Eurographics/ IEEE-VGTC Symposium on Visualization (2006)
Thomas Ertl, Ken Joy, and Beatriz Santos (Editors)

Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering

Gerd Marmitt and Philipp Slusallek†

Computer Graphics Group, Saarland University, Germany

Abstract

The importance of high-performance rendering of unstructured or curvilinear data sets has increased significantly, mainly due to its use in scientific simulations such as computational fluid dynamics and finite element computations. However, the unstructured nature of these data sets lead to rather slow implementations for ray tracing. The approaches discussed in this paper are fast and scalable towards realtime ray tracing applications.
We evaluate new algorithms for rendering tetrahedral and hexahedral meshes. In each algorithm, the first cell along a ray is found using common realtime ray tracing techniques. For traversing subsequent cells within the volume, Plücker coordinates as well as ray-bilinear patch intersection tests are used. Since the volume is rendered directly, all algorithms are applicable for isosurface rendering, maximum-intensity projection, and emission-absorption models.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism I.4.10 [Image Processing and Computer Vision]: Image Representation

1. Introduction

In the past several years the demand for volume rendering of scalar data sets has increased steadily with a focus on structured volume data, often generated by CT and NMR devices. This data is usually organized in rectilinear or even regular grids and can be used more or less directly for rendering.

On the other hand, numerical simulations often operate on curvilinear or even unstructured domains. Fully unstructured volume data is often organized by imposing a tetrahedral topology between adjacent sample points in 3D. In the same way, curvilinear grids can be converted into a tetrahedral mesh. One common technique for visualization is isosurface extraction [LC87], which was adapted to tetrahedral primitives [DK91]. However, in this paper we want to investigate algorithms suitable for direct volume rendering. This allows not only the use of emission-absorption models but also isosurface rendering and maximum-intensity projection.

Rendering volumetric data is still a demanding task mainly due to the handling and processing of huge amounts of data. This is especially true for curvilinear or even unstructured volumes, which additionally cannot take advantage of the inherent structure and the predictable access patterns of regular grids. In this paper, we present therefore two advanced traversal algorithms based on ray tracing. They allow for quickly locating the entry point into the volume data set while using efficient algorithms based on Plücker tests and bilinear patch intersections to traverse the set of connected tetrahedra respectively hexahedra along a ray.

The remainder of this paper is organized as follows. We first give a brief overview of related research on rendering unstructured volume data (Section 2). Section 3 presents the algorithmic background used in the following Sections. In Section 4, we cover tetrahedral meshes and show how to traverse them in Plücker space, while Section 5 evaluates Plücker coordinates and bilinear patches for the traversal of curvilinear data. In Section 6, we report the results of using these approaches for direct volume rendering. Conclusion and ideas for future work are discussed in Sections 7 and 8.

† e-mail: {marmitt, slusallek}@cs.uni-sb.de

2. Previous Work

Rendering isosurfaces by extracting the implicit surface from a tetrahedral mesh [DK91] is the most obvious approach. Recently it became popular to use consumer graphic cards as the main tool for isosurface rendering [Pas04, KSE04, KW05].

However, we want to traverse the unstructured or curvilinear volume data directly. Most of the suggested algorithms for this task can be divided into two groups: object space or image space methods. The former ones are usually referred as projective methods. Hybrid algorithms use a combination of both approaches [HK99, BKS99].

Many researchers have developed projective algorithms [Luc92, WG91]. One drawback of purely projective methods was the lack of an efficient early termination, i.e. stopping traversal once the opacity value reached 1.0. Wang et al. [WSW05] proposed therefore storing a dynamic ray-cast like traversal link which stops if the opacity reaches 1.0. Since this link has to be rebuild whenever the viewpoint changes, a performance gain requires many occluded cells.

Another approach is the implementation of a hybrid ray casting/projective system. Hong et al. [HK99] calculate the initial face by scan-converting all boundary faces on the image plane and apply depth-sort along the given viewing ray. Each cell is decomposed into 12 triangles, which are projected onto the view-port for traversal. Bunyk et al. [BKS99] extend this method with a modified z-buffer approach and a view-independent preprocessing step. However, the projection step introduces additional accuracy problems.

On the other hand, we have methods solely based on ray casting. Frühauf [Frü94] presents such a method for curvilinear volume data. The volume is traversed in computational space which makes a two-step processing necessary. Whenever the viewpoint changes, a redirection vector needs to be recomputed for all nodes within the grid. Forsell [For94] adapts this method to Line Integral Convolution (LIC). This method generates a texture of the grid size that can be convolved with a filter kernel and the texture pixel indicated by the vector field.

Weiler et al. [WKME03] implemented an efficient ray-caster for tetrahedral meshes on a consumer graphics card. The initial tetrahedron is found by rasterizing the extracted boundary faces of a given model. For their ray-casting approach, a ray-plane intersection is used. However, this precomputation leads to a total memory requirement of 160 bytes per tetrahedron, and hence the size of the model is restricted to 600,000 tetrahedra on the card used. Additionally, a convexification needs to be performed as suggested by Williams et al. [WM92], which further increases the number of tetrahedrons of the data set.

Additionally, it is still hard to extend GPU ray casters for combining different rendering primitives in one scene with full interaction. Ray tracing, on the other hand, lets different rendering primitives interact with each other including reflection and refraction. It was always considered as too slow to use for interactive rendering tasks [Lev90, Gar90]. Recently Parker [PPL*99] and Wald [WFM*05] showed, that this is no longer true.

Unfortunately, Parker's [PPL*99] approach was restricted to a supercomputer for achieving practical rendering times, while Walds [WFM*05] algorithm is restricted to rectilinear isosurface rendering. In this paper we therefore investigate ray tracing algorithms for rendering curvilinear and unstructured volume data for interactive purposes. Especially for the tetrahedral mesh traverser we are using the results by Marmitt et al. [MS05] and their proposal for traversing curvilinear grids [MFS05]. All algorithms are suitable for both, isosurface and emission-absorption volume rendering.

3. Algorithmic Background

Before we continue with our evaluation of interactive volume traversal algorithms, we first provide some algorithmic background. In particular we discuss properties of the so-called Plücker space.

Plücker coordinates are a way of specifying directed lines in three-dimensional space [Eri97]. Basically, these coordinates represent a ray by an *oriented line*. Suppose a ray $r(t) = o + dt$ is given, this results in the following six-vector:

$$\pi_r = \{d : d \times o\} = \{p_r : q_r\} \tag{1}$$

In particular we use this representation to determine whether an oriented line passes clockwise or counter-clockwise around another oriented line. This information is simply given by the *permuted inner product* of their Plücker coordinates, which is rather easy to compute. Given two lines in Plücker space r and s this results in:

$$\pi_r \odot \pi_s = p_r \cdot q_s + q_r \cdot p_s \tag{2}$$

Note that this is also the volume of a tetrahedron spanned by the 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 (see Figure 1). Care has to be taken about the direction, i.e. the Plücker value for a line $r \rightarrow s$ differ from $s \rightarrow r$: the sign of the permuted inner product changes.

4. Tetrahedral Meshes

One way to organize unstructured volume data is to add an explicit structure like a tetrahedral mesh. This can be achieved e.g. by Delaunay-tetrahedralization. As Platis et al. [PT03] showed, Plücker coordinates can be used for a fast computation of the intersection point.

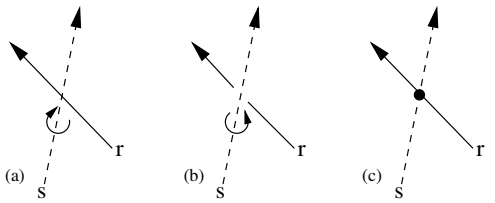

Figure 1: *Three possible cases for the Plücker test: (a) ray r passes clockwise line s, (b) ray r passes counter-clockwise line s, and (c) ray r intersects line s.*

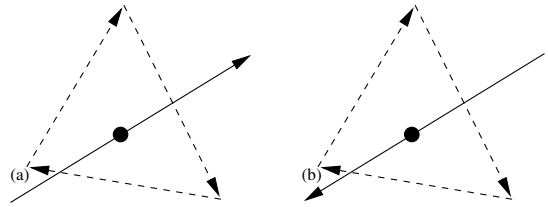

Figure 2: *Two different configurations with the same result: (a) the ray passes all line segments of the triangle clockwise, hence all signs are positive, and (b) the ray passes all line segments counter-clockwise, and therefore all signs are negative.*

In the following we describe how to use the Plücker space to traverse a tetrahedral mesh. In a first step we need to identify the initial tetrahedron along a ray. This can be found using a well-known acceleration structure for ray tracing, i.e. kd-trees are employed on the boundary faces of the tetrahedral mesh. Once the first tetrahedron is known, all subsequent tetrahedra are traversed using Plücker coordinates determining the tetrahedra traversed along a given ray. In each cell either isosurface rendering or an emission-absorption model can then be applied.

We extend and accelerate an approach suggested by Garrity [Gar90]. He finds the nearest tetrahedron along a ray using a grid as acceleration structure. Connectivity information between adjacent tetrahedra is then used to traverse the cells incrementally by calculating plane intersections.

4.1. Finding the Initial Tetrahedron

To find the initial tetrahedron along a ray, we first extract all outer faces during a preprocessing step. It is easy to see, that this is the set of all tetrahedral faces not shared with another tetrahedra in the set. A kd-tree can then be used as acceleration structure, which has been proven as a fast and efficient technique when using ray tracing [Hav01]. Especially we used the implementation suggested by Wald et al. [Wal04]. The resulting triangle provides the tetrahedron as well as its entry face so that we can continue in Plücker space.

4.2. Mesh Traversal in Plücker Space

Here, we extend the ray-tetrahedron intersection algorithm introduced by Platis et al. [PT03] for the traversal of tetrahedral meshes. They especially used the fact that each tetrahedron can be decomposed into four triangles. The entry face is already known due to the shared-face property. To find the exiting face, we check each triangle whether it is intersected by the ray. This can be achieved by converting all three edges and the intersecting ray into Plücker space. Using the properties described in Section 3, a ray intersects the triangle if and only if all results have the same sign. The only condition is that Plücker coordinates are either clockwise or counter-clockwise calculated. Figure 2 illustrates the basic idea.

A naïve approach would be now to check all three possible exit faces separately. This would lead to three to nine tests. As an optimization we can make use of the property that each edge is shared by two faces we already know. Hence, we can reuse the Plücker tests from the exit face, i.e. we first test the lines $v_0 \rightarrow v_3$ and $v_1 \rightarrow v_3$ against our ray. If the sign of these two tests differs, we also have to check $v_2 \rightarrow v_3$. An important premise is, that all interior faces are shared by exact two tetrahedra. Hence, we do not allow sliding interfaces, i.e. tetrahedra that share only part of their faces. Figure 3 illustrates both methods.

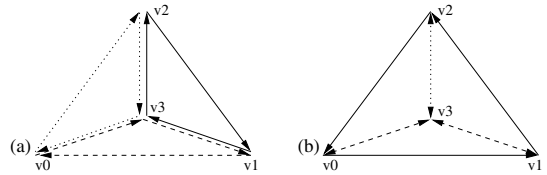

Figure 3: *(a) Naïve approach: All three exiting faces of the triangle are tested independently although each line is shared by two faces, and (b) Optimized approach: The solid lines tests are given from the previous tetrahedron and the dotted line needs only to be computed if the test with the dashed lines failed. The direction of each line can be swapped by turning the sign of the Plücker test.*

Applying the above mentioned optimizations, the number of tests drops to 2.67 on average. Also, the raw performance of tetrahedra processed per second raises by 55% on average compared to the naïve approach. Another advantage is that only one vertex coordinate, i.e. v_3, and the corresponding scalar value have to be fetched per tetrahedron. This keeps the memory bandwidth low and improves the overall performance due to better cache usage. Once the exit face is identified, we need connectivity information to get to the neighboring tetrahedron. This requires 16 additional bytes per tetrahedron.

The baseline performance for determining the exit face including the barycentric coordinates and the distance of

the hit point reaches 15.5 million tetrahedra per second on our test system (Dualcore-Opteron with 2GHz, both CPUs used). Note that the conversion into Plücker space was not pre-computed, since this would double the memory requirements.

4.3. Cell Intersection

For all rendering tasks, the interpolated value at the entry/exit points has to be computed first. Plücker tests provide directly the scaled barycentric coordinates, which is a major advantage to plane intersection-based approaches. Thus each Plücker value needs only to be divided by the sum of all three values belonging to that face:

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

For the emission-absorption model these values can either be directly used for accumulation, or additional super-sampling is applied. To find the implicit isosurface, we have to interpolate between the entry and exit face. Of course, the isovalue has to be within the range of the interpolated values at these faces. A user may even modify this isovalue during the rendering task.

Given a tetrahedron with its vertices v_i and corresponding scalar values $s_i (i \in \{0,1,2,3\})$, we calculate this linear function by solving the system of four equations:

$$s_i = ax_i + by_i + cz_i + d \tag{4}$$

with $v_i = (x_i, y_i, z_i)$ for the unknowns a, b, c and d. The intersection is then found by substituting the ray equation into (4) and setting s_i to the chosen iso-value. Figure 4 illustrates the results for the described rendering modes as well as applying a transfer function to highlight regions of interest.

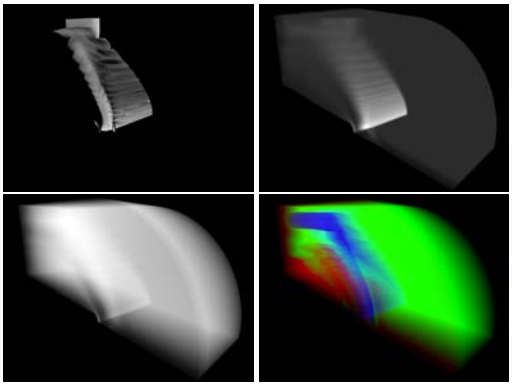

Figure 4: *The tetrahedral Blunt-fin data set rendered as an iso-surface (upper-left), maximum-intensity-projection (upper-right), direct volume rendering (lower-left) with transfer functions (lower-right).*

4.4. Normal Calculation

From the previous observations follows directly that the orientation of the plane describing the iso-surface is independent from the iso-value. Hence the plane normal $N_t = (a\ b\ c)^T$ is constant within the cell. Unfortunately, this results in severe discontinuities between the tetrahedra surfaces, which decreases the rendering quality significantly. For better results, one can calculate a normal N_v per vertex of the tetrahedron. This can be expressed as the sum of all n tetrahedra h_t connected to this vertex weighted with their volume $V(h_t)$:

$$N_v(a\ b\ c)^T = \sum_{t=0}^{n} N_t(a\ b\ c)^T * V(h_t) \tag{5}$$

These vertex normals can be pre-computed and stored into the data file. In a final step, four barycentric coordinates of the intersection point within the tetrahedra are calculated and weighted with the corresponding vertex normal to obtain a smooth isosurface normal. To achieve this, the 2D barycentric coordinates for triangles are extended for the 3D (i.e. tetrahedral) case. Finally, the vector resulting from equation 5 needs itself to be normalized.

5. Hexahedral Meshes

Hexahedral meshes are better known as curvilinear data sets. The topology of data points is still a grid, but the space between two neighboring points varies. Ideally, a bijective function is supplied, which allows to convert points from the physical space into computational space and vice versa. In reality, this function is often not available. Therefore we assume here, that only scalar values together with the 3D mesh vertices are supplied.

Again, the initial hexahedron along a ray is found using the same kd-tree as described in Section 4.1. For this purpose we extract all boundary faces of the data set in a preprocessing step, and decompose each resulting quadrilateral into two triangles. This makes the kd-tree traversal as efficient as for the tetrahedral traversal.

5.1. Mesh Traversal

Two approaches are evaluated in the following subsections. First, it is easy to see, that the Plücker test introduced in Section 4.2 can be extended for traversing hexahedral grids. However, each face needs to be decomposed into two triangles for calculating the coordinates for the interpolation. This leads to parametric discontinuities along this additional diagonal, but allows to determine the intersected faces unambiguously. The second method extends this approach using bilinear patches. Here, no singularities can occur, but holes within a cell are introduced.

There is no extra memory for connectivity required as the topology is implicit in the grid structure. Once the exit

face has been determined, we only have to modify the index pointer by incrementing respectively decrementing with respect to the dimension. If our pointer exits the volume bounds, the traversal stops.

5.1.1. Plücker Space

Applying the same optimizations as for the tetrahedra-traversal from Section 4.2 leads to at most four tests to decide which one of the five possible faces is the exit face. This algorithm is optimized for convex hexahedral faces but can be easily extended to handle concave faces too. Figure 5 illustrates our algorithm. Suppose that the entry face is given by the vertices v_0, v_1, v_2, and v_3. The opposite face shall be determined by v_4, v_5, v_6, and v_7.

In a first step, we perform the Plücker test with edges $v_4 \rightarrow v_5$ and $v_6 \rightarrow v_7$. This results in three areas A_0, A_1, and A_2. Note that each area contains exactly three faces of the hexahedron, i.e. there are only three faces left to check. To do this we have to treat each area differently, e.g. we need to test the edges $v_2 \rightarrow v_6$ and $v_3 \rightarrow v_7$ for A_0, $v_4 \rightarrow v_6$ and $v_5 \rightarrow v_7$ for A_1, and $v_0 \rightarrow v_4$ and $v_1 \rightarrow v_5$ for A_2. The second test only needs to be performed if the first does not lead to a decision. This second step is illustrated in Figure 5(b). Now simple sign comparisons are sufficient to determine the correct exit face.

Fetching the data for the next cell and saving the appropriate data for reusing needs more time compared to the tetrahedral mesh. Our analysis showed that the costs for these operations are seven times higher compared to our tetrahedral traverser, where only one vertex and scalar value needs to be fetched in each step. Unfortunately it is not possible to

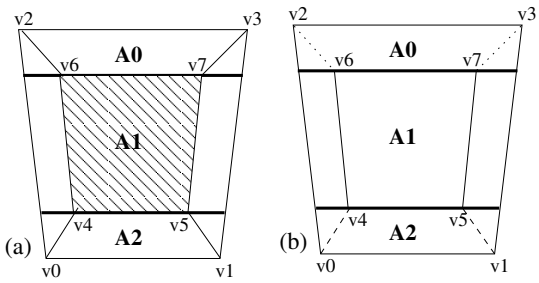

Figure 5: *(a) To determine the exiting face, the hexahedron is subdivided into three areas. (b) The next step is then to check which face is intersected by applying two additional Plücker tests (A_0: dotted edges, A_1: solid edges, and A_2: dashed edges).*

just scale the computed Plücker values to receive the parametric coordinates directly, since they are only suitable for triangles. This makes it necessary to split each face into two triangles so that we again can compute the barycentric coordinates. Depending on the previously computed Plücker tests, one or two additional tests are required, which reduces the performance significantly. Furthermore, decomposing a face into two triangles leads to unwanted discontinuities if the values of the hexahedra are badly distributed (see Figure 6). The number of hexadedra processed per second, including the calculation of (u,v), as well as the distance to the intersection point reaches 3.2 million on our test system (Dualcore-Opteron 2 GHz, both CPUs used). Again, all line segments are converted on-the-fly into Plücker space during traversal.

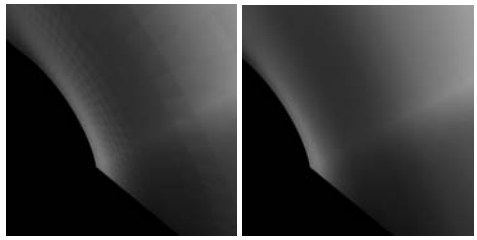

Figure 6: *Blunt-fin: Decomposing produces artifacts (left), while the bilinear patch delivers smooth results (right).*

5.1.2. Bilinear Patch-Extension

To avoid these singularities, we found bilinear patches to be an interesting alternative. Note that it is even possible to traverse the curvilinear data set directly by applying this test to each of the five possible exit faces and stop as soon as a patch is hit, i.e. $(u,v) \in [0,1]^2$. However, in some configurations this is not possible since bilinear patches are generally not flat, i.e. we introduce holes in our hexahedron. These holes are caused by the implicit edge representation of bilinear patches. In particular, an edge of the bilinear patch depends on all four face vertices. In contrast, Plücker coordinates allow a unique decision whether a ray passes clockwise or counter-clockwise since they depend solely on the two edge points. Hence, we need to handle this inconsistency so that a ray can traverse the correct cells within a volume.

We found that a combination of Plücker and bilinear patch tests works best. Hence, the first step is again to compute the Plücker values of the lines $v_4 \rightarrow v_5$, $v_6 \rightarrow v_7$, and the ray, which results again in the three areas A_0, A_1, and A_2 (see Figure 5). In contrast to the previous method, we check all three possible exit faces using the bilinear patch intersection. As a result, raw traversal performance is 15% lower compared to the pure Plücker-traverser. However, we save time in the overall performance, since the surface parameters (u,v) are already known at this point. A fast and robust implementation was described by Ramsey et al. [SDRH04] which was also implemented for this paper.

In case of inconsistencies due to numerical issues (i.e. insufficient floating point accuracy), we perform two additional Plücker tests equivalent to the second step described

in Section 5.1.1. These numerical issues depend on the geometric average unit of the cells and are therefore data dependent. However, this hardly occurred in our two tested models and leads therefore to a performance gain of 25% on average. In all, this algorithm is able to process up to 4.5 million hexahedra per second on our system (Dualcore-Opteron 2 GHz, both CPUs used).

5.2. Cell Intersection

In addition to the traversal operation we need to interpolate the scalar values at these points for shading the volume. This introduces one or two additional Plücker tests depending on what can be reused from the exit face decision. When using bilinear patches we can make use of the fact, that the parametric coordinates are already known, and we only need to compute the interpolated scalar value on the entry and exit faces. Discontinuities are completely avoided this way. Applying isosurface rendering is now

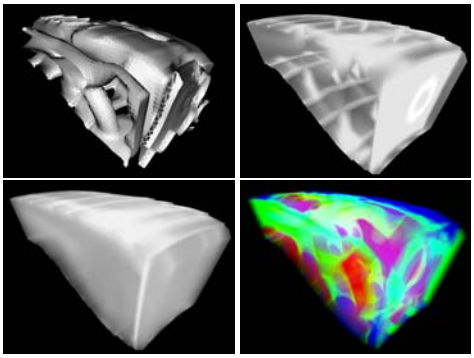

Figure 7: *Combustion chamber data set rendered as isosurface (upper-left), maximum-intensity-projection (upper-right), direct volume rendering (lower-left) with transfer functions (lower-right).*

fairly simple. If the user-defined value is within the values interpolated at the entry and exit faces, an additional linear interpolation determines the intersection with the implicit surface. For emission-absorption models or maximum-intensity-projection we use the interpolated faces values directly. It is of course also possible to super-sample these values in the computational domain to obtain smoother results.

5.3. Normal Calculation

If we assume for a moment, that we know the normal vectors at each of the eight vertices of a hexahedron we can proceed as follows. We reuse the same parametric coordinates as for interpolating the scalar values, except that we interpolate a vector at the entry and exit face. Another linear interpolation between these two vectors based on their distance of the isosurface yields the normal. This corresponds to a kind of trilinear interpolation.

For computing the vertex normals of a hexahedron we adapted a method suggested by Frühauf [Frü94]. The basic idea is to compute the normal like in a regular grid with unit length, i.e. in computational space. The following relationship can thereafter be used to convert between physical coordinates x_i and computational coordinates ξ_j, determined using the Jacobian matrix J_{ij}:

$$x_i = J_{ij} \cdot \xi_j \qquad \text{and} \qquad J_{ij} = \frac{\delta x_i}{\delta \xi_j} \tag{6}$$

Note that we can also use this equation to convert vectors from one space into another. Furthermore, we approximate the unknown Jacobian matrix for a grid node n by using central differences:

$$J_{ij} = \frac{1}{2} \cdot \left(\frac{x_i(n) - x_i(n-1)}{\xi_j(n) - \xi_j(n+1)} + \frac{x_i(n+1) - x_i(n)}{\xi_j(n+1) - \xi_j(n)} \right) \tag{7}$$

Hence we can calculate three vectors, since we operate in three dimensions. A linear combination of these vectors with the normal in computational space N^c results in the converted normal N^p.

6. Results

For the evaluation we measured the rendering performance with four common volume data sets. Buckyball and Combustion chamber are available as tetrahedral respectivily hexahedral meshes only. The Blunt-fin on the other hand was available for both types of volume organization and therefore allows for a direct comparison.

For measuring the actual rendering performance we equipped our system with a Dualcore-Opteron 2 GHz and 32 GB main memory running Linux, using both cores in parallel. We decided to measure the frame-rate for three of the most common techniques used for volume rendering, namely isosurface rendering *(iso)*, maximum-intensity-projection *(mip)* and emission-absorption models *(eam)* using a 512x512 view-port. *Base* indicates the obtained frames per second for finding the initial surface triangle only. We evaluated the tetrahedral Plücker against the hexahedral Plücker *(hex-P)* and our hybrid *(hex-H)* approach. Since the number of tetrahedra respectively hexahedra varies largely within the views, we decided to use an average of three views from each dimension together with one perspective view along the cell diagonal.

Table 1 shows that finding the initial face using a kd-tree needs 6 - 18% of the total rendering time (*base* measurement. More importantly, our 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, we suggest using this hybrid approach for traversing curvilinear grids on a ray tracing basis. Compared to our tetrahedral implementation the performance is about the same. We believe that this is

Table 1: *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.*

Data set	base	iso	mip	eam
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

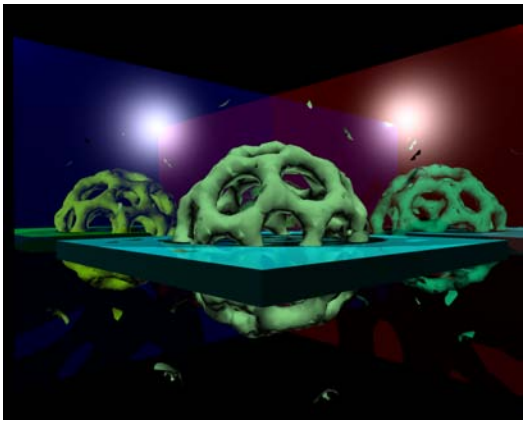

Figure 8: *Seamless integration with surface ray tracing. The volume data set, in this case the Bucky-ball, is augmented and surrounded by reflective surfaces and light sources.*

caused by higher memory bandwidth and poor cache performance.

Using OpenRT [Wal04] as ray tracing framework, it is also easy to combine and let interact different primitives in one scene. Figure 8 shows the Bucky-ball in a polygonal environment. Once the code to compute the intersection with an unstructured volume data set is available, all ray tracing effects (e.g. reflection, intersection between polygonal objects) work without any additional effort. It is furthermore possible to combine different volume organizations into one scene (see Figure 9). OpenRT also allows efficient distribution among a cluster of consumer PCs, or shared memory systems in our case [MS05].

7. Conclusion

We have demonstrated several traversal algorithms for rendering tetrahedral and hexahedral grids. It was shown that the properties of Plücker tests are especially useful when traversing neighboring tetrahedra since this reduces the number of tests to 2.67 in average. To achieve this, we only need to store additional 16 bytes of connectivity information per tetrahedron. Only one vertex needs to be fetched per new traversed tetrahedron keeping the memory bandwidth low.

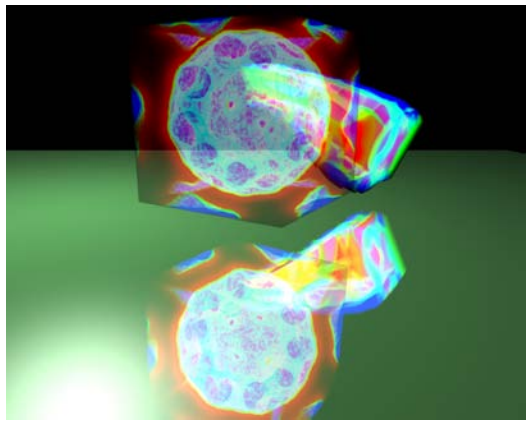

Figure 9: *Unstructured (Bucky-ball) and curvilinear (Combustion Chamber) data sets can not only be rendered into one scene. Even more important is that all primitives interact with each other.*

For rendering curvilinear data sets our measurements suggests to use a hybrid method of Plücker and bilinear patch intersections. This approach is not only faster but delivers better visual results since discontinuities are avoided. Hence, curvilinear grids should not be converted into a tetrahedral mesh. This increases the number of cells by five times and hence the memory consumption, too. Storing the connectivity information needs additional memory, and the performance is about the same.

It is also worth noting, that ray tracing is well suited for scalability. Hence, doubling the number of processors nearly doubles the performance of our traversal algorithms [MS05].

8. Future Work

Since we have demonstrated a fast software renderer for unstructured and curvilinear grids using ray tracing we would like to further improve our algorithms. Especially the memory access for cell points is time consuming. We believe that this is due to the poor cache performance, which needs further investigation. Finally, the linear scaling with respect to the number of used processors suggests to adopt this algorithm to the upcoming multi-core processors currently in development by all major chip manufactures.

Acknowledgements

The authors wish to thank the Visualization and Interactive Systems Group from the University of Stuttgart and Ralf Sonderhaus for providing data sets as well as Heiko Friedrich, Sven Woop, and Holger Theisel for discussion.

References

[BKS99] BUNYK P., KAUFMAN A. E., SILVA C. T.: Simple, fast, and robust ray casting of irregular grids. In *Dagstuhl '97, Scientific Visualization* (Washington, DC, USA, 1999), IEEE Computer Society, pp. 30–36.

[DK91] DOI A., KOIDE A.: An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE Trans Commun. Elec. Inf. Syst E-74*, 1 (1991), 213–224.

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

[For94] FORSSELL L. K.: Visualizing flow over curvilinear grid surfaces using line integral convolution. In *VIS '94: Proceedings of the conference on Visualization '94* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 240–247.

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

[Gar90] GARRITY M. P.: Raytracing irregular volume data. In *VVS '90: Proceedings of the 1990 workshop on Volume visualization* (New York, NY, USA, 1990), ACM Press, pp. 35–40.

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

[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.

[KSE04] KLEIN T., STEGMAIER S., ERTL T.: Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of Pacific Graphics '04* (2004), pp. 186–195.

[KW05] KIPFER P., WESTERMANN R.: Gpu construction and transparent rendering of iso-surfaces. In *Proceedings Vision, Modeling and Visualization 2005* (2005), Greiner G., Hornegger J., Niemann H., Stamminger M., (Eds.), IOS Press, infix, pp. 241–248.

[LC87] LORENSEN W. E., CLINE H. E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (Proceedings of ACM SIGGRAPH) 21*, 4 (1987), 163–169.

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

[Luc92] LUCAS B.: A scientific visualization renderer. In *VIS '92: Proceedings of the 3rd conference on Visualization '92* (Los Alamitos, CA, USA, 1992), IEEE Computer Society Press, pp. 227–234.

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

[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.

[Pas04] PASCUCCI V.: Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. In *Eurographics - IEE TCVG Symposium on Visualization (2004)* (2004), pp. 293–300.

[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.

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

[SDRH04] SHAUN D. RAMSEY K. P., HANSEN C.: Ray Bilinear Patch Intersections. *Journal of Graphics Tools 9*, 3 (2004), 41–47.

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/.

[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.

[WG91] WILHELMS J., GELDER A. V.: A coherent projection approach for direct volume rendering. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1991), ACM Press, pp. 275–284.

[WKME03] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-Based View-Independent Cell Projection. *IEEE Transactions on Visualization and Computer Graphics 9*, 2 (2003), 163–175.

[WM92] WILLIAMS P. L., MAX N.: A volume density optical model. In *VVS '92: Proceedings of the 1992 workshop on Volume visualization* (New York, NY, USA, 1992), ACM Press, pp. 61–68.

[WSW05] WANG W., SUN H., WU E.: Projective volume rendering by excluding occluded voxels. *Int. J. Image Graphics 5*, 2 (2005), 413–432.