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 [83, 88]. 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 [20, 30]. One alternative approach is to base the classification on the employed visualization algorithms, rather than on the data itself, compare [87].

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.

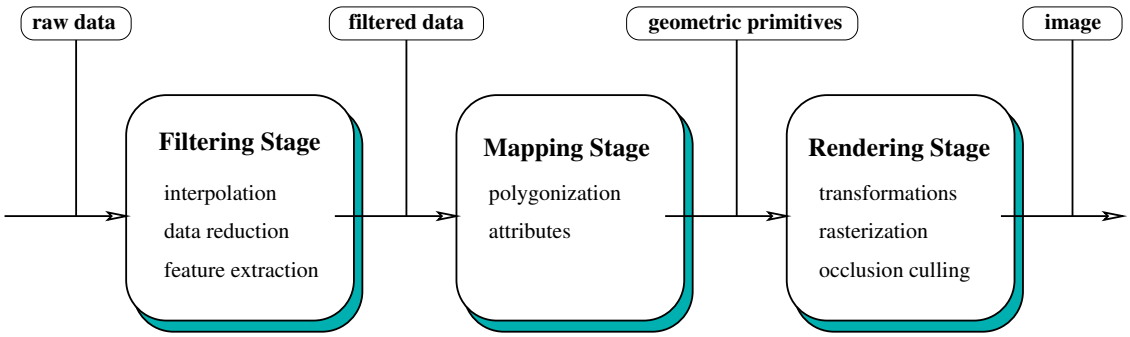


Figure 3.1: The main stages of the visualization pipeline.

- 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.

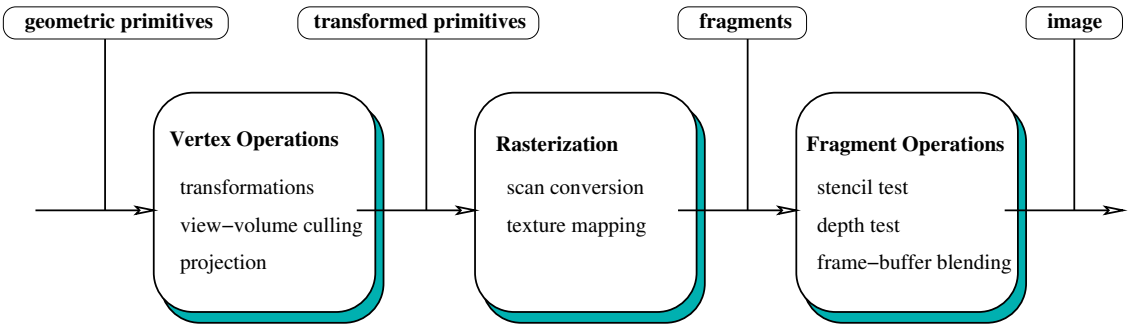


Figure 3.2: The main stages of the rendering pipeline.

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 [76], introduced in 1992 by SILICON GRAPHICS, which is available on almost every platform and supported by many programming languages and DIRECTX [8], 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 [82, 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 [53] 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 $f(\mathbf{x}(\lambda, \mu))$ within the intersection of the data volume Ω and an arbitrarily oriented plane

$$\mathcal{P} = \{\ \mathbf{x}(\lambda, \mu) \mid \mathbf{x}(\lambda, \mu) = \mathbf{a} + \lambda \mathbf{v}_1 + \mu \mathbf{v}_2\ \}.$$

Often a colormap $\boldsymbol{C}$ 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 $\boldsymbol{C}_i \in [0, 1]$, $i = r, g, b$ for the red, green and blue color components, as well as opacity value $\boldsymbol{C}_\alpha \in [0, 1]$

$$\boldsymbol{C} : Im(f) \mapsto \mathbb{R}^4, \quad \text{with} \quad f(\mathbf{x}) \mapsto (\boldsymbol{C}_r, \boldsymbol{C}_g, \boldsymbol{C}_b, \boldsymbol{C}_\alpha).$$

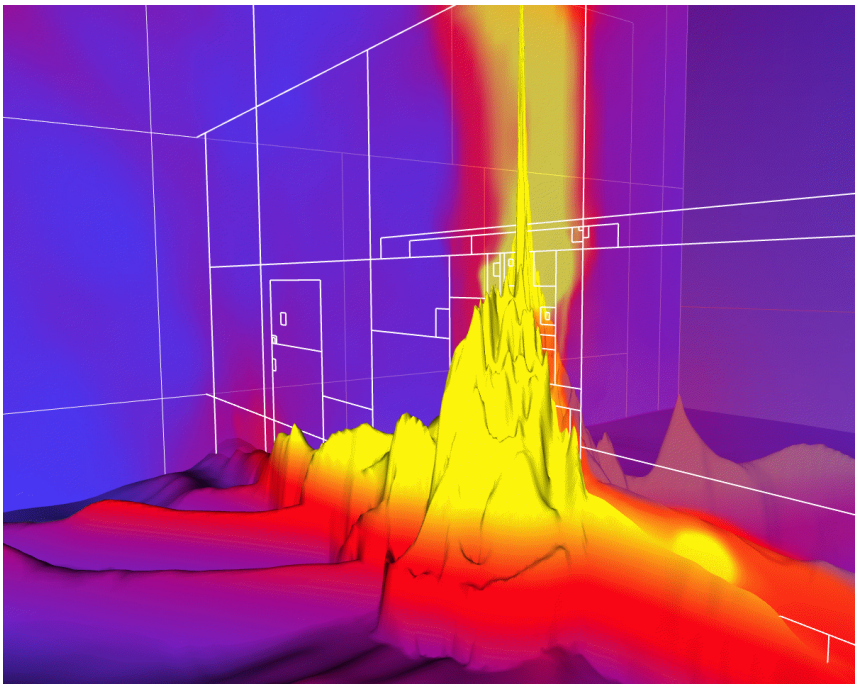


Figure 3.3: Scalar field visualization using combined height field and slicing techniques. *(dataset courtesy of M. Norman, National University of California)*

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

$$\mathbf{s}(\lambda, \mu) = \mathbf{x}(\lambda, \mu) + k\ f(\mathbf{x}\ (\lambda, \mu)) \frac{\mathbf{v}_1 \times \mathbf{v}_2}{|\mathbf{v}_1||\mathbf{v}_2|},$$

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} [1]

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

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

There exist a number of natural requirements for this approximation $\mathcal{S}_{app}$. It should at least be C^0-continuous for continuous interpolants and topologically consistent with $\mathcal{S}_{int}$, 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 $\mathcal{S}_{int}$ and $\mathcal{S}_{app}$ 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. [53]. 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 $\mathcal{S}_{int}$ 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.

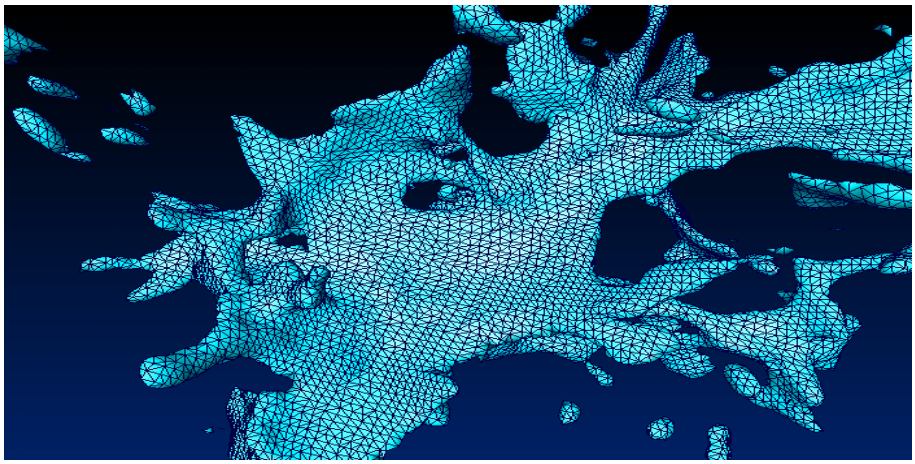


Figure 3.4: Example of an isosurface approximation by a triangular mesh that was generated by the *Marching Cubes* algorithm. *(dataset courtesy of T. Abel, Stanford University)*

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 $\mathcal{S}_{int}$ 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 $\mathcal{S}_{int}$ and that it is not invariant under the sign inversion operation.

Nielson et al. [59] 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 $\mathcal{S}_{int}$ 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

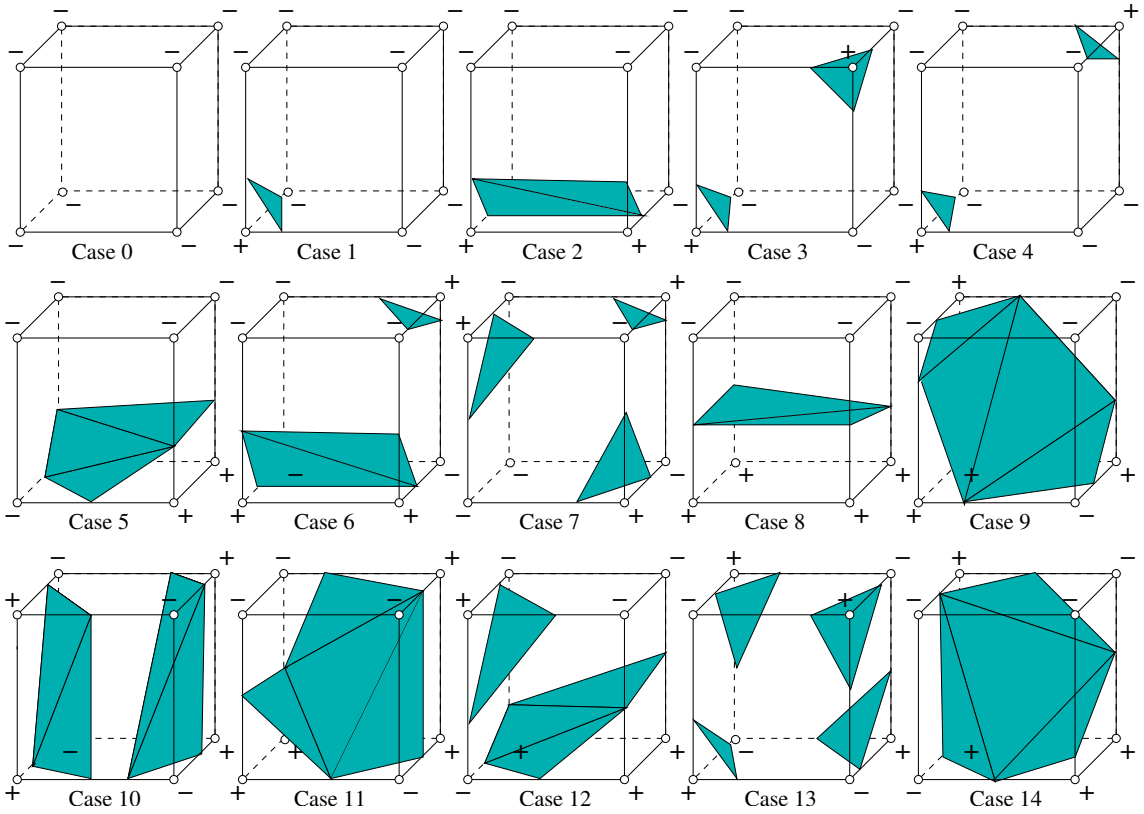


Figure 3.5: Using symmetry operations the 256 triangle patch configurations for hexahedral cells can be reduced to 15 topologically different ones.

cubes algorithm for this cell type, which is also called *marching tetrahedra* [14], 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 [33].

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 [51]. 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 [50], where only the visible portion of the isosurface is extracted, accelerating both, the extraction as well as the rendering phase. Chiang et al. [21, 22] 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

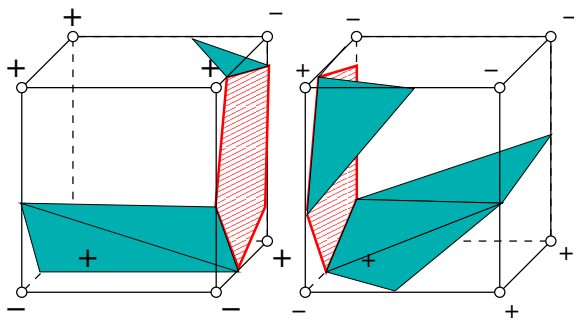


Figure 3.6: Example of two adjacent cells of class 6 and 12 with inconsistent local triangulations that lead to artifacts in the resulting triangular mesh. These become visible as cracks (area with red pattern) at the common interface.

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 [35]. Livnat et al. [52] present a hybrid approach, in which sub-pixel triangles are replaced by point primitives.

Kobbelt et al. [42] 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. [93] 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 $\boldsymbol{x} \in \Omega$ 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 [19]. In this section we will closely follow the discussion presented in [32]. Another good presentation of the underlying physical models can be found in [56].

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 dependence of the radiation field at each point, such that

$$dE = I(\mathbf{x}, \mathbf{n}, \nu)\, cos\vartheta\, da\, d\Omega\, d\nu\, dt$$

gives the amount of radiant energy per time unit dt and frequency interval $d\nu$ that emerges at the location $\mathbf{x}$ and is radiated into the solid angle $d\Omega$ in the direction $\mathbf{n}$ through the surface element da. Here ϑ defines the angle between $\mathbf{n}$ and da, compare Figure 3.7.

Another important quantity which describes the decrease of radiation traveling through material is the *absorption* or *extinction* coefficient χ. It is defined such, that

$$dE^{(ab)} = \chi(\mathbf{x}, \mathbf{n}, \nu) I(\mathbf{x}, \mathbf{n}, \nu)\; ds\; da\; d\Omega\; d\nu\; dt$$

yields the amount of energy removed from a beam with radiance $I(\mathbf{x}, \mathbf{n}, \nu)$ passing through a cylindric volume element of length ds with a cross section da. Is is convenient to split the absorption coefficient into the so-called *true absorption* coefficient κ and the *scattering* coefficient σ

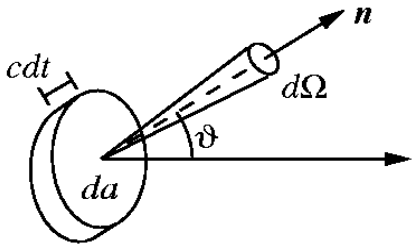


Figure 3.7: Illustration adopted from [32].

$$\chi(\mathbf{x}, \mathbf{n}, \nu) = \kappa(\mathbf{x}, \mathbf{n}, \nu) + \sigma(\mathbf{x}, \mathbf{n}, \nu). \tag{3.1}$$

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 η is defined such, that

$$dE^{(em)} = \eta(\mathbf{x}, \mathbf{n}, \nu)\; ds\; da\; d\Omega\; d\nu\; dt$$

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 $\boldsymbol{x}$ 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

$$\eta(\mathbf{x}, \mathbf{n}, \nu) = q(\mathbf{x}, \mathbf{n}, \nu) + j(\mathbf{x}, \mathbf{n}, \nu). \tag{3.2}$$

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

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

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

$$j(\mathbf{x}, \mathbf{n}', \nu') = \int\int p(\mathbf{x}, \mathbf{n}, \mathbf{n}', \nu, \nu')\, \sigma(\mathbf{x}, \mathbf{n}, \nu)\, I(\mathbf{x}, \mathbf{n}, \nu)\, d\nu\, d\Omega. \tag{3.3}$$

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 the amount of emitted energy at that location, reduced by the amount of absorbed and scattered energy:

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

With $\mathbf{x}(s) := \mathbf{p} + s\mathbf{n}$, where $\mathbf{p}$ 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 $ds \to 0$

$$\frac{\partial}{\partial s} I(\mathbf{x}, \mathbf{n}, \nu) = -\chi(\mathbf{x}, \mathbf{n}, \nu) I(\mathbf{x}, \mathbf{n}, \nu) + \eta(\mathbf{x}, \mathbf{n}, \nu). \tag{3.4}$$

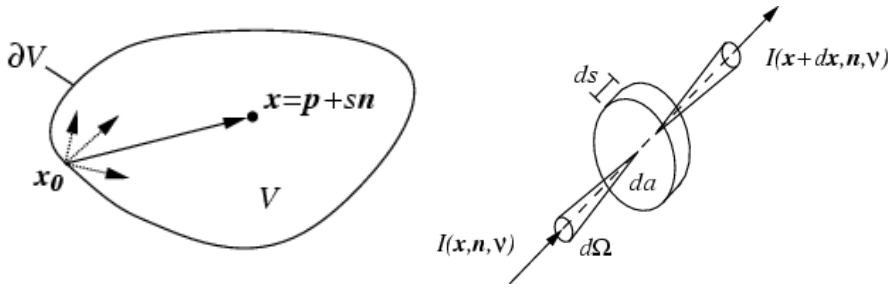


Figure 3.8: adopted from [32].

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

$$I(\boldsymbol{x}(s), \mathbf{n}, \nu) = I(\boldsymbol{x}(s_0), \mathbf{n}, \nu) e^{-\tau_\nu(\boldsymbol{x}(s_0), \boldsymbol{x}(s))} + \int_{s_0}^{s} \eta(\boldsymbol{x}(s'), \mathbf{n}, \nu) e^{-\tau_\nu(s', s)} ds', \tag{3.5}$$

where the *optical depth* $\tau_\nu(s_1, s_2)$, between the two points $\mathbf{x}(s_1) = \mathbf{p} + s_1\mathbf{n}$ and $\mathbf{x}(s_2) = \mathbf{p} + s_2\mathbf{n}$ is defined as

$$\tau_\nu(s_1, s_2) = \int_{s_1}^{s_2} \chi\big(\mathbf{x}(s')\big)\, ds'. \tag{3.6}$$

The formulation (3.5) allows the following intuitive interpretation: The total specific intensity at the location $\mathbf{x}$ emitted into the direction $\mathbf{n}$ consists of two parts: the sum of all specific intensity emitted along the ray segment $\mathbf{p} + s\frac{||\mathbf{x}-\mathbf{p}||}{||n||}\,\mathbf{n}$ 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 $\mathbf{x}_0$ into direction $\mathbf{n}$.

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 $\chi(\boldsymbol{x})$[2], respectively $\eta(\boldsymbol{x})$ 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 $\mathcal{T} : Im(f) \rightarrow \mathbb{R}^4$ with

$$f \mapsto \mathcal{T}(f) := (\eta_r(f), \eta_g(f), \eta_b(f), \chi(f)).$$

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_{color})

$$\mathcal{C}_{pre}(\mathbf{x}) = I_{color}(\mathcal{T} \circ f \mid \boldsymbol{p}_0, ...\boldsymbol{p}_n)(\mathbf{x}).$$

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

$$\mathcal{C}_{post}(\mathbf{x}) = \mathcal{T} \circ I_{data}(f \mid \boldsymbol{p}_0, ...\boldsymbol{p}_n)(\mathbf{x}).$$

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

[2] In order to ease the discussion we will neglect the direction and frequency dependency of the coefficients in this section.

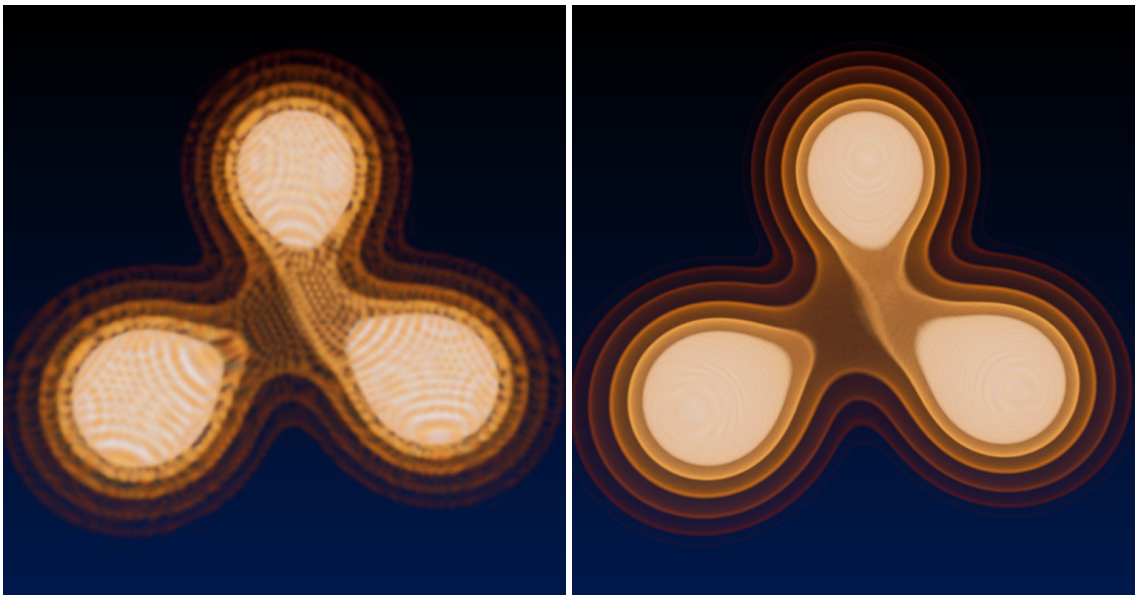


Figure 3.9: Comparison of pre-, and post-classification for volume rendering using a transfer function that contains high frequencies. The image on the left, rendered using pre-classification, shows severe aliasing artifacts, whereas on the right side post-classification was employed. *(data courtesy of P. Diener, Louisiana State University)*

Let us assume that the original signal $f(\boldsymbol{x})$ was sampled with a frequency $\nu > \nu_{max}(f)$. 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 $\mathcal{T} \circ f$, since its Fourier spectrum may contain frequency components larger than ν_{max}, which prohibits a faithful reconstruction of $\mathcal{T} \circ f$.

In contrast to this, post-classification allows an ideal reconstruction. In this case the original signal $f(\boldsymbol{x})$ is reconstructed first and next the composed signal $\mathcal{T} \circ I(f)$ 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 $\eta(\mathbf{x}, \mathbf{n}, \nu)$, that at each location $\boldsymbol{x}$ 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 [72], 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 $\eta(\mathbf{x}, \mathbf{n}) = q(\mathbf{x})$, respectively $\chi(\mathbf{x}, \mathbf{n}) = \kappa(\mathbf{x})$. 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

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

According to $\tau(s_1, s_2) := \int_{s_1}^{s_2} \kappa(s)\, ds$, the optical depth between $\mathbf{x}(s_1)$ and $\mathbf{x}(s_2)$ 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* [95] 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* [96], 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* [43] 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 $[s_i, s_{i+1}], i = 0, ...n-1$. 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.

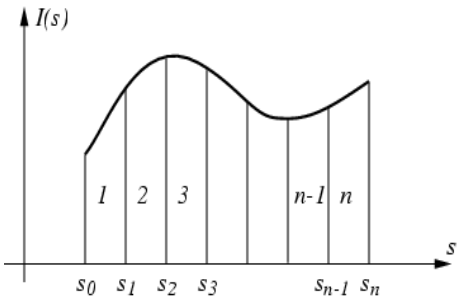


Figure 3.10: Figure adopted from [32].

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:

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

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

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

and its *emission* by

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

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

$$\begin{aligned} I(s_n) &= I(s_{n-1})T_n + b_n \qquad (3.11)\\ &= (I(s_{s-2})T_{n-1} + b_{n-1})\,T_n + b_n \\ &= \ldots \\ &= ((((...(b_0)T_1 + b_1)T_2...)T_{n-1} + b_{n-1})\,T_n + b_n \\ &= \sum_{k=0}^{n}(\; b_k \prod_{j=k+1}^{n} T_j\;) = \sum_{k=n}^{0}(\; b_k \prod_{j=n}^{k+1} T_j\;) \qquad (3.12) \end{aligned}$$

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:

$I \leftarrow b_0$;
for $(k = 1;\ k < n;\ k = k + 1)$ {
    $I \leftarrow T_k\, I + b_k$;
}

According to (3.12) the summation may alternatively be performed in the opposite *front-to-back* order

$I \leftarrow b_n$;
$T \leftarrow T_n$;
for $(k = n - 1;\ k > 0;\ k = k - 1)$ {
    $I \leftarrow I + b_k\, T$;
    $T \leftarrow T_k\, T$;
}
$I \leftarrow I + b_0\, T$;

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

$$\begin{aligned} b_k &= \int_{s_{k-1}}^{s_k} q_k\, e^{-\int_s^{s_k} \kappa(s)ds} \\ &= q_k \int_{s_{k-1}}^{s_k} e^{-\kappa_k (s_k - s)} ds \\ &= \frac{q_k}{\kappa_k}\,(1 - T_k). \end{aligned}$$

Combining this with Equation 3.11 yields

$$I(s_n) = I(s_{n-1})\, T_n + \frac{q_n}{\kappa_n}(1 - T_n). \tag{3.13}$$

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 integration.

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 employed 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

$$\boldsymbol{C}_{comp} = (1 - \alpha)\, \boldsymbol{C}_{old} + \alpha\, \boldsymbol{C}_{new}, \tag{3.14}$$

which is usually also supported by the graphics hardware. Here $\boldsymbol{C}_{old}$ is the color triple of the previous fragment stored in the frame buffer, and α and $\boldsymbol{C}_{new}$ are the opacity, respectively the colors of the incoming fragment.

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

$$\begin{aligned} C_{new} &= \frac{q_n}{\kappa_n}, && (3.15)\\ C_{old} &= I(s_{n-1}), && (3.16)\\ C_{comp} &= I_n, && (3.17)\\ \alpha &= 1 - T_n = (1 - e^{-\kappa_k \Delta s}). && (3.18) \end{aligned}$$

So storing $(\frac{q_{red}}{\kappa}, \frac{q_{green}}{\kappa}, \frac{q_{blue}}{\kappa}, (1 - e^{-\kappa_k \Delta s}))$ 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 Δs, 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 with the smallest angle between the actual viewing direction and the slice normals 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 [67], that provide direct hardware implementations for raycasting.

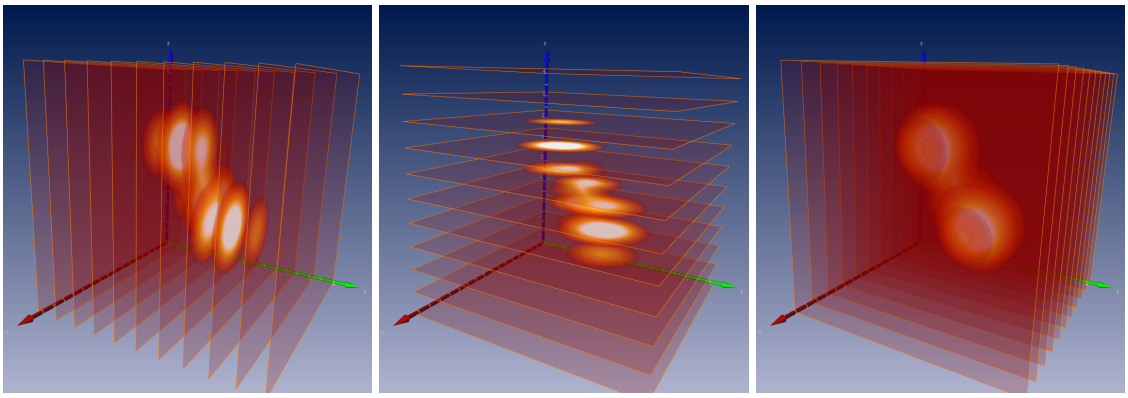


Figure 3.11: For volume rendering via 2D textures, three stacks of planar slices are employed. For each viewpoint, the set that is most perpendicular to the viewing direction is rendered.

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 [18, 97] 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

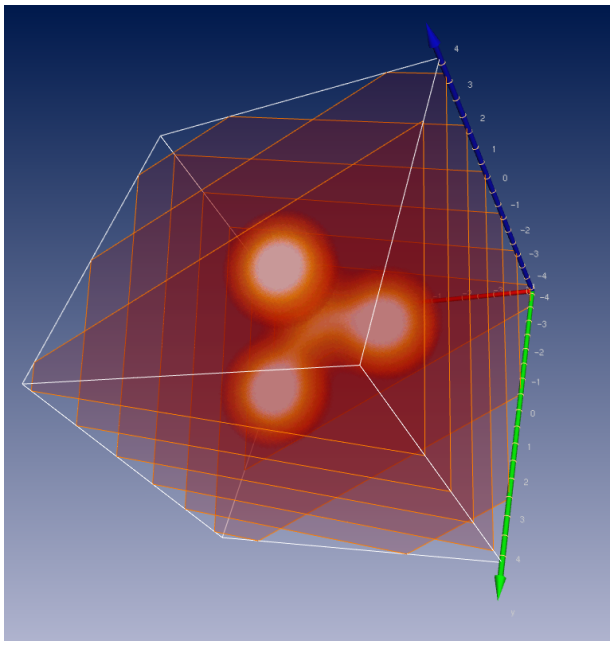


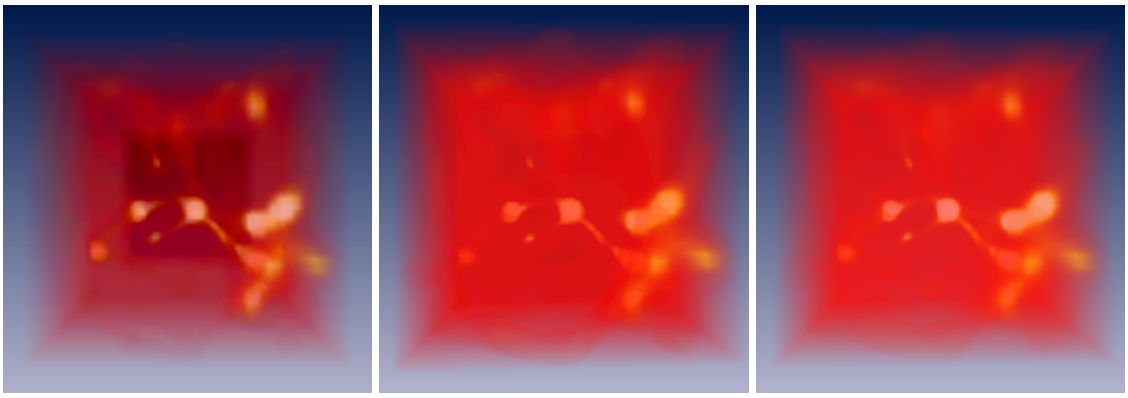
Figure 3.12: For volume rendering via 3D textures, a set of slices that are perpendicular to the actual viewing direction is extracted on-the-fly and blended in the frame buffer.

proxy geometries is described in [44]. 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-



(a) (b) (c)

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)*

ticular 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 [70].

Texture-based volume rendering has been greatly improved in the last years. Engel et al. [27] 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. [40] 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 [41]. 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.