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. [47] presented an algorithm that skips empty space utilizing an octree data structure to encode the presence of non-transparent material. Danskin [24] extended this approach by exploiting data homogeneity and accumulated opacity as well.

Subramanian et al. [84] 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. [45] 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. [99] has been extended by Cohen et al. [23], 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. [62, 61] 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. [46] 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. [44] 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. [92] 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. [15] presented an error and importance driven strategy for selecting a set of octree nodes from the full pyramidal structure. Fang et al. [29] 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. [79]. 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 [80].

Another approach for accelerating texture-based volume rendering by leaping of empty regions has been proposed by Tong et al. [86]. 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 [84] 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 [99, 23] is disadvantageous for 3D texture-based volume rendering, since the volume is not subdivided into axis-aligned regions.

The approach in [46] 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 multigrid quadrature algorithm [25]. 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 Simpsons 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 [25].

4.3.1 Numerical Background

Let $I(g) := \int_a^b g(s)\, ds$ denote the integral of the scalar function $g : [a,b] \mapsto \mathbf{R}$, that is to be approximated numerically over the interval $\Omega := [a,b]$. Consider an initial, coarse discretization $\mathcal{G}$ of Ω, that is given by a set of intervals

$$\mathcal{I}_i := [s_i, s_{i+1}], \text{ for } i = 0, ..., n, \text{with } s_0 := a,\ s_n = b$$

as indicated in Figure 4.1.

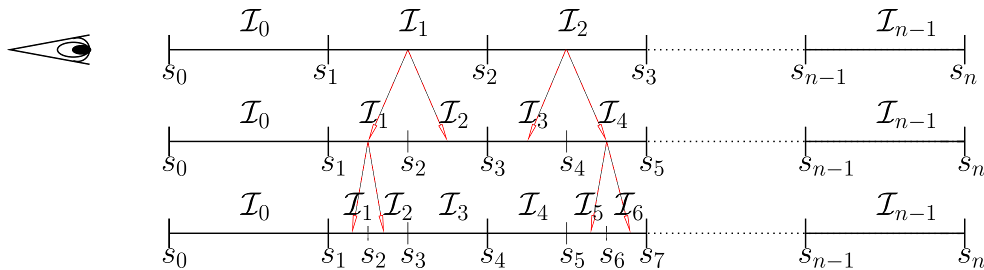


Figure 4.1: Refinement of ray-intervals by bisection.

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

$$\mathcal{T}(\mathcal{I}) = \frac{h}{4}(f(s_l) + 2f(s_m) + f(s_r)), \text{ and} \tag{4.1}$$

$$\mathcal{S}(\mathcal{I}) = \frac{h}{6}(f(s_l) + 4f(s_m) + f(s_r)). \tag{4.2}$$

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

Analogously

$$\mathcal{T}(\mathcal{G}) := \sum_{i=0}^{n} \mathcal{T}(\mathcal{I}_i) \text{ and} \tag{4.3}$$

$$\mathcal{S}(\mathcal{G}) := \sum_{i=0}^{n} \mathcal{S}(\mathcal{I}_i) \tag{4.4}$$

denote the approximations of $I(g)$ over the whole domain Ω. The local approximation error on $\mathcal{I}$ can be estimated by

$$e_{loc}(\mathcal{I}) := \mid \mathcal{T}(\mathcal{I}) - \mathcal{S}(\mathcal{I}) \mid. \tag{4.5}$$

Intervals $\mathcal{I}_n$ that are tagged for refinement, as discussed below, are split up into the two subintervals by simple bisection

$$\mathcal{I} = [s_l, s_r] \quad \longrightarrow \quad [s_l, \frac{s_l + s_r}{2}], [\frac{s_l + s_r}{2}, s_r] =: (\mathcal{I}_l^{ref}, \mathcal{I}_r^{ref}),$$

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

$$\mathcal{P}(\mathcal{I}_r^{ref}) = \mathcal{P}(\mathcal{I}_l^{ref}) = \mathcal{I}.$$

The refinement strategy of the integration approach is to obtain an equidistribution of the local approximation errors $e_{loc}(\mathcal{I}_i)$ for all intervals $\mathcal{I}_i$. Following the abstract suggestion in [6] the decision which intervals should be refined in order to achieve this is based on a second error estimator $e_{extr}(\mathcal{I})$, that yields information about the local approximation error of the subintervals $\mathcal{I}_l^{ref}, \mathcal{I}_r^{ref}$ in case $\mathcal{I}$ would be refined. This information is obtained by extrapolation of the local approximation errors on $\mathcal{I}$ and $\mathcal{P}(\mathcal{I})$. The estimated error for the trapezoidal rule is of the form

$$e_{loc}(\mathcal{I}) \doteq Ch(\mathcal{I})^{\gamma}. \tag{4.6}$$

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

$$e_{loc}(\mathcal{I}^{ref}) \doteq \big(Ch(\mathcal{I})^{\gamma}\big)2^{-\gamma} \doteq e_{loc}(\mathcal{I})\big(\frac{e_{loc}(\mathcal{I})}{e_{loc}(\mathcal{P}(\mathcal{I}))}\big).$$

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

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

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

$$m_{extr}(\mathcal{G}) := \max_{\mathcal{I} \in \mathcal{G}} e_{extr}(\mathcal{I}). \tag{4.8}$$

Since the goal of the refinement process is to obtain an equidistribution of the local approximation errors, it is reasonable to restrict the refinement at each integration step to those intervals for which $e_{loc}(\mathcal{I}) \geq m_{extr}$ holds. A modification of this criterion is discussed in [63].

A suitable measure for the unknown global approximation error

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

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

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

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

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

where c_{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

$$\mathcal{T}(\mathcal{I}) = \frac{h(\mathcal{I})}{4}(q_l + 2\,q_m e^{-\frac{h(\mathcal{I})}{4}(\kappa_l+\kappa_m)} + q_r e^{-\frac{h(\mathcal{I})}{4}(\kappa_l+2\kappa_m+\kappa_r)}),\ \text{and} \tag{4.11}$$

$$\mathcal{S}(\mathcal{I}) = \frac{h(\mathcal{I})}{6}(q_l + 4\,q_m e^{-\frac{h(\mathcal{I})}{4}(\kappa_l+\kappa_m)} + q_r e^{-\frac{h(\mathcal{I})}{6}(\kappa_l+4\kappa_m+\kappa_r)}). \tag{4.12}$$

Here the segments are oriented such, that $\mathcal{I}_0$ denotes the segment that is closest to the viewpoint, see Figure 4.1.

Instead of defining the local approximation error for the interval $\mathcal{I}_i$ 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

$$\begin{aligned} e_{loc}(\mathcal{I}_i) &:= e^{-\tau(s_0,s_i)}|\mathcal{T}(\mathcal{I}_i) - \mathcal{S}(\mathcal{I}_i)| \\ &= (\prod_{k=0}^{i-1} T_k)|\mathcal{T}(\mathcal{I}_i) - \mathcal{S}(\mathcal{I}_i)|, \end{aligned} \tag{4.13}$$

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 (𝓘_i; i = 0,...,n) {
        if (𝓘_i is newly refined) {
          compute 𝓣(𝓘_i), 𝓢(𝓘_i);
          compute e_loc(𝓘_i), e_extr(𝓘_i);
        }
        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

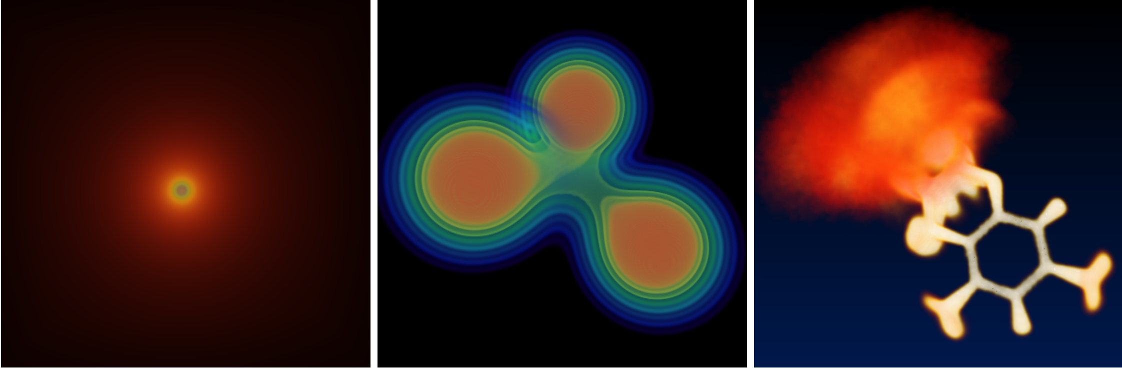


Figure 4.2: Renderings of the three tested datasets. The first one was generated by sampling the function $f(\boldsymbol{x}) = (\boldsymbol{x}^2 + 10^{-3})^{-1}$ on a uniform grid a resolution of 256^3 voxels. The second dataset, the result of an astrophysical simulation describing the collision of three 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.

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.

	fps(standard)	**fps(adaptive)**	**reduction of evaluations**
Dataset I	0.118 fps	0.335 sec	76.1%
Dataset II	0.028 fps	0.041 fps	54.3%
Dataset III	0.109 fps	0.097 fps	53.1%

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.

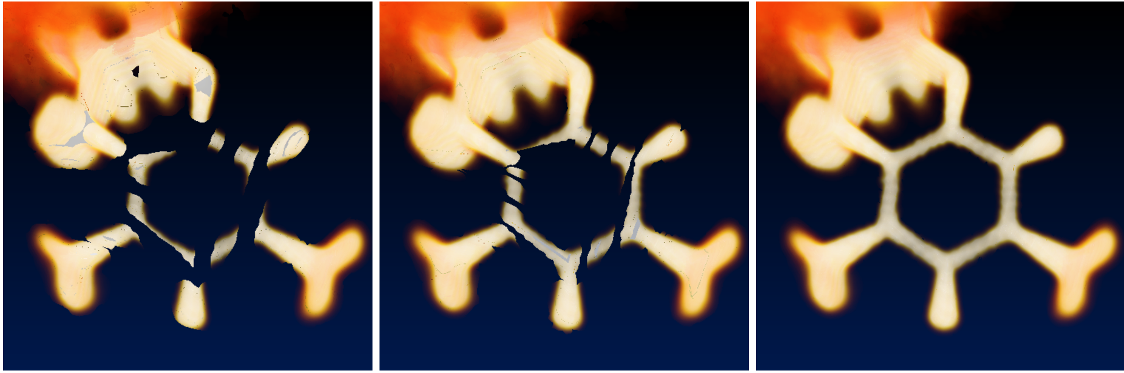


Figure 4.3: From left to right the initial refinement of the rays is increased.

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. Is 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 preceeding 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

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

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.

  $$(\mathcal{B}^i \cap \mathcal{B}^j) = \emptyset \vee (\mathcal{B}^i \cap \mathcal{B}^j) \subset (\partial\mathcal{B}^i \cup \partial\mathcal{B}^i)$$

  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 $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ and let the relation $(\mathcal{A}, \mathcal{B})$ denote that $\mathcal{A}$ (partially) occludes $\mathcal{B}$ for the given viewpoint. The polyhedra build a *visibility cycle* if the relations $(\mathcal{A}, \mathcal{B})$, $(\mathcal{B}, \mathcal{C})$ and $(\mathcal{C}, \mathcal{A})$ 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.

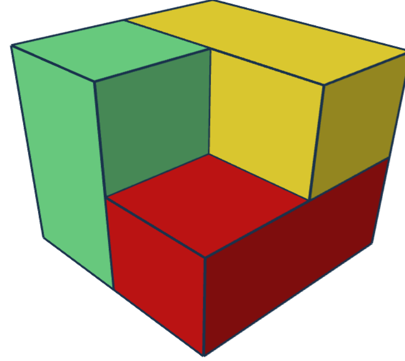


Figure 4.4: Example of three axis-aligned blocks that form a visibility cycle for the considered viewpoint.

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

$$V(\mathcal{B}_m) := \frac{|\{\Omega_{ijk} \in \mathcal{B}_m \mid R(i,j,k) = 0\}|}{|\mathcal{B}_m|},$$

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 $\mathcal{B}_m$. 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

$$r(\mathcal{B}_m) := \frac{|\{\mathcal{C}(\mathcal{B}_m) \mid \mathcal{C}(\mathcal{B}_m) \text{ is leaf node}\}|}{|\mathcal{B}_m|},$$

[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 $\mathcal{B}_m$ and the node's volume itself. If $r(\mathcal{B}_m)$ 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 $R^l(.)$ for the pyramid level Λ^l is related to the one of the next higher resolved level, $R^{l+1}(.)$, via

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

Here we adopted the notation introduced in Subsection 2.2.1. In particular Λ^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

$$S_{yz}(i)^l = \sum_{j=p_1^m}^{(p_1^m+n_1^m)} \sum_{k=p_2^m}^{(p_2^m+n_2^m)} R(i,j,k)^l. \tag{4.16}$$

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 [11] 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:

```
// 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)
{
    Γ = new subgrid(bounding box, level);
    if (level ≠ maxlevel)
    {
        markedCells = inspect_cells(Γ, importance_criterion);
        newBoxList  = cluster(markedCells);
        for all (boxes[i] ∈ newBoxList)
        {
            Γ_sub = create_subgrid(boxes[i], level+1);
            set Γ_sub as C(Γ);
        }
     }
     return Γ;
}
```

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 Ω^l_{ijk} in the pyramid we rather store the minimal and maximal data value

$$(f^{min}_{ijk} := \min_{\mathbf{x}\in\Omega^l_{ijk}} f(\mathbf{x}),\ f^{max}_{ijk} := \max_{\mathbf{x}\in\Omega^l_{ijk}} f(\mathbf{x})).$$

Determining the min-max intervals is inexpensive for trilinear interpolation, since it equals the minimal and maximal data samples of the vertices covered by Ω^l_{ijk}. During the clustering procedure each of the intervals $\mathbf{C}_\alpha\big([f^{min}_{ijk}, f^{max}_{ijk}]\big)$ in the colormap's alpha channel is inspected, and the cell is tagged, if its maximal opacity $\mathbf{C}_\alpha = 1 - T$ 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 κ via $\mathbf{C}_\alpha(f(\boldsymbol{x})) = 1 - exp(-\kappa(\boldsymbol{x})d_0)$, assuming that κ 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 Ω_{ijk} by

$$\begin{aligned} A(\Omega_{ijk}) &:= 1 - exp(-\kappa_{max}(\Omega_{ijk})\ d) \\ &= 1 - (1 - \tilde{A}_{max})^{\frac{d}{d_0}}. \end{aligned}$$

Here d is the cells diagonal and

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

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 [18, 97]. 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

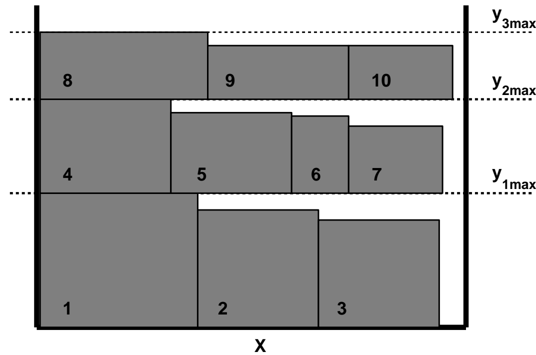


Figure 4.5: 2D example of the "next-fit-decreasing-heigh" packing algorithm: subvolumes are inserted from left to right, starting at the lower left corner.

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 [48]. We adopt a three-dimensional version of the "next-fit-decreasing-height" (NFDH) algorithm [36].

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 $\mathcal{H}_1, \mathcal{H}_2$. 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:

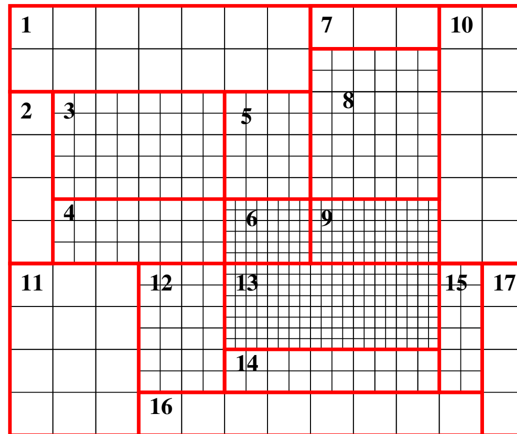
```
traverse_subtree(subnode)
{
  if (subnode is a leaf node) {
    render (subnode);
  }
  else {
    H1 := 'halfspace of subnode.childA';
    H2 := 'halfspace of subnode.childB';
    if (viewpoint ⊂ H1 )
      traverse_subtree(subnode.childB);
    else
      traverse_subtree(subnode.childA);
  }
}
```

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 $\boldsymbol{x}_k$ of the bounding boxes onto the planes normal direction $\boldsymbol{n}$. Indexing the slice through location $\boldsymbol{a}$ by 0 and slices in normal direction by positive indices, the slice index the corresponds to the vertex $\boldsymbol{x}_k$ is given by $i = [((\boldsymbol{x}_k - \boldsymbol{a}) \cdot \boldsymbol{n}) / d_0]$. Thus the intersection computations can be restricted to the slices with indices within $[i_{min}, i_{max}]$.

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



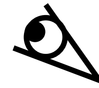


Figure 4.6: The order of a view-consistent node traversal for the decomposition of the AMR grid example from Figure 2.6 for a viewpoint in the lower right.

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 764×793 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 $\alpha_{thres} = 0.03$ 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 $\alpha_{thres} = 0.1$ 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 4.2 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 INFINITEREALITY2 the texelsize 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 4.2 to 4.6. Also less cells that are not tagged 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.

	levels	bricks	volume	texsize	preproc.	fps
standard	1	7	103.0%	208.0 MB	8.1s	**0.1**
Octree I	8	1503	2.9%	3.1 MB	6.5s	**10.2**
Octree II	8	615	3.6%	5.1 MB	17.3s	**13.1**
AMR I	4	333	0.8%	2.0 MB	21.3s+**2.7s**	**28.6**
AMR II	4	333	0.8%	2.0 MB	21.3s+**2.7s**	**28.6**

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.

	levels	bricks	volume	texsize	preproc.	fps
standard	1	7	103.0%	208.0 MB	8.0s	**0.1**
Octree I	8	1786	3.4%	3.7 MB	17.2s	**8.2**
Octree II	8	857	5.3%	6.4 MB	15.1s	**11.8**
AMR I	4	417	1.4%	4.0 MB	19.5s+**4.6s**	**24.6**
AMR II	4	417	1.4%	4.0 MB	19.5s+**4.6s**	**24.6**

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

	levels	bricks	volume	texsize	preproc.	fps
standard	1	23	103.1%	736.0 MB	13.3s	**0.1**
Octree I	7	3277	7.7%	15.9 MB	34.6s	**3.2**
Octree II	7	1907	8.7%	19.0 MB	11.7s	**3.9**
AMR I	5	1388	4.3%	16.0 MB	34.4s+**13.9s**	**6.3**
AMR II	5	1525	3.9%	16.0 MB	34.4s+**13.5s**	**6.5**

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

	levels	bricks	volume	texsize	preproc.	fps
standard	1	1	100.0%	16.0 MB	0.5s	**3.5**
Octree I	8	652	8.9%	0.6 MB	4.1s	**13.7**
Octree II	8	587	7.9%	0.9 MB	6.1s	**15.6**
AMR I	4	61	8.8%	1.0 MB	0.9s + **0.7s**	**14.3**
AMR II	4	333	6.9%	1.0 MB	0.9s + **0.8s**	**16.7**

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

	levels	bricks	volume	texsize	preproc.	fps
standard	1	2	101.0%	64.0 MB	2.0s	**0.4**
Octree I	7	2947	37.6%	14.5 MB	11.0s	**3.7**
Octree II	7	1814	37.0%	15.0 MB	7.2s	**4.5**
AMR I	3	619	32.1%	32.0 MB	6.1s + **11.4s**	**4.5**
AMR II	3	501	35.0%	32.0 MB	6.1s + **13.2s**	**4.7**

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

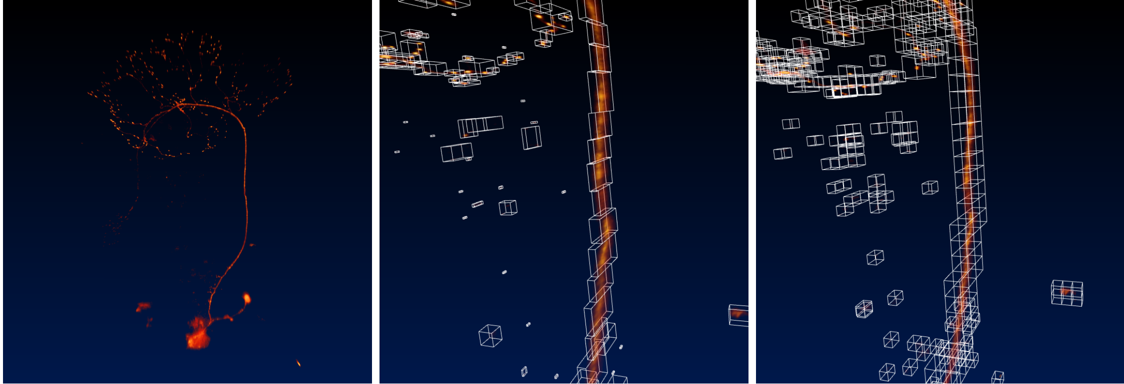


Figure 4.7: Dataset I: The images show volume renderings for the three different approaches. left: standard, middle: AMR, right: octree. *(dataset courtesy of R. Menzel, Freie Universität Berlin)*

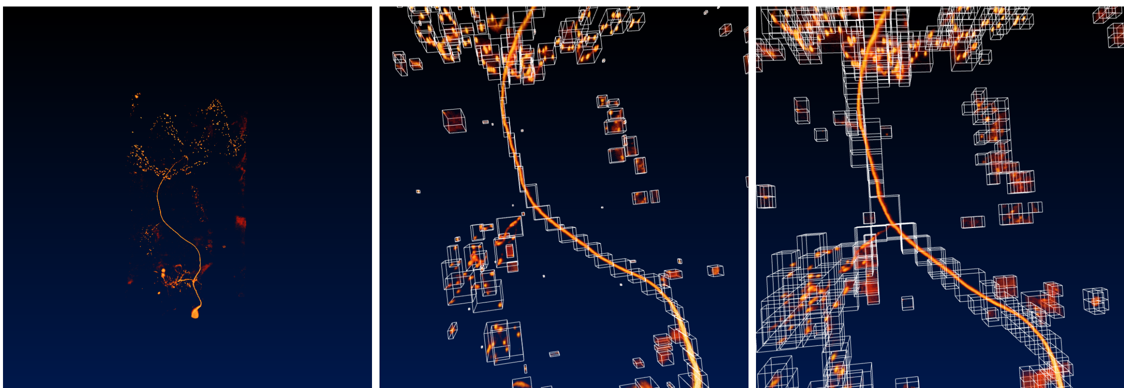


Figure 4.8: Dataset II: bee brain neuron, left: standard, middle: AMR, right: octree. *(dataset courtesy of R. Menzel, Freie Universität Berlin)*

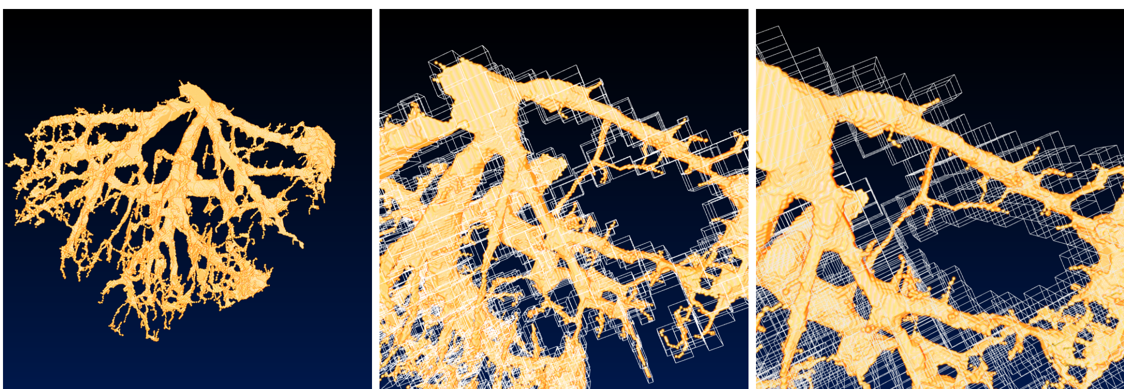


Figure 4.9: Dataset III: vascular tree, left: standard, middle: AMR, right: octree. *(dataset courtesy of P. Schlag, Robert-Rössle-Klinik and Unversitätsklinikum Charité)*

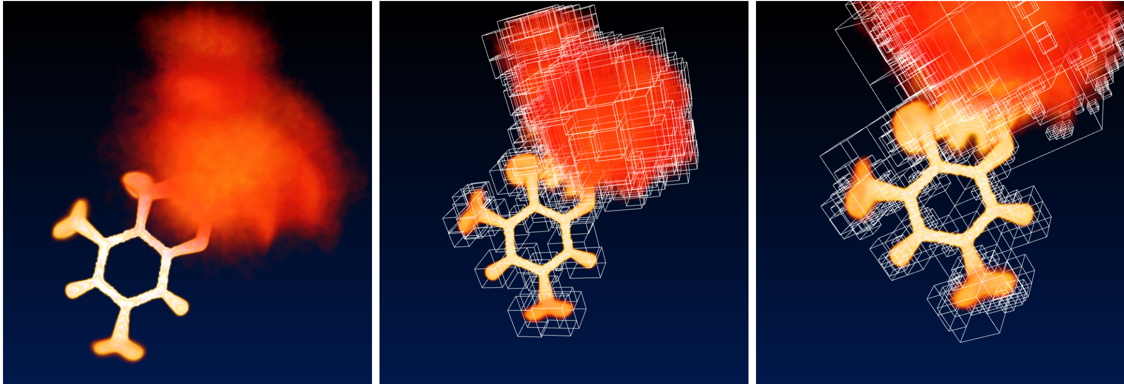


Figure 4.10: Dataset IV: molecule conformation, left: standard, middle: AMR, right: octree. *(dataset courtesy of D. Baum, Zuse Institute Berlin)*

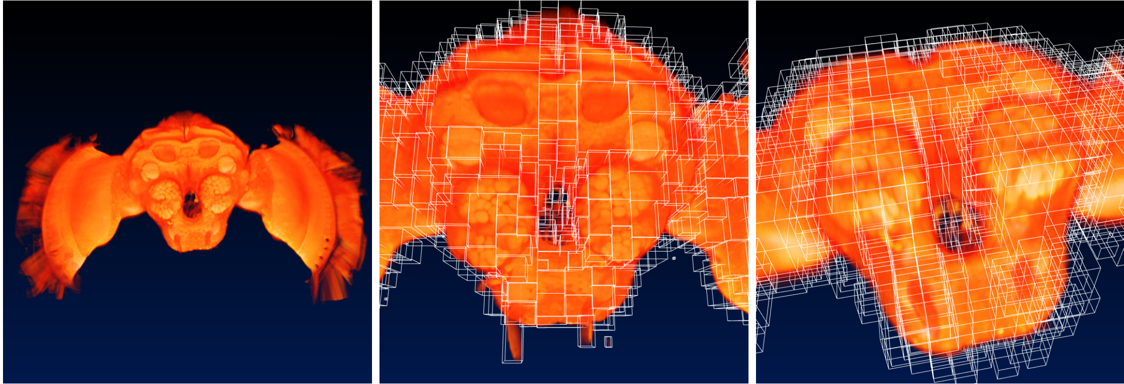


Figure 4.11: Dataset V: bee brain, left: standard, middle: AMR, right: octree. *dataset courtesy of R. Brandt and R. Menzel, Freie Universität Berlin)*