and 13 of Figure 3.5. For these cases all four edges of the face are intersected and the correct connection of these intersection points can not be determined solely based on by the four node values, compare Figure 3.6.

A simple method to generate a continuous triangulation is to modify the lookup table in order to ensure that the same edges are connected on both sides of adjacent cells. Notice that this operation does not guarantee a triangulation that is topologically consistent with the surface $S_{\text{int}}$ and that it is not invariant under the sign inversion operation.

Nielson et al. [58] presented a more sophisticated solution to this problem. It is based on an inspection of the bilinear interpolant in the face domain. For the problematic cases mentioned above, the intersection between the isosurface $S_{\text{int}}$ and the face domain is given by two hyperbolas. The face edges are connected differently, depending on whether the interpolated value at the intersection between the both asymptotics of the hyperbola is below or above the given isovalue. Hence an evaluation of the interpolant at this additional location can be used to determine the correct connection of the intersected edges and guarantee continuity across cell faces.

Another approach is to decompose cells with problematic configurations into sets of tetrahedra with consistent edges on both sides of the face. The variant of the marching
Figure 3.5: Using symmetry operations the 256 triangle patch configurations for hexahedral cells can be reduced to 15 topologically different ones.

cubes algorithm for this cell type, which is also called marching tetrahedra [13], generates triangulations with coinciding isocontours on common interfaces between adjacent tetrahedra. However, there still remains some freedom in how the decompositions is carried out, resulting in slightly different surface approximations and the number of generated triangles in increased. A comprehensive overview about these and other methods to obtain consistent triangulation for the marching cubes approach is given in [32].

A lot of work has been carried out in the field of isosurface visualization in the last decades and it is still an active area of research. An important problem that was addressed is the optimization of the surface extraction phase. Livnat et al. presented the “Near Optimal Isosurface Extraction” (NOISE) algorithm for optimized isosurface generation from structured and unstructured grids [50]. They use a span space representation of the data domain to obtain a worst case complexity of $O(\sqrt{n} + k)$, where $n$ is the number of cells of the dataset and $k$ is the number of cells that are intersected by the considered isosurface. Livnat et al. further proposed an algorithm for view-dependent isosurface extraction in [49], where only the visible portion of the isosurface is extracted, accelerating both, the extraction as well as the rendering phase. Chiang et al. [20, 21] presented out-of-core isosurface generation approaches for datasets that are too huge to be held in main memory. They generate search data structures in a preprocessing step in order to minimize
I/O operations during surface extraction by restricting disc access to data of cells that are intersected by the considered isosurface.

In the recent years point-based techniques have been employed for accelerating the rendering phase of the isosurface visualization. Ji et al. presented a pure point-based approach for efficient isosurface rendering for remote data in [34]. Livnat et al. [51] present a hybrid approach, in which sub-pixel triangles are replaced by point primitives.

Kobbelt et al. [41] proposed an extension of the marching cubes approach that avoids aliasing artifacts at sharp features on the extracted surfaces.

While the work reviewed so far is based on isosurface representations by triangle or point primitives, Westermann et al. [91] proposed a fundamentally different approach for real-time extraction and rendering of lighted and shaded isosurfaces using texture mapping hardware. This approach does not require a polygonal representation of the surface. Further a lot of work has been carried out in the field of multi-resolution isosurface extraction. We will give an overview about this in Section 5.1.
3.4 Direct Volume Rendering

As mentioned above, direct volume rendering approaches assign radiometric quantities, that depend of the considered grid function \( f \), to each point \( x \in \Omega \) and compute the resulting intensity distribution in the image plane. The governing equations of this process can be derived within the framework of linear transport theory [18]. In this section we will closely follow the discussion presented in [31]. Another good presentation of the underlying physical models can be found in [55].

Let us first introduce some radiometric definitions needed in the following. The basic quantity in radiometry is the specific intensity \( I \), which is also called radiance. It completely describes the angle and frequency dependency of the radiation field at each point, such that

\[
dE = I(x, n, \nu) \cos \vartheta \, da \, d\Omega \, d\nu \, dt
\]
gives the amount of radiant energy per time unit \( dt \) and frequency interval \( d\nu \) that emerges at the location \( x \) and is radiated into the solid angle \( d\Omega \) in the direction \( n \) through the surface element \( da \). Here \( \vartheta \) defines the angle between \( n \) and \( da \), compare Figure 3.7.

Another important quantity which describes the decrease of radiation traveling through material is the absorption or extinction coefficient \( \chi \). It is defined such, that

\[
dE^{(ab)} = \chi(x, n, \nu)I(x, n, \nu) \, ds \, da \, d\Omega \, d\nu \, dt
\]
yields the amount of energy removed from a beam with radiance \( I(x, n, \nu) \) passing through a cylinic volume element of length \( ds \) with a cross section \( da \). It is convenient to split the absorption coefficient into the so-called true absorption coefficient \( \kappa \) and the scattering coefficient \( \sigma \)

\[
\chi(x, n, \nu) = \kappa(x, n, \nu) + \sigma(x, n, \nu).
\]

Figure 3.7: Illustration adopted from [31].

This reflects the fact that there are two main sources for absorption, namely true or thermal absorption, a process that converts radiation energy into thermal energy of the material, and scattering, whereby incoming light is redirected after interaction with the atoms of the material. The latter process usually also includes a change of the light frequency.

Analogously the emission coefficient \( \eta \) is defined such, that

\[
dE^{(em)} = \eta(x, n, \nu) \, ds \, da \, d\Omega \, d\nu \, dt
\]
is the amount of radiation energy emitted per time unit and frequency interval \( d\nu \) by a cylindrical volume element with length \( ds \) and cross section \( da \) at \( x \) into the solid angle.
$d\Omega$ in direction $\mathbf{n}$. Like the absorption coefficient, the emission can be split up into two parts, the thermal emission coefficient $q$ and the scattering part $j$

$$\eta(x, n, \nu) = q(x, n, \nu) + j(x, n, \nu).$$

The phase function $p(x, n, n', \nu, \nu')$ relates the amount of incoming radiant energy from direction $\mathbf{n}$ and with the frequency $\nu$ to the amount of energy with frequency $\nu'$ that is scattered into the direction $\mathbf{n}'$

$$dE^{(\text{scatt})} = (\sigma I ds da d\Omega d\nu dt) \times (p(x, n, n', \nu, \nu') d\Omega' d\nu').$$

So the part of the emission which is due to scattering is given by

$$j(x, n', \nu') = \int \int p(x, n, n', \nu, \nu') \sigma(x, n, \nu) I(x, n, \nu) d\nu d\Omega.$$

The equation of transfer can be derived from the assumption that the change of radiation energy at each location within the radiation field is equal to the amount of emitted energy at that location, reduced by the amount of absorbed and scattered energy:

$$\{I(x, n, \nu) - I(x + ds, n, \nu)\} ds da d\Omega d\nu dt = \{-\chi(x, n, \nu) I(x, n, \nu) + \eta(x, n, \nu)\} ds da d\Omega d\nu dt.$$

With $x(s) := p + sn$, where $p$ is some reference point at the boundary of the radiation field, see Figure 3.8, we immediately obtain the differential formulation of the equation of transfer by $ds \to 0$

$$\frac{\partial}{\partial s} I(x, n, \nu) = -\chi(x, n, \nu) I(x, n, \nu) + \eta(x, n, \nu).$$

![Figure 3.8: adopted from [31].](image)

The following (formal) solution to Equation 3.4 is called the integral formulation of the equation of transfer

$$I(x(s), n, \nu) = I(x(s_0), n, \nu) e^{-\tau_\nu(x(s_0), x(s))} + \int_{s_0}^{s} \eta(x(s'), n, \nu) e^{-\tau_\nu(s', s')} ds'.$$
where the optical depth $\tau_\nu(s_1, s_2)$, between the two points $x(s_1) = p + s_1 n$ and $x(s_2) = p + s_2 n$ is defined as

$$\tau_\nu(s_1, s_2) = \int_{s_1}^{s_2} \chi(x(s')) ds'.$$  \hspace{1cm} (3.6)

The formulation (3.5) allows the following intuitive interpretation: The total specific intensity at the location $x$ emitted into the direction $n$ consists of two parts: the sum of all specific intensity emitted along the ray segment $p + s ||x-p|| n$ for $s \in [0,1]$, which is attenuated due to absorption along the ray and the attenuated background intensity that is emitted at the boundary location $x_0$ into direction $n$.

### 3.4.1 Transfer Functions

As discussed in the last section the underlying physical model for direct volume rendering requires the specification of absorption and emission coefficients $\chi(x)^2$, respectively $\eta(x)$ at each location within the data volume. These mappings are also called transfer functions in this context, whereas the process of mapping data values to radiometric coefficients is called classification.

In general the transfer functions are not given as analytical expression, but are rather specified via four user-defined lookup tables; three for the emission coefficients for red, green and blue frequency interval components and one for the absorption coefficients. Intermediate values are obtained from the these entries by interpolation.

Let the transfer function be denoted by $T : Im(f) \rightarrow \mathbb{R}^4$ with

$$f \mapsto T(f) = (\eta_r(f), \eta_g(f), \eta_b(f), \chi(f)).$$

There are two ways of performing the classification, which correspond to the order in which interpolation and the mapping to the radiometric coefficients are carried out: For so-called pre-classification the discrete data samples are first mapped to radiometric quantities, followed by an interpolation of the resulting emission and absorption coefficients ($I_{\text{color}}$)

$$C_{\text{pre}}(x) = I_{\text{color}}(T \circ f \mid p_0, \ldots, p_n)(x).$$

Here $p_i$ denote the sample locations of $f$ that contribute to the interpolated function value at the considered location $x$. In contrast to this, the term post-classification is used if the transfer function is applied after interpolation of the data samples ($I_{\text{data}}$)

$$C_{\text{post}}(x) = T \circ I_{\text{data}}(f \mid p_0, \ldots, p_n)(x).$$

Since in general $C_{\text{pre}}(x) \neq C_{\text{post}}(x)$ holds, the question which order is preferable arises. The answer is provided by sampling theory, see also Section 2.3.

\[\text{2In order to ease the discussion we will neglect the direction and frequency dependency of the coefficients in this section.}\]
Let us assume that the original signal $f(x)$ was sampled with a frequency $\nu > \nu_{\text{max}}(f)$. In this case in principle an ideal reconstruction of $f$ according to Theorem 1 is possible. However, this is in general not the case for the composed function $T \circ f$, since its Fourier spectrum may contain frequency components larger than $\nu_{\text{max}}$, which prohibits a faithful reconstruction of $T \circ f$.

In contrast to this, post-classification allows an ideal reconstruction. In this case the original signal $f(x)$ is reconstructed first and next the composed signal $T \circ I(f)$ has to be interpolated using a sample distance that corresponds to the highest frequency components in the spectrum of the composed function. A direct comparison of volume rendering via pre- and post-classification is shown in Figure 3.9.

### 3.4.2 Emission-Absorption Models

Due to the emission term $\eta(x, n, \nu)$, that at each location $x$ takes into account the amount of incident, scattered light from all possible directions, solving Equation 3.5 is a computationally intensive task, that does not allow for interactive image generation rates even for moderately sized datasets.

The simplified emission-absorption model, introduced by Sabella in 1988 [70], is an attempt to reduce this complexity. In this approach the scattering of light is completely ignored and it is assumed that the emission and absorption coefficients are isotropic.
Hence Equation 3.2 and 3.1 reduce to \( \eta(x, n) = q(x) \), respectively \( \chi(x, n) = \kappa(x) \). Notice that this includes that any frequency dependency can be omitted, since according to Equation 3.3 the absence of scattering prohibits the mixing of frequencies. Hereby Equation 3.5 simplifies to

\[
I(s) = I(s_0)e^{-\tau_\nu(s_0, s)} + \int_{s_0}^{s} q(s')e^{\tau(s', s)}ds'.
\] (3.7)

According to \( \tau(s_1, s_2) := \int_{s_1}^{s_2} \kappa(s) ds \), the optical depth between \( x(s_1) \) and \( x(s_2) \) depends solely on the true absorption coefficient \( \kappa(s) \).

There exist various approaches for solving this equation numerically, which may be classified as image-order or object-order methods. In image-order approaches, also called raycasting, the resulting intensity for each image pixel is computed by integrating the simplified equation of transfer 3.7 along a ray through the viewpoint and the pixel location. Raycasting will be discussed in more detail in Subsection 3.4.3.

In contrast to this, object-order approaches traverse the cells of the data in a specific order and composite the contributions of each cell to the final image. Splatting [93] is an example for this. In this approach for each voxel a semi-transparent polygonal surface primitive called footprint is composited onto the image plane, usually in a back-to-front order. Also texture-based approaches, which leverage the texture units of modern graphics hardware, fall into this category. We will discuss them in more detail in Section 3.5. Another example is cell-projection [94], which can be viewed as object-order raycasting. In contrast to the latter one the ray-integration is carried out on a per-cell bases, followed by a step in which the separate ray-segments are merged, in order to obtain the final pixel intensities.

The shear-warp-algorithm [42] incorporates aspects of both, image and object order approaches. The basic idea is to shear the axes-aligned slices of the data volume, such that the rows of voxels are aligned with rows of pixels of an intermediate image. The sheared slices are composited along one of the major axes, replacing trilinear by bilinear interpolation within each slice. In a last step the intermediate image is transformed (“warped”) in image space to generate the final image. The shear-warp approach is considered as the fastest software-based volume rendering algorithm. A drawback is that three copies of the dataset have to be kept in main memory during rendering, one set of slices perpendicular to the three major axes. Further the shear-warp approach tends to suffer from artifacts due to the simplified interpolation scheme.
3.4.3 Raycasting

Since in general Equation 3.7 has no analytical solution, in most cases the integration has to be carried out numerically. Therefore the whole ray interval is divided into a set of subintervals $[s_i, s_{i+1}], i = 0, \ldots, n - 1$. Here $s_n$ corresponds to the camera position, as indicated in Figure 3.10, and $s_0$ is the parameter where the ray enters the data volume.

Notice that subintervals do not necessarily have to be of equal lengths. In Section 4.3 we will present a raycasting approach where the intervals are chosen adaptively using local error criteria. According to Equation 3.7 the specific intensity at $s_k$ is associated to the one at $s_{k-1}$ as follows:

$$I(s_k) = I(s_{k-1})e^{-\tau(s_{k-1},s_k)} + \int_{s_{k-1}}^{s_k} q(s)e^{-\tau(s,s_k)}ds.$$  \hspace{1cm} (3.8)

We define the transparency of the ray-segment $[s_{k-1}, s_k]$ by

$$T_k := e^{-\tau(s_{k-1},s_k)}$$  \hspace{1cm} (3.9)

and its emission by

$$b_k := \int_{s_{k-1}}^{s_k} q(s)e^{-\tau(s,s_k)}ds.$$  \hspace{1cm} (3.10)

Setting $b_0 := I(s_0)$ (background intensity), Equation 3.8 can be rewritten as

$$I(s_n) = I(s_{n-1})T_n + b_n$$  \hspace{1cm} (3.11)

$$= (I(s_{n-2})T_{n-1} + b_{n-1}) T_n + b_n$$

$$= \ldots$$

$$= (((((\ldots(b_0)T_1 + b_1)T_2)\ldots)T_{n-1} + b_{n-1}) T_n + b_n$$

$$= \sum_{k=0}^{n} (b_k \prod_{j=k+1}^{n} T_j ) = \sum_{k=n}^{0} (b_k \prod_{j=n}^{k+1} T_j )$$  \hspace{1cm} (3.12)

This gives rise to two kinds of recursive evaluation schemes for the ray integration. Equation 3.11 motivates a summation starting at the rays entry point to the camera position (back-to-front), as indicated by the following piece of pseudo code:
According to (3.12) the summation may alternatively be performed in the opposite front-to-back order:

\[
\begin{align*}
I & \leftarrow b_n; \\
T & \leftarrow T_n; \\
\text{for} \ (k = n - 1; \ k > 0; \ k = k - 1) \ \{ \\
I & \leftarrow I + b_k T; \\
T & \leftarrow T_k T; \\
\} \\
I & \leftarrow I + b_0 T;
\end{align*}
\]

Though for the front-to-back traversal besides the accumulated intensity also the accumulated transparency has to be computed, it has the advantage that the summation can be stopped once the accumulated transparency is small enough, so that the contribution of the remaining segments does not change the resulting intensity significantly (early ray termination).

### 3.5 Texture-Based Volume Rendering

Another simplification of Equation 3.8 is possible for absorption and emission coefficients \(\kappa(s)\), respectively \(q(s)\), that are piecewise constant within each ray-segment \([s_{k-1}, s_k]\). In this case Equations 3.9 and 3.10 reduce to

\[
T_k = e^{-\int_{s_{k-1}}^{s_k} \kappa(s) ds} = e^{-\kappa_k \Delta s},
\]

where \(\Delta s := s_k - s_{k-1}\), and

\[
b_k = \int_{s_{k-1}}^{s_k} q_k e^{-\int_{s_{k-1}}^{s_k} \kappa(s) ds} ds
\]
\[
= q_k \int_{s_{k-1}}^{s_k} e^{-\kappa_k (s_k - s)} ds
\]
\[
= \frac{q_k}{\kappa_k} (1 - T_k).
\]

Combining this with Equation 3.11 yields

\[
I(s_n) = I(s_{n-1}) T_n + \frac{q_n}{\kappa_n} (1 - T_n).
\]  

Although the underlying emission-absorption model removes much of the original complexity of Equation 3.5, the performance of raycasting is still limited by the large number
of interpolation and compositing operations that have to be performed during the integra-

The formulation 3.13 allows the utilization of texturing capabilities offered by modern
graphics hardware in order to accelerate the interpolation and accumulation step. Since
standard graphics hardware supports only polygonal rendering primitives \(^3\), the data
volume has be mapped onto a set of polygons, called proxy geometries in this context. Usually
planar slices, aligned with the data volume (2D texture-based volume rendering) or
perpendicular to the viewing direction (3D texture-based volume rendering), are em-
ployed as discussed in Subsections 3.5.1 and 3.5.2. Texture hardware interpolates the
color and transparency values for the polygon fragments within the rasterization stage.
The slices are blended in a back-to-front order in the frame buffer, according to the blending
equation

\[
C_{\text{comp}} = (1 - \alpha) C_{\text{old}} + \alpha C_{\text{new}},
\]

which is usually also supported by the graphics hardware. Here \(C_{\text{old}}\) is the color triple
of the previous fragment stored in the frame buffer, and \(\alpha\) and \(C_{\text{new}}\) are the opacity,
respectively the colors of the incoming fragment.

The basic principle of texture-based volume rendering is that Equation 3.14 is equiv-
alent to 3.13 if one identifies

\[
C_{\text{new}} = \frac{q_n}{\kappa_n},
\]

\[
C_{\text{old}} = I(s_{n-1}),
\]

\[
C_{\text{comp}} = I_n,
\]

\[
\alpha = 1 - T_n = (1 - e^{-\kappa \Delta s}).
\]

So storing \((\frac{q_{\text{red}}}{\kappa}, \frac{q_{\text{green}}}{\kappa}, \frac{q_{\text{blue}}}{\kappa}, (1 - e^{-\kappa \Delta s}))\) as the texture’s RGBA-components, blending
according to (3.14) approximates the solution of the Equation 3.7 via Riemann summation.

Notice that the last component depends on the sample distance \(\Delta s\), which requires
the adaption of these components, respectively the associated colormap, if the sample
distance is altered.

### 3.5.1 2D Texture-Based Volume Rendering

In the 2D texture approach three stacks of planes parallel to the three coordinate planes
are generated in a preprocessing step (object-aligned slices), as indicated in Figure 3.11. Within
these proxy geometries bilinear interpolation is employed. During rendering the
stack which is most perpendicular to the actual viewing direction is blended back-to-front,
according to Equation 3.14.

\(^3\)There exist also specialized hardware solutions dedicated to volume rendering, like the VOLUMEPRO
graphics board [65], that provide direct hardware implementations for raycasting.
Figure 3.11: For volume rendering via 2D textures, three stacks of planar slices are employed. For each viewpoint, the set that is most perpendicular to the viewing direction is rendered.

The advantage of this approach is that hardware support for 2D textures is available on almost all consumer graphics cards. Since bilinear interpolation is applied, the graphics performance is high, provided that enough texture memory is available to hold all three stacks of slices simultaneously in graphics memory.

Drawbacks are artifacts that may arise due to the restriction to bilinear interpolation and at viewing angles at which the set of texture stack has to be changed. Further the assumption of a constant slice distance is only fulfilled for parallel projection and a viewing direction which is (anti-) parallel to one of the coordinate planes, though this effect is usually less apparent than the artifacts due to changing the stack of textures. Another drawback of this approach is the high (texture-)memory consumption, since three stacks of slices have to be generated.

3.5.2 3D Texture-Based Volume Rendering

Some of the drawbacks of 2D texture-based volume rendering can be eliminated or at least reduced, if 3D textures are supported by the graphics hardware. In the 3D texture approach for volume rendering [17, 95] the set of proxy geometries is not precomputed, but is rather generated on-the-fly. According to the actual viewpoint, the data is sampled on slices perpendicular to the actual viewing direction (view-aligned slices), see Figure 3.12. Due to this smooth adaption of the sampling positions, artifacts caused by the abrupt change of the texture stacks as for 2D textures are avoided.

The 3D texture hardware supports fast trilinear interpolation of texture samples and further allows to vary the slice distance interactively. For parallel projection correct sampling distances are obtained for all rays and all viewing directions. Though for perspective projection the distance is correct only for the rays parallel to the viewing direction, the small errors are hardly visible for normal viewing angles. An computationally more expensive approach that utilizes tessellated spherical shells centered at the viewpoint as
Figure 3.12: For volume rendering via 3D textures, a set of slices that are perpendicular to the actual viewing direction is extracted on-the-fly and blended in the frame buffer. Proxy geometries is described in [43]. It is intended for immersive environments that usually involve large viewing angles.

Texture-based approaches may suffer from the limited precision of the frame buffer and the color lookup-table, which may cause round-off errors for intermediate results in the blending step. This can result in visible artifacts especially if a large number of highly transparent slices is blended in the frame buffer. Figure 3.13 shows a comparison between two hardware-supported renderings performed with 8-, and 12-bit precision per color-channel and a pure software-based renderer that carries out all computations using floating point precision. At the time this thesis was written, graphics hardware with full hardware-supported floating-point precision frame-buffer blending was announced by the leading hardware manufactures.

Further problems arise if the 3D texture is too large to fit into memory. In this case it is subdivided into smaller sub-textures, usually called texture bricks, which are rendered separately in a view-consistent order. To avoid artifacts at the boundaries, adjacent texture bricks have to share a layer of texels, to allow for consistent trilinear interpolation. Bricking usually results in a large performance decrease, since for each frame multiple bricks have to be transferred between main and graphics memory.

Pre-classification as discussed in Subsection 3.4.1 is realized if 4-channel textures are used, storing three color and one transparency value per texel. In this case the interpolation is carried out on the colors. In addition to the general drawbacks of pre-classification, the high memory requirements of at least four bytes per texel are disadvantageous, in par-
Figure 3.13: Comparison of the rendering quality of texture-based approaches using 8-Bit (a) and 12-Bit (b) blending precision and a pure software-based raycasting approach (c) for a SAMR dataset. In (a) a discontinuous change of the overall transparency for the different levels is clearly visible, caused by round-off errors due to small opacity values for the higher resolved levels according to Equation 5.2. This effect is less apparent in Figure 3.13 (b) and not noticeable for the floating point precision rendering in (c). (dataset courtesy of G. Bryan, Princeton University)

particularly since one of the limiting factors of texture-based approaches is the available texture memory. Further a change of the transfer function requires a complete redefinition of the textures, which decreases the performance due to the limited memory bandwidth between main- and graphics-memory.

However, recent graphics hardware architectures allow post-classification also for texture-based approaches. On Silicon Graphics workstations, it is supported directly via the so-called SGI_TEXTURE_COLOR_TABLE extension. Instead of four-channel (RGBA)-textures, one-channel 3D textures are employed, which store indices into a color lookup table. The color lookup for each fragment is performed after trilinear interpolation within the one-channel 3D texture. On consumer graphics hardware this mechanism can be realized by means of dependent textures, as described in detail in [68].

Texture-based volume rendering has been greatly improved in the last years. Engel et al. [26] utilize multi-textures and pixel-shader capabilities of recent graphics boards for efficient texture-based volume rendering for high-frequency transfer functions, an approach known as pre-integration. Kniss et al. [39] presented interactive volume rendering for multi-dimensional transfer functions, based on data value, gradient magnitude and the second directional derivative. Kniss further presented a hardware-accelerated shading model for volumetric light attenuation effects to produce shadows and translucency effects in [40]. Of course this is only a small fraction of the work that has been carried out in the field of volume rendering in the last years. We will further summarize research on hardware-accelerated multi-resolution volume rendering approaches in Section 4.1 and 4.2.

Nowadays texture-based approaches can be considered as the state-of-the-art approach
for volume rendering, since they allow interactive frame rates even for larger datasets. In contrast to indirect rendering methods, where the achievable rendering performance is usually limited by the number of polygonal primitives that can be rendered per time unit (polygon-rate limitation), the performance of direct volume rendering approaches is mainly limited by the rate of fragment operations (fill-rate limitation), as well as the amount of available texture memory. In the next chapter we will propose an algorithm that addresses both of these drawbacks for large, sparse datasets.
Chapter 4

Accelerated Volume Rendering

3D imaging and computational science produce increasingly large volumetric datasets. While data volumes consisting of $O(10^9)$ voxels are not unusual today, future imaging devices and large scale simulations are supposed to create tera-scale datasets. Hence the development of volume rendering approaches dedicated to handle large datasets is getting more and more important.

The main performance limitations for volume rendering is the large amount of interpolation operations during the evaluation of the integral (3.7) and the texture memory limitations for hardware-accelerated approaches.

This chapter proposes algorithms for software-based raycasting as well as hardware-accelerated volume rendering that address these problems. Both approaches have in common that they aim at reducing the number of interpolation operations by focusing the computational effort to regions of the data volume that have the most contribution to the resulting pixel intensities.

In the next subsection, related work in the field of volume rendering for large data is discussed. In Section 4.3 we present an algorithm that employs an error-controlled adaptive integration scheme to accelerate the computation of the raycasting integral. In Section 4.4 we propose an approach for accelerated texture-based volume rendering for large, sparse datasets.

4.1 Related Work on Software-Based Volume Rendering

Various papers that deal with the topic of accelerating emission-absorption approaches have been published in the last years. In 1990 Levoy et al. [46] presented an algorithm that skips empty space utilizing an octree data structure to encode the presence of non-transparent material. Danskin [23] extended this approach by exploiting data homogeneity and accumulated opacity as well.

Subramanian et al. [82] designed a ray-tracer that works efficiently for cases where the data of interest is distributed sparsely through the volume. A space partitioning of the data volume is carried out by a median-cut subdivision. Empty spaces are pruned of and the subdivision is repeated until each region contains exactly one voxel.
Laur et al. [44] proposed a splatting algorithm that operates on a pyramidal representation of the volume and determines the number of splats adaptively, according to user-supplied error criteria. Storing data mean and root mean square at each node permits rendering by progressive refinement. Nodes within the user-specified tolerance are rendered as single splats by utilizing texture mapping capabilities.

The idea of exploiting the distance transform to speed up the background traversal by Zuiderveld et al. [98] has been extended by Cohen et al. [22], who introduced so-called ‘proximity clouds’ that store ‘uniformity information’ (typically encoded in a space partitioning tree) directly in the voxel raster: Voxel data either contain a data value or information indicating how far incident rays may leap without missing important features.

Novins et al. [61, 60] utilizes a BSP-tree to store lower and upper bounds of the intensity and opacity for subregions of the data volume. Based on these bounds, an initial coarse ray discretization is refined in regions where the error exceeds a user-defined threshold, aiming at an equally distributed discretization error.

Lee et al. [45] reduced raycasting overheads by an adaptive block subdivision. Their algorithm applies an uniform space subdivision and then merges coherent uniform blocks in order to generate adaptively-sized blocks which are efficient for leaping space.

A common drawback of all these approaches is that they require some kind of preprocessing, for example in order to detect homogeneous regions and/or the computation of error bounds on the volume rendering integral. This procedure usually has to be repeated each time the transfer function is changed, which might be computationally expensive for larger datasets.

In Section 4.3 we will present a scheme that accelerates raycasting without the need of preprocessing the data. Its underlying idea is to start with an approximation of the ray-integral based on a coarse discretization of the whole interval. Employing local error criteria this coarse discretization is recursively refined by point enrichment in regions that require higher resolution, i.e. regions where the local error exceeds a certain threshold.

4.2 Related Work on Hardware-Accelerated Approaches

Spatial data structures have been employed in combination with 3D texture-based volume rendering approaches, too.

LaMar et al. [43] proposed a multi-resolution techniques for interactive volume visualization of large datasets. They employ an octree representation and a node selection scheme for adapting the distance of texture slices for regions depending on their distance to the viewpoint. They further introduced the use of spherical shells as proxy geometries. Weiler et al. [90] improved the latter approach by techniques that avoid rendering artifacts due to discontinuous texture interpolation and varying sample distances at boundaries between regions with different resolutions.

Boada et al. [14] presented an error and importance driven strategy for selecting a set of octree nodes from the full pyramidal structure. Fang et al. [28] introduced an approach for rendering deformable volume data utilizing an octree encoded target volume.
Octree-based data structures have further been applied for efficient handling of time-dependent datasets by Shen et al. [77]. They proposed a combination of a ‘spatial’ octree and a binary ‘time’ partitioning tree. It effectively captures both, the spatial and the temporal coherence in a time-varying field. Originally intended to accelerate raycasting algorithms, this work has been extended to hardware volume rendering via 3D texture mapping [78].

Another approach for accelerating texture-based volume rendering by leaping of empty regions has been proposed by Tong et al. [84]. The data volume is partitioned into equal-sized blocks that are pruned in one direction. The resulting blocks are merged to reduce texture I/O and rendered in parallel projection. Similar to the octree approach this algorithm introduces partitioning axes independent of the underlying spatial data distribution, and it is not clear which brick size to choose for optimal rendering performance.

Combining the approach presented in [82] with texture-based volume rendering would suffer from the simple median-cut subdivision strategy which produces regions containing too many irrelevant voxels, or too many regions, if the subdivision is carried out until voxel-size is reached.

The techniques presented in [98, 22] is disadvantageous for 3D texture-based volume rendering, since the volume is not subdivided into axis-aligned regions.

The approach in [45] is not efficient for volume rendering via 3D textures, since it usually generates many boxes of voxel size and it does not guarantee that the resulting blocks can be traversed in a view-consistent order.

In Section 4.4 we propose an algorithm that employs an AMR hierarchy of nested subgrids in order to accelerate texture-based volume rendering of sparse data. The contents of the scalar data are evaluated based on relevance criteria. This approach achieves significant performance gains if compared space leaping approaches using an octree data structure.
4.3 Acceleration by Adaptive Integration

The raycasting algorithm discussed in this section is based on the application of an error-controlled adaptive quadrature algorithm [96, 24]. We chose this approach since it does not require a preprocessing of the data and in contrast to other adaptive integration schemes, it is applicable to integrands that are only \(C^0\)-continuous, which is important for our purposes, since we employ piecewise trilinear interpolation.

The local error criterion is based on the comparison of two integration methods of different order, the trapezoidal and Simpson’s rule in this case, though other schemes might be applied. The local refinement is carried out by interval bisections. We will briefly describe the employed numerical integration scheme in the next section, for more details the reader might refer to [24].

4.3.1 Numerical Background

Let \(I(g) := \int_a^b g(s) \, ds\) denote the integral of the scalar function \(g : [a, b] \mapsto \mathbb{R}\), that is to be approximated numerically over the interval \(\Omega := [a, b]\). Consider an initial, coarse discretization \(\mathcal{G}\) of \(\Omega\), that is given by a set of intervals \(I_i := [s_i, s_{i+1}]\), for \(i = 0, ..., n\), with \(s_0 := a, s_n = b\)

as indicated in Figure 4.1.

![Figure 4.1: Refinement of ray-intervals by bisection.](image)

Further let \(h(I_i) := |I_i| = |s_{i+1} - s_i|\) denote the length of the interval \(I_i\), and \(T(I)\), \(S(I)\) the results of trapezoidal rule, respectively the Simpson’s rule applied to the interval \(I = [s_l, s_r]\)

\[
T(I) = \frac{h}{4} (f(s_l) + 2f(s_m) + f(s_r)), \quad \text{and} \quad (4.1)
\]
\[
S(I) = \frac{h}{6} (f(s_l) + 4f(s_m) + f(s_r)) \quad (4.2)
\]

Here \(s_m = \frac{(s_l + s_r)}{2}\) is chosen as the third node which is required for the Simpson’s rule.
Analogously

$$T(G) := \sum_{i=0}^{n} T(I_i) \quad \text{and} \quad (4.3)$$

$$S(G) := \sum_{i=0}^{n} S(I_i) \quad (4.4)$$

denote the approximations of \( I(g) \) over the whole domain \( \Omega \). The local approximation error on \( I \) can be estimated by

$$e_{loc}(I) := | T(I) - S(I) |. \quad (4.5)$$

Intervals \( I_n \) that are tagged for refinement, as discussed below, are split up into the two subintervals by simple bisection

$$I = [s_l, s_r] \rightarrow [s_l, s_l + s_r/2], [s_l + s_r/2, s_r] =: (I_{ref}^l, I_{ref}^r),$$

compare Figure 4.1. In the following \( P(I) \) will denote the next coarser interval that contains \( \hat{I} \), i.e.

$$P(I_{ref}^r) = P(I_{ref}^l) = I.$$ 

The refinement strategy of the integration approach is to obtain an equal distribution of the local approximation errors \( e_{loc}(I_i) \) for all intervals \( I_i \). The decision which intervals should be refined in order to achieve this is based on a second error estimator \( e_{extr}(I) \), that yields information about the local approximation error of the subintervals \( I_{ref}^l, I_{ref}^r \) in case \( I \) would be refined. This information is obtained by extrapolation of the local approximation errors on \( I \) and \( P(I) \). The estimated error for the trapezoidal rule is of the form

$$e_{loc}(I) \doteq Ch(I)\gamma. \quad (4.6)$$

Here \( C \) is a local constant and \( \doteq \) denotes equality up to 'higher-order' terms in \( h \). The actual value of \( \gamma \) is not important for the derivation, because it will cancel out in the following. Since \( h(P(I)) = 2h(I) \) holds, it follows that \( e_{loc}(P(I)) \doteq 2\gamma e_{loc}(I) \) and therefore

$$e_{loc}(I_{ref}) \doteq (Ch(I)\gamma)2^{-\gamma} \doteq e_{loc}(I)\left( \frac{e_{loc}(I)}{e_{loc}(P(I))} \right).$$

This gives rise to the following estimation of the extrapolated local error

$$e_{extr}(I) := \frac{e_{loc}(I)^2}{e_{loc}(P(I))}. \quad (4.7)$$

Based on definition (4.7) the maximal local error after global refinement of \( G \) can be estimated via

$$m_{extr}(G) := \max_{I \in G} e_{extr}(I). \quad (4.8)$$
Since the goal of the refinement process is to obtain an equal distribution of the local approximation errors, it is reasonable to restrict the refinement at each integration step to those intervals for which $e_{\text{loc}}(I) \geq m_{\text{extr}}$ holds. A suitable measure for the unknown global approximation error

$$|(\int_a^b g(t)dt) - (\sum_{i=0}^n S(I))|$$

is obtained by comparing the approximations of the integral via the actual discretization $G$ and the next coarser one $P(G)$

$$e_{\text{glob}}(G) := |S(P(G)) - S(G)|. \quad (4.9)$$

This global error estimation is used in order to define a “stopping”-criterion for the iteration. The refinement process is stopped, if

$$e_{\text{glob}}(G) \leq c_{\text{tol}} |S(G)|, \quad (4.10)$$

where $c_{\text{tol}}$ is a relative precision threshold.

### 4.3.2 Application to Volume Rendering

Evaluation of Equation (4.3) and (4.4) for the emission-absorption model (3.7) yields

$$T(I) = \frac{h(I)}{4}(q_l + 2q_me^{-\frac{h(I)}{4}(\kappa_l+\kappa_m)} + q_re^{-\frac{h(I)}{4}(\kappa_l+2\kappa_m+\kappa_r)}), \quad \text{and} \quad (4.11)$$

$$S(I) = \frac{h(I)}{6}(q_l + 4q_me^{-\frac{h(I)}{6}(\kappa_l+\kappa_m)} + q_re^{-\frac{h(I)}{6}(\kappa_l+4\kappa_m+\kappa_r)}). \quad (4.12)$$

Here the segments are oriented such, that $I_0$ denotes the segment that is closest to the viewpoint, see Figure 4.1.

Instead of defining the local approximation error for the interval $I_i$ directly according to Equation (4.5), we multiply it by a weighting factor $e^{-\tau(s_0,s_i)}$, which specifies the accumulated absorption between the $i$-th ray-segment and the viewpoint. This is motivated by the fact, that the contribution of a ray-segment to the total intensity depends on the amount of absorption between the segment and the viewpoint. So we obtain the following expression for the local approximation error

$$e_{\text{loc}}(I_i) := e^{-\tau(s_0,s_i)}|T(I_i) - S(I_i)|$$

$$= (\prod_{k=0}^{i-1} T_k)|T(I_i) - S(I_i)|, \quad (4.13)$$

where $T_k$ denotes the transparency of segment $k$, according to definition (3.9).

The overall algorithm for the the adaptive ray-integration scheme is summarized in the following pseudocode:
generate initial discretization;
do
{
    // traverse intervals from front--to--back
    for all (Iᵢ : i = 0, ..., n) {
        if (Iᵢ is newly refined) {
            compute T(Iᵢ), S(Iᵢ);
            compute e_{loc}(Iᵢ), e_{extr}(Iᵢ);
        }
        update global transparency;
        update global intensity;
        update m_{extr}(G);
    }
    compute e_{glob};
    refine all I, with e_{loc}(I) > m_{extr}(G);
} while (e_{glob} > c_{tol}|S(G)|);

4.3.3 Results and Discussion

Figure 4.2: Renderings of the three tested datasets. The first one was generated by sampling the function \( f(x) = (x^2 + 10^{-3})^{-1} \) on a uniform grid a resolution of \( 256^3 \) voxels. The second dataset, the result of an astrophysical simulation describing the collision of three black holes, had a resolution of \( 512^3 \) voxels. The third dataset, the result of a molecular dynamics simulation and conformational analysis, contained \( 185 \times 202 \times 157 \) voxels. (datasets courtesy of P. Diener, Louisiana State University (b) and D. Baum, Zuse Institute Berlin (c))

We applied the adaptive ray-integration scheme discussed above to three datasets with different frequency characteristics. The first example was a smooth analytical grid function, the spectrum of the second datasets contained high frequencies and the last tested dataset contained large homogeneous regions as well as sharp edges in other parts. More information about the examples is given in Figure 4.2.

51
All measurements were performed on a SGI onyx3 with a 500 MHz MIPS R14000 processor. The screen resolution was 400 × 400 pixels for all examples. We compared the performance of the adaptive ray-integration scheme with that of a standard raycasting algorithm. Our implementation performs “early ray-termination” with a transparency threshold of $10^{-2}$ and the trapezoidal rule was chosen as quadrature rule. For the adaptive scheme a relative error threshold of $10^{-2}$ was chosen. The following table shows the resulting rendering times as well as the relative reduction of interpolation operations of the adaptive scheme in comparison to the standard raycasting method.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>fps(standard)</th>
<th>fps(adaptive)</th>
<th>reduction of evaluations</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0.118 fps</td>
<td>0.335 sec</td>
<td>76.1%</td>
</tr>
<tr>
<td>II</td>
<td>0.028 fps</td>
<td>0.041 fps</td>
<td>54.3%</td>
</tr>
<tr>
<td>III</td>
<td>0.109 fps</td>
<td>0.097 fps</td>
<td>53.1%</td>
</tr>
</tbody>
</table>

Table 4.1: This table states the frame rates of the standard and adaptive raycasting scheme for the three different datasets as well as the reduction of interpolation operations due to the adaptive approach.

The adaptive scheme achieves best results for the first dataset. In this case the number of interpolation operations was reduced by about 76 percent and the rendering performance was almost three times higher if compared to the non-adaptive renderer. Here the adaptive scheme takes advantage of the smoothness of the data, which allows for very coarse initial discretizations. The error estimator efficiently decreased the interval length in the central region of the data volume, where the sharp peak is located.

In order to generate the semi-transparent isoshells for the higher resolved second dataset, a high-frequency colormap was applied. This required an increased sampling rate. For this example the number of interpolation operations was reduced by about 55% and the rendering performance of the adaptive scheme was about 45% faster, if compared to the non-adaptive method.

For the third dataset the performance of the standard scheme was about 10% higher, though the adaptive scheme needed about 50% less interpolation operations. This is due to the fact that for this data set a rather fine initial discretization for the rays was necessary in order to ensure that the sharply edged ring was not missed by the integrator. Figure 4.3 shows a sequence of three renderings with increasing resolution of the initial discretizations. In this case the overhead due to the error estimator computation and the additional data structures involved was higher than the performance gain due to the reduced interpolation operations. For datasets that contain high frequencies the adaptive approach, like any adaptive integration scheme, potentially misses features like sharp edges. In order to chose appropriate initial ray discretizations for this type of data it would be advantageous to detect regions of different frequency characteristics in a preprocessing step. As future work it would be interesting to investigate if the adaptive integration scheme can be employed for this purpose.

52
As a conclusion one can state that the application of the adaptive integration scheme reduced the number of necessary function evaluations for all tested data sets if compared to the standard algorithm. The amount of decrease ranged from 76% to 53%, though the frame rates do not increase at the same rate, due to overhead of the adaptive solver which is caused by allocation of data structures for storing the ray-segments and the computations of the estimated local errors. The adaptive scheme is advantageous during the preparation phase of the visualization session, when a suitable colormap is determined and hence modified and adjusted frequently, since it does not require any preprocessing of the data, which usually depends on the chosen colormap. It achieves best performance for smooth datasets, since in this case the error estimation process is most efficient.
4.4 Acceleration by Exploiting Sparsity of Data

4.4.1 Motivation

As discussed in Section 3.5, the availability of powerful texturing units on consumer graphic hardware enormously increased the popularity of 2D-, and 3D texture-based volume rendering algorithms. Though they allow interactive frame-rates for moderately sized data volume for large datasets the rendering performance of texture-based approaches is still too slow for interactive exploration. The main performance limiting factors are

- the fill-rate of the texture hardware,
- the amount of texture memory, since it determines the number of texture bricks necessary to render the data volume and, closely related to this,
- the texture I/O-bandwidth, since this (mainly) determines the time for placing new texture chunks in texture memory during each pass.

Often just a small portion of the highly resolved dataset is of interest for the user. This holds, for example, if the dataset contains large transparent and/or homogeneous regions that do not contribute any information to the final image. For segmented image data the user often needs to visualize just a subset of the various segmented regions. For instance, in biomedical visualization only line-like structures, such as vessel trees, neuron trees, trabeculae in bones, or filament structures in muscles, have to be visualized. Thin structures, occupying only a tiny fraction of a volume, occur on all length scales in nature – ranging from chain molecules in chemistry to filaments of galaxies and galaxy clusters. Additionally, in numerical simulations, often large computational volumes have to be considered to take care of boundary conditions, though the interesting phenomena happen in very small spatial regions.

4.4.2 Ansatz

Sparsity of data can be exploited to increase the rendering performance by assigning individual “texture bricks”, only to regions that have been classified as relevant. The optimal coverage of these subvolumes would be achieved by assigning a texture to each relevant elementary cell of the dual voxel grid. This approach is unfeasible, since it requires a large amount of texture memory: texture bricks need to share rows of texels at common boundaries, to ensure consistent trilinear interpolation. Further it would result in an enormous number of texture-, and polygon-coordinates to be computed and specified, since every brick has to be intersected with the proxy geometries to be rendered. A good balance between the volume enclosed by texture bricks and the number of created bricks is therefore crucial.

In the following we present an algorithm that achieves this balance. It consists of the following steps:
• Recursive subsampling of the dataset, in order to build a pyramidal structure.

• Application of a signature-based clustering algorithm to generate an AMR hierarchy that efficiently encloses the relevant regions.

• Assignment of separate 3D texture bricks to each leaf of the hierarchy.

• Minimization of the usage of texture memory due to power-of-two restrictions (of the graphics API or graphics hardware) by utilizing a 3D-packing algorithm.

For evaluation we apply our algorithm to several 3D-image and simulation datasets with different characteristics and compare its performance and texture memory requirements with the standard as well as an octree-based algorithm. We show that significant gains in rendering performance are achieved for sparse datasets. The AMR-based algorithm has the additional advantage that it requires less parameter adjustment, i. e. it reduces the amount of user interaction, since standard parameter settings already yield good rendering performance – almost independent of the topology and spatial distribution of the interesting subregions.

This Section is organized as follows: the octree-based and AMR approaches are presented in Subsections 4.4.4 and 4.4.5. Subsections 4.4.7 to 4.4.9 address issues related to the hardware-accelerated rendering of the resulting data structures. Finally we apply the algorithms to several datasets and discuss the results in Subsection 4.4.11. But first we list some requirements that the desired coverage of the relevant regions should fulfill.

4.4.3 Data Structure Requirements

In order to decide if a cell is relevant or not an importance criterion is necessary. It might for example be based on opaqueness or on a voxel classification performed by a preceding segmentation step. The first case will be discussed in more detail in Subsection 4.4.6. Let us for the moment consider that the relevance of the cells is encoded by a function

\[ R(i, j, k) := \begin{cases} 1, & \text{if cell } \Omega_{ijk} \text{ is “relevant”}, \\ 0, & \text{otherwise}. \end{cases} \] (4.14)

We aim at generating an efficient coverage of the relevant cells for accelerating 3D texture-based volume rendering. It should fulfill the following conditions:

• The coverage should consist of a set of axis-aligned blocks \( \mathcal{B}_i \), since these geometries are most appropriate for rendering via 3D textures.

• The blocks should have pairwise disjoint interiors, i. e.

\[ (\mathcal{B}_i \cap \mathcal{B}_j) = \emptyset \lor (\mathcal{B}_i \cap \mathcal{B}_j) \subset (\partial \mathcal{B}_i \cup \partial \mathcal{B}_j) \]

for \( i \neq j \), in order to avoid multiple rendering of subregions of the data volume.
• The coverage should not contain any visibility cycles, in order avoid expensive occlusion computations and allow independent rendering of the separate blocks.

For a definition of visibility cycles assume three convex polyhedra \( A, B \) and \( C \) and let the relation \((A, B)\) denote that \( A \) (partially) occludes \( B \) for the given viewpoint. The polyhedra build a visibility cycle if the relations \((A, B), (B, C)\) and \((C, A)\) are fulfilled for this viewpoint. A simple 3D example of a set of three axis-aligned bounding boxes that form a visibility cycle from the considered viewpoint is given in Figure 4.4.

In addition to the requirements listed above, the blocks should further build an efficient coverage of the relevant cells in the sense that that the total number of boxes, as well as

\[
V(B_m) := \frac{|\{\Omega_{ijk} \in B_m \mid R(i, j, k) = 0\}|}{|B_m|},
\]

i. e. the ratio of the non-relevant cells to the whole number of cells in each block, are small.

In the following we will compare two approaches that employ spatial data structures to generate these coverings. The leaf nodes of the data structures will correspond to the axis-aligned blocks \( B_m \). Regarding the discussion in Subsection 2.4, only two data structures fulfill at least the first three requirements listed above, namely the octree and the kD-tree\(^1\).

### 4.4.4 Octree Generation

In this subsection we describe the construction of the octree structure. In a first step the data volume is subdivided into blocks of pre-selectable extensions\(^2\). Only sub-branches with leaf nodes that cover relevant regions of the data volume are inserted into the hierarchy.

In order reduce the number of blocks, we recursively visit each node and check if all its subnodes are leaves. In this case they are pruned off and the node itself becomes a leaf. We achieve another reduction of the number of bricks by taking into account

\[
r(B_m) := \frac{|\{C(B_m) \mid C(B_m) \text{ is leaf node}\}|}{|B_m|},
\]

\(^1\)It is obvious that the first two requirements are fulfilled. For the view-consistent traversal of the nodes we refer to Section 4.4.9.

\(^2\)We choose powers of two, in order to meet the restriction on the texture dimensions given by the graphics hardware, respectively graphics API.
i.e. the ratio between the volume covered by the sub-branch of \( B_m \) and the node’s volume itself. If \( r(B_m) \) is smaller than a pre-selectable threshold, this sub-branch is pruned and the node gets a leaf node. To find a good balance between the performance limiting factors, this parameter has to be adjusted carefully.

In the last step the data of the subvolumes corresponding to the resulting leaves of the octree is copied and stored in the hierarchy’s nodes. Here some care is necessary at the boundary faces of the grids, since trilinear interpolation is applied; to avoid artifacts caused by discontinuities between adjacent grids during the rendering using 3D textures, it has to be ensured that they share one row of data samples at their common interfaces.

A drawback of the octree data structure is that the spatial locations of the partitioning boundaries are not adapted to the underlying data contents. As shown in Subsection 4.4.11, for large, sparse datasets this often results in an unnecessarily high number of boxes and/or boxes that capture only very few relevant voxels.

### 4.4.5 Adaptive Approach

In contrast to the octree, the AMR data structure allows to freely place subgrids and thereby to better adapt the shape of the subregions that contain relevant cells. Consider for example the simple case of a rectangular region of opaque voxels in the center of a larger volume. The AMR structure can enclose this with just one child node and without covering many transparent cells, whereas an octree requires at least eight or more subnodes, thereby introducing a higher number of interior boundaries.

Since AMR subgrids in general might build visibility cycles we use a generation scheme that arranges the nodes in a kD-tree manner. In order to construct an AMR hierarchy that fulfills the constraints listed above, we generate a pyramidal representation of the original dataset by subsampling it with an integer factor in a preliminary step. This factor usually equals 2, but larger values (e.g. to reduce the amount of additional memory necessary for storing the hierarchy), possibly different for the three coordinate directions, are also admissible.

The relevance function \( R^l(\cdot) \) for the pyramid level \( \Lambda^l \) is related to the one of the next higher resolved level, \( R^{l+1}(\cdot) \), via

\[
R^l(i, j, k) := \begin{cases} 
1, & \text{if } \left( \sum_{i=2i}^{2i+1} \sum_{j=2j}^{2j+1} \sum_{k=2k}^{2k+1} R^{l+1}(\hat{i}, \hat{j}, \hat{k}) \right) \geq 1 \\
0, & \text{else.}
\end{cases}
\]  \hspace{1cm} (4.15)

Here we adopted the notation introduced in Subsection 2.2.1. In particular \( \Lambda^0 \) denotes the coarsest level of resolution, while the highest level is given by the original dataset. Equation 4.15 states that a cell on the coarser level is tagged as relevant, if it contains at least one tagged cell in the original dataset.

After this preparing step we build the AMR hierarchy utilizing the clustering algorithm discussed in Section 2.2.3. The clustering starts at the coarsest resolution of the pyramid, which defines the root node of the hierarchy and it is recursively repeated on the
newly generated subregions, until the finest level, i.e. the original data, is reached. The signature lists (2.7) are computed from the relevance functions $R(i, j, k)^l$ via

$$S_{yz}(i)^l = \sum_{j=\mu_{p_1}^n}^{(p_2^m+n_2^n)} \sum_{k=\mu_{p_2}^m}^{(p_3^m+n_3^m)} R(i, j, k)^l.$$  \hfill (4.16)

The choice of the splitting planes in the clustering algorithm ensures that the resulting blocks, respectively subgrids are arranged in an adaptive kD-tree, which will be important for the view-consistent traversal, compare Subsection 4.4.9. As for the octree, the leaf nodes have to share one row of data samples at their common boundary faces.

Notice that one could alternatively cluster directly on the original data volume, without first creating the pyramid. This usually slightly decreases the number of blocks needed for the coverage, but in turn increases the running time of the algorithm. As discussed in [10] the running time is $O(k(P + M))$, where $k$ is the total number of grids upon termination of the algorithm, $P$ is the number of flagged cells, and $M$ is related to the determination of the inflection points. Keeping the preprocessing times small is crucial in our application, since the clustering has to be repeated each time the colormap is changed. Further this hierarchical approach offers the possibility to obtain a multi-resolution representation of the interesting regions by applying appropriate averaging methods in order to compute the data samples on the coarser grids.

The overall approach is summarized in the following the pseudo code:

```plaintext
// subsampling of data
create pyramidal structure;

// start grid generation
create_subgrid(0, data domain);

// render of resulting leaves
traverse_subtree(root-node);

// recursive grid generation
create_subgrid(level, bounding box)
{
  \Gamma = \text{new subgrid}(\text{bounding box, level});
  \text{if (level} \neq \text{maxlevel})
  {
    \text{markedCells} = \text{inspect_cells}(\Gamma, \text{importance\_criterion});
    \text{newBoxList} = \text{cluster(markedCells)};
    \text{for all (boxes[i] \in newBoxList)}
    {
      \Gamma_{\text{sub}} = \text{create_subgrid}(\text{boxes[i], level+1});
      \text{set } \Gamma_{\text{sub}} \text{ as } C(\Gamma);
    }
  }
  \text{return } \Gamma;
}
```

The pseudo code for the view-consistent traversal (traverse_subtree) is given in Section 4.4.9.
4.4.6 Opacity as Relevance Criterion

In the following we will briefly discuss the special, but nevertheless important case that the relevance criterion is based on the opacity of the cells. Since the opacity, respectively transparency depends on the colormap, which usually is changed several times during a visualization session, storing the values of relevance functions \( R(i, j, k)^l \) directly for each cell on the pyramid level \( l \) would be disadvantageous, because in this case the subsampling stage would have to be carried out after each modification of the colormap.

In order to avoid this, for each cell \( \Omega_{ijk}^l \) in the pyramid we rather store the minimal and maximal data value

\[
(f_{ijk}^{\text{min}} := \min_{x \in \Omega_{ijk}^l} f(x), f_{ijk}^{\text{max}} := \max_{x \in \Omega_{ijk}^l} f(x)).
\]

Determining the min-max intervals is inexpensive for trilinear interpolation, since it equals the minimal and maximal data samples of the vertices covered by \( \Omega_{ijk}^l \). During the clustering procedure each of the intervals \( C_\alpha([f_{ijk}^{\text{min}}, f_{ijk}^{\text{max}}]) \) in the colormap’s alpha channel is inspected, and the cell is tagged, if its maximal opacity \( C_\alpha = 1 - T \) is below a user selectable threshold. Recall that according to Equation (3.18), the opacity entry stored in the colormap is related to the absorption factor \( \kappa \) via \( C_\alpha(f(x)) = 1 - \exp(-\kappa(x)d_0) \), assuming that \( \kappa \) is piecewise constant per slice segment of thickness \( d_0 \). Hence we can estimate an upper bound of the transparency \( A(\Omega_{ijk}) \) of the cell \( \Omega_{ijk} \) by

\[
A(\Omega_{ijk}) := 1 - \exp(-\kappa_{\text{max}}(\Omega_{ijk}) d) = 1 - (1 - \tilde{A}_{\text{max}})^\frac{d}{d_0}.
\]

Here \( d \) is the cells diagonal and

\[
\tilde{A}_{\text{max}} := \max_{f(x) \in [f_{ijk}^{\text{min}}, f_{ijk}^{\text{max}}]} \{ A(f(x)) \}.
\]

Due to the subdivision of the preprocessing step into pyramid creation and clustering, only the clustering step has to be repeated if the colormap is changed.

4.4.7 Rendering

We render each of the generated blocks separately utilizing the standard 3D texture mapping approach [17, 95]. Individual 3D texture bricks are assigned to the separate leaf nodes of the resulting hierarchies and each block is rendered separately in a back-to-front order. One-channel textures and the SGI color table extension were used. In the following we describe the differences during rendering for both types of hierarchies, AMR tree and octree.

The graphics hardware assumes the dimensions of 3D textures to be equal to a power of two. This could be achieved by extending the data subvolume of each leaf grid of the AMR hierarchy to the next bigger power of two, for example by clamping the boundary
Figure 4.5: 2D example of the “next-fit-decreasing-height” packing algorithm: subvolumes are inserted from left to right, starting at the lower left corner.

texels and restricting the generated texture coordinates to the unextended area. Regarding the potentially large number of textures to deal with, this would typically result in a large overhead of unused texture memory. We decided to reduce this overhead by utilizing a packing algorithm that inserts the texture bricks into one single texture.

4.4.8 Texture Packing

For our purposes the following variant of the three-dimensional packing problem is appropriate:

pack a given number of axis-aligned boxes into one container with fixed width and depth sizes, such that its height is minimized.

This problem belongs to the class of NP-hard problems, but a couple of useful heuristics have been suggested, compare [47]. We adopt a three-dimensional version of the “next-fit-decreasing-height” (NFDH) algorithm [35].

First, the boxes are inserted into a list in the order of decreasing height. The packing algorithm starts at the lower left corner of the container and the boxes are inserted from left to right until the right border is reached. Then a new row is opened, with a depth coordinate given by the largest depth of the already inserted boxes. This procedure is repeated until the lowest layer of the container is filled. Then a new layer is opened and this process continues until all boxes are inserted. See Figure 4.5 for a 2D example.

We iterate this procedure with different values for the base layer extensions of the container, chosen as powers of two. For the resulting containers the height is extended to the next power of two, and the one with smallest volume is taken. Then a 3D texture of this size is defined with the sub-textures are inserted at the appropriate positions. For each block its offset in the merged texture is stored.

If the packed texture is still too large to fit entirely into texture memory, a separate texture brick is generated for each leaf node of the hierarchy.
4.4.9 View-Consistent Node Traversal

As discussed above, it is not possible in general to render arbitrary arranged axis-aligned blocks separately in a back-to-front order, because occlusion cycles might occur for certain viewpoints. However, since the clustering algorithm performs a partition of the data volume in a kD-tree style, no cycles occur in this AMR approach. We employ the tree in order to traverse the generated blocks in the view-consistent order.

The traversal starts at the root node of the kD-tree. At each internal node the associated axis-aligned partition plane divides the domain into two half-spaces $H_1, H_2$. For a back-to-front traversal the subtree corresponding to the half-space which does not contain the viewpoint, has to be visited first. In case the viewpoint is located on the plane, the subtrees can be traversed in any order. The procedure is outlined in the following pseudocode:

```plaintext
traverse_subtree(subnode)
{
    if (subnode is a leaf node) {
        render(subnode);
    } else {
        $H_1 := \text{halfspace of subnode.childA}$;
        $H_2 := \text{halfspace of subnode.childB}$;
        if (viewpoint $\subset H_1$)
            traverse_subtree(subnode.childB);
        else
            traverse_subtree(subnode.childA);
    }
}
```

4.4.10 Generation of the Proxy Geometries

Computing the intersection points of the slices and the bounding boxes of the blocks is done in software. We speed up this procedure by first determining the interval of slices that intersect the actual block. This is accomplished by projecting the vertices $x_k$ of the bounding boxes onto the planes normal direction $n$. Indexing the slice through location $a$ by 0 and slices in normal direction by positive indices, the slice index the corresponds to the vertex $x_k$ is given by $i = \left[\frac{(x_k - a) \cdot n}{d_0}\right]$. Thus the intersection computations can be restricted to the slices with indices within $[i_{\text{min}}, i_{\text{max}}]$.

In order to compute the coordinates of the polygons that result from the intersections between the slices and the block edges, the bounding box vertices that lie above and below the oriented slice are determined. For each configuration a table lookup returns which edges are intersected, as well as their correct order (needed for the definition of the associated texture polygons). The coordinates are computed by linear interpolation between the endpoints of these edges.

The view-consistent order of the octree nodes can easily be determined by a table lookup at each node, which returns the correct order to visit the subnodes with respect
Figure 4.6: The order of a view-consistent node traversal for the decomposition of the AMR grid example from Figure 2.6 for a viewpoint in the lower right.

to the actual viewpoint, as discussed for example in [5]. As in the AMR case, for each block the interval of intersecting slices is precomputed and the intersection points are determined using the fast table-lookup approach mentioned above.

4.4.11 The Applications

We applied our algorithm to several datasets with decreasing degrees of sparseness, ranging from extremely sparse neuron data to non-sparse bee-brain scans. The performance was tested on an SGI Onyx2 InfiniteReality2 with two RM7 raster managers with 64 MBytes texture memory. The runs were performed on a single 195 MHz MIPS R10K processor.

Since texture-based volume rendering is fill-rate limited, the frame rates depend on the size of the viewer window, the number of slices, and the area in screen space covered by the data volume (and thus on the actual position of the viewpoint). We averaged the frame rates for several positions inside and outside the data volume by choosing viewpoints located on different circles with varying radii and orientations. For all examples the size of the rendered images was $764 \times 793$ pixels; the numbers of slices are listed below. In the examples we used the opacity as the importance criterion. Cells with an associated opacity value greater than $\alpha_{\text{thres}} = 0.03$ were marked as relevant.

4.4.11.1 The Datasets

The datasets I and II are confocal microscopy images of neurons inside a honey bee’s brain. About 0.1% respectively 0.2% of the cells were marked as relevant and the volumes were rendered with 1200 slices. For dataset II a greater threshold of $\alpha_{\text{thres}} = 0.1$ was used,
in order to eliminate noise contained in the microscopy image. Dataset III represents a part of a human vascular tree. Here 1.2% of the cells were tagged as relevant. This dataset also was rendered with 1200 slices. Example IV contains data from a molecular dynamics simulation and conformational analysis. About 14% of the cells were marked as relevant. The volume was rendered with 320 slices. This is the only dataset which fitted into memory without the need of bricking. The last example is a non sparse dataset containing 23% of relevant cells, which was rendered with 900 slices.

Images of the different datasets are shown in Figure 4.7 to Figure 4.11. boxes of the octree, and the right images display the AMR bounding boxes.

4.4.11.2 Results

The statistics are displayed in Tables 5.1 to 4.6. For each dataset we list the results for standard volume rendering, for the octree and the AMR approach. The first rows (‘Standard’) show the results for the standard volume rendering approach. Rows labeled ‘Octree I’ contain the octree result with leaf dimensions that result in optimal frame rates. Rows labeled ‘Octree II’ show the best octree results we achieved by adjusting the ratio threshold parameters. The fourth rows (‘AMR I’) display AMR results with the clusterer’s efficiency parameter set to 0.85 and a minimal extension bound of 8. The last rows, labeled ‘AMR II’, report the optimal results achieved by adjusting these parameters for each dataset.

The table columns list the depth of the hierarchies, the number of created texture bricks, the percentage of the data volume covered by them, the amount of texture memory (after extension to the next power of two and packing them into one texture), the preprocessing times, and finally the frame rates. On the INFINITREALITY2 the texel size is two bytes, so the internal size of the textures is twice the number given in the table. Note that for the datasets I, II, III and IV the volume entries for the standard case give values greater than 100%, because the bricks share a common row of data samples at their boundaries. For the standard approach the preprocessing times are just given by the times to allocate and define the texture or textures (in cases where bricking is necessary). The preprocessing times for the AMR hierarchies are split into the part needed for the resampling and a second part for clustering and packing. Note that only the clustering and packing step has to be updated, when the colormap is changed.

4.4.12 Discussion and Conclusions

The number of generated texture bricks in the AMR case is much smaller than the bricks created by the corresponding octree hierarchy, compare Tables 5.1 to 4.6. Also less non-relevant cells are enclosed, especially for the sparse datasets I, II and III. So the number of interpolation operation as well as the rendered area in screen space are drastically reduced, resulting in significant performance gains for the AMR approach for datasets I, II and III.

For these examples the best AMR results required less texture memory compared to the octree results with optimal parameters. This is due to the efficient clustering that
captures only few non-relevant cells and the smaller number of bricks, resulting in less duplicated texels on common boundaries of bricks. For the dataset IV the amount of texture memory was almost equal for both hierarchical approaches. For the non-sparse dataset V the amount was twice as high as for the octree case. Here the sizes of the covered volume are comparable for octree and AMR (choosing a better heuristics for packing could improve this for AMR). The frame rates for these two datasets are also comparable.

The frame rates for the standard and optimal parameter settings in the AMR cases are similar for all datasets. This shows that the standard setting usually yields good performance and hence almost no user interaction is necessary for finding good rendering parameters. In contrast to this the optimal parameter settings for the octree based algorithm vary strongly even for similarly structured datasets and are therefore difficult to predict. Adjusting the parameters typically requires a higher amount of user interaction.

To summarize: The AMR-based algorithm yields best results for sparse and large datasets. For these kind of data it benefits from the capability of tightly covering complex shaped domains of relevant voxels with a smaller number of boxes, resulting in significant performance gains.
Table 4.2: The table columns list the depth of the hierarchies, the number of created texture bricks, the percentage of the data volume covered by them, the amount of texture memory (after extension to the next power of two and packing them into one texture), the preprocessing times, and finally the frame rates for dataset I: Neurons inside a bee brain, containing $654 \times 993 \times 200$ voxels and rendered with 1200 slices.

<table>
<thead>
<tr>
<th>levels</th>
<th>bricks</th>
<th>volume</th>
<th>texsize</th>
<th>preproc.</th>
<th>fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>1</td>
<td>7</td>
<td>103.0%</td>
<td>208.0 MB</td>
<td>8.0s</td>
</tr>
<tr>
<td>Octree I</td>
<td>8</td>
<td>1503</td>
<td>2.9%</td>
<td>3.1 MB</td>
<td>6.5s</td>
</tr>
<tr>
<td>Octree II</td>
<td>8</td>
<td>615</td>
<td>3.6%</td>
<td>5.1 MB</td>
<td>17.3s</td>
</tr>
<tr>
<td>AMR I</td>
<td>4</td>
<td>333</td>
<td>0.8%</td>
<td>2.0 MB</td>
<td>21.3s+2.7s</td>
</tr>
<tr>
<td>AMR II</td>
<td>4</td>
<td>333</td>
<td>0.8%</td>
<td>2.0 MB</td>
<td>21.3s+2.7s</td>
</tr>
</tbody>
</table>

Table 4.3: Dataset II: Neurons inside a bee brain, containing $566 \times 990 \times 200$ voxels and rendered with 1200 slices.

<table>
<thead>
<tr>
<th>levels</th>
<th>bricks</th>
<th>volume</th>
<th>texsize</th>
<th>preproc.</th>
<th>fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>1</td>
<td>23</td>
<td>103.1%</td>
<td>736.0 MB</td>
<td>13.3s</td>
</tr>
<tr>
<td>Octree I</td>
<td>7</td>
<td>3277</td>
<td>7.7%</td>
<td>15.9 MB</td>
<td>34.6s</td>
</tr>
<tr>
<td>Octree II</td>
<td>7</td>
<td>1907</td>
<td>8.7%</td>
<td>19.0 MB</td>
<td>11.7s</td>
</tr>
<tr>
<td>AMR I</td>
<td>5</td>
<td>1388</td>
<td>4.3%</td>
<td>16.0 MB</td>
<td>34.4s+34.5s</td>
</tr>
<tr>
<td>AMR II</td>
<td>5</td>
<td>1525</td>
<td>3.9%</td>
<td>16.0 MB</td>
<td>34.4s+34.5s</td>
</tr>
</tbody>
</table>

Table 4.4: Dataset III: Vascular tree, containing $528 \times 574 \times 700$ voxels and rendered with 1200 slices.

<table>
<thead>
<tr>
<th>levels</th>
<th>bricks</th>
<th>volume</th>
<th>texsize</th>
<th>preproc.</th>
<th>fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>1</td>
<td>2</td>
<td>101.0%</td>
<td>16.0 MB</td>
<td>0.5s</td>
</tr>
<tr>
<td>Octree I</td>
<td>8</td>
<td>652</td>
<td>8.9%</td>
<td>0.6 MB</td>
<td>4.1s</td>
</tr>
<tr>
<td>Octree II</td>
<td>8</td>
<td>587</td>
<td>7.9%</td>
<td>0.9 MB</td>
<td>6.1s</td>
</tr>
<tr>
<td>AMR I</td>
<td>4</td>
<td>61</td>
<td>8.8%</td>
<td>1.0 MB</td>
<td>0.9s+0.8s</td>
</tr>
<tr>
<td>AMR II</td>
<td>4</td>
<td>333</td>
<td>6.9%</td>
<td>1.0 MB</td>
<td>0.9s+0.8s</td>
</tr>
</tbody>
</table>

Table 4.5: Dataset IV: Molecular conformation dataset, containing $185 \times 202 \times 157$ voxels and rendered with 320 slices.

<table>
<thead>
<tr>
<th>levels</th>
<th>bricks</th>
<th>volume</th>
<th>texsize</th>
<th>preproc.</th>
<th>fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>1</td>
<td>2</td>
<td>101.0%</td>
<td>64.0 MB</td>
<td>2.0s</td>
</tr>
<tr>
<td>Octree I</td>
<td>7</td>
<td>2947</td>
<td>37.6%</td>
<td>14.5 MB</td>
<td>11.0s</td>
</tr>
<tr>
<td>Octree II</td>
<td>7</td>
<td>1814</td>
<td>37.0%</td>
<td>15.0 MB</td>
<td>7.2s</td>
</tr>
<tr>
<td>AMR I</td>
<td>3</td>
<td>619</td>
<td>32.1%</td>
<td>32.0 MB</td>
<td>6.1s+11.4s</td>
</tr>
<tr>
<td>AMR II</td>
<td>3</td>
<td>501</td>
<td>35.0%</td>
<td>32.0 MB</td>
<td>6.1s+13.2s</td>
</tr>
</tbody>
</table>

Table 4.6: Dataset V: Bee brain dataset, containing $749 \times 495 \times 100$ voxels and rendered with 900 slices.

<table>
<thead>
<tr>
<th>levels</th>
<th>bricks</th>
<th>volume</th>
<th>texsize</th>
<th>preproc.</th>
<th>fps</th>
</tr>
</thead>
<tbody>
<tr>
<td>standard</td>
<td>1</td>
<td>2</td>
<td>101.0%</td>
<td>64.0 MB</td>
<td>2.0s</td>
</tr>
<tr>
<td>Octree I</td>
<td>7</td>
<td>2947</td>
<td>37.6%</td>
<td>14.5 MB</td>
<td>11.0s</td>
</tr>
<tr>
<td>Octree II</td>
<td>7</td>
<td>1814</td>
<td>37.0%</td>
<td>15.0 MB</td>
<td>7.2s</td>
</tr>
<tr>
<td>AMR I</td>
<td>3</td>
<td>619</td>
<td>32.1%</td>
<td>32.0 MB</td>
<td>6.1s+11.4s</td>
</tr>
<tr>
<td>AMR II</td>
<td>3</td>
<td>501</td>
<td>35.0%</td>
<td>32.0 MB</td>
<td>6.1s+13.2s</td>
</tr>
</tbody>
</table>
Figure 4.7: The images show volume renderings of dataset I (a bee brain neuron) for the three different approaches: standard (left), AMR (middle) and octree (right). (dataset courtesy of R. Menzel, Freie Universität Berlin)

Figure 4.8: Dataset II: bee brain neuron, left: standard, middle: AMR, right: octree. (dataset courtesy of R. Menzel, Freie Universität Berlin)

Figure 4.9: Dataset III: vascular tree, left: standard, middle: AMR, right: octree. (dataset courtesy of P. Schlag, Robert-Rössle-Klinik and Universitätsklinikum Charité)
Figure 4.10: Dataset IV: molecule conformation, left: standard, middle: AMR, right: octree. (dataset courtesy of D. Baum, Zuse Institute Berlin)

Figure 4.11: Dataset V: bee brain, left: standard, middle: AMR, right: octree. dataset courtesy of R. Brandt and R. Menzel, Freie Universität Berlin)
Chapter 5

Visualization of Structured Adaptive Mesh Refinement Data

Due to the growing popularity of structured AMR schemes for representing multi-scale phenomena in the last years, an increasing number of scientists is in need of appropriate interactive visualization techniques to interpret and analyze time-dependent simulation data defined on this kind of grids. Tools for both, 2D analysis to quantitatively convey the information within single slices and a 3D representation to quickly grasp the overall structure are required.

A naive method for rendering structured adaptive mesh refinement data is to resample the grid function onto a uniform mesh with a cell size corresponding to highest resolved level. The data for the new grid nodes that are not present in the original AMR grid, has to be obtained by interpolation. Though this approach has the advantage that it allows to apply all standard visualization routines available for uniform grids, it is unfeasible even for moderately resolved AMR data due to the enormous amount of memory requirements. As an example consider a time-dependent hierarchy with five levels of refinement, a root level subgrid with $128^3$ cells, an overall refinement factor of 2 and a grid function with 4 bytes per node. Resampling such an hierarchy would result in an uniform grid that requires 256 GByte of memory for each time step.

An alternative approach is to convert AMR data to an unstructured, locally refined hexahedral grid with explicit connectivity information. Though not as disadvantageous as the approach mentioned above, it also increases the memory requirements for storing the data and it further does not take advantage of the regular structure of the subgrids, that allows for fast and efficient rendering.

Another drawback of these conversions is that information about the grid function on different levels of resolution that is present in the original data is lost. Scientists are often interested in comparing the grid functions on different levels of refinement in order to analyze their codes and numerical algorithms, and/or inspect the behavior of the solution at the level boundaries.

Thus converting AMR data to standard grid types is neither a feasible nor a desirable strategy. It is much more advantageous to develop visualization techniques that can
directly handle and utilize the hierarchical structure of AMR data and at the same time take advantage of the fact, that it consists of structured subgrids, in order to accelerate the rendering.

This will be our basic paradigm for all visualization approaches presented in the next sections. This chapter is organized as follows: In the next Section related work in the field of hierarchical visualization techniques is discussed. Next the topics of point location and interpolation for AMR data are addressed. In Section 5.4 we discuss the adaption of indirect volume rendering methods for AMR data, whereas in Section 5.5 direct volume rendering approaches for AMR data are presented.

5.1 Related Work

To our knowledge the first paper in the literature that deals with visualization of AMR data was published in ’93 by Max [56]. Max describes a back-to-front cell-sorting algorithm for AMR grids and employs it for a software-based volume rendering approach and for contour surface extraction.

Norman et al. [59] describe an approach of resampling AMR data to uniform as well as unstructured grids, which allows them to apply standard rendering algorithms, though their approach is feasible only for AMR hierarchies with a small number of refinement levels, as discussed at the beginning of this chapter.

There exist quite a number of papers that deal with the problem of artifacts in triangulations that result from the standard marching cubes algorithm, if it is applied to non-conforming hexahedral grids. Shu et al. [79] use axis-aligned patches to fill the cracks in the multi-resolution isosurface mesh. A disadvantage of this approach is the visually noticeable change of the surface normal between triangles resulting from the marching cubes approach and the triangles of the planar patches.

Shekkar et al. [76] propose a crack-fixing scheme that shifts vertices of triangle edges located on the higher resolved face parallel to the face in order to align with triangle edges of adjacent lower resolved cells. By this some details present in the higher resolved regions might get lost and the resulting meshes contain T-vertices, which might lead to shading artifacts.

Westermann et al. [92] discuss two strategies for dealing with the crack problems for octrees. Their first approach utilizes a conforming split scheme in order to eliminate dangling nodes. Since this approach can lead to a noticeable increase of the number of generated triangles, they propose a second scheme in which coarse triangles which abut higher resolved cells are split up into triangle fans, in order to match the shape of the adjacent smaller triangles. Compared to [76] this has the advantage that the detail information of the higher resolved regions is preserved, though it is feasible only for grids, where adjacent cells differ by at most one level of resolution and dangling nodes that are dependent nodes. Fang et al. [27] extended the conforming split approach proposed in [92] to deal with vertex-centered data defined on unrestricted AMR grids. Their approach still requires that the data at dangling nodes are dependent nodes.
In contrast to the approaches mentioned above, Weber et al. [87] address the problem of isosurface generation for cell-centered data defined on AMR grids. They propose the use of several types of stitching cells to connect cells on different levels of resolution. This approach avoids resampling of cell-centered data, but has the drawback that it restricts the applicability of their algorithm to restricted AMR grids, which guarantee that refined levels are surrounded by at least one layer of cells from the next coarser level.

Two parallel, direct volume rendering approaches for octree data generated by the PARAMESH framework [54] are compared by Ma [53]. The first one performs a resampling of the data to a uniform grid, similar to [59], rendering subregions of the resulting grid on individual processors, whereas the other methods utilizes the octree hierarchy and distributes its leaf nodes.

In [88] Weber et al. apply the stitching scheme presented in [87] to a cell-projection algorithm. Park et al. [64] present a hierarchical splatting approach for AMR data. Kel- ley et al. [33] describe a framework for interactive, parallel volume rendering of remote AMR data. Sub-trees of the AMR hierarchy are distributed on individual processors and composed on the local rendering client. Weber et al. [89] further investigate load-balancing strategies for parallel volume rendering of AMR data.

Visualization methods for time-dependent simulations carried out on locally refined, unstructured grids, have been presented by Polthier et al. [66], Happe et al. [30] and Schmidt et al. [72]. These approaches have in common that they assume the existence of two unstructured grids and associated grid functions at each time step at which the underlying grid structure is changed: the solution before and after grid refinement, respectively grid coarsening. This ensures that on each pair of consecutive time steps the interpolation can be carried out on identical grids.

5.2 Point Location for AMR Data

Since in AMR grids subregions of the computational domain might be covered by multiple cells from different levels of resolution, we have to specify the point location operation, which is important for many visualization methods, as follows:

Given a maximal search depth \( \tilde{l} \) with \( 0 \leq \tilde{l} \leq l_{\text{max}} \), locate the unique cell \( \Omega_{ijk}^l \in \Lambda^l \) with \( 0 \leq l \leq \tilde{l} \), for which

\[
(p \in \Omega_{ijk}^l) \land (p \notin \Lambda^{l+1} \cup \ldots \cup \Lambda^{l_{\text{max}}})
\]

holds.

Notice that \( \tilde{l} \) does not necessarily have to equal the highest resolved level. It is often beneficial to use coarser approximations of the grid function in some subregions, for example in order to accelerate rendering and/or preprocessing times. In this case it should be avoided to interpolate data samples on the highest available level for coarser representations. This will in general violate the sampling theorem and result in aliasing artifacts, since the sampling rate on the coarse levels is too low.
A direct approach for cell location is to inspect each subgrid on all refinement levels \(0 \leq l \leq \tilde{l}\), starting at level \(\tilde{l}\), until a subgrid that covers the considered location is determined. Due to the uniform structure of the subgrids, the associated local coordinates \((u, v, w)\) are given by

\[
(u, v, w) = \left(\frac{p_x - m_x}{d_x} - \left\lfloor \frac{p_x - m_x}{d_x} \right\rfloor, ..., ...\right)
\]

where \(\lfloor \cdot \rfloor\) denotes the floor function, \(m\) the minimal coordinate of the subgrids bounding box and \(d\) the cell size. This search strategy is not feasible for most visualization algorithms, like for example direct volume rendering via raycasting, which involve a large number of point location operations.

A way to speed up this process is to exploit the inherent hierarchical structure of AMR hierarchies. Simulation packages, like for example the AMR code Enzo [63], often do generate nested subgrids, so each \(\Gamma^l_j\) (except for the root level subgrids) is completely contained in exactly one subgrid \(\Gamma^{l-1}_j = \mathcal{P}(\Gamma^l_j)\) on the next coarser level of refinement \(\Lambda^{l-1}\). This tree-structure can be directly exploited for a hierarchical search strategy.

In case the subgrids are not arranged in a nested manner, this tree structure can always be induced in a preprocessing step, as indicated in the following pseudocode:

```plaintext
set minimal bounding box of all \(\Lambda^0\) as \(\Gamma^{-1}\);
insert all \(\Gamma^0_i \in \Lambda^0\) as \(\mathcal{C}(\Gamma^{-1})\);
for all \((\Lambda^l, l = 0, ..., l_{max} - 1)\) {
    for all \((\Gamma^l_i \in \Lambda^l)\) {
        for all \((\Gamma^{l+1}_j \in \Lambda^{l+1})\) {
            if \((\Gamma^l_i \cap \Gamma^{l+1}_j) \not\subseteq (\partial \Gamma^l_i \cup \partial \Gamma^{l+1}_j))\)
                set \((\Gamma^l_i \cap \Gamma^{l+1}_j)\) as \(\mathcal{C}(\Gamma^l_i)\)
        }
    }
}
```

Though for large \(n\) this hierarchical approach is preferable to the linear search strategy mentioned above, it still suffers from the drawback that AMR hierarchies tend to be not well balanced, i.e. the degree of the nodes can vary considerably. Subgrids located on the coarser levels usually have a much larger number of descendants than the higher resolved ones. Subgrids with hundreds of subnodes are not unusual. Thus the traversal of the hierarchy becomes less beneficial. This in particular decreases the efficiency of local search strategies. Even if the next query requests a point that is located in the same subgrid, it is still necessary to inspect the potentially large number of the child nodes.

We address this problem by subdividing the computational domain into axis-aligned blocks \(\mathcal{B}_m^l\) that enclose cells from the same resolution level, i.e. we decompose the grid domain into regions of cells that are either completely refined or unrefined

\[
\Lambda^l = \bigcup_{m=0}^{n} \mathcal{B}_m^l \quad \text{for} \ 0 \leq l \leq l_{max},
\]
Figure 5.1: **Left:** 2D example of a grid, that contains 3 subgrids on the next level of resolution. In order to subdivide the volume into axis-aligned, non-overlapping regions containing only cells of the same refinement level, two split axes are determined. **Right:** Subregions that contains just one subgrid, are finally partitioned in up to 4 subregions (up to 6 in the 3D case)

such that

\[
(B_i^l \subseteq \Lambda^l \setminus (\Lambda^{l+1} \cup \ldots \cup \Lambda^{l_{\text{max}}})) \lor (B_m^l \subseteq \Lambda^{l+1} \cup \ldots \cup \Lambda^{l_{\text{max}}}).
\]

Regarding the discussion above this decomposition should further have the following properties

- \((B_i^l \cap B_j^l) = \emptyset \lor (B_i^l \cap B_j^l) \subset (\partial B_i^l \cup \partial B_j^l)\) for \(i \neq j\), i.e. the interiors of the blocks on the same level should be disjoint,
- the subdivision should consist of a small number of blocks,
- the blocks should be nested, that is for each \(B_i^l\) exists exactly one \(B_j^{l-1}\) with \(B_i^l \subseteq B_j^{l-1}\),
- and the nodes of this nested hierarchy should be of small degree.

In principle one could exploit an octree-data-structure for this purpose. The root node, covering the domain of the AMR root level, is recursively subdivided into 8 child nodes, until each leaf contains only cells on the same level of resolution. The drawback of this approach is that it usually leads to a large number of bricks, since often the partition has to carried out until the subregions contain exactly one cell. As an example consider a subgrid of \(n \times n \times n\) cells, that is separated from the nearest boundary of its parent level by just one layer of cells. Here the octree subdivision scheme would generate \(n \times n\) leaf nodes to cover this layer. Besides high storage requirements this large number of small nodes is disadvantageous for local search strategies.

In the following we propose a different approach that aims at generating a smaller number of blocks. The decomposition starts at the root level. In order to ease the discussion we assume that the root level consists of just one subgrid \(\Gamma_0\). This is no loss of
Figure 5.2: (a): Subgrid configuration where the unmodified decomposition leads to an unnecessarily high number of bricks. (b): By first enclosing the subgrids with a minimal bounding box, and decomposing the outer region, the number of created bricks is reduced!  

...
The splitting procedure described above is continued on each block that consists of cells that are refined by grids on the next level of resolution, until the highest resolved level is reached. This resulting space decomposition results in blocks that are arranged in the manner of an adaptive kD-tree data structure, compare Section 2.4, consisting of three different types of nodes:

- The first type represents blocks that cover cells that are not further refined. These nodes are leaf nodes of the associated kD-tree.

- Nodes representing blocks that cover cells which are refined on the next level on resolution. These also store information about the partition axis and references to the two subnodes, associated with the next two subvolumes. We use nodes of this type to prune the traversal of the hierarchy once a certain level of resolution is reached.

- The third type of nodes is primarily used for tree traversal. It holds information about the next partition plane and references to the next two subnodes.

Information of the number of generated block for several application examples will be given in Section 5.5.5.

5.3 Interpolation

Assuming continuous fields, e.g. those that do not contain jumps, which are for example present at material boundaries or shock fronts, a prerequisite for artifact-free rendering is a globally continuous interpolation of the discrete data samples.

![Cracks in a height field mesh](image)

Figure 5.3: Cracks in a height field mesh, resulting from discontinuous interpolation.
For AMR data this requires special attention at the boundaries of the refinement levels. Problems might arise at dangling nodes, present at common faces of cells that belong to different resolution levels.

5.3.0.1 Vertex-Centered Data

Let us first discuss the case of vertex-centered data, for which typically piecewise trilinear interpolation is employed, if \( C^0 \)-continuity of the interpolant is desired. \( C^0 \)-continuous trilinear interpolation at common faces of adjacent cells of the same resolution level is guaranteed, since the interpolants on both sides degenerate to the same bilinear interpolant on the face. Continuity at boundaries of grid cells with different resolution requires the following:

Firstly, the data values on the boundary nodes that are no dangling nodes (white circles in Figure 5.4) have to coincide on the coarse and fine grid. This is usually ensured in numerical AMR schemes by the restriction step, in which the solution on the coarser grid nodes is updated by the more accurate grid function of the refined regions, compare Section 2.2. In general non-dependent hanging nodes (black circles in Figure 5.4) lead to discontinuities of the piecewise trilinear interpolant, resulting in visible artifacts for most visualization methods, as for example shown in Figure 5.3. There are two alternatives to deal with this problem: One is to replace the data values at hanging nodes by the result obtained via bilinear interpolation of the coarse cell faces, i.e. to convert them to dependent nodes. This typically introduces only minor corrections, because the difference between the fine and coarse solutions is usually small at level boundaries, since most error estimators detect regions that require refinement based on this difference. Further the number of dangling nodes is small compared to the number of all nodes in the grid hierarchy.

Another method to obtain continuity which does not change the grid function, is to modify the underlying grid structure in order to construct a conforming grid. In this approach (conforming split), the coarse cells at level boundaries are replaced by a set of sub-cells of different topology, such that each internal face is shared by exactly two adjacent cells. In Subsection 5.4.3.3, we will present a conforming split scheme for AMR grids, that is capable of handling arbitrary level differences between adjacent cells and requires only two types of split cells.

We apply both techniques in the visualization routines discussed in the following. The first method is more advantageous for direct volume rendering approaches, since the
Figure 5.5: Resampling from cell-centered data to vertex-centered data: Data at the non-dangling vertices (circles) is obtained via trilinear interpolation between the adjacent cell-centered data samples (crosses) using a stencil width that corresponds to the mesh size of the highest resolved adjacent grid. At boundaries missing samples (circles with crosses) are generated via trilinear interpolation on the coarse grid. Data at dangling nodes is obtained via bilinear interpolation between the adjacent non-dangling nodes.

Performance of the methods would be decreased significantly if the uniform structure of the subgrids is partially replaced by hybrid grid patches. For isosurface algorithms the second approach is appropriate, since the conforming split avoids cracks, which arise at level boundaries, even for a globally continuous interpolant, compare Section 5.4.3.

5.3.0.2 Cell-Centered Data

Let us now discuss the case of cell-centered data. Nearest-neighbor interpolation is the natural interpolation scheme this kind of data, since it allows a direct inspection of the data samples. We therefore support this kind of interpolation for cell-centered data in all visualization techniques discussed in the next sections. However, the 'blocky' appearance of the resulting representations, that is especially pronounced on lower resolved regions, might obstruct the perception of the qualitative features of the data, consider for example the isosurface shown in Figure 5.10. Hence higher-order interpolation scheme are
Figure 5.6: Comparison of two height field meshes: a blocky mesh extracted from the original cell-centered data using nearest neighbor interpolation (a) and a smooth mesh extracted from the resampled vertex-centered data using tri-linear interpolation (b). (dataset courtesy of M. Norman, National University of California)

Desirable also for cell-centered data.

One approach is to work on the dual grid, that has vertices at the locations of the data samples, i. e. at the cell centers, and to use trilinear interpolation on this grid structure. Problems arise at the boundaries of the levels, where the regular structure of the dual grid is disturbed. Since one of our requirements is to deal with AMR grids, that allow adjacent cells to differ by more than one level, i. e. unrestricted AMR hierarchies, a stitching cell approach to connect the cells of the dual grid like the one proposed by Weber et al. [87], compare Section 5.1, is not feasible for our purpose.

We therefore decided to resample the cell-centered data into a vertex-centered form, which is better suited for trilinear interpolation, as discussed above. In order to determine the value of the grid function at the grid nodes, we need to construct an interpolation rule for the grid function based on the cell data. According to Figure 5.5, we have to distinguish the following cases:

- Data for vertices $x_{ijk}$ in the interior of a refinement level $\Lambda^l$ is obtained by trilinear interpolation of the eight data samples $\bar{f}$ at the adjacent cell centers, i. e. (assuming cubic cells for simplicity)

  $$f(x_{ijk}) := \frac{1}{8} \sum_{l,m,n=-1,1} \bar{f} \left( x_{ijk} + \frac{1}{2}(lh_0, mh_1, nh_2) \right), \quad (5.1)$$

  where $h$ is the grid spacing on $\Lambda^l$.  

78
• For boundary nodes \( x_{ijk} \in \partial \Omega_l \) that are not hanging nodes, the same interpolation kernel as for interior nodes is used. In this case the width of the kernel is determined by the highest resolved cell that shares the vertex \( x_{ijk} \). Data samples at locations \( x_{ijk} + \frac{1}{2}(\pm h_0, \pm h_1, \pm h_2, ) \) that are not present on this level are obtained by trilinear interpolation on the highest resolved refinement level, that contains the point.

• In order to achieve a \( C^0 \)-continuous trilinear interpolant the data associated with the hanging nodes are obtained by bilinear interpolation between the data at the adjacent non-hanging nodes.

• Boundary values on the root level \( \Lambda^0 \) are generated via extrapolation.

We end up with a vertex-centered hierarchy that ensures continuous trilinear function interpolation, even if adjacent cells differ by more than one level of resolution.

5.4 Indirect Volume Rendering

In this section we will discuss the modifications to some popular indirect volume rendering algorithms that are necessary when they are applied to structured AMR data. The first part of the section deals with approaches for displaying data within planar slices and in the second part a variant of the marching cubes algorithm, suitable for unrestricted, refined hexahedral grids will be presented.

5.4.1 Planar Slices

Our approach for rendering data within planar slices utilizes 2D textures, in order to allow interactive performance even if a large number of extracted slices is displayed simultaneously. For each subgrid that is intersected by the slice \( S \), a separate 2D texture is allocated and initialized with the extracted data samples. Multiple rendering of regions that are covered by subgrids on different resolution levels would result in noticeable artifacts, in particular if semi-transparent colormaps are applied. In order to avoid this, we employ the stencil-buffer, offered by standard graphics APIs.

First the textures for the highest resolved level are processed. Each rendering of a texture updates the stencil buffer, thus preventing the associated regions in the frame buffer from being overwritten by subsequently processed textures from the coarser levels, as indicated in the following pseudocode:

```c
/* enable stencil test */
glEnable(GL_STENCIL_TEST);
/* suppress rendering in regions where stencil is 1 */
glStencilFunc(GL_EQUAL, 0x1, 0x1);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
/* reset buffer */
glClearStencil(0x0);
```
Figure 5.7: Texture offsets: For cell-centered data (a) textures are aligned with cell centers, whereas for vertex-centered data (b), texels are aligned with the vertices. The latter one allows smooth transitions between the different texture patches if bi-linear interpolation is employed and the “dangling” texels are adjusted. In (b) nearest-neighbor interpolation was used, in order to better depict the texel offsets.

\[
\text{for } (l = l_{\text{max}}, \ldots, 0) \{ \\
\quad \text{for all } (\Gamma_l^i \in \Lambda_l \text{ with } (\Gamma_l^i \cap S) \neq \emptyset) \{ \\
\quad\quad \text{render texture of } (\Gamma_l^i \cap S); \\
\quad\} \\
\} \\
\]

For cell-centered data we apply nearest-neighbor interpolation, and the texels are aligned with the centers of the squares that result from the intersection between the slice and the cells, compare Figure 5.7 (a). For vertex-centered data the texels are aligned with grid nodes, respectively with the intersections between the cell edges and the slice, as indicated in Figure 5.7 (b). In this case the textures are rendered using bilinear interpolation. In order to ensure a continuous interpolation across boundaries of textures associated with different levels, we compute the boundary texels that belong to dangling nodes by bilinear interpolation between the adjacent coarse texels, compare Figure 5.8.

5.4.2 Height Fields

Let us now discuss the extraction of height fields for AMR data. For cell-centered data we display them as axis-aligned bars with top-faces perpendicular to the considered data
Figure 5.8: Texture-based rendering of AMR data within orthogonal slices. In the left image nearest neighbor interpolation was applied. The subgrid boundaries are indicated by the white lines. In the middle image the textures are interpolated using bilinear interpolation. Since the ‘dangling texels’ are adjusted appropriately, no interpolation artifacts at the interface between textures of different resolution occur. Utilizing the stencil buffer allows to render semi-transparent textures without changes in the overall transparency in regions that are covered by multiple subgrids, as shown in the right image. *(dataset courtesy of M. Norman, National University of California)*

slice. The height is proportional to the value of the grid function at the projection of the top-face onto the considered slice as shown in Figure 5.6 (a). For vertex-centered data we approximate the resulting surface by a triangular mesh. The data of cells in the interior of a refinement level, which are intersected by the considered slice, is represented by two triangles, compare Figure 5.9 (a).

Problems arise at coarse cells that abut on a boundary of a higher level of resolution. Although adjusting the data at the dangling nodes to obtain slave nodes, as discussed above, ensures a continuous interpolation across the boundary, the resulting surface still contains T-vertices. These can result in shading artifacts, compare Figure 5.9 (c).

So instead of modifying the grid function at dangling nodes, we rather decided to remove these nodes by replacing the triangles of coarse boundary cells by sets of smaller triangles, which provide a transition from the coarse triangles of the interior cells to the higher resolved ones from the adjacent finer level. Therefore we introduce a new central vertex and connect it to the triangle nodes from the adjacent cells that lie on the coarse cell faces, as shown in Figure 5.9 (b).
Figure 5.9: Image (a) shows a height field approximation by a triangular mesh that contains dangling nodes. These can lead to shading artifacts, as depicted in image (c). Replacing coarse triangles at level boundaries by strips of smaller triangles removes the dangling nodes (b) and the shading artifacts (d). (dataset courtesy of M. Norman, National University of California)
5.4.3 Isosurfaces

5.4.3.1 Cell-Centered Data

For cell-centered AMR data in combination with nearest-neighbor interpolation we generate the isosurface by primitives that are coplanar to one of the three coordinate planes. The construction is straight-forward:

- If two neighboring cells on the same level of resolution have associated data values below and above the iso-value, a surface element with the shape of the common face is generated.
- If the two cells belong to different levels of resolution, the smaller face segment of the higher resolved cell is chosen.

An example of a resulting surface is shown in the following Figure:

![Isosurface for cell-centered AMR data. Surface primitives represent isosurface patches between two adjacent cells with data values below and above the iso-value. (dataset courtesy of G. Bryan, Princeton University)](image)

5.4.3.2 Vertex-Centered Data

For vertex-centered data and trilinear interpolation the application of the standard marching cubes algorithm is problematic, since it generates artifacts at the level boundaries, visible as cracks in the isosurface, compare Figure 5.11 (a). In contrast to the display of height field surfaces discussed in the last subsection, the problem remains even for globally $C^0$-continuous, piecewise trilinear interpolants.
Figure 5.11: The left image shows cracks in the triangulation generated by the marching cubes algorithm applied to an SAMR dataset. The right image shows a conforming mesh extracted from the hybrid grid that is obtained by replacing coarse cells at level boundaries by sets of tetrahedral and pyramidal cells. *(dataset courtesy of G. Bryan, Princeton University)*

The origin of the artifacts is the following: Depending on the signature of the face vertices, the one-dimensional isocontour of the intersection between the isosurface of the trilinear interpolant and the face consists of up to two hyperbolas, defined by

\[ s(t) = \frac{(f(x_{00}) - f(x_{10}))t - f(x_{00}) + v_{iso}}{(f(x_{00}) - f(x_{10}) + f(x_{11}) - f(x_{01}))t + f(x_{01}) - f(x_{00})}, \]

where \(x_{ij}\) are the locations of the face nodes.

On the coarse face each of these hyperbolas is approximated by one line segment, whereas on the higher resolved faces a finer approximation is generated, since the bilinear interpolant is sampled at a larger number of locations, compare Figure 5.12 (a). Though for a continuous interpolant the starting-, and end-points of the line-sets, which lie on the edges of the coarse cell, coincide, the polylines will in general not be congruent in the interior of the face. This is only the case, if the hyperbolas degenerate to a straight line, i. e. if

\[ f(x_{00}) - f(x_{10}) + f(x_{11}) - f(x_{01}) = 0, \]

which is holds if the two vectors

\[(1, 0, f(x_{10}) - f(x_{00})), (1, 0, f(x_{01}) - f(x_{11}))\]

are coplanar.

In case the values of the dangling nodes are not dependent nodes, the artifacts are even more severe. Consider for example a central dangling node with a signature that differs
Figure 5.12: Reason for cracks at interfaces between fine and coarse grid cells: On the left side the isocontour on the coarse face is approximated by a straight line, whereas on the refined faces a larger number of isopoints (rectangles) is generated. In general these additional points do not lie on the coarse isoline, which leads to cracks in the resulting isosurface for the three-dimensional case. The figure on the right shows a situation for dangling nodes that are not dependent nodes. In this example no isocontour is generated on the coarse face, though a (discontinuous) isocontour results from the line segments of the refined faces.

from the one of the adjacent, non-dangling nodes. On the refined side the marching cubes algorithm would generate a closed isocontour, whereas on the coarse side no line-segment is present at all, as shown in Figure 5.12 (b).

5.4.3.3 Conforming Split

According to the discussion above we need a generalization of the marching cubes algorithm, that

- can handle non-restricted AMR hierarchies,
- can handle dangling nodes, that are not dependent nodes,
- and generates meshes without T-vertices, in order to avoid shading artifacts.

Due to these constraints we decided to generalize the conforming split approach discussed in [92] to handle locally refined hexahedral grids, with adjacent cells that differ by an arbitrary number of refinement levels.

Coarse cells that abut on a level boundary are replaced by two types of cells, namely tetrahedra and pyramids. Therefore a new node is generated in the center of these cells, with a data value obtained by trilinear interpolation. For faces that are shared by a cell on the same level of resolution, one pyramidal cell with the new vertex in the center as its apex and the face as its base is generated.
Figure 5.13: Illustration of the conforming split scheme. Pyramidal cells are employed to remove dangling nodes in the interior of the coarse face (left image), while tetrahedral cells are inserted according to the configuration of dangling nodes at the cell edges (right image).

For faces at the level boundary, a set of pyramids is generated, depending on the number of dangling nodes at the interior of the face. These pyramidal share the newly generated center node as a common apex, compare left image in Figure 5.13.

Figure 5.14: The 16 possible vertex configurations for tetrahedral cells can be reduced to 3 topologically different ones. Two of them lead to triangulations, as shown on the left side. Analogously the topologically different cases for pyramidal cells are shown on the right.

In the case that the dangling nodes that are not dependent nodes, it is further necessary to remove dangling nodes that are located on the edges of the cells. Depending on the configuration of these nodes the pyramidal cells are further split up into sets of tetrahedra as illustrated in the right part of Figure 5.13. Notice that this is necessary also for coarse cells that share only an edge with a refined cell.
During the isosurface extraction step each cell is processed separately. The topologically different cases for tetrahedral and pyramidal cells are shown in Figure 5.14. Figure 5.15 shows a rendering, in which triangles generated from different types of cells are highlighted in different colors.

A drawback of the approach discussed so far is that due to the increased number of cell primitives in the hybrid grid, also the number of generated triangles is increased, if compared to isosurfaces (with cracks), that result from the unmodified, standard marching cubes. In order to reduce the number of triangles, we employ the compact cubes algorithms [57] in a post-processing step.

5.4.3.4 Results and Discussion

We applied the algorithm to two AMR datasets - the results from galaxy cluster formation simulations. The first one consists of 984 subgrids on 7 levels of refinement; the second one contains 1939 subgrids, distributed on 8 refinement levels. The measurements were performed on a 2.0 GHz PENTIUM4-SYSTEM equipped with a 128 MBytes ATI-RADEON 9200 graphics board. The results are given in Table 5.16.
Figure 5.16: The table lists the preprocessing times for the detection of the location of dangling nodes, as well as the generation of the cell configurations for the conforming split, which has to be performed only once per dataset. The next three entries state the times for the actual extraction of the isosurface and the number of generated triangles for the standard marching cubes that only operates on the original hexahedral cells and leads to cracks at the boundaries, as well as for the adaptive variant discussed above, in both cases with and without triangle reduction utilizing the compact cubes approach.

The number of triangles resulting from the adaptive algorithm was about 60% larger if compared to the isosurfaces with cracks resulting from the standard marching cubes algorithm. Post-processing both surfaces using the compact cubes algorithm, reduced the increase to 25%. The extraction times where comparable for the different methods. Figure 5.4.3.4 shows a resulting rendering.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>preprocessing</th>
<th>standard MC</th>
<th>standard CC</th>
<th>adaptive MC</th>
<th>adaptive CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>(7.0+5.0) s</td>
<td>107,000 (1.6 s)</td>
<td>58,000 (1.8 s)</td>
<td>164,000 (2.1 s)</td>
<td>72,000 (2.3 s)</td>
</tr>
<tr>
<td>II</td>
<td>(10.0+12.0) s</td>
<td>332,000 (4.4 s)</td>
<td>190,000 (4.6 s)</td>
<td>527,000 (4.8 s)</td>
<td>240,000 (5.0 s)</td>
</tr>
</tbody>
</table>

Figure 5.17: Nested semi-transparent isosurfaces depicting regions of constant gas density inside a proto-galaxy. *(dataset courtesy of T. Abel, Stanford University)*

88
5.5 Direct Volume Rendering for Structured AMR Data

In this section we present a hardware-accelerated volume rendering algorithm as well as a software-based raycasting approach for structured AMR data, that directly exploit the hierarchical structure in order to achieve fast rendering performance even for highly resolved datasets.

5.5.1 3D Texture-Based Volume Rendering

A possible approach for volume rendering of AMR data via 3D textures is based on the utilization of the stencil buffer, similar to the algorithm for slice extraction discussed in Section 5.4. In this case a separate 3D texture is allocated for each subgrid and a stack of slices $S_i$, oriented perpendicular to the viewing direction, is extracted and blended back-to-front in the frame buffer:

```c
define 3D textures;
enable stencil-buffer test;
enable blending;
/* loop over all slices from back-to-front */
for all ($S_i$, $i = 0, ..., n$) {
    clear stencil-buffer;
    /* loop over all levels from fine to coarse */
    for all ($\Lambda^l$, $l = l_{max}, ..., 0$) {
        /* loop over all subgrids */
        for all ($\Gamma^l_k \in \Lambda^l$) {
            render ($S_i \cap \Gamma^l_k$);
        }
    }
}
```

However, this approach has some drawbacks: Volume rendering is fill-rate limited and the stencil-buffer test is performed in the last stage of the rendering pipeline, compare Section 3.1, so the time consuming interpolation operations are still performed for regions that do not contribute to the final image at all.

Secondly, frequent texture-I/O operations will decrease the rendering performance, since the volume is processed in the order of slices, which increases the number of texture switches. This is especially disadvantageous if the total size of textures required to represent the AMR hierarchy exceeds the amount of available texture memory.

It is more advantageous to avoid multiple processing of regions that are covered on different levels of resolution. We therefore decompose the data domain into axis-aligned blocks $B^l_m \subset \Lambda^l$, with

$$ (B^l_i \cap B^l_j) = \emptyset \lor (B^l_i \cap B^l_j) \subset (\partial B^l_i \cup \partial B^l_j) \quad \text{for } i \neq j, $$

that consists either of cells that are refined by subgrids, or of cells which are not further refined. Each block is processed separately during rendering phase, so it has to be ensured

89
that no subsets of the blocks build visibility cycles for any viewpoint. In Section 5.2 we proposed a decomposition that fulfills all of these constraints. In particular the resulting blocks are arranged in a kD-tree structure, allowing efficient determination of the viewpoint-consistent order for each viewpoint.

To reduce the amount of additional texture memory required by the “power-of-two” restrictions of the graphics-API, respectively graphics-hardware, we apply the texture-packing approach discussed in Section 4.4.

As discussed above we employ nearest-neighbor interpolation for cell-centered AMR data and trilinear interpolation for vertex-centered data. In the first case the texels are aligned with the centers of the cells, while in the second one they are aligned with the vertices of the grid. To avoid artifacts originating from discontinuities between sibling subgrids, adjacent texture-blocks share a row of data samples at their common boundary faces and the data at dangling nodes has to be replaced to the interpolated texel values of the abutting, coarse texture.

5.5.2 Opacity Corrections

If a block is selected for rendering, as discussed in Section 5.5.4, it is processed as in the standard approach for volume rendering via 3D textures. Each texture is sampled on slices perpendicular to the viewing direction, which are blended in the frame buffer.

Since texture-based volume rendering is fill-rate limited, it is advantageous to reduce the number of interpolation operations by adapting the distance of the textured slices according to the resolution of the associated subgrids. If blocks on level 0 are rendered with slice distances $\Delta_0$, the slice distance for blocks on level $l$ is given by $\Delta_l = r\Delta_0$, where $r$ denotes the refinement factor. Slices for the different blocks are aligned as indicated in Figure 5.18.

The texel alignment ensures $C^0$-continuity for vertex-centered data, if texels that correspond to hanging nodes are obtained by bilinear interpolation between the texels on the coarse face. Nevertheless, since blocks from different levels are rendered with varying sample distances, the opacity entries of the colormap have to be adjusted in order to avoid noticeable differences in the overall transparency between blocks from different resolution levels.

Let $\alpha_i(0)$ denote the i-th colormap's opacity entry used for rendering root level blocks.
According to Equation (3.18), the opacity is related to the absorption coefficient $\kappa_i$ via

$$\alpha_i(0) = 1 - T_i = 1 - e^{-\kappa_i \Delta_0}.$$  

So one yields the following relation for the opacity entries for rendering blocks on refinement level $l$

$$\alpha_i(l) = 1 - e^{-\kappa_i \Delta_l} = 1 - \left( e^{-\kappa_i r^{-l} \Delta_0} \right) = 1 - \left( 1 - \alpha_i(0) \right)^{\frac{1}{r^l}} \quad (5.2)$$

For each level of the hierarchy a separate colormap is precomputed and activated prior to rendering the separate blocks.

Even with these opacity correction artifacts might remain in the resulting renderings. They are due to small regions at level boundaries, where the sample distance does not correspond to the opacity correction according to Equation (5.2). Weiler et al. addressed this problem in detail in [90]. They present an efficient algorithm to detect these problematic regions and render the corresponding slice parts with the correct opacity that corresponds to the actual sample distance in these regions.

### 5.5.3 Raycasting

For the raycasting approach we also employ the decomposition of the data domain into blocks $B_i$ of cells from the same resolution level, since it accelerates the point location operation. In contrast to the hardware-accelerated algorithm, the tree is traversed in a front-to-back order, to allow for “early-ray-termination” once the opacity of the ray exceeds a certain threshold. Each block is processed separately. First the bounding box is scan-converted and for each pixel the intensity contribution of its ray-segment, which results from the intersection between the ray and the blocks bounding box, is computed according to (3.11)

$$I(s_n) = I(s_{n-1})T_n + b_n.$$  

Here $I(s_{n-1})$ is the accumulated intensity of the pixels ray-segments that have already been processed. We support the standard as well as the adaptive integration scheme described in Chapter 4 for the numerical approximation of $T_n$ and $b_n$. In the last step the updated intensity values for the pixel are written into the frame-buffer and the next ray-segment of the block is processed.

### 5.5.4 Adaptive Block Selection

The kD-tree structure is utilized for traversing the separate blocks in a view-consistent order. If a node that is associated with a block is processed, two cases have to be distinguished:
• The node is a leaf node, indicating that the covered cells are not further refined and thus have to be rendered.

• The node is not a leaf node, i.e. it represents a region that is further refined.

In the second case, there are two alternatives: to render the region with the actual level of detail, or to further decent the subtree, since higher visual accuracy is required.

We base this decision upon the projected extend of the subgrid cells in screen space. If the projected size is smaller than the extend of a selectable number of pixels, the block is selected for rendering and the traversal of the subtree is stopped. In order to quickly estimate the maximal screen space extent of the subgrid cell, we project a ball centered at the grids bounding box corner closest to the viewpoint and with a diameter equal to the grid cells diagonal. In addition a maximal level at which the hierarchy traversal is stopped can be specified. Lower resolution can be used during user interaction like rotation or zooming, while a deeper traversal of the hierarchy is performed for still images. A combination of the methods can be used to guarantee a desired lower bound of the frame rate.

5.5.5 Results and Discussion

The measurements were performed on a SGI ONYX2-SYSTEM with a 195 MHz MIPS R10K processor and a single INFINITEREALITY2 graphics-pipeline with two RM7 raster managers and 64 MBytes of texture memory each. The size of the viewport was $800 \times 800$ pixels. We applied the algorithms to three different datasets with increasing complexity. See Figures 5.19 to 5.21 for detailed information and resulting renderings.

The results of the measurements are shown in Table 5.1. It lists the number of generated leaf blocks, the preprocessing time for allocating the block hierarchy and texture packing as well as resampling in case of cell-centered data, the percentage of texture memory reduction achieved by texture packing and the size of the resulting texture. An average number of 3 to 4 leaf blocks per subgrid was created, independent of the depth and total number of grids of the hierarchy. The average texture memory reduction achieved by packing was about 45%.

Table 5.2 shows the associated frame rates for the root level data, the full hierarchy and the close-up view on the refined part for the viewer positions chosen in Figure 5.19.

<table>
<thead>
<tr>
<th>Dataset</th>
<th># leaf blocks</th>
<th>preprocessing</th>
<th>ratio</th>
<th>texture memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset I</td>
<td>345</td>
<td>0.2 s</td>
<td>45%</td>
<td>1 MB</td>
</tr>
<tr>
<td>Dataset II</td>
<td>970</td>
<td>1.2 s</td>
<td>43%</td>
<td>16 MB</td>
</tr>
<tr>
<td>Dataset III</td>
<td>3370</td>
<td>5.8 s</td>
<td>46%</td>
<td>16 MB</td>
</tr>
</tbody>
</table>

Table 5.1: This table lists the number of generated leaf blocks, the preprocessing times for allocating and packing the textures as well as resampling, the achieved texture memory reduction and the resulting size of the packed texture.
Figure 5.19: Dataset I, resulting from an AMR galaxy cluster simulation, consists of 91 grids on 7 levels of refinement. The (resampled) root level contained $33^3$ samples and was rendered with 120 slices, the more refined grids with respectively more, as discussed in Section 5.5.2. If resampled to a uniform grid, the grid would contain more than $4.000^3$ data samples, corresponding to about 70 GByte of texture memory. (left) root level, (middle) full hierarchy, (right) associated bounding boxes. (dataset courtesy of M. Norman, National University of California)

Figure 5.20: Dataset II represents a hierarchy consisting of 359 grids on 4 levels of refinement. The root level contains $95 \times 63 \times 14$ data and was rendered with 200 slices. This AMR hierarchy was generated from an uniform confocal microscopy dataset with $749 \times 495 \times 100$ cells utilizing an opacity based importance criterion. Regions with associated opacity values below a certain threshold are represented at coarser resolution, based on the algorithm proposed in Section 4.4. Rendering the uniform dataset with the standard approach for texture-based volume rendering resulted in frame rates below 2 fps. The amount of texture memory in this case was 64 MBytes. (left) root level, (middle) full hierarchy, (right) associated bounding boxes. (uniform dataset courtesy of R. Brandt and R. Menzel, Freie Universität Berlin)
Figure 5.21: Dataset III is another AMR hierarchy resulting from a cosmological simulation that consists of 813 grids distributed on 9 levels of refinement. The (resampled) root grid contains $129^3$ data samples and was rendered with 250 slices. If resampled to an uniform grid, the grid would contain about $66,000^3$ data samples, resulting in an amount of $2.7 \times 10^8$ MBytes of texture memory. (left) root level, (middle) full hierarchy, (right) associated bounding boxes. (dataset courtesy of G. Bryan, Princeton University)

to 5.21. The last entry represents the frame rate achieved by rendering the full hierarchy in the mode described in Subsection 5.5.4, i.e. the subtree traversal is stopped, once the cells of the grid associated to the subtree root node have a screen space extension that is smaller than a pixel.

For all datasets (almost) interactive frame rates were achieved $^1$. The frame rates were minimal for the close-up views, since the covered screen space is maximal for these viewpoints. As the performance results for the third dataset show, the subpixel criterion for block selection can result in significant performance gains. In general the effect is more pronounced for deep hierarchies with a large number of subgrids on the more refined levels. Rendering the datasets with the stencil buffer approach as discussed in Section 5.5.1 was about three times slower than the approach that utilizes the domain decomposition.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>root</th>
<th>full</th>
<th>close-up</th>
<th>full adap.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset I (hardware)</td>
<td>10.4</td>
<td>6.7</td>
<td>2.0</td>
<td>7.2</td>
</tr>
<tr>
<td>Dataset II (hardware)</td>
<td>10.1</td>
<td>3.2</td>
<td>2.0</td>
<td>3.2</td>
</tr>
<tr>
<td>Dataset III (hardware)</td>
<td>6.5</td>
<td>1.4</td>
<td>1.1</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Table 5.2: This table shows the frame rates for the root level data, the full hierarchy, the close-up view on the refined part and the view-dependent rendering.

$^1$Notice that the frame rates are more than two times higher on actual graphics hardware.
5.6 Visualization of Time-Dependent AMR Data

For analysis purposes of scientific data, like animation, feature identification, or feature tracking, the underlying non-discrete time-dependent function has to be faithfully reconstructed from the grid function. This is done by spatial and temporal interpolation.

Appropriate interpolation methods may improve the results of data analysis greatly. Furthermore, they help to reduce the amount of grid data to be saved and handled. For instance in large numerical simulations this allows to store fewer time steps and nevertheless create smooth animations that display the underlying process without discontinuities or cracks. Even if numerical methods that provide dense output by maintaining and internally evaluating some interpolants are available, it is usually unfeasible to store all the data due to the immense storage requirements. Only information that can not be regenerated by spatio-temporal interpolation methods should be saved.

In this section we describe an approach that allows the generation of dense output for time-dependent AMR data. In principle temporal interpolation approaches for unstructured grids like proposed by Polthier et al. [66], Happe et. al. [30] or Schmidt et al. [72], compare Subsection 5.1, could be applied also in the case of AMR data. These approaches require the storage of the grid structures and grid functions before and after grid adaption for each time step. In this case the temporal interpolation can be performed on pairs of identical grids.

But for realistic AMR simulations, which often contain dozens of refinement levels, and typically evolve several scalar and vector quantities, this would require to store huge amounts of data, because the temporal step size usually doubles between two consecutive levels of refinement, i.e. increases exponentially. AMR simulation codes therefore often store data only for time steps that correspond to root level updates.

We therefore propose a different approach in which intermediate grid hierarchies are generated in order to connect the given key-frames hierarchies. An imaginable approach for this would be to identify corresponding subgrids present in the key-frames, and to interpolate their position and sizes for the intermediate steps. But this would in general result in overlapping grids on the same level of refinement and it is also not clear how to proceed in case a subgrid has no corresponding ‘partner’ on the next frame. Instead we propose an approach that

- generates an intermediate grid hierarchy by merging the cells on each refinement level that is present in the key-frames,
- uses a clustering algorithm to induce a nested grid structure on the resulting collection of cells,
- projects the grid functions of each key-frame to such an ‘merged’ grid hierarchy and finally
- generates the intermediate grid functions by interpolating between each set of corresponding data samples on these ‘merged’ hierarchies.
5.6.1 Generation of Intermediate Grids

The temporal refinement scheme for AMR data was described in Subsection 2.2.4. The resulting change of the underlying grid structure complicates the interpolation of intermediate time steps during the visualization phase. We can propose the problem as follows:

Given a set of grid hierarchies and associated grid functions \( (H(t), f^l(t)) \)

at discrete time steps \( t_0, t_1, ..., t_m \), with potentially different topology, we want to generate intermediate grid hierarchies \( H(t) \), as well as interpolated grid functions \( f^l(t) \) for \( t \in ]t_i, t_{i+1}[ \).

Let us first address the construction of the intermediate grid hierarchies. We make the following assumptions, which are usually fulfilled by the numerical schemes:

- The root-grid structure \( \Lambda^0(t) \) remains constant for all time steps \(^3\), and
- the spatial refinement factors between two consecutive levels \( (\Lambda^l(t), \Lambda^{l+1}(t)) \) does not change in time.

In a first step, the refinement levels of the intermediate grid are generated. This is done by merging the subgrids for each level of the key-frame hierarchies:

\[
H(t) = H(t_0) \cup H(t_1) \cup ... \cup H(t_m)
\]

\[
= \bigcup_{l=0}^{t_{\text{max}}} \left( \Lambda^l(t_0) \cup \Lambda^l(t_1) \cup ... \cup \Lambda^l(t_m) \right).
\]

Here \( t_{\text{max}} \) denotes the maximum number of refinement levels present in the set of considered time steps. By this merging of corresponding levels, in general we loose the subgrid structure present in the keyframe hierarchies, as illustrated in Figure 5.22. The resulting collection of cells on each level could be stored as an unstructured hexahedral grid with explicit connectivity information. But in terms of memory efficiency and performance it is more advantageous to reintroduce a structure of disjoint subgrids on these unions of cells, since the rendering algorithms work faster on blocks with implicitly given (trivial) connectivity.

So for each \( \Lambda^l(t) \) of the intermediate hierarchy, we require a partition into axis-aligned, non-overlapping rectangular subgrids \( \Gamma^l(t_i) \), such that \( \Lambda_i(t) \subseteq \bigcup \Gamma_i^l(t) \). We achieve this by utilizing the clustering algorithm described in Subsection 2.2.3. In principle the clustering could be applied at once to all subgrids on the same level, but this would result in too high computational efforts for computing the signature lists, due to the large number of cells contained in the minimal bounding box enclosing the higher resolved levels. Thus we perform the clustering procedure per subgrid rather than per level.

\(^2\)The number of key-frames required for this depends on the order of the interpolation function that is applied to obtain the associated grid function, compare Subsection 5.6.2.

\(^3\)This assumption can be weakened. It just has to guaranteed that the cells on the coarsest level of refinement are not shifted or rotated against each other.
Per assumption $\Gamma^0$ remains unchanged, i.e. $\Gamma^0_0(t) := \Lambda^0_0(t) = ... = \Lambda^0_{m}(t)$. Suppose we want to generate the subgrids of a grid $\Gamma^l_q(t) \subset \Lambda^l(t)$. The signature list for its index field is initialized as follows:

$$S(i, j, k) = \begin{cases} 
1, & \text{if } \exists t_s \in (t_0, ..., t_m) \text{ with } \Omega^l_{ri,rj,rk} \subset \bigcup_{s=0}^{m} \Lambda^{l+1}(t_s) \\
0, & \text{otherwise}, 
\end{cases}$$

that is, a cell in $\Gamma^l_q(t)$ is marked for clustering, if it is refined by the next finer level in at least one of the key-frames. This signature list is passed to the clustering algorithm, which generates a set of subgrids $\Gamma^{l+1}_{0}(t), \Gamma^{l+1}_{1}(t), ...$ that belong to the next level $\Lambda^{l+1}(t)$. This procedure is recursively repeated for each of the newly created subgrids, until the maximal level $l_{\text{max}}$ is reached.

### 5.6.2 Temporal Interpolation of Grid Functions

Next the vertex-, respectively cell-centered grid functions $f^l(t), \overline{f}^l(t): \Lambda^l(t) \mapsto \mathbb{R}$ associated with the intermediate grid hierarchy $\mathcal{H}(t)$, are constructed. Two cases have to be distinguished for each cell $\Omega^l_{ijk} \in \Lambda^l(t)$:

- $\Omega^l_{ijk} \in \Lambda^l(t_s) \forall t_s \in (t_0, ..., t_m)$,
- $\exists t_s \in (t_0, ..., t_m) \text{ with } \Omega^l_{ijk} \notin \Lambda^l(t_s)$.

In the first case $f^l_{ijk}(t)$ can simply be computed from the set of given functions values $f^l_{ijk}(t_s)$ at the given keyframes. In the second case, in at least one of the given hierarchies no corresponding cell on the refinement level $l$ exists, so we have to apply some
form of interpolation on the grid function on the coarser levels of resolution to obtain it. Let us assume that this is the case for $\Lambda^l_i(t_s)$ in the following. Because of the nesting property of the levels, the cell is covered by at least one coarser cell. Let $\Omega^l_{ijk}$ denote the cell on the finest level $\tilde{l} \leq l$ that contains $\Omega^l_{ijk}$. In the cell-centered case we perform a nearest-neighbor interpolation, i.e. $f^l_{ijk}(t_s) := f_{\tilde{l} \tilde{i} \tilde{j} \tilde{k}}(t_s)$. For vertex-centered grid functions, $f^l_{ijk}(t_s)$ is obtained by trilinear interpolation within $\Omega^l_{\tilde{i} \tilde{j} \tilde{k}}$.

For larger hierarchies determining which of the two cases holds for each cell and collecting the associated data samples can be an expensive operation. In order to accelerate this procedure we resample each grid function $f^l(t_s)$ onto a grid with the topology of the intermediate hierarchy in a first step. This involves some interpolation in order to obtain data values for fine cells or vertices that are not present in the given set of hierarchies. After this preprocessing the grid function can be computed much more efficient, since now for each subgrid $\Gamma^l(t)$ there exist corresponding subgrids $\Gamma^l(t_s), \Gamma^l(t_{s+1}), ...$. This is especially advantageous if more than one intermediate time step for the same subset of key-frames has to be generated.

We employ three different temporal interpolation schemes. The first one is $C^0$-continuous piecewise linear interpolation

$$f^l_{ijk}(t) = \frac{t_{s+1} - t}{t_{s+1} - t_s} f^l_{ijk}(t_s) + \frac{t_s - t}{t_s - t_{s+1}} f^l_{ijk}(t_{s+1})$$

(5.6)

The second one is $C^1$-continuous cubic Hermite interpolation. So besides the function values for $t_s$ and $t_{s+1}$, also the first derivative at these time steps has to be taken into account

$$f^l_{ijk}(t_{s+1}) := f^l_{ijk}(t_s) H^3_0(t) + \left( \frac{d}{dt} f^l_{ijk}(t_s) \right) H^3_1(t) +$$

$$f^l_{ijk}(t_{s+1}) H^3_2(t) + \left( \frac{d}{dt} f^l_{ijk}(t_{s+1}) \right) H^3_3(t) .$$

(5.7)

(5.8)
Figure 5.24: Three interpolated frames from dataset II, a simulation describing a supernova explosion visualized by texture-based volume rendering. (data set courtesy of T. Abel, Stanford University)

In case the first derivative of the grid functions is not available during the visualization phase, we generate a Catmull-Rom spline, as discussed in Section 2.3. This implies that we require the grid function values at four successive time steps in order to obtain an approximation of the first derivatives at $t_s$ and $t_{s+1}$.

It is possible to obtain more precise values for the first derivative of the grid functions in the case that the grid function represents some conserved quantity that fulfills a conservation law of the form

$$\frac{\partial}{\partial t} \rho(\vec{x}, t) = \text{div } \vec{j}(\vec{x}, t). \quad (5.9)$$

Here $\rho(\vec{x}, t)$ denotes the density of the conserved quantity and $\vec{j}(\vec{x}, t)$ is the associated current. An common example is the case of mass conservation in hydrodynamic simulations. Here $\rho(\vec{x}, t)$ denotes the mass density and the current is computed from the density and the velocity field $\vec{v}(\vec{x}, t)$ via $\vec{j} = \rho \vec{v}$. Since usually in hydrodynamic simulations besides the mass, respectively density fields, also the associated velocity vector fields are stored, we can compute the derivative of the scalar field according to equation (5.9). The divergence of the current is approximated by the flux of the mass through each of the cells faces

$$\text{div } \vec{j} = \text{div } (\rho \vec{v}) = \frac{1}{V_{\text{cell}}} \sum_{i=0}^{5} \rho_i \, A_i \, \vec{n}_i \, \vec{v}_i. \quad (5.10)$$

$\rho_i, \vec{v}_i$ denote the density and velocity fields evaluated at the $i$-th face and $A_i, V_{\text{cell}}, \vec{n}_i$ are the face area, its volume and the outward-oriented face normals.

### 5.6.3 Results and Discussion

The performance was tested on a SGI Onyx3 system on a single 500 MHz MIPS R14000 processor. Dataset I is a result from a cosmological simulation of the formation of stars in the early universe with a root grid resolution of $128^3$ cells, 8 levels of
Figure 5.25: Comparison of three different time interpolation schemes for the analytical dataset III, a damped cosine wave traveling along the x-axis with constant velocity. For linear interpolation oscillations occur, whose minimal peak amplitude during the considered time interval is depicted by the semi-transparent plane. The oscillations decrease for the Hermite interpolation (middle image) and vanish for the flux-based approach (right image). The cell-centered data was rendered using constant spatial interpolation.

refinement and about 2000 grid per time step, compare Figure 5.23. Dataset II depicts a supernova explosion with 8 levels of refinement and about 1600 grids per time step. We took 10 time steps and generated 8 intermediate frames for each pair using linear and Hermite interpolation, with estimated first derivatives. Figure 5.24 shows some volume rendered images of the sequences. The resulting animations show slight oscillations in some parts for linear interpolation, which decrease for Hermite interpolation. Besides that the resulting animations are smooth.

For illustrating the differences of animation quality of the different interpolation schemes we choose an analytical example as dataset III. It shows a damped cosine oscillation that moves along the x-axis with constant velocity. The $64 \times 32 \times 32$ root grid is refined two times and contains about 1100 sub grids at each of the 20 keyframes. We compared the animation quality of linear, Hermite and flux-based interpolation by generating 8 intermediate frames per pair of time steps. Figure 5.25 shows volume rendered images of the resulting sequences. The animation shows disturbing oscillations for linear interpolation were visible. They decreased for Hermite interpolation with estimated derivatives and vanished for the flux-based interpolation, where the knowledge of the velocity vector fields are taken into account.

Information about performance and memory requirements is given in Figure 5.26. The number of subgrids in the merged hierarchies is decreased by about 20% compared to the number of subgrids present in the stored hierarchies. As can be seen in the table the amount of additional memory requirements and the times for grid generation where highest for the Hermite interpolation, since 4 keyframes had to be merged in this case. But the space increase was still less than 30% in all examples. The middle row depict the times for grid generation and keyframe projection, which has to be carried out only if the subset of keyframes used for the interpolation is changed. Again it was highest for Hermite interpolation, but with less than 7 seconds even for the 2000 grid dataset it
<table>
<thead>
<tr>
<th>Dataset</th>
<th>increase of cells</th>
<th>grid generation</th>
<th>interpolation</th>
</tr>
</thead>
<tbody>
<tr>
<td>I (linear)</td>
<td>7%</td>
<td>3.4 sec</td>
<td>0.2 sec</td>
</tr>
<tr>
<td>I (Hermite)</td>
<td>12%</td>
<td>6.6 sec</td>
<td>0.8 sec</td>
</tr>
<tr>
<td>II (linear)</td>
<td>11%</td>
<td>2.2 sec</td>
<td>0.1 sec</td>
</tr>
<tr>
<td>II (Hermite)</td>
<td>15%</td>
<td>3.5 sec</td>
<td>0.3 sec</td>
</tr>
<tr>
<td>III (linear)</td>
<td>20%</td>
<td>1.8 sec</td>
<td>0.1 sec</td>
</tr>
<tr>
<td>III (Hermite)</td>
<td>30%</td>
<td>4.2 sec</td>
<td>0.3 sec</td>
</tr>
<tr>
<td>III (flux-based)</td>
<td>20%</td>
<td>3.5 sec</td>
<td>2.0 sec</td>
</tr>
</tbody>
</table>

Figure 5.26: The first two columns denote the increase in the number of cells for the intermediate time steps relative to the given keyframes. The third column states the times for generating the intermediate grid and projecting the given grid functions. This has to be carried out only if the keyframes change. The last column gives the time for interpolation of the intermediate grid function.

still admits a on-the-fly generation during the visualization phase. Due to the keyframe projection step the times for the interpolation (right row) are short, which is advantageous if more than one intermediate frame is generated for a constant set of keyframes.

5.6.4 Future Work

There are several ways to extend the presented algorithm. Higher order interpolation schemes could be implemented for spatial interpolation during the prolongation step. Further it would be interesting to combine the presented approach with feature tracking algorithms. Also it seems promising to adapt the order of temporal interpolation to the rate of change of the underlying data. It might be beneficial to use lower order temporal interpolation for subgrids with slowly varying data and higher order interpolation for subgrids with rapidly changing data (which is usually the case for the higher resolved levels).
5.7 Rendering the First Star in the Universe - A Case Study

The remainder of this section reports on the application of the texture-based volume rendering algorithm in a collaboration with physicists, to render a time-dependent AMR simulation of the evolution of the first stars in the universe, starting with its formation out of gaseous interstellar matter up to its death in a supernova explosion. Parts of the data contained up to 27 levels of refinement, resolving an initial coarse computational domain of several thousand light years down to the region the newborn protostar. The volume renderer was applied to display the 3D density distribution of the interstellar gas. The voice and gesture controlled CAVE application VIRTUAL DIRECTOR [83] was utilized to define camera paths following the interesting features in the data domain. Background images were created from another cosmological simulation, with the STAR RENDERER, developed at PIXAR ANIMATION STUDIOS. Parts of the resulting renderings were broadcasted on the DISCOVERY CHANNEL program “The unfolding Universe”, aired in summer 2002.

5.7.1 The Simulation

The first star datasets [2] were computed with the AMR code Enzo [16, 63], designed for high-resolution cosmological structure formation simulations. The simulations follow the hydrodynamics, gravity of the gas and the dark matter, the chemical rate equations
following the non-equilibrium chemistry of the formation of ions and molecules, as well as all relevant radiative processes. It represents and ab initio calculation of the formation of the first star in the universe. It has no free variables. The simulation starts at about 400,000 years after the big bang and evolves it for approximately 200 million years. We use frames typically spaced by hundreds of thousands years in the dynamic sequence.

The initial coarse grid, representing the root node of the hierarchical data structure, covered a computational domain of about 18,000 light years.

To visualize the formation of the first star we zoom into the static data set of an equal but further evolved simulation of the same type. The simulations find that a single massive star is formed per proto–galactic object of $\sim 10^6 M_\odot$. Consequently, it is known that it will die in a supernova. To follow the explosion the simulation ejects the typical energy of a supernova explosion in ten solar masses of material at a scale of approximately a thousand times the distance between earth and the sun. As this simulations evolves a dense shell is formed and expands very rapidly. The typical timescales between frames is now less than thousands of years as compared to the hundreds of thousands of years for the first sequence. Look at Figures 5.27 to 5.30 for more details about the three different parts of the simulation.

5.7.2 VR Environment

To analyze the AMR data and create camera paths to highlight the interesting aspects for the wide-screen visualization, the team used VIRTUAL DIRECTOR [83] in a room sized CAVE immersive VR environment. VIRTUAL DIRECTOR is a voice and wand gesture
controlled application for navigation and camera choreography, compare Figure 5.29.

For smooth VR interaction, we needed a lightweight subset of the AMR data. Since the AMR computation naturally creates refined grids in dense regions, a time-dependent animation of the bounding boxes of the grid domains provided a very good guide to visually interesting areas for the star-formation scenes, where opacity was derived from gas density, as shown in Figure 5.27. Display toggles for each level of grids helped in reducing visual clutter. In the supernova scene, compare Figure 5.30, the temperature field was the visually appropriate one; but it was poorly correlated with density, and AMR grid placement was less helpful. Temperature-colored particles from a subset of grid cells served as an interactive guide.

Due to the dramatic range of scale inherent in the data, controlling the user’s scale is crucial. Virtual Director allows the user to shrink down to a very small virtual size, enabling controlled flight around the tiny newly born protostar, or to become large enough to reach across the 18,000 light year dataset. The proper scale enables the user to make sense of the data when immersed within the nested domains.

A background provides motion cues and enhances the perception of depth in visualization. It is particularly desirable when artistically crafting visualizations for large, general audiences. We chose a background to suit the first-star theme: a simulation of dark matter formation over a $100^3$ Mpc volume of the early universe, by Norman, Bryan, and O’Shea, also computed with the AMR code Enzo, but as a pure N-body gravitational calculation.
Figure 5.30: The images are renderings of the third dataset for three different time steps and camera positions. The simulation describes the supernova explosion explosion of the star, and consists of 760 time steps with between 6 and 10 levels of refinement and about 1,350 to 2,500 grids per time step. (dataset courtesy of T. Abel, Stanford University)

Its periodic boundary conditions allowed tiling many copies of the data to cover the sky smoothly. It is most clearly visible in the background of the right image in Figure 5.27.

The dark matter was given in particle form; images were created with the STAR RENDERER, a fast Gaussian-spot particle renderer from PIXAR ANIMATION STUDIOS. Particle velocity, a proxy for mass density, determined the choice of color from a palette of black-body radiation colors.

5.7.3 Results

The volume rendering was carried out on a SGI ONYX2 INFINITEREALITY2 with two RM7 raster managers with 64 MBytes of texture memory. Information about the separate datasets and their rendering times (for a viewport size of 864 × 486 pixels) is given in the captions of Figures 5.27 to 5.30.

We learned from this project that hardware supported, texture-based volume rendering can be used for broadcast quality rendering of even highly resolved AMR simulations. The VR environment proved to be extremely helpful for defining smooth camera paths deep inside the computational domains on very diverse spatial scales.
Chapter 6

Summary and Concluding Remarks

SAMR (Structured Adaptive Mesh Refinement), introduced in 1989 by Berger and Collela [8], is a special adaptive method for solving partial differential equations. In this approach the computational domain is initially covered by a set of coarse, structured subgrids. During the computation local error estimators are utilized to detect cells that require higher resolution. These cells are covered by a set of axis-aligned, structured subgrids, which overlay the refined regions of the coarse base grid.

In this thesis we developed direct and indirect volume visualization algorithms for scalar fields that are defined on SAMR grids. In particular we investigated the applicability of SAMR data structures for volume rendering of large datasets. Nowadays state-of-the-art volume rendering approaches leverage the powerful texture units of modern graphics hardware, which perform fast bi-, respectively trilinear interpolation of texture samples and thereby allow interactive frame rates even on standard consumer PC systems, compare Section 3.5.

But due to the enormous rate of increase in resolution of datasets, which are for example generated by 3D imaging devices or numerical simulations, performance is still an issue even when utilizing highly specialized graphics hardware. The performance limiting factors are the fill rate of the hardware, i.e. the number of pixels values that can be computed per time unit, the available amount of texture memory, and the I/O-bandwidth. In Section 4.4 of Chapter 4 we investigated approaches that accelerate texture-based volume rendering for the frequently occurring case of large, but sparse data, i.e. highly resolved data where only a small fraction of the voxels contains relevant information. The relevance criterion might for example be given by voxel transparency or by some cardinal function, e.g. choosing material subsets of segmented data.

Our strategy was to restrict most of the rendering work to the relevant parts of the volume. In order to benefit from the strength of texture hardware, the resulting coverage has to consist of few axis-aligned, non-overlapping regions, that can be processed separately in a view-consistent order for each viewpoint. We presented an approach, in which these regions are covered by subvolumes that are leaf nodes of a SAMR tree representation of the data volume. The tree was constructed by hierarchical clustering of cells that contain relevant data.
The AMR approach obtains a good balance between the number of boxes needed for tightly capturing the relevant voxels of the data volume and the number of covered non-relevant ones. The amount of created texture bricks is much smaller than the number of bricks created by an alternative octree approach, resulting in significant performance gains for the AMR approach, especially for the sparse datasets.

In Section 4.3 we further discussed a raycasting approach that is based on the utilization of an adaptive integration scheme. In particular for smooth datasets this results in a considerable reduction of interpolation operations and in an increase of the achieved rendering performance compared to the standard approach.

In Chapter 5 we described our work in the field of visualizing SAMR data. We discussed the topics of fast point location and spatial interpolation of discrete SAMR data in Sections 5.3 and 5.2. In Section 4.4 we presented indirect volume rendering methods for artifact-free visualization of carpet plots and planar slices as well as an algorithm for extracting $C^0$-continuous isosurfaces from data defined on unrestricted SAMR grids, i.e. grids that contain adjacent cells that differ by more than one level of refinement.

We further presented a hardware accelerated volume rendering approach for SAMR data that utilizes 3D textures in Section 5.5. Since current texture hardware requires axis-aligned texture bricks which contain cells on the same resolution level, some preprocessing was necessary. For this we proposed a decomposition of the data domain into disjoint axis-aligned areas of constant cell size that results in a small number of blocks. Since the partition maintains the multi-resolution representations of the grid function in the refined areas, the traversal of sub-branches of the tree can be pruned in the rendering phase, based on a view-dependent projection criterion.

Generating smooth animations of time-dependent data requires dense output. If the frequency of available time steps is too coarse, dense output is usually obtained by some sort of temporal interpolation between the given key-frames. Creation of dense output from a given set of time steps via interpolation is especially problematic for AMR data, due to potential changes of the underlying grid topology during the time evolution and the fact that usually not all generated subgrids are stored by the simulation. These problems are related to the fact that in AMR approaches the subgrid structure generally changes after each update, due to a regridding procedure that aims to place grids in regions where higher resolution is required. The time step size is usually halved between two consecutive levels of refinement, i.e. it increases exponentially, so storing all intermediate subgrids would result in huge amounts of data. In Section 5.6 we proposed an approach for temporal interpolation of AMR data that addresses these problems. In order to handle the problem of varying grid topology during time evolution, intermediate grid hierarchies are generated by merging the grids on the corresponding refinement levels. In a second step a nested grid structure is induced, employing a clustering algorithm. We investigated different interpolation schemes like linear, Hermite or flux-based interpolation, that allows a better approximation of the underlying grid function in the case that the data represents a conserved quantity. The overall algorithm is fast and allows an on-the-fly generation of interpolated frames.
In summary, we developed algorithms for

- accelerated software- and hardware-based volume rendering of 3D scalar data,
- planar slicing and the display of height fields for cell- and vertex-centered data on unrestricted SAMR grids,
- $C^0$-continuous isosurface extraction for cell- and vertex-centered data on unrestricted AMR grids,
- software- and hardware-based direct volume rendering for cell- and vertex-centered data on unrestricted SAMR grids and
- temporal interpolation of SAMR data with varying grid topology.

All algorithms have been implemented within the visualization framework AMIRA. They have been employed in several visualization projects and are now used for data analysis by several international research groups in the fields of cosmology and numerical relativity.
Bibliography


[39] J. Kniss, G. Kindlmann, and C. Hansen. Interactive volume rendering using multi-
dimensional transfer functions and direct manipulation widgets. In Proceedings of

[40] J. Kniss, S. Premoze, C. Hansen, and D. Ebert. Interactive translucent volume ren-

extraction from volume data. In E. Fiume, editor, SIGGRAPH 2001, Computer


texture-based volume visualization. In D. Ebert, M. Gross, and B. Hamann, editors,

[44] D. Laur and P. Hanrahan. Hierarchical splatting: A progressive refinement algo-
rithm for volume rendering. Computer Graphics (Proceedings of SIGGRAPH 91),


[48] S. L. Liebling. The singularity threshold of the nonlinear sigma model using 3D

[49] Y. Livnat and C. Hansen. View dependent isosurface extraction. In D. Ebert, H. Hagen,

algorithm using the span space. IEEE Transactions on Visualization and Computer

[51] Y. Livnat and X. Tricoche. Interactive point-based isosurface extraction. In H. Rush-


Curriculum Vitae

Personal Information:
   Name: Ralf Kähler
   Date of birth: 05/16/1972
   Place of birth: Tönning
   Nationality: German

Education:
   08/1978 - 06/1982 Primary School (Grund- und Hauptschule Tönning)
   10/1992 - 04/1999 Studies of Physics (FU Berlin)
   03/1999 Diploma Degree in Theoretical Physics

Work Experience:
   08/1991 - 09/1992 Civilian Service (Rettungsdienst Nordfriesland)
   10/1997 - 04/1999 Tutor (FU Berlin)
   since 10/1999 Scientific Assistant at the Zuse-Institute Berlin in a Cooperation with the MPI for Gravitational Physics (Albert-Einstein-Institute)