Interactive Volume Rendering with Ray Tracing

Gerd Marmitt, Heiko Friedrich, and Philipp Slusallek

Computer Graphics Group, Saarland University, Germany

Abstract

Recent research on high-performance ray tracing has achieved real-time performance even for highly complex surface models already on a single PC. In this report we provide an overview of techniques for extending real-time ray tracing also to interactive volume rendering. We review fast rendering techniques for different volume representations and rendering modes in a variety of computing environments. The physically-based rendering approach of ray tracing enables high image quality and allows for easily mixing surface, volume, and other primitives in a scene, while fully accounting for all of their optical interactions. We present optimized implementations and discuss the use of upcoming high-performance processors for volume ray tracing.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Raytracing

1. Introduction

This report presents the current state in interactive volume rendering based on ray tracing extending [MHB*00, MFS05]. The following introductory sections provide a brief overview about ray tracing, volume rendering, their correlations and introduce the terminology used in this report.

1.1. Ray Tracing

Ray tracing is a physically-based image synthesis technique [PH04] which is well-known for its excellent image quality. In past this rendering algorithm was considered too slow for interactive applications but recently even real-time frame rates on commodity PCs were achieved for polygonal scenes [RSH05, WBS02, WIK*06, WBS06] and for volume data sets (e.g. [GBKG04]). Its core concept is the *ray* for computing visibility and simulating the distribution of light in a virtual environment. A camera model is used to generate *primary rays* that are cast through the pixels in the image plane to determine the objects visible along a ray $R(t) = O + D * t$ with origin O and direction D. For computing the light received at O all contributions along $R(t)$ need to be sampled and accumulated while *secondary rays* may be used to include global lightning effects such as shadows, or multiple scattering of light (see Figure 1). However, in this report we concentrate on basic volume rendering without secondary rays.

We define the ray tracing algorithm as an algorithm that given a ray – or set of rays – enumerates or accumulates the contributions along the ray(s):

```
for all pixels do
  enumerate primitives along ray
    accumulate contribution
```

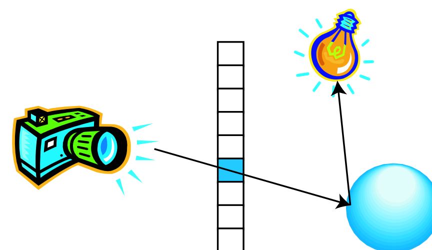

Figure 1: *A simple 2D ray tracing example: A ray is cast from a camera through a pixel in the image plane into the scene. After hitting the sphere, a color is computed at the hit point using a secondary ray (shadow ray) to calculate if the hit-point on the sphere is lit by the light source or blocked by an occluder.*

© The Eurographics Association 2006.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

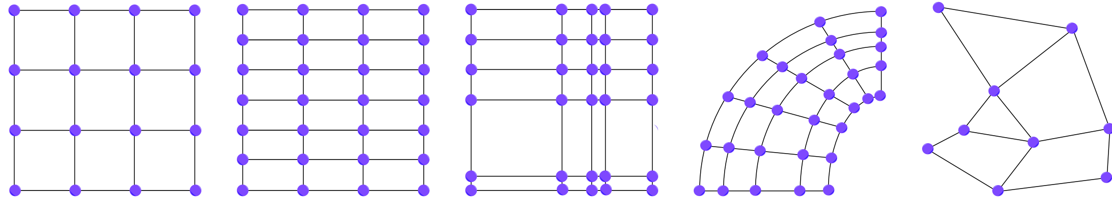

Figure 2: *2D examples of volumetric grid types (from left): regular, anisotropic regular, rectilinear, curvilinear, and unstructured.*

This is exactly the inverted *rasterization* algorithm, which iterates over primitives, determines the rays affected, and then accumulates contributions to each of them:

```
for all primitives do
  determine covered rays
    accumulate contribution
```

Ray tracing corresponds to the common classification of *image-order* rendering, while rasterization corresponds to the classification of *object-order* rendering. Hybrid methods exist as well. Note that the necessary sorting is implicit in ray tracing but not in rasterization.

In this paper we discuss volume rendering algorithms that follow the presented ray tracing definition and only briefly summarize other volume rendering approaches for reference.

1.2. Volume Rendering

Volume rendering is a process to generate 2D images from 3D volumetric data. Primary acquisition sources for volumetric data sets are (among others) Computed Tomography (CT) and Magnetic Resonance Imaging (MRI) scanners as well as computational simulations like field and fluid simulations.

1.2.1. Volumetric Data and Interpolation

Volumetric data sets consist of (multiple) scalar, vector, or tensor entities that are given at discrete locations in space. However, here we focus solely on single scalar values called *voxels* (volumetric elements). They are the smallest element of computation in volume rendering and represent physical quantities like pressure or density. The data type of the voxel values can be arbitrary ranging from binary up to floating point numbers. The structure of these scalar fields of voxels is versatile. Principle categories are whether the data set is *structured* or *unstructured* (aka. *irregular*). Structured data sets have an inherent organization that allows for simple addressing of the voxels given a position in the data set. On the other hand, unstructured data sets require additional information in the form of an adjacency list for addressing operations i.e. neighborhood computations. Figure 2 depicts some widely used volumetric grid types and their terminology.

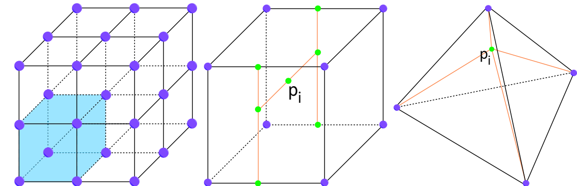

Figure 3: *Left: a cell defined by eight voxel locations. Middle: within a cell at each location p_i a value can be calculated by trilinear interpolation. Right: Within a tetrahedron a piecewise linear interpolation can be used to interpolate in-between values.*

For regular and rectilinear data sets we can define *cells* in the grid as a box defined by eight grid points. For curvilinear data sets a transformation from *physical space* to *computational space* [WCA*90] maps the distorted values into a regular grid which allows for using the box cell structure. Since volume data is defined only at discrete locations in space an interpolation must be performed to reconstruct in-between values. In the case of regular, rectilinear and curvilinear data sets, a piecewise trilinear interpolation is usually applied but higher order interpolations are also possible [The01] (see Figure 3).

Unstructured data sets have an irregular cell structure that is typically obtained by partitioning the data set into a tetrahedra mesh such that a piecewise linear (or higher order) interpolation can be applied within each tetrahedron. The tetrahedral partitioning yields also a useful adjacency information that can be used for rendering.

In general a value at point p_i can be reconstructed as weighted sum of the scalar values at the vertices of the cells. The weights are obtained by computing the local coordinates of p_i relative to the cell using barycentric coordinates (see Figure 3).

1.2.2. Ray Casting

In contrast to surface ray tracing, the *ray casting* algorithm (ray tracing with just primary rays) has to be changed slightly in order to render images of volumetric data sets. Since we do not only have empty space together with an

© The Eurographics Association 2006.

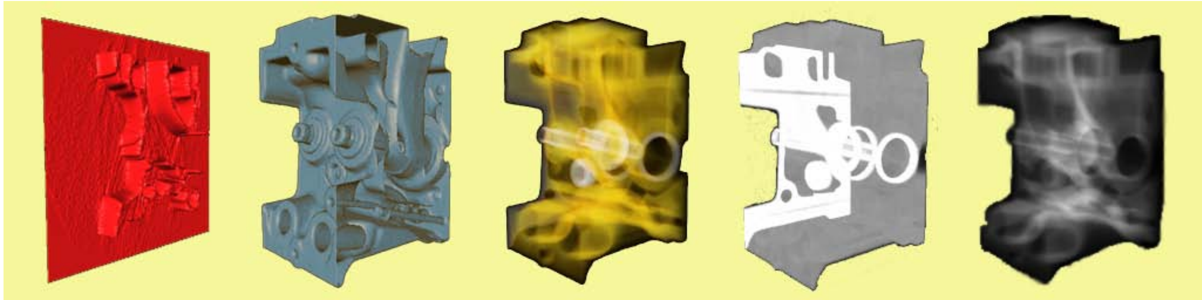

Figure 4: *Various volume rendering methods used to render the engine data set (from left): decomposition (one slice rendered as hight map), iso-surface, semi-transparent, maximum intensity projection (MIP), and x-ray.*

opaque surface definition, it is necessary to step along the ray through the volume to accumulate the contribution from the semi-transparent voxels (see Section 1.3). The stepping distance is not necessarily equidistant and can be set adaptively i.e. for importance sampling.

Ray casting based volume rendering has many benefits compared to other techniques like superior image quality, many acceleration techniques to speed-up the rendering process, and finally is embarrassingly parallel which can be exploited in various ways i.e. via software SIMD computations, multiple CPUs, multiple hardware pipelines, and compute cluster.

1.3. Volume Rendering Techniques

In order to obtain a meaningful visualization, a *mapping* is used to translate voxel values to optical properties, i.e. absorption and emission coefficients (aka. color and opacity), and other entities that convey visual information.

In general the literature differentiates between five volume rendering techniques that operate in image order each with a special field of application [SM00,LCN98,HJ04]. See Figure 4 for typical images and Figure 5 for a schematic overview.

Decomposition: Decomposition methods convert the data set into geometric primitives, e.g. spheres, or slices. The primitives are placed in space and are scaled and colored based on the mapping. Slices are commonly rendered as pseudo-colored textures, height maps, or are used for further processing like segmentation.

Iso-surface: In some applications it is important to examine the distribution of a certain single value (the *iso-value*) within the data set. The visualization of all points p within the data set with the same iso-value (the so called *level set*) yields a surface and therefore this method is widely called iso-surface rendering. We can define iso-surfaces formally as $f(x,y,z) = const$. A particular important application for this rendering method is virtual endoscopy.

Maximum Intensity Projection: For each view and pixel to be rendered, the Maximum-Intensity-Projection method computes the maximum value encountered along the ray. This method is often used for Magnetic Resonance Angiograms where thin structures, e.g. blood vessels, have to be rendered accurately.

Semi-transparent: Semi-transparent rendering considers the volume as a transparent medium. If light passes a volume it may be *absorbed*, *scattered*, or light may be *emitted*. To render a volume as a transparent medium a *transfer-function* (the mapping) maps the scalar voxel values to absorption, scattering, and emission parameters. Then, the contribution along a ray can be computed by solving the general volume rendering integral. The volume rendering integral in its low

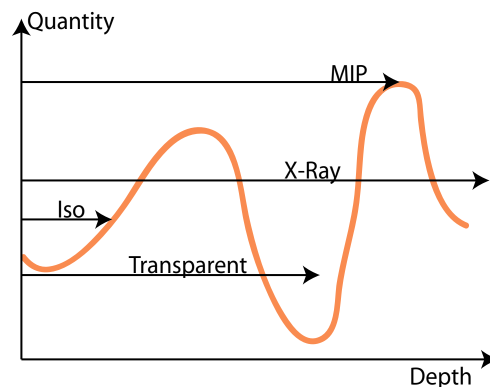

Figure 5: *A one dimensional example for ray tracing based volume rendering: MIP seeks along the ray the maximum value. X-Ray computes the absorption along the ray. Iso-surface rendering ends at the first hit-point of the ray with a user specified iso-value. The semi-transparent method accumulates emission and in-scattered light until the ray is saturated.*

© The Eurographics Association 2006.

albedo form (scattering is ignored) [HJ04]:

$$I_\lambda(x,y,D) = \int_0^L c_\lambda(p)\mu(p)\exp^{-\int_0^p \mu(t)dt}\,dp \tag{1}$$

computes for every pixel x,y with ray direction D and ray length L the incoming light I. λ is the wavelength and μ denotes the absorption factor [HJ04].

For the sake of rendering performance the general integral is usually greatly simplified and scattering is completely neglected. This simplified integral becomes after some transformations (i.e. expansion to Riemann sum) two recursive front-to-back compositing formulas, one for the color c (emission coefficient) and one for the opacity α (absorption coefficient). This is ideally suited for ray casting.

$$\begin{aligned} c_{i+1} &= c(p_{i+1})\alpha(p_{i+1})(1-\alpha)+c_i \\ \alpha_{i+1} &= \alpha(p_{i+1})(1-\alpha)+\alpha_i \end{aligned} \tag{2}$$

At each sampling point (p_i) along the ray this equations are computed. Whereby $c(p_i)$ denotes the color of the interpolated voxel value from the transfer function, and $\alpha(p_i)$ is the opacity. A nice property of this solution is that an early ray termination can be performed when α is above some threshold $(1-\zeta)$. This means that light behind that sample point which would have little effect on the pixel of the image plane is ignored. A back-to-front approximation of the volume rendering integral exists as well but since no early ray termination can be performed [SM00] this method is seldom used in volume ray casting applications.

X-Ray: rendering approximates also the volume rendering integral but ignores additionally the emission and only accumulates the absorption along a ray. This results in pictures which look like typical x-ray images. No mapping is used except for the final pixel value.

1.4. Content

In this report we present efficient software implementations for MIP, iso-surface, and semi-transparent volume rending in the context of regular/rectilinear data sets (see Section 3) as well as curvilinear and unstructured grids (Section 4). We discuss efficient traversal of the data structures and fast rendering computations within traversed cells. We also focus on optimized algorithms and efficient data layouts. For reference we briefly summarize in Section 2 non-ray tracing based volume rendering algorithms.

2. Alternative Rendering Approaches

In this section, we briefly cover alternatives to software volume ray tracing. We begin with projection, since this is one of the most often used techniques. Projection methods can be further categorized in cell-projection, vertex-projection (splatting) and texture-mapping. Software as well as hardware implementations exist for this approach. However, since graphics hardware became more and more powerful all approaches were adapted to GPUs for several years.

2.1. Cell Projection

An early software implementation based on projection was proposed by Lucas [Luc92]. The projected faces of an irregular grid are sorted using the Painter's Algorithm known from polygonal rendering. Lucas used only the centroids of each face as sorting criteria which may lead to an incorrect sorting. Wilhelm's et al. [WGTG96] uses a software scan conversion optimizing the sorting by using coherence between adjacent pixels as well as scan-lines. Cell faces are always decomposed into triangles. A kd-tree culls invisible regions from the current view point.

Williams et al. [WMS98] developed an elaborate volume rendering system purely based on projection. The paper includes a detailed description of analytically solving the rendering equation for producing high-quality images. As Bennet showed, this system can be well parallelized [BCM*01]. Another distributed renderer was earlier suggested by Ma et al. [MC97]. Here, the data set is partitioned allowing to render larger volumes. The volume cells are distributed round-robin-like among the nodes. Each node scan converts its local cells and sends the calculated ray segments to their final destination in screen space for sorting and merging.

It is also possible to project irregular meshes onto screen space but then process them in image-order using conventional ray tracing. It is a hybrid approach since the covered cells by a ray are partly determined before accumulating. Bunyk et al. [BKS97] decompose the entire tetrahedral mesh into triangles. Before applying ray casting, the complete mesh is projected onto the screen, which simplifies the intersection test from a 3D to a 2D problem. Adjacency information computed during preprocessing is used to accumulate in correct order.

Hong [HK98] uses the same approach for curvilinear volumes. Each hexahedral face is therefore decomposed into two triangles, resulting in twelve triangles per hexahedron. In a first version, each triangle was tested independently. In a follow-up paper, Hong [HK99] suggested therefore to group all twelve triangles together and apply a ray crossing technique. The number of intersections determines the triangle where the ray exits the hexahedron.

Rendering unstructured data consisting of a tetrahedral mesh was also early adapted to rasterization hardware. Shirley and Tuchman [ST90] decompose each tetrahedra in up to four triangles depending upon the view-point and project each triangle onto the image plane. Correct colors are only computed for the triangle vertices and linear interpolation is used for intersection points in-between a triangle.

© The Eurographics Association 2006.

Using 3D texture mapping, Röttger et al. [RKE00] were able to extend the projected tetrahedra algorithm [ST90] for hardware-accelerated and accurate rendering of volumetric data. Parts of the volume integral depending upon the length of the segment are linearly approximated by a modulation of the vertex colors. The remaining parts depend upon the textural coordinates only and can therefore tabulated in a 2D texture map.

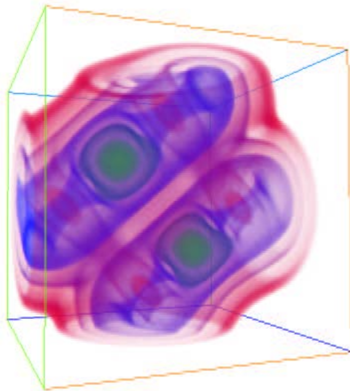

Figure 6: *The orbital data set rendered with a transfer function with Weilers GPU implementation using tetrahedral strips [WMKE04].*

Projection-based ray-casting for tetrahedral meshes were also adopted for Graphics boards. Weiler et al. [WKME03] uses the ray-plane intersection [Gar90] to determine the face exiting a tetrahedron. The ray integration relies on pre-integration, as described in [EKE01]. These computations are performed in the fragment program. Ray tracing is restricted to a single cell since multiple rendering passes have to be applied due to the limited flexibility of graphics boards. Interactive rendering of mid-sized models is possible with 2 to 5 fps. A more compact data set representation taking advantage of implicit neighbors [WMKE04] achieves the same performance with less memory consumption.

Another adaption of a CPU based renderer was suggested by Hong et al. [HQK05]. While Mora [MJC02] demonstrated an object-order ray casting approach for the CPU, here rasterization hardware allows for fast perspective projection of the volume cells. Both approaches are restricted to regular volumes. A min-max octree is first used for an efficient classification of cells. Sub-volumes of $N \times N \times N$ voxels are then projected onto the image plane. Fragments are generated corresponding to the rays intersecting with that cell. The correct order between sub-volumes is implicitly given by the min-max octree. Dividing each sub-volume into pre-computed layers further reduce the visibility ordering.

A fast iso-surface ray casting algorithm also based on object-order ray casting is proposed by Neubauer et al. [NMHW02] (see Figure 7 for two example images). The

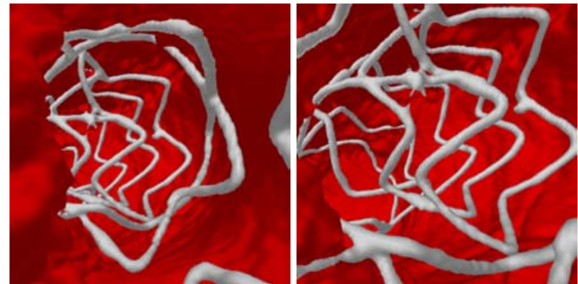

Figure 7: *Two example images of the chest data set rendered with two iso-surfaces (iso-values 440 and 1100) and Neubauers [NMHW02] approach. An average frame rate of* 0.8 *is reported with an image resolution of* 512×512 *pixel on a single PIV 1.9 GHz.*

complete data set is subdivided into *macro-cells* of size m^3 where m is usually between four and ten. This macro-cells are then used to build a min/max octree (similar to [WV92] and [WFM*05]).

For each pixel on the image plane that has not yet processed, the octree is traversed and at each traversal step the min/max values are checked whether boundary-cells are in the next sub-tree or not. At a leaf node the boundaries of the macro-cell are projected/rasterized onto the image plane. This yields a hexagonal footprint. For each pixel in this hexagon *local rays* are used to traverse the macro-cell grid. This reduces the number of traversal steps for the octree structure since the pixels that are covered by the hexagon would perform all the same traversal steps. For the macro-cell traversal, the method of Amanatides and Woo [AW87] is used. If a boundary cell is encountered, an intersection test is performed with the iso-surface and eventually normal and shading are computed.

2.2. Vertex Projection (Splatting)

Splatting is a forward mapping algorithm where the reconstruction of the original signal is achieved by spreading each data sample's energy across the cell's footprint in image space. This is often approximated by some 2D basis function. The volume is therefore represented as an array of overlapping basis functions where their amplitudes are given by the voxel values. Common kernels are Gaussian, which are stored in a footprint table. It was first described by Westover [Wes90] rendering regular grids on an CPU.

However, this approach is not well suited for perspective projection. Müller et al. [MY96] suggests therefore a hybrid method. The voxel contributions are partly pre-computed by splatting in object space. However pixel accumulations are

© The Eurographics Association 2006.

then ray tracing-like processed by shooting rays intersecting the splats in space.

Splatting can be further distinguished between compositing all splats back-to-front and the so-called sheet-buffer method [MC98]. Here, the splats are organized in cache-sheets, which are subsequently added back-to-front. These cache-sheets are aligned with the volume face most parallel to the image plane. This results in popping artifacts, if the orientation of the compositing sheets suddenly changes. Arranging the sheet-buffer parallel to the image plane overcome this problem [MC98]. To this end it is necessary to add slabs of partial kernels within the sheet.

Aliasing effects caused by the discrete evaluation of the splatting equation can be relaxed by adapting Heckberts elliptical weighted average (EWA) re-sampling filter for volume splatting [ZPvBG01]. The footprint function is replaced with a re-sampling filter. Each footprint function is now separately band-limited and hence respecting the Nyquest frequency of the rasterized image. Chen et al. [CRZP04] proposed an adaptive EWA filter to get rid of further aliasing artefacts. Such artifacts are caused by voxels far away where the sampling rate of diverging viewing rays falls below the sampling rate of the volume grid. For close voxels, the sampling rate of the rays is higher then the volume sampling rate. Approximating the EWA re-sampling filter with the dominant reconstruction filter only yields better rendering quality.

2.3. Texture Mapping

Cabral et al. [CCF94] was one of the first showing that texture capabilities of graphics boards can be used directly for rendering volumetric data sets. It can be seen as a hybrid approach between backward and forward projection, since the iteration over texture coordinates is ray tracing like while a forward projection loops over all values to find the appropriate texture coordinates to sum into. The slicing plane tri-linearly interpolates the scalar value and the texture mapped slices are blended into the frame-buffer in a back-to-front manner.

Engel et al. [EKE01] improved the rendering quality by proposing pre-integrated volume rendering. The ideas presented in [RKE00] are extended and improved for regular grids. Basically, all slices are converted to slabs, i.e. they are enriched with a thickness so that interpolated values in-between cannot be missed by an "unfavorable" defined transfer function. The numerical integration is split into two parts. One for the continuous scalar field and one for the transfer functions. The lookup tables needs modification only when the transfer function is changed. Röttger also combined this approach later with volumetric clipping and advanced lighting [RGW*03].

The approaches discussed so far compute all scalar values of the grid for rendering the volume, no matter if they are visible or not. Li [LMK03] proposed therefore to partition the volume into smaller sub-volumes with similar properties. These properties depend on the transfer function, e.g. scalar values within a certain range are grouped together. A kd-tree is used to render this partitioned volume with correct visibility order, where each node in the tree corresponds to a sub-volume. Each sub-volume is culled and clipped against an opacity map. This opacity map corresponds to a region of the frame buffer and stores the minimum opacity of the frame buffer pixels in that region.

Krüger et al. [KW03] addressed speed-optimizations reducing per-fragment operations. The early Z-test is exploited to terminate fragment processing for implementing early ray termination and empty space skipping.

2.4. Shear-Warp

Shear-warp [LL94] is still one of the fastest software implementations for volume rendering. The basic idea is to factorize the projection matrix into a 3D shear and 2D warp. Using the shearing the data set is transformed into sheared object space. In this space all viewing rays are parallel to one coordinate axis and the volume is considered as a stack of 2D slices. The 2D slices are then aligned and re-sampled such that they are perpendicular to the viewing directing. Then an intermediate image can be composed using the sheared object space values. Finally the intermediate image is warped to the image plane. For a perspective transformation each slide needs an individual scaling during re-sampling.

Rendered images are prone to show stair-casing artifacts near 45° viewing angle. Intermediate slices lying halfway between two adjacent volume slices overcome this problem. Images may furthermore blur during a zoom-in, since the re-sampling of the warp matrix is not adaptive. An enhanced version solving these problems can be found in [SM02] but increase the computational cost.

2.5. Custom Hardware

The need for custom graphics hardware arise with the demand for real-time volume rendering systems. Neither GPUs nor CPUs were fast enough at that time to achieve this goal. Most systems has been developed for rendering regular data, e.g. *Cube* [KK99], *Vizard* [KS97] and, *VolumePro* [PHK*99]. Due to the highly regular computation all of them achieved real-time frame-rates allowing for interactive rendering. However, changing or extending custom hardware is tedious and costly. Another disadvantage which they share with GPUs is the limited memory. Out-of-core solutions are in general not an alternative due to the high bandwidth needed.

Cube [KK99] is based on a hybrid-order algorithm based upon Shear-Warp. However, it was planed to compute only the shear-step on-board and let the graphics board warp and render the image. Eight identical rendering pipelines are able to render a 256^3 volume at 30 fps.

© The Eurographics Association 2006.

Never commercially realized, Cube [KK99] was the predecessor of the well-known VolumePro board [PHK*99]. Although the scalability was enhanced, perspective rendering was still not possible. The latest generation consists of separate sample and voxel processing pipelines. Voxel processors traverse data slice-by-slice in memory order and store them in on-chip buffers. These buffers are traversed by sample processors responsible for illumination, filtering and compositing. More interestingly, perspective rendering is now possible [WBLS03].

Vizard [KS97] and Vizard II [MKW*02] was based on an image-order offering full ray casting including early-ray termination. Phong shading was implemented using look-up tables. The performance is not comparable to *VolumePro* due to the FPGA implementation. This makes this system more flexible at the cost of the achieved frames rates.

The methods discussed so far provide fast and reliable volume rendering. Parallization is in almost all approaches possible which works, e.g. in favor of modern GPUs. However, as we will see in section 5.3 the flexibility of modern graphics boards allows the implementation of ray casting directly. This significantly improves the image quality while preserving the speed. Splatting, Texture-Mapping and Shear-Warp are also fast and memory-efficient but lacks rendering quality. Custom hardware is fast and delivers high-quality but offers so far only a limited flexibility.

Scientific visualization demands high-quality *and* flexibility. Ray tracing naturally offers both, while being always considered as too slow. In the following sections we will present algorithms and data structures to accelerate volume rendering and techniques to improve image quality.

3. Ray Tracing based Rendering of Rectilinear Data

Regular and rectilinear data sets are the most common grid types in volume rendering. Almost all scanner devices and many simulation applications output these grids. In the following three sub-sections we will discuss efficient algorithms for iso-surface rendering, semi-transparent volume rendering, and MIP in addition to the necessary data structures to speed up the rendering process or to increase the image quality.

3.1. Ray Tracing based Iso-Surface Rendering

The performance of iso-surface ray tracing depends heavily on two algorithms: the intersection test of the ray with the implicitly defined surface function within a cell, and the identification of cells which contain a piece of the iso-surface in front-to-back order.

3.1.1. Ray Iso-Surface Intersection Tests in a Cell

Various methods have been proposed to calculate the intersection point of a ray with the implicit iso-surface function in a cell.

Analytic Methods

The iso-surface within a cell can be reconstructed using a trilinear interpolation. To do so, the ray equation $R(t)$ is substituted into the trilinear interpolation equation. Solving the resulting equation for t results in a cubic polynomial (see [PSL*98] for a complete derivation and discussion).

This cubic polynomial can be solved analytically by applying i.e. Cardano's formula. Nonetheless, an efficient and numerical stable implementation is non-trivial [Sch90], even when using `double` precision computations. Furthermore, computational expensive calculations like `(a)cos` are involved. Marmitt et al. [MFK*04] proposed a faster and numerical more stable method, although only `single` precision computations are used. The key to their fast algorithm is the observation that actually only one root of the cubic polynomial is needed. They isolate the roots by computing the extrema of the polynomial. These extrema split the ray segment into at most three parts. Then they step through these segments from front to back, computing the data values at its start and end point from the cubic polynomial. Once an interval containing zero is found, it can be guaranteed that it contains exactly one root, that the root lies in the interval and that it is the first relevant root.

The root in the interval can then be calculated by a simple recursive bi-section until the desired accuracy is reached or a small number of bi-section steps is performed. The latter one is more efficient and after three to four iterations commonly no visual differences can be observed. The calculation of an intersection point with this method is exact, numerical more stable, and approximately three times [MFK*04] faster compared to i.e. Schwarze's algorithm [Sch90] to solve Cardanos formula.

Approximative Methods

Faster methods are only reported for intersection tests that roughly approximate the intersection points but may fail to find valid intersections although existing. One representative for this is the method of Neubauer [NMHW02]. He suggested to use repeated linear interpolations. At the intersection points p_{in} and p_{out} of the ray with the cell the values v_{in} and v_{out} are calculated. Then it is assumed that the function along p_{in} and p_{out} is linear. Solving the linear interpolation formula for t_i an intersection point t_i can be calculated. By applying a trilinear interpolation at t_i and looking at v_{in} and v_{out} a ray segment can be identified where a more accurate intersection point can be found. This process is repeated recursively. Although this method is computational very cheap it is error prone since it only linearly approximates the trilinear function.

Intersection Test Satisfying Smoothness Conditions

Until now we have only considered intersection tests that operate solely in one cell and thus there is no higher order continuity, i.e. C^1, guaranteed at cell boundaries.

© The Eurographics Association 2006.

Rössel et al. showed in [RZNS04] how to exploit *quadratic super splines* to reconstruct iso-surfaces that satisfy a pseudo C^1 continuity criterion along cell boundaries and thus produce smooth iso-surfaces contours. To do so, each cell of the data set is partitioned into 24 congruent tetrahedra and 65 Bernstein-Bézier coefficients are calculated. This allows for performing a ray iso-surface intersection test by solving *quadratic equations* along a ray, for each tetrahedron. The quasi-interpolating spline is a univariate piecewise quadratic polynomial.

The required 65 coefficients per cell (50 on the cells faces, 14 on the midpoints of the tetrahedra edges and one at the center position) are computed by a repeated averaging pattern using 27 data values at the centers from the surrounding cells. This results in ten coefficients per tetrahedron (four at the vertices and six at the middle of the edges).

To intersect a ray with the iso-surface within a tetrahedron, first three values w_1, w_2 and w are calculated applying de Casteljaus algorithm at the intersection points of the ray with the tetrahedron q_1 and q_2 as well as their midpoint $(q_1+q_2)/2$ using the ten Bézier coefficients of the tetrahedron (see Figure 8).

The required quadratic equation

$$\alpha\tau^2+\delta\beta\tau+\delta^2 w_1=0 \qquad ,\tau\in[0,\delta], \tag{3}$$

specifying the intersection point p of the ray with the iso-surface can be now set up using w_1, w_2, and w. δ denotes the maximum in the hit interval, $\alpha=2(w_1+w_2-2w)$, $\beta=4w-3w_1-w_2$.

This intersection test is quite efficient in terms of computational requirements and allows for rendering high quality iso-surfaces which satisfy a pseudo C^1 continuity along cell boundaries of the iso-surface. Additionally, shading gradients can be directly computed from the polynomial pieces of the splines. Nevertheless, the memory requirements for the needed coefficients is high and a careful tradeoff must be made between pre-processing of all values and on-the-fly computation.

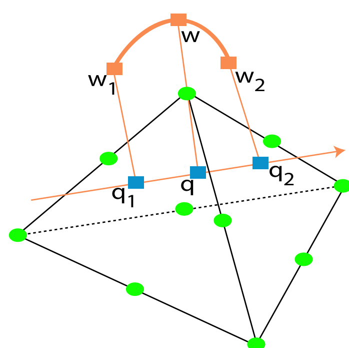

Figure 8: *The ten Bézier points (green points) of a quadratic polynomial inside a tetrahedron are associated with the Bernstein-Bézier coefficients. The restriction of this trivariate polynomial piece to an arbitrary ray is a quadratic, univariate polynomial (orange curve) which is uniquely determined by the values (orange boxes) at three points (blue boxes).*

3.1.2. Fast Boundary Cell Traversal

A naïve ray casting algorithm would step along a ray and test all pierced cells whether a piece of the iso-surface can be found within the cell. This is only applicable for small data sets due to the high number of memory requests for the voxels which would cause cache thrashing. Additionally, it is computational expensive to determine if a cell is a boundary cell. In order to quickly locate cells that contain the specified iso-value along the ray, acceleration structures and -techniques are required.

Implicit Min/Max KD-Trees

An approach by Wald et al. [WFM*05] utilizes min/max kd-trees for *coherent* iso-surface ray tracing (see Figure 9 for some example screen-shots). In coherent ray tracing, SIMD extensions can be successfully exploited for traversing *packets* of rays in parallel. This can be realized efficiently with kd-trees, which require only a single binary decision per ray in each traversal step. Although kd-trees seem ill-suited for data-parallel packet traversal, as all of the cells in a regular data set are small and thus rays in a packet could diverge and may traverse/intersect different cells, this argument is less relevant for iso-surface rendering.

A particular iso-surface defined by the data set is only located within a small subset of all cells. These cells, called *boundary cells*, are in general irregularly distributed, sparse, and often enclosed by large regions of empty space. These properties are ideal for kd-trees [Hav01] (see Figure 10).

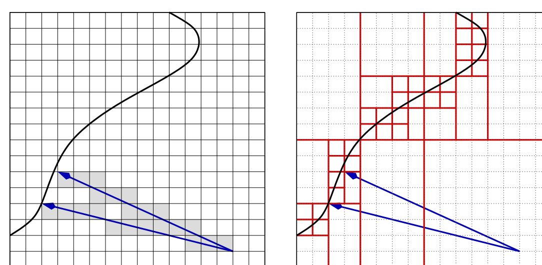

Figure 10: *Rather then traversing all cells along the rays until an iso-surface is hit (left), a top down traversal in a min/max hierarchy can be used to quickly skip regions without boundary cells (right). Since the traversal of neighboring rays bears high coherence,at least on the higher levels of the hierarchy, SIMD packet traversal can be efficiently applied.*

© The Eurographics Association 2006.

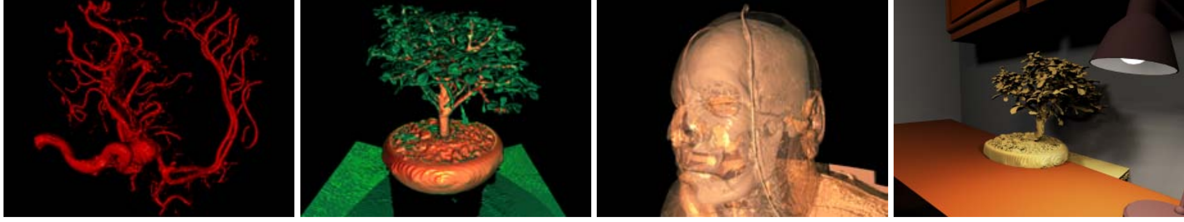

Figure 9: *Various iso-surfaces rendered with Wald's kd-tree approach [WFM*05]. From left to right: the aneurysm, the bonsai data set with two opaque iso-surfaces, the head of the visible female project with two transparent surfaces, and finally the bonsai tree in a polygonal environment including instant global illumination [WKB*02]. All images can be rendered at interactive frame rates on commodity PCs.*

Building such an implicit kd-tree is quite easy. First, a kd-tree over all the voxels of the entire data set is build. This is done in a way that a kd-tree split plane always coincides with the cell boundaries of the volume's cells, yielding a one-to-one mapping between the volume's cells and the voxels of the kd-tree. A simple way is to split the volume at the cell boundary that is closest to the center in the largest dimension. Second, the minimum/maximum values are computed recursively for all kd-tree nodes: Each leaf node stores the min/max values of its associated cell, and each inner node stores the min/max values of its children. Note that this acceleration structure is similar to the one used by Wilhelm and van Gelder [WV92].

In a naïve implementation, one would store the entire tree simply using the same node layout as for surface ray tracing [WSBW01] simply by adding the two min/max values to each node to allow early subtree culling if no parts of the iso-surface can be found in the subtrees. Assuming default 16-bit data values this naïve approach however requires 12 bytes for each node: 8 bytes for specifying the plane and pointers, plus 4 bytes for the min/max values. As a kd-tree of N leaves has an additional $N-1$ inner nodes, for N 16-bit voxels it requires $(2N-1)\times 12$ bytes for the kd-tree. At 2 bytes per input data value, the size of the acceleration structure would then be 12 times the size of the input data. Obviously, this overhead is too high. For 8-bit voxels, the relative overhead would be even worse.

Fortunately, the memory overhead can be significantly reduced. If we assume for a moment that the number of cells in each dimension is a power of two (the number of voxels is then 2^N+1), the resulting kd-tree would be a balanced binary tree, i.e. all its leaves are in the same level. In a balanced binary tree however it is easy to show that all the nodes in the same level l will use the same splitting dimension d_l. Therefore, we only have to store this value once per level and not in all nodes of level l. Furthermore, in a balanced kd-tree no pointers are needed for address computations.

The remaining split plane positions in the nodes can also be avoided. The number of distinct splitting planes for a particular level l and splitting dimension $i \in \{x,y,z\}$ can be calculated by 2^n, where n is the number of i as splitting dimension up to level l. This allows to store all splitting position of a particular level in a small array.

Scene	Single PC C	Single PC SIMD	Ratio	5-Node Cluster C	5-Node Cluster SIMD	Ratio
Bonsai	3.4	5.2	1.5	16.2	24.6	1.5
Aneurysm	3.0	6.2	2.0	14.6	29.8	2.0
ML 512^3	1.2	2.3	1.8	6.1	11.3	1.8
Female	2.7	4.2	1.5	13.6	20.7	1.5
" (zoom)	2.3	7.9	3.5	11.2	39.1	3.5
LLNL	0.9	1.3	1.5	–	–	–
" (zoom)	1.6	5.4	3.9	7.6	28.7	3.8

Table 1: *Overall rendering performance data when running our framework in various scenes including diffuse shading, for both a single (dual-CPU) PC, as well as with a 5-node dual-Opteron cluster. The overview of the LLNL data set could not be rendered, because the memory footprint at this view was larger than the 2 GB RAM per client in the cluster setup.*

All that remains to be stored are the min/max values in the kd-tree nodes. A further memory saving can be accomplished if the min/max values at the leaf nodes are not stored, and instead are computed on the fly from the cell's voxels. This is tolerable since this computations have to be performed only at leaf nodes.

As mentioned before, this implicit kd-tree construction works only if the data set has 2^N cells in each dimension. One simple method of making arbitrary data sets comply to this constraint would be to *pad* them to a suitable size. Instead, a better solution is to assume that all nodes are embedded in a larger *virtual* grid that exceeds the scene's original bounding box and to build the kd-tree over that virtual grid.

By properly assigning the splitting plane positions, we can make sure that all virtual nodes lie outside the real scene's bounding box. As the kd-tree traversal code always clips the ray to that bounding box we know that rays will never be traversed outside the box, and thus can guarantee that no ray will ever touch any of these virtual nodes. As such, we do not have to store them, either. For more details see [WFM*05].

© The Eurographics Association 2006.

Implicit KD-Tree Traversal

The described data structure is – except for the min/max values stored per node – very similar to the polygonal case where kd-trees are usually used. Thus, the already existing traversal code requires only a minor modification: During each traversal step we first test whether the current iso-value lies in the min/max range specified by the current node. If this is not the case, we immediately cull this subtree, and jump to the next node in the traversal stack. Otherwise, we perform exactly the same operations as in the polygonal case (see [Wal04] for a thorough discussion).

Results

Table 1 provides some performance numbers of Wald's approach. The data sets used for their experiments are: The bonsai tree (256^3), the aneurysm (256^3), various resolutions of the synthetic Marschner-Lobb data set (512^3), the Visible Female ($512^2 \times 1734$), and the Lawrence-Livermore (LLNL) Richtmyer-Meshkov simulation ($2048^2 \times 1920$) These data span a wide range of different data, from low (ML) to high surface frequency (bonsai, LLNL), from medical (aneurysm and female) to scientific data (LLNL), and from very small (aneurysm) to extremely large data sets.

The performance is compared between a pure C and SIMD implementation (parallel ray kd-tree traversal and intersection test) on a single PC with an 1.8 GHz Opteron 246 processor and on a 5-Node cluster each equipped with two of these CPUs linked via Gigabit Ethernet. At a resolution of 512×512 pixels one can achieve interactive performance for all test scenes even on the single PC. An almost perfect performance increase can be achieved by running the framework on multiple PCs in parallel. As can be seen, this setup allows up to 39 fps, even including the most complex data sets. Finally, since the kd-tree structure is stored in the implicit form, rectilinear data sets can be supported as well.

3.1.3. Interactive Large Iso-Surface Ray Tracing

An interactive out-of-core iso-surface ray tracing engine was presented by DeMarle et al. [DPH*03]. They exploit a PC cluster to render volume data sets that are too large to fit into the main memory of a single PC (see Figure 11). The core rendering engine is basically a port of Parkers [PSL*98,PPL*99,PSL*99] *-Ray engine. The *-Ray ray tracer is considered as one of the first interactive ray tracing systems and runs on an parallel shared memory SGI Onyx 2000 with typically 128 - 256 processors.

For volumes, the ray tracer uses a hierarchical grid acceleration structure and a three level bricking approach with a very low memory overhead for the acceleration structure. In some sense this is similar to storing min/max values only every N levels in a tree structure.

For the time-varying Lawrence Livermore National Laboratory (LLNL) data set of a Richtmyer-Meshkov instability simulation (270 time steps) with a voxel resolution of

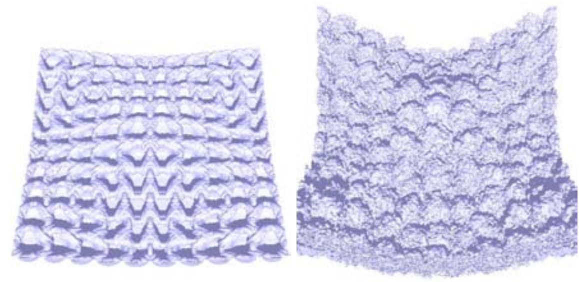

Figure 11: *Two time steps (45, 270) of the LLNL data set rendered with DeMarles approach [DPH*03] between* 6.1 *(left) and* 2.1 *(right) fps using a PC cluster with 31 machines equipped with two Pentium IV 1.7 GHz each. The resolution for each time step is* $2048 \times 2048 \times 1920$ *voxel.*

$2048 \times 2048 \times 1920$ and 8 bit voxel values (7.5 GB per time step) only 8.5 MB of memory are needed per time step for the hierarchical grid data structure.

Their application uses a typical client/server approach similar to Wald's distributed OpenRT rendering system [Wal04]. A master system divides the image into rectangular tiles and distributes the render tasks to the render nodes on a per tile basis. If a node is a multi-processor system the render task is broken down into sub tasks i.e. scan-lines. A `DataServer` container on each node is used as an abstraction layer for data management. The `DataServer` is shared between all processors using *semaphores* and *shared memory*. If a ray touches a brick, it must request the brick from the `DataServer` which loads the data from the network if the data are not present.

The hierarchical grid is similar to Neubauer's approach (see Section 2). At the leaves a macro-cell size of approx. 8^3 voxels is used and the min/max values are propagated up in the hierarchy. Rays start traversal at the top hierarchy level using an incremental grid walking algorithm. If an iso-surface is contained within a macro cell, the traversal starts again at the lower level grid. This process is executed recursively through the hierarchy until the ray hits the iso-surface or the ray exits the volume. Ray iso-surface intersection tests at the lowest level are performed using an analytic approach solving Cardano's formula (see Section 3.1.1).

Results

DeMarle presents performance numbers for two data sets: the torso part of the visible female (428 MB) and the above mentioned LLNL data set (7.5 GB). Their test system consists of 32 Linux PC each equipped with 2 Pentium IV 1.7 GHz processors and 1 GB of RAM. All renderings are done with a screen resolution of 512×512 pixels.

The first benchmark, performed with the visible female data set, is a scaling test. As expected the performance in-

© The Eurographics Association 2006.

creases linearly with the number of render nodes. With one render node approx. 1 fps can be achieved and with 32 nodes up to 22 fps. All in between values lie on a almost perfect linear line. The LLNL data set can be rendered, dependent on the complexity of the iso-surface, with 2.2 up to 6.7 fps using the complete cluster system.

3.1.4. Comparisons

In comparison to Wald's kd-tree approach the system of DeMarle consumes much less additional memory for its acceleration structures. However, the reported performance is almost five times slower for the visible female data. Wald reports a performance of approximately 8.1 fps on a single PC with two AMD Opteron 1.8 GHz CPUs. Furthermore, their out-of-core extension from [WDS04] allows to render the LLNL data set interactively on a *single* PC. Reasons for this are mainly the optimized intersection test and the kd-tree acceleration structure which in general compares favorably to the multi-level grid structure used by DeMarle.

3.1.5. Discussion and Conclusions

In this section we have presented several algorithms to intersect rays with implicit iso-surfaces, acceleration structures and further optimization techniques as well as out-of-core techniques for the rendering of extremely large datasets.

For iso-surface rendering the intersection tests of Marmitt [MFK*04] and Rössel [RZNS04] have both their strength and weakness. If memory overhead and computational costs are not an issue and high accuracy, i.e. C^1 continuity, is required the approach of [RZNS04] is probably the algorithm of choice. In all other cases the approach of Marmitt et al. [MFK*04] should be considered since correct results are obtained, i.e. all intersections are found, the computational costs are very low, and no additional memory is needed.

Similar as for polygonal scenes, kd-trees have proven to be an efficient acceleration structure for iso-surface rendering and the memory footprint is also within practical limits. Furthermore, just a single technique is necessary and no "*magic*" parameters have to be selected to speed up the rendering performance.

In recent work of Wald et al. [WIK*06, WBS06], Reshetov et al. [RSH05] and Lauterbach et al. [LYTM06] new traversal algorithms for grids, kd-trees and bounding volume hierarchies have been proposed to speed up the ray tracing performance for polygonal scenes up to a factor of ten. It would be interesting to exploit these new techniques also in the domain of volume rendering.

3.2. Semi-Transparent Rendering

Maybe the most important visualization method for volumetric data sets is the semi-transparent rendering model. Many interactive solutions have been proposed during recent years, each trading rendering speed for quality or flexibility.

The key for fast semi-transparent volume rendering is the optimization of the following three elements (beside the early ray termination previously described in Section 1.3):

60% to 80% of all voxel values are classified transparent by the transfer function in real world applications and thus are not visible. Efficient *space leaping* structures and techniques are required to skip cells and complete volume regions that do not contribute to the final image (Section 3.2.1).

Many results of computations, i.e. normal estimations for shading, are (re)computed whenever they are needed. As neighboring rays tend to perform very much the same computations, it is worthwhile to cache values which can be reused for the next ray, or packet of rays (Section 3.2.2).

Volume rendering requires a very high memory bandwidth and may causes cache thrashing by displacing frequently used data values from the caches. In order to achieve high rendering performance this cache thrashing should be reduced as much as possible since memory requests can be extremely expensive in terms of clock cycles (Section 3.2.3).

3.2.1. Space Leaping/Empty Space Skipping

Space leaping is maybe the most effective acceleration technique for semi-transparent volume rendering as it reduces the number of reconstruction samples. In general we can differentiate between techniques which use a spatial data structure to skip empty regions and methods that exploit coherence without pre-computing any data structures.

Space Leaping with Spatial Data Structures

In Section 3.1.2 we have described two acceleration structures namely multi-level grids and kd-trees to speed up iso-surface ray tracing. Other structures like octrees [WV92] and bounding volume hierarchies can be exploited as well. The important point is that min/max values are available at every granularity level of the acceleration structure in order to know the range of values within the sub-structures.

This min/max values are not effective for semi-transparent rendering as the transfer function determines which voxels are visible and which are not. Commonly a *summed area table* is build over the opacity values of the transfer function. With two simple lookups in this table, one for the min and one for the max of a data structure element, it can then be determined if a spatial region is fully transparent or not. However, this scheme is only correct if the color and opacity lookup in the transfer function occurs after sampling (post-classification). Pre-classification first determines the colors and opacities for the cell's voxels and then use them for interpolation. Thus, it might be possible that a color value of zero is computed when the mapping of the voxels specifies

© The Eurographics Association 2006.

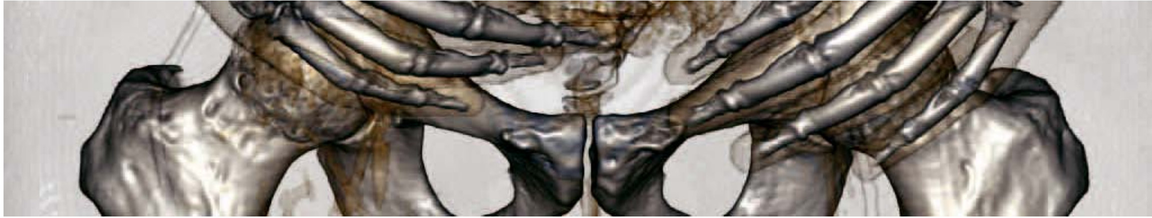

Figure 12: *A close up of the visible male data set (hip region) rendered with the approach of Grimm et al. [GBKG04].*

the opacities to be zero. In post-classification a voxel value might be computed which has a non-zero color. This behavior occurs often when high opacity frequencies are specified in the transfer function for large voxel value intervals.

To avoid a false transparent/opaque classification for a particular node in the spatial hierarchy Grimm et al. [GBKG04] proposed for pre-classification applications the use of *quantized binary histograms*. These histograms are conservative such that a transparent region can be classified as opaque but not vice versa and thus no non-transparent regions are skipped.

Basically this histogram is a list of size s where s is the range of voxel values i.e. 255 if 8 bit voxel values are given in a data set. During the pre-processing phase for each brick such a histogram is build. By iterating over the brick's voxels, a flag is set to every $s[v]$, where v is the voxel value, and the rest is left to zero. To save memory and increase the lookup efficiency this list is quantized into buckets of size 32. A similar histogram can be build for the opacity values of the transfer function. To determine if a brick is fully transparent one has to iterate over all buckets in both lists and check if all buckets are classified as transparent.

Space Leaping without Spatial Data Structures

Sarang et al. [SK04] proposed a space leaping technique that exploits coherence in brute force ray casting applications. Their idea is driven by the fact that a group of rays very likely traverses the same amount of empty space until a semi-transparent region is reached.

To exploit this observation two different kinds of rays are used in two render passes: *detector* rays, shoot in the first pass, and *leap* rays used in the second pass. In the first render pass one detector ray is shoot for a group of four pixels (see Figure 13 for the sampling pattern) on the image plane. All detector rays are shot and behave like in a traditional ray casting system performing ray marching, sampling etc. Additionally they keep track of the first non-empty region they reach and store this information in a *leap buffer*.

For every pixel the leap buffer has in the image plane a corresponding entry that encodes a distance. For detector rays this distance is the number of transparent samples until a semi-transparent cell is found. The empty entries for

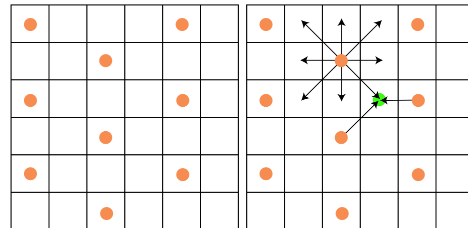

Figure 13: *Left: a part of the image plane showing the location of the detector rays (orange dots). For the rest of the pixels leap rays are used. Right: the distances of the detector rays are spread to the remaining pixels by using a minimum operator.*

the leap rays are filled with the *minimum* number of samples during which the rays encounter empty space by spreading the values from the detector rays to the eight neighboring pixels. The space leaping rays are then shoot in the second pass starting at the minimum position encoded in the leap buffer.

Please note that this algorithm is only correct if the volumetric object is not too far away from the camera. If this distance exceeds a certain value rays diverge too much i.e. in perspective rendering and sampling artefacts can occur due to missed features. However, Sarang et al. [SK04] report that with this algorithm brute force ray casting can be accelerated up to 165% for various models with a size between 64^3 and $512 \times 512 \times 361$. This speed up yields interactive frame rates of up to ten fps with a view-port size of 256×256 pixel.

3.2.2. Caching and Pre-Computations

As previously mentioned it can be worthwhile to cache some values during ray casting that are likely to be reused in near future. Grimm et al. [GBKG04] proposed to use a *gradient cache* on a per brick basis (see Section 3.2.1), for gradients calculated in the *derivative first* manner [MMMY97] (see Figure 12 for an example image).

They use two data structures for efficient caching. The gradient cache itself and a bit list which indicates if a particular gradient has already been computed and is in the cache or not. The number of entries in the cache is equal to the

© The Eurographics Association 2006.

number of voxels in the brick. Every time a gradient has to be computed the corresponding presence bit is checked if the gradient is already computed. If not, the gradient is calculated. After processing the brick, the cache is flushed for the next brick. Therefore the gradients on the border of a brick have to be calculated up to eight times, which is rarely the case and thus no significant performance is lost. In average a speed up factor of approximately 2.2 can be expected by using the gradient cache. Nevertheless, this high speed up can only be expected if orthogonal projection is used. In perspective rendering the rays can diverge for bricks that are far away from the image plane. As a result the distance between neighboring rays is large, which will lower the reuse ratio for the caches.

3.2.3. Memory and Cache Optimizations

Knittel [Kni00] presented with UltraVis a highly optimized ray casting approach for semi-transparent volume rendering which is especially targeted for Intel's Pentium III processor. His approach depends heavily on concrete details of the cache replacement strategies of the PIII processor. He

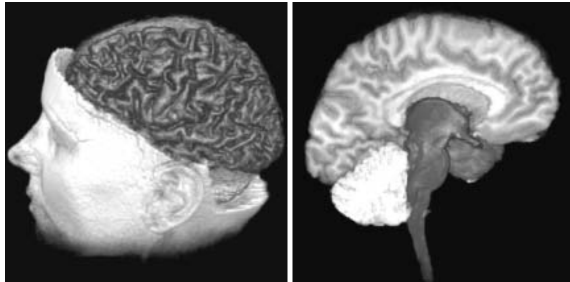

Figure 14: *Two example images of the UNC MRI-head, with* $256 \times 256 \times 110$ *voxel resolution, rendered with the UltraVis system on a PIII 500MHz system and* 256×256 *image resolution. The images are rendered between* 1.7 *and* 7.0 *fps.*

observed that a major obstacle for high performance volume rendering is the limited memory bandwidth. Ray tracing based volume rendering has a memory access pattern that does not fit today's cache architectures and thus many data have to be reloaded numerous times due to cache thrashing.

Cache Optimizations and Spread Memory Layout

In Knittel's UltraVis system three different data structures are accessed: the *volume data* itself, the *transfer function* and *additional parameters* such as thresholds and shading parameters.

In order to avoid the replacement of frequently used data in the cache, i.e. the transfer function values, a spread memory layout is used to virtually *lock* the data in the level one cache once they are loaded. The Pentium III processor uses a *4-way associative cache* which means that there are four cache lines (the so called *set*) where a particular data from the memory can be mapped to. Considering that the main memory is internally subdivided into equal sized *pages* whereby the offset of a particular data in this page determines the set in the cache, four times more memory for the volume data set is allocated as actually needed. The voxel data are then only placed in the first quarter of each page. Now we know that data which are placed in the remaining parts of the pages will never be replaced by voxel data, within the cache.

In order to reduce the cache thrashing of the voxel data, the order of the voxel values is altered via a cubic interleaved address function. Using such an interleaved memory storage a cube of n^3 voxels occupies $n \times n \times n$ cache locations (assuming n is a power of two, and n^3 is smaller than the cache size). This reduces cache thrashing and thus increases the cache hit ratio since neighboring rays very likely access some of the previous cached voxel values.

3.2.4. Discussion and Conclusions

Many techniques have been proposed to speed up semi-transparent volume rendering and only a fraction could be discussed here. However, today there is no software implementation that is able to render high quality images of mid-sized data sets on a single PC without any limitation either in terms of quality or flexibility i.e. orthographic rendering is used to simplify the rendering process. Furthermore, acceleration techniques like object-order ray casting [MJC02] are only useful for primary rays. If secondary rays are needed there is no single basis like the image plane where these tricks can emanate from.

However, new processors like IBMs Cell are on the way which can avoid some of the problems that are inherent in semi-transparent volume rendering. For example Knittels approach to avoid cache thrashing is not necessary on the Cell architecture since there is simply no cache on the Cell's SPUs. The new challenges are then mainly how to keep the SPUs busy using i.e. *virtual hypertheading* techniques in software as well as "hand tuned" caching for data which are not frequently used and sometimes altered. Additionally, todays solutions which rely on massive computational power using compute clusters can maybe shrunk to a single PC using the Cell processor.

3.3. Maximum Intensity Projection

MIP is an important technique mainly used in real world medical imaging applications to visualize thin structures like blood vessels or contrast-enhanced tissues. Not very much research has been done for this kind of volume rendering [ME05].

Parker et al. suggested in [PPL*99] the use of a priority queue in conjunction with a spatial data structure i.e. min/max multilevel grids. The priority queue is used to keep

© The Eurographics Association 2006.

track of cells or macro-cells in the grid with the maximum values. Although they use the priority queue in conjunction with a min/max multi-level grid all other common min/max hierarchical data structures could be exploited as well.

First the priority queue is initialized with the root node and its maximum value. Iteratively, the node with the highest max value is taken from the list and its children are inserted instead. At a leaf node, a macro-cell traversal is started and at each pierced cell-face a bi-linear interpolation is performed since they assume a linear function along the ray and thus no extremal value can be found *within* a cell. The maximum value encountered along the ray segment through a macro-cell is again stored in the priority queue. The algorithm terminates if one of these leaf cells appears at the head of the priority queue.

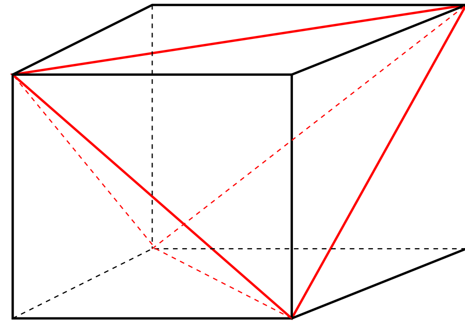

Figure 15: *A convex hexahedron can be approximated by five tetrahedra if the renderer supports tetrahedral meshes only.*

4. Ray Tracing based Rendering of Irregular Data Sets

Handling curvilinear or even unstructured data is more demanding compared to regular grid structures. However, software ray-casting systems were proposed as early as 1990 [Gar90, Lev90, WCA*90]. Different methods have been developed in the following years. Usually, two steps needs to be taken for rendering irregular data. In a first step, the initial cell (tetrahedron or hexahedron) is determined using some acceleration structure over the volume boundary faces. The following step iteratively traverses all subsequent cells along a ray until the last cell is reached. Incremental traverser require adjacency information and hence do not allow for sliding interfaces, i.e. complete faces must be shared between adjacent cells

4.1. Locating the initial Cell

One obvious way to find the initial cell is the extraction of boundary faces from the data set for tetrahedral and hexahedral meshes. Such faces are easily identified as they are the only faces not shared between two cells. A spatial index structure over those faces of the cells then allows to quickly locate the entry point and cell using standard ray tracing techniques.

Several researchers [BKS97, HK98, HK99, KN91] proposed to project the boundary faces onto the image-plane and fetch the associated cell from the data set. A disadvantage of these approaches is, that they require an update each time the view-point changes. This is avoided by using a spatial index as acceleration structure. So far grids [Gar90, PPL*99] and kd-trees [MS06] were proposed.

4.2. Ray-Primitive Traversal

Image order ray casting was not only one of the first methods proposed for rendering irregular grids but also reached interactive frame rates as early as 1999 [PPL*99]. We will therefore cover recent implementations in this area in more detail.

Garrity [Gar90] and Williams et al. [WCA*90] were the first who considered software ray casting for irregular grids. Wiliams decomposes each hexahedral cell into twelve triangles and uses the barycentric coordinates for interpolation of data values. Garrity decomposes each hexahedral cell into five tetrahedra (see Figure 15) and performs ray-plane intersections for his incremental traversal.

4.2.1. Computational Space Traversal

Frühauf [Frü94] traverses a curvilinear grid in computational space. This greatly reduces the difficulties of resampling along the ray since the computational space is regular. First, the Jacobian is approximated using the central differences of the adjacent vertices in computational and physical space. Each vertex or vector can then be transformed from computational into physical space and backwards using the Jacobian matrix.

An incremental grid-traversal for regular grids is then employed to resample the scalar values. At each face intersection of a grid cell, the ray is bend according to the precomputed vectors, using bilinear interpolation. To reduce the computational costs, all computed bending vectors along a ray are stored and updated only when changing the viewpoint. Hence, if the mapping parameters, e.g. transfer function, changes, the pre-computed ray paths can be used directly. Unfortunately no performance measurements were reported.

4.2.2. Ray-Plane-based Traversal

Ma [Ma95] was one of the first exploiting the parallelism of ray tracing for rendering tetrahedral data. The data set as well as the rendering is distributed among processing nodes. In a pre-processing step, the volume is partitioned such that each processor handles only a sub-volume. Boundary faces are projected orthographically onto the screen to determine

© The Eurographics Association 2006.

the first cell along a ray. Exit faces are determined as described in [Gar90]. All processors accumulate the interpolated values along a ray independently. These contributions need to be sorted with respect to the visibility order before the final image can be composed. Ma showed that the frame rate scales linearly with the number of processors.

4.2.3. Grid-based Traversal

To our knowledge Parker et al. [PPL*99] showed one of the first methods that was able to render irregular volume data at interactive rates on a supercomputer. The hierarchical data structure for rectilinear grids is used to quickly determine the tetrahedra possibly hit by the traversed grid cells. All tetrahedra intersecting a given grid cell are attached to this cell, as depicted in Figure 16. Here a multilevel grid with three layers was used. Minimum and maximum values are again helpful to cull regions outside of the current iso-value. Since no connectivity information is used, all tetrahedra assigned to grid leafs needs to be checked, like in a traditional surface ray tracer. First, the barycentric coordinates from the intersected faces, i.e. triangles are determined for interpolating the scalar value. The intersection with the iso-surface can then be computed using linear interpolation.

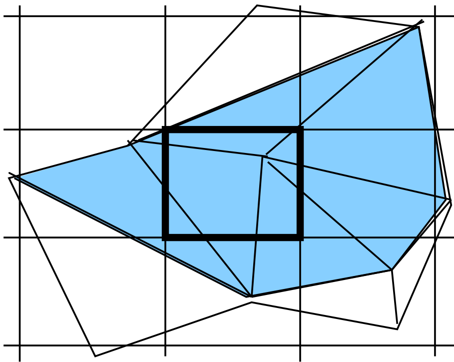

Figure 16: *Parker [PPL*99] sort the tetrahedral mesh into a grid. Once a suitable grid-cell is determined, all tetrahedra intersecting the grid-cell are tested sequentially.*

Results

The rendered bioelectric field consists of over one million tetrahedra. Using a 512x512 view-port, interactive rendering rendering was possible with 16 processors or more. Curvilinear volumes are not supported by this system although possible when decomposing each hexahedron into five tetrahedra as demonstrated in Figure 15. Parker demonstrated high-quality volume rendering of large data sets on supercomputer. The iso-surface was not extracted but computed on the fly and could hence be changed during rendering. The measured performances showed again that a ray tracing system scales well with the number of processors.

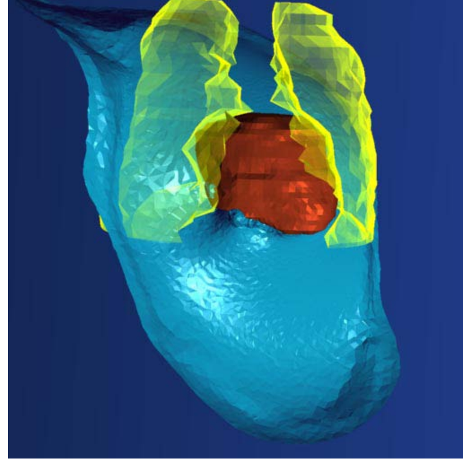

Figure 17: *Ray tracing of over 1 million tetrahedra from bioelectric field simulation. Heart and lungs are represented as polygonal mesh for orientation [PPL*99].*

4.2.4. Plücker Space Traversal

An optimized traversal algorithm suitable for unstructured and curvilinear data sets was presented by Marmitt et al. [MS06]. The volume is traversed using so-called Plücker tests after finding the initial cell using a kd-tree. Basically, the Plücker coordinates allow for easy determination whether an *oriented* line passes clockwise or counter-clockwise around another *oriented* line.

The exiting face of a cell can be determined by testing all edges with the traversing ray in Plücker coordinates. Using connectivity information, the next cell can be processed in the same way until the last cell along a ray is reached. Since this test is line based, it can be used for arbitrary polygons and hence especially for triangles and quadrilaterals, as the faces of tetrahedra or hexahedra respectively.

Properties of Plücker Space

Plücker coordinates are a way of specifying directed lines in three-dimensional space [Eri97]. Plücker coordinates π_r represent a ray $R(t) = O + D * t$ by an *oriented line*:

$$\pi_r = \{d : d \times o\} = \{p_r : q_r\} \qquad (4)$$

Then the inner product of Plücker space

$$\pi_r \odot \pi_s = p_r \cdot q_s + q_r \cdot p_s \qquad (5)$$

defines the relative orientation of the two lines r and s. A positive result means that r passes s clockwise, while in the negative case r passes s counter-clockwise. If this product is zero both lines intersect each other. Note that this inner product is proportional to the signed volume of a tetrahedron spanned by the origin and direction values of r and s.

© The Eurographics Association 2006.

Furthermore, the Plücker test directly provides the scaled barycentric coordinates of the intersection of a ray with the three edges e_i of a triangle, which is a major advantage to plane intersection-based approaches. Thus to compute barycentric coordinates each Plücker value only needs to be divided by the sum of all three values obtained from the three edges of a triangle:

$$w_i = \pi_r \odot \pi_{e_i} \qquad \text{and} \qquad u_i = w_i / \sum_{i=0}^{3} w_i \qquad (6)$$

Tetrahedral Traversal

Intersecting a single tetrahedron with a ray by converting all lines into Pücker space was already demonstrated by Platis et al. [PT03]. To achieve this, the tetrahedron was decomposed into four faces resp. triangles. In general a ray hits two of the four triangles when intersecting the tetrahedron. In a tetrahedral mesh, all inner tetrahedra are connected and therefore one intersected face is already known when reaching a tetrahedra during traversal.

The advantage of the Plücker space is however, that not all faces need to be tested separately, as in the ray-plane intersection case. Instead we benefit from the fact, that each edge of the tetrahedron is shared by two faces. Since also each face is shared by two tetrahedrons, the computed Plücker tests for the exiting face corresponds to the subsequent entry face and need not to be calculated again. These known edges from the previous exiting face are given by the vertices v_0, v_1, v_2, and v_3 in Figure 18. To determine the exit face, lines

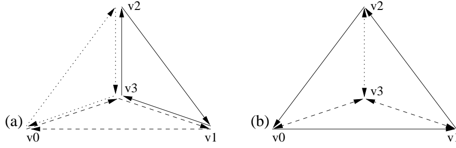

Figure 18: *(a) Naïve approach: All three exiting faces of the triangle are tested independently although each line is shared by two faces, and (b) Optimized approach: The solid lines tests are given from the previous tetrahedron and the dotted line needs only to be computed if the test with the dashed lines failed to provide an unambiguous result.*

$v_0 \longrightarrow v_3$ and $v_1 \longrightarrow v_3$ are tested against the ray which already provides a final result for one third of all cases on average. Otherwise an additional test of the ray against $v_2 \longrightarrow v_3$ finally determines the exit face. With these optimizations the number of tests is reduced to 2.66 in average [MS06].

The tetrahedral mesh itself needs to be enriched with connectivity information that requires 16 additional bytes per tetrahedron. Since all edges as well as the traversing ray are converted into Plücker coordinates on-the-fly, no further memory is consumed. For the final rendering, the interpolated scalar value at the intersection point is needed for visualization. As already shown, the computed Plücker values provide the scaled barycentric coordinates directly. To visualize iso-surfaces it is sufficient to check whether the user-defined value is within the interpolated scalar values computed at the entry and exit face with the barycentric coordinates. Figure 19 shows the Bucky-ball in a polygonal environment with several light sources. For semi-transparent rendering, the values interpolated at the faces can be accumulated directly.

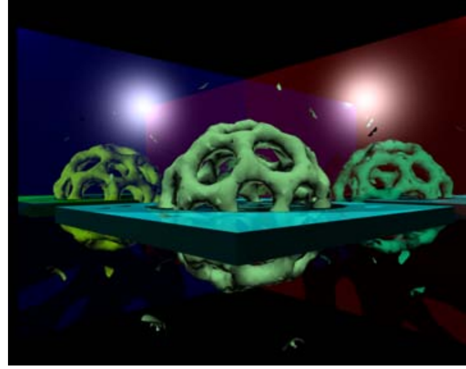

Figure 19: *Seamless integration with surface ray tracing. The volume data set, in this case the Bucky-ball, is augmented and surrounded by reflective surfaces and light sources [MS06].*

Hexahedral Traversal

In [Gar90] it was suggested to decompose each hexahedral of the curvilinear grid into five tetrahedra as depicted in Figure 15. This, however, does not only increase memory consumption but diminishes the speed of the cell traversal itself. The preferable way is therefore to traverse the hexahedral cells directly.

For this purpose Marmitt et. al [MS06] use again the properties of the Plücker space and basically apply the same optimizations as in the tetrahedral case. When optimized for convex hexahedral faces at most four tests per tetrahedron are necessary. The algorithm is illustrated in Figure 20. In this figure the entry face is determined by v_0, v_1, v_2, and v_3. The opposite face is determined by v_4, v_5, v_6, and v_7. In a first step the ray is tested against the edges $v_4 \longrightarrow v_5$ resp. $v_5 \longrightarrow v_7$ (bold horizontal lines in Figure 20). Note that each area contains at most three faces of the hexahedron, i.e. there are only two edges left to check. These are different for each area, e.g. $v_2 \longrightarrow v_6$ and $v_3 \longrightarrow v_7$ for A_0, $v_4 \longrightarrow v_6$ and $v_5 \longrightarrow v_7$ for A_1, and $v_0 \longrightarrow v_4$ and $v_1 \longrightarrow v_5$ for A_2. Testing the second edge is only necessary, if the first test does not lead to a decision. Again simple sign comparisons are sufficient to determine the correct exit face.

Unfortunately, fetching new data is not as efficient as in the tetrahedral case. Analysis revealed that the costs for this operation is seven times higher compared to the tetrahedral

© The Eurographics Association 2006.

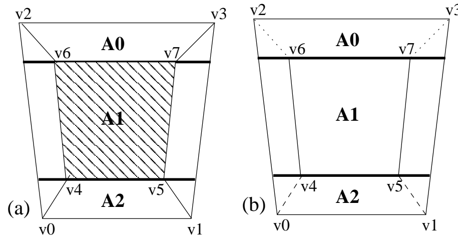

Figure 20: *(a) To determine the exiting face, the hexahedron is subdivided into three areas (A_1 - A_3). (b) The next step is then to check which face is intersected by applying two additional Plücker tests (A_0: dotted edges, A_1: solid edges, and A_2: dashed edges).*

traverser. This is probably caused by re-arranging vertices relative to the entry face. Pücker coordinates represent *oriented lines* and hence the correct processing order needs to be guaranteed. Another problem is that Plücker coordinates cannot be used directly for computing barycentric coordinates, since they are only suitable for triangles. Splitting each face into two triangles decrease the performance and introduce discontinuities.

Instead of using additional Plücker tests within the areas A_0 - A_3 it is also possible to apply three ray-bilinear patch intersections [RPH04]. The correct exiting face is found iff the intersection $(u, v) \in [0, 1]^2$. Since the bilinear patches deliver meaningful parametric coordinates, they can be directly used for interpolation. The result can be ambiguous at shared edges due to numerical issues. Holes or overlaps are introduced in this case, but an additional Plücker test can be applied to uniquely decide this case. Basically, the same types of visualization are possible like in the tetrahedral traversal.

Results

The rendering performance of a 512x512 view-port on a dual-core Opteron with 2 GHz are reported in Table 2 for iso-surface rendering (*iso*), maximum-intensity projection (*mip*), and semi-transparent (*semi*) rendering. Finding the initial face using a kd-tree needs 6 - 18% of the total rendering time (*initial*). More importantly, the hybrid approach (*hex-H*) delivers not only a better quality but is in most cases significantly faster compared to the pure Plücker approach (*hex-P*) when rendering hexahedral meshes. In general, it is suggested to use this hybrid approach for traversing curvilinear grids on a ray tracing basis. Using OpenRT [Wal04] as ray tracing framework, it is also easy to combine and let interact different primitives in one scene and account for all their optical interactions (see Figure 21). OpenRT also supports efficient ray tracing on PC clusters and shared memory systems [MS05] providing almost linear scalability in both cases.

Data set	initial	iso	mip	semi
Blunt-fin (tetra)	27.35	1.67	1.99	1.74
Bucky-ball (tetra)	21.68	0.92	0.95	0.84
Blunt-fin (hex-P)	27.35	1.86	1.35	1.55
Blunt-fin (hex-H)	27.35	2.00	2.16	1.77
Comb (hex-P)	18.43	2.48	0.88	3.14
Comb (hex-H)	18.43	2.43	1.25	3.55

Table 2: *While the performance (fps) of the hexahedral Plücker traverser is sometimes even lower than the tetrahedra Plücker, our hybrid approach outperforms the other two algorithms in every rendering task.*

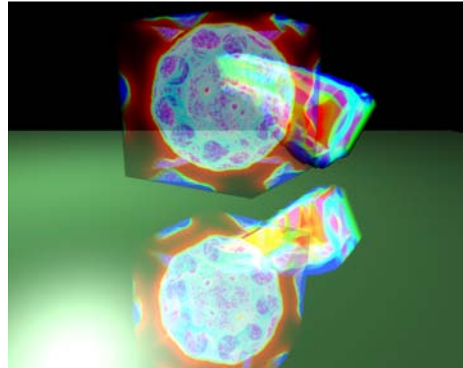

Figure 21: *Unstructured (Bucky-ball) and curvilinear (Combustion Chamber) data sets can not only be rendered into one scene. Even more important is that all primitives interact with each other [MS06].*

4.3. Discussion and Conclusion

Despite the more complicated handling of irregular data sets, the previous discussed algorithms demonstrate interactive rendering of unstructured and curvilinear meshes. Even Parker's [PPL*99] brute-force approach was able to render a large volume with up to 16 fps on a larger shared memory system. Marmitts [MS06] traversal in Plücker space was optimized in many ways allowing for rendering mid-sized models at near-interactive rates on a single dual-core CPU. Both methods benefit from the easy parallelization of ray tracing. Due to the linear scaling with the number processors, future chip generations with four and eight cores or more will certainly enable interactive frame rates on a consumer PC.

Ray tracing is also well known for its easy implementation of advanced rendering effects. The demand for such features is rising steadily, especially for visualization tasks. There can be no doubt, that such advanced rendering effects together with parallelization will increase research activities in the near future.

© The Eurographics Association 2006.

5. Hardware Support

The discussion of previous ray tracing approaches for volume rendering leads naturally to the conclusion, that parallelism will be a key component of future ray tracing systems. Using shared-memory systems [PPL*99] or a cluster of consumer PCs [WFM*05] it is already possible to achieve interactive frame rates. In contrast to consumer graphics boards their flexibility allows for a far easier implementation even for sophisticated rendering effects like global illumination (see Figure 22). The gap between processor speed and memory latency is steadily increasing which is bad for applications like volume rendering since they require an enormous memory bandwidth. Even worse, the size of volumetric data sets is rapidly increasing too. One possibility to reduce the memory bandwidth requirements is parallelism. This parallelism can be exploited on three different levels.

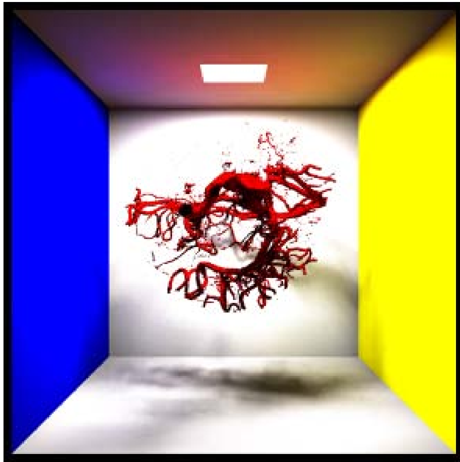

Figure 22: *The aneurysm data set in the Cornell Box including global illumination using the IGI approach of Wald et al. [WKB*02].*

5.1. Single Instruction Multi Data

Single Instruction Multi Data (SIMD) instructions allow to operate on several rays in parallel. Today, SIMD can operate on four 32 bit floating point or integer data at the same time. If data coherency can be exploited, SIMD significantly reduces the instruction count and thus improves performance. Furthermore, SIMD instruction sets allow often to avoid costly conditionals [Kni00] using masking operations.

5.2. Multi-Core CPUs

All major chip manufacturers are currently developing multi-core processors which will certainly work in favor of ray tracing. Preliminary measurements show that the scaling is close to linear.

Volume visualization usually requires a high memory bandwidth to conquer the large amount of data. Therefore such processor architectures need to maintain or even improve the memory connection bandwidth. Recently, IBM realized a processor aiming for both high parallelism and high memory bandwidth. The *Cell* [Int05] processor architecture consists of a single simplified PowerPC that is in charge of controlling eight Synergistic Processor Elements (SPEs) by a fast interconnection bus. Using this bus more than 20 GB/sec of data can be transfered. Each of the SPEs acts as an independent vector processor equipped with 256 KB local store and 128 SIMD registers.

Since SPEs are vector processors they also support a SIMD instruction set similar to AltiVec or SSE. As previously demonstrated, ray traversal and computations within a cell e.g. ray iso-surface intersection test can be significantly speed-up [WFM*05, MFK*04] and it should be relatively straight forward to adapt this technique for the *Cell*.

It needs to be added however, that developers and compilers are faced with new challenges. The SPUs in-order execution makes it necessary to carefully arrange crucial instructions by hand. Additionally, the SPU's local store has no hardware caching support and a software implementation has to be used. Due to high memory latency it is costly to wait for requested data from main memory and thus a hand tuned virtual multi-threading must be exploited to keep maybe several rays – or ray bundles – busy. These and other restrictions demand a more intelligent implementation to exploit the theoretical possible performance.

In general, multi-core processors will enable a series of applications for future scientific visualization. One important issue is the large amount of data produced by scientific simulations or devices. An example is the Richtmyer-Meshkov instability 11 consisting of 200 time steps with 7.5 GB of data per time step.

5.3. GPU Fragment Programs

Finally, the latest graphics boards offer enough flexibility to implement ray casting on the GPU. Especially fragment programs are suitable for such implementations which fully exploits the build-in parallelism of modern graphics boards.

In 2005, Stegmaier et al. [SSKE05] already presented an implementation of a rectilinear volume ray caster on modern consumer graphics boards. A fragment programs simulates ray tracing of individual pixels through the volume. The volume rendering integral is approximated by sampling at a finite number of positions. After the initial intersection with the bounding box is found, subsequent voxels are fetched from a 3D texture map. The opacity and color values accumulated so far are updated according to the chosen optical model. The sampling position is then advanced along the ray by a given step size. Even more parallelism can be exploited by clustering several graphics boards together [MSE06]. The

© The Eurographics Association 2006.

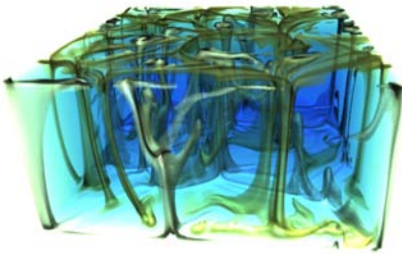

Figure 23: *Simulation of the convection flow in the earths crust including refractive effects [SKB*].*

.

quality of Stegmaiers approach can be enhanced as demonstrated by Strengert [SKB*] by implementing the Kubelka-Munk approach for trecking reflectance and transmittance. Figure 23 shows a rendered image.

However, the programming model as well as the application interface of GPUs is still tedious to use. For example, the number of loops in fragment program is restricted to 256 at the current state. A ray caster therefore has to use nested loops for traversing the volume. Even then, secondary rays for advanced shading can hardly be implemented, since no recursion is available. First steps in porting ray tracing to GPUs have already been taken [PBMH02], but it depends upon future flexibility of graphic boards whether this will be a sufficient basis for allowing for full featured ray tracing.

6. Summary and Conclusions

In this report we presented an overview of ray tracing techniques suitable for volumetric rendering. After briefly recapping the main visualization tasks for volume visualization we grouped all major contributions in two categories. Object-order algorithms and hybrids project the voxel data onto the image plane and thus follow the basic rasterization principle. They can be further classified into cell projection, splatting, and texture mapping. Important alternatives were summarized as alternative approaches including shear-warp and custom hardware. With their increasing flexibility, high memory bandwidth, and excellent floating point performance, graphics boards establish themselves as a serious competitor.

Ray tracing itself has several advantages. Volumes, whether they are regular, curvilinear or even unstructured, are just another primitive for the ray tracer and hence can be easily combined even with polygonal surfaces in a simple plug 'n' play fashion. This was demonstrated by Parker [PPL*99], Wald [WFM*05], and Marmitt [MS06]. Additionally, shadows, reflections, refractions or translucency interact with each other without additional effort. Even advanced shaders, e.g. for global illumination need just to be implemented once and subsequently work for all primitives. The flexibility which is necessary for supporting all of these effects is today only available for software ray tracing systems. Nevertheless, recent developments in custom ray tracing hardware for surface models [WSS05] have shown that programmable ray tracing hardware may become available and might also be extended for volume rendering.

The lack of realtime rendering performance especially for large data sets can be reduced by exploiting coherence and parallelism. Right now it is not clear which hardware architecture will be best suited in future for volume rendering since GPUs and custom hardware become more flexible, and CPUs more parallel.

Scientific visualization becomes more and more important in major research activities today. Natural disasters, understanding weather and climate changes, improving human health, and the simulation of physical processes are just a few key words of an endless list. Such tasks require flexible rendering approaches that are readily provided using ray tracing.

7. Acknowledgements

We would like to thank all the people that contributed to this report, either by comments and suggestions or by making images available: In particular we thank Holger Theisel for discussion and Thomas Ertl, Stephen Parker, Sören Grimm, and Günther Knittel for providing images.

References

[AW87] AMANATIDES J., WOO A.: A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics*. 1987, pp. 3–10. 5

[BCM*01] BENNETT J., COOK R., MAX N., MAY D., WILLIAMS P.: Parallelizing a high accuracy hardware-assisted volume renderer for meshes with arbitrary polyhedra. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics* (2001), pp. 101–106. 4

[BKS97] BUNYK P., KAUFMAN A. E., SILVA C. T.: Simple, fast, and robust ray casting of irregular grids. In *Dagstuhl '97, Scientific Visualization* (1997), pp. 30–36. 4, 14

[CCF94] CABRAL B., CAM N., FORAN J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 Symposium on Volume Visualization* (1994), pp. 91–98. 6

[CRZP04] CHEN W., REN L., ZWICKER M., PFISTER H.: Hardware-Accelerated Adaptive EWA Volume Splatting. In *VIS '04: Proceedings of the conference on Visualization '04* (2004), pp. 67–74. 6

© The Eurographics Association 2006.

[DPH*03] DEMARLE D. E., PARKER S., HARTNER M., GRIBBLE C., HANSEN C.: Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)* (2003), pp. 87–94. 10

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (2001), pp. 9–16. 5, 6

[Eri97] ERICKSON J.: Pluecker Coordinates. *Ray Tracing News* (1997). http://www.acm.org/tog/resources/-RTNews/html/rtnv10n3.html#art11. 15

[Frü94] FRÜHAUF T.: Raycasting of nonregularly structured volume data. In *Proceedings of Eurographics 1994* (1994), pp. 295–303. 14

[Gar90] GARRITY M. P.: Raytracing irregular volume data. In *VVS '90: Proceedings of the 1990 workshop on Volume Visualization* (1990), pp. 35–40. 5, 14, 15, 16

[GBKG04] GRIMM S., BRUCKNER S., KANITSAR A., GRÖLLER M. E.: Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data. In *Proceedings IEEE/SIGGRAPH Symposium on Volume Visualization and Graphics* (Oct. 2004), pp. 1–8. 1, 12

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001. 8

[HJ04] HANSEN C., JOHNSON C. R.: *The Visualization Handbook*. 2004. 3, 4

[HK98] HONG L., KAUFMAN A.: Accelerated ray-casting for curvilinear volumes. In *VIS '98: Proceedings of the conference on Visualization '98* (1998), pp. 247–253. 4, 14

[HK99] HONG L., KAUFMAN A. E.: Fast projection-based ray-casting algorithm for rendering curvilinear volumes. *IEEE Transactions on Visualization and Computer Graphics 5*, 4 (1999), 322–332. 4, 14

[HQK05] HONG W., QIU F., KAUFMAN A.: GPU-based Object-Order Ray-Casting for Large Datasets. In *VG '05: Volume Graphics 2005* (2005), pp. 177–186. 5

[Int05] INTERNATIONAL BUSINESS MACHINES: The Cell Project at IBM Research. http://www.research.ibm.com/cell/, 2005. 18

[KK99] KREEGER K., KAUFMAN A.: Hybrid volume and polygon rendering with cube hardware. In *HWWS '99: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (1999), pp. 15–24. 6, 7

[KN91] KOYAMADA K., NISHIO T.: Volume visualization of 3D finite element method results. *IBM J. Res. Dev. 35*, 1-2 (1991), 12–25. 14

[Kni00] KNITTEL G.: The ULTRAVIS System. In *Proceedings of the 2000 IEEE symposium on Volume visualization* (2000), pp. 71–79. 13, 18

[KS97] KNITTEL G., STRASSER W.: VIZARD - Visualization Accelerator for Realtime Display. In *HWWS '97: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (1997), pp. 139–146. 6, 7

[KW03] KRÜGER J., WESTERMANN R.: Acceleration Techniques for GPU-based Volume Rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), pp. 287–292. 6

[LCN98] LICHTENBELT B., CRANE R., NAQVI S.: *Introduction to Volume Rendering*. Person Education, 1998. 3

[Lev90] LEVOY M.: Efficient Ray Tracing for Volume Data. *ACM Transactions on Graphics 9*, 3 (July 1990), 245–261. 14

[LL94] LACROUTE P., LEVOY M.: Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (1994), pp. 451–458. 6

[LMK03] LI W., MÜLLER K., KAUFMAN A.: Empty space skipping and occlusion clipping for texture-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), pp. 317–324. 6

[Luc92] LUCAS B.: A scientific visualization renderer. In *VIS '92: Proceedings of the 3rd conference on Visualization '92* (1992), pp. 227–234. 4

[LYTM06] LAUTERBACH C., YOON S.-E., TUFT D., MANOCHA D.: Interactive Ray Tracing of Dynamic Scenes using BVHs. *Technical Report, University of North Carolina at Chapel Hill* (2006). 11

[Ma95] MA K.-L.: Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures. In *PRS '95: Proceedings of the IEEE symposium on Parallel rendering* (1995), pp. 23–30. 14

[MC97] MA K.-L., CROCKETT T. W.: A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *PRS '97: Proceedings of the IEEE symposium on Parallel rendering* (1997), pp. 95–104. 4

[MC98] MÜLLER K., CRAWFIS R.: Eliminating popping artifacts in sheet buffer-based splatting. In *VIS '98: Proceedings of the conference on Visualization '98* (1998), pp. 239–245. 6

[ME05] MORA B., EBERT D. S.: Low-complexity maximum intensity projection. *ACM Transactions on Graphics 24*, 4 (2005), 1392–1416. 13

© The Eurographics Association 2006.

[MFK*04] MARMITT G., FRIEDRICH H., KLEER A., WALD I., SLUSALLEK P.: Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)* (2004), pp. 429–435. 7, 11, 18

[MFS05] MARMITT G., FRIEDRICH H., SLUSALLEK P.: Recent Advancements in Ray-Tracing based Volume Rendering Techniques. In *Vision, Modeling, and Visualization (VMV) 2005* (November 2005), pp. 131–138. 1

[MHB*00] MEISSNER M., HUANG J., BARTZ D., MUELLER K., CRAWFIS R.: A practical evaluation of popular volume rendering algorithms. In *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization* (2000), pp. 81–90. 1

[MJC02] MORA B., JESSEL J. P., CAUBET R.: A new object-order ray-casting algorithm. In *VIS '02: Proceedings of the conference on Visualization '02* (2002), pp. 203–210. 5, 13

[MKW*02] MEISSNER M., KANUS U., WETEKAM G., HIRCHE J., EHLERT A., STRASSER W., DOGGETT M., FORTHMANN P., PROKSA R.: VIZARD II: A Reconfigurable Interactive Volume Rendering System. In *Eurographics/ACM SIGGRAPH Workshop on Graphics Hardware* (2002), pp. 137–146. 7

[MMMY97] MOLLER T., MACHIRAJU R., MUELLER K., YAGEL R.: A comparison of normal estimation schemes. In *IEEE Visualization* (1997), pp. 19–26. 12

[MS05] MARMITT G., SLUSALLEK P.: *Fast Ray Traversal of Unstructured Volume Data using Plucker Tests*. Tech. rep., Saarland University, 2005. Available at http://graphics.cs.uni-sb.de/Publications. 17

[MS06] MARMITT G., SLUSALLEK P.: Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering. In *Proceedings of Eurographics/IEEE-VGTC Symposium on Visualization (EuroVIS) '06* (2006), pp. 235–242. 14, 15, 16, 17, 19

[MSE06] MÜLLER C., STRENGERT M., ERTL T.: Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV06)* (2006), pp. 59–66. 18

[MY96] MÜLLER K., YAGEL R.: Fast perspective volume rendering with splatting by utilizing a ray-driven approach. In *VIS '96: Proceedings of the 7th conference on Visualization '96* (1996), pp. 65–72. 5

[NMHW02] NEUBAUER A., MROZ L., HAUSER H., WEGENKITTL R.: Cell-based first-hit ray casting. In *Proceedings of the Symposium on Data Visualisation 2002* (2002), pp. 77–86. 5, 7

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray Tracing on Programmable Graphics Hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH) 21*, 3 (2002), 703–712. 19

[PH04] PHARR M., HUMPHREYS G.: *Physically Based Rendering : From Theory to Implementation*. Morgan Kaufman, 2004. 1

[PHK*99] PFISTER H., HARDENBERGH J., KNITTEL J., LAUER H., SEILER L.: The VolumePro real-time ray-casting system. *Computer Graphics 33*, Annual Conference Series (1999), 251–260. 6, 7

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P.-P., HANSEN C., SHIRLEY P.: Interactive Ray Tracing for Volume Visualization. *IEEE Transactions on Computer Graphics and Visualization 5*, 3 (1999), 238–250. 10, 13, 14, 15, 17, 18, 19

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization* (October 1998), pp. 233–238. 7, 10

[PSL*99] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics* (1999), pp. 119–126. 10

[PT03] PLATIS N., THEOHARIS T.: Fast Ray-Tetrahedron Intersection Using Plücker Coordinates. *Journal of graphics tools 8*, 4 (2003), 37–48. 16

[RGW*03] RÖTTGER S., GUTHE S., WEISKOPF D., ERTL T., STRASSER W.: Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003* (2003), pp. 231–238. 6

[RKE00] RÖTTGER S., KRAUS M., ERTL T.: Hardware-accelerated volume and isosurface rendering based on cell-projection. In *VIS '00: Proceedings of the conference on Visualization '00* (2000), pp. 109–116. 5, 6

[RPH04] RAMSEY S., POTTER K., HANSEN C.: Ray Bilinear Patch Intersections. *Journal of Graphics Tools 9*, 3 (2004), 41–47. 17

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics 24*, 3 (2005), 1176–1185. (Proceedings of ACM SIGGRAPH). 1, 11

[RZNS04] RÖSSL C., ZEILFELDER F., NÜRNBERGER G., SEIDEL H.-P.: Reconstruction of Volume Data with Quadratic Super Splines. *IEEE Transactions on Visualization and Computer Graphics 10*, 4 (2004), 397–409. 8, 11

[Sch90] SCHWARZE J.: Cubic and Quartic Roots. In *Graphics Gems*, Glassner A., (Ed.). Academic Press, 1990, pp. 404–407. 7

[SK04] SARANG L., KAUFMAN A.: Light weight space leaping using ray coherence. In *VIS '04: Proceedings of the conference on Visualization '04* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 19–26. 12

[SKB*] STRENGERT M., KLEIN T., BOTCHEN R.,

© The Eurographics Association 2006.

STEGMAEIER S., CHEN M., ERTL T.: Spectral Volume Rendering using GPU-based Raycasting. *The Visual Computer (to appear)*. 19

[SM00] SCHUMANN H., MUELLER W.: *Visualisierung - Grundlagen und allgemeine Methoden*. Springer, 2000. 3, 4

[SM02] SWEENEY J., MUELLER K.: Shear-warp deluxe: the shear-warp algorithm revisited. In *VISSYM '02: Proceedings of the symposium on Data Visualisation 2002* (2002), pp. 95–104. 6

[SSKE05] STEGMAIER S., STRENGERT M., KLEIN T., ERTL T.: A Simple and Flexible Volume Rendering Framework for Graphics-Hardware–based Raycasting. In *Proceedings of the International Workshop on Volume Graphics '05* (2005), pp. 187–195. 18

[ST90] SHIRLEY P., TUCHMAN A.: A polygonal approximation to direct scalar volume rendering. *ACM Computer Graphics (Proceedings San Diego Workshop on Volume Visualization 1990 24*, 5 (1990), 63–70. 4, 5

[The01] THEISEL H.: CAGD and Scientific Visualization, Habilitation thesis, 2001. 2

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. 10, 17

[WBLS03] WU Y., BHATIA V., LAUER H., SEILER L.: Shear-image order ray casting volume rendering. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics* (2003), pp. 152–162. 7

[WBS02] WALD I., BENTHIN C., SLUSALLEK P.: *OpenRT - A Flexible and Scalable Rendering Engine for Interactive 3D Graphics*. Tech. rep., Saarland University, 2002. Available at http://graphics.cs.uni-sb.de/Publications. 1

[WBS06] WALD I., BOULOS S., SHIRLEY P.: Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies (revised version). *Technical Report, SCI Institute, University of Utah, No UUSCI-2006-023 (conditionally accepted at ACM Transactions on Graphics)* (2006). 1, 11

[WCA*90] WILHELMS J., CHALLINGER J., ALPER N., RAMAMOORTHY S., VAZIRI A.: Direct volume rendering of curvilinear volumes. In *Computer Graphics (San Diego Workshop on Volume Visualization)* (1990), pp. 41–47. 2, 14

[WDS04] WALD I., DIETRICH A., SLUSALLEK P.: An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering* (2004), pp. 81–92. 11

[Wes90] WESTOVER L.: Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (1990), pp. 367–376. 5

[WFM*05] WALD I., FRIEDRICH H., MARMITT G., SLUSALLEK P., SEIDEL H.-P.: Faster Isosurface Ray Tracing using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics 11*, 5 (2005), 562–573. 5, 8, 9, 18, 19

[WGTG96] WILHELMS J., GELDER A. V., TARANTINO P., GIBBS J.: Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. In *VIS '96: Proceedings of the 7th conference on Visualization '96* (1996), pp. 57–64. 4

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM SIGGRAPH 2006* (2006). 1, 11

[WKB*02] WALD I., KOLLIG T., BENTHIN C., KELLER A., LEK P. S.: Interactive Global Illumination using Fast Ray Tracing. In *Rendering Techniques 2002* (Pisa, Italy, June 2002), Debevec P., Gibson S., (Eds.), Eurographics Association, Eurographics, pp. 15–24. (Proceedings of the 13th Eurographics Workshop on Rendering). 9, 18

[WKME03] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-based ray casting for tetrahedral meshes. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (2003), pp. 333–340. 5

[WMKE04] WEILER M., MALLON P. N., KRAUS M., ERTL T.: Texture-encoded tetrahedral strips. In *VV '04: Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics (VV'04)* (2004), pp. 71–78. 5

[WMS98] WILLIAMS P. L., MAX N. L., STEIN C. M.: A High Accuracy Volume Renderer for Unstructured Data. *IEEE Transactions on Visualization and Computer Graphics 4*, 1 (1998), 37–54. 4

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics). 9

[WSS05] WOOP S., SCHMITTLER J., SLUSALLEK P.: RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *Proceedings of ACM SIGGRAPH* (2005). 19

[WV92] WILHELMS J., VAN GELDER A.: Octrees for faster isosurface generation. *ACM Transactions on Graphics 11*, 3 (July 1992), 201–227. 5, 9, 11

[ZPvBG01] ZWICKER M., PFISTER H., VAN BAAR J., GROSS M.: EWA volume splatting. In *VIS '01: Proceedings of the conference on Visualization '01* (2001), pp. 29–36. 6

© The Eurographics Association 2006.