Interactive Volume Visualization of General Polyhedral Grids

Philipp Muigg, Markus Hadwiger, *Member, IEEE*,
Helmut Doleisch, *Member, IEEE*, and Eduard Gröller, *Member, IEEE*

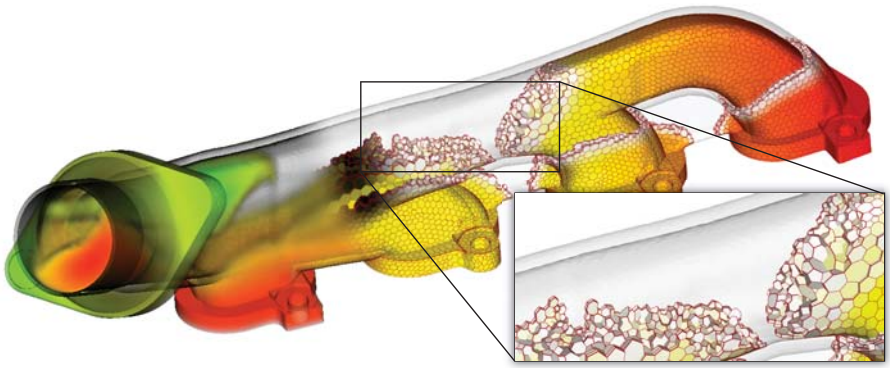


Fig. 1. Interactive ray-casting of the temperature distribution in an exhaust manifold that was simulated using a state-of-the-art CFD solver on a complex grid composed of general polyhedral cells. Red color indicates warm, green cool regions. The complex structure of the underlying mesh is illustrated through cell faces. The cells in this mesh are not only tetrahedra or other predefined cell types, but also are very general, often non-convex, polyhedra with non-planar faces, which our approach can nevertheless visualize directly.

Abstract—This paper presents a novel framework for visualizing volumetric data specified on complex polyhedral grids, without the need to perform any kind of a priori tetrahedralization. These grids are composed of polyhedra that often are non-convex and have an arbitrary number of faces, where the faces can be non-planar with an arbitrary number of vertices. The importance of such grids in state-of-the-art simulation packages is increasing rapidly. We propose a very compact, face-based data structure for representing such meshes for visualization, called *two-sided face sequence lists (TSFSL)*, as well as an algorithm for direct GPU-based ray-casting using this representation. The TSFSL data structure is able to represent the entire mesh topology in a 1D TSFSL data array of face records, which facilitates the use of efficient 1D texture accesses for visualization. In order to scale to large data sizes, we employ a mesh decomposition into bricks that can be handled independently, where each brick is then composed of its own TSFSL array. This bricking enables memory savings and performance improvements for large meshes. We illustrate the feasibility of our approach with real-world application results, by visualizing highly complex polyhedral data from commercial state-of-the-art simulation packages.

Index Terms—Volume rendering, unstructured grids, polyhedral grids, GPU-based visualization.

1 Introduction

Unstructured grids are a very important representation for volume data, such as the results of computational fluid dynamics (CFD) simulations computed using finite volume methods. Over the years, the complexity of the grids produced by state-of-the-art meshers and simulation packages, such as OpenFOAM [2] or STAR-CCM+ [3], has increased tremendously. The volume meshes used nowadays in complex multi-physics simulations, for example, consist not only of an arbitrary combination of fixed cell types, such as tetrahedra, hexahedra, and octahedra, but contain a significant number of essentially arbitrary polyhedral cells. These cells can have an arbitrary number of faces, each of which can consist of an arbitrary number of vertices. Cells can be non-convex and even degenerate, while their faces can be non-convex and non-planar. Moreover, different regions of a mesh are often generated using different meshing strategies. Taking all of these properties into account in visualization systems has become very important. However, most existing unstructured-grid visualization-methods are constrained to tetrahedral meshes for performance reasons and simplicity of implementation. Thus, these approaches have to tetrahedralize more complex grids before visualization. However, for meshes with complex cells, the increased number of cells resulting from tetrahedralization is a significant burden for visualization performance and memory usage. Moreover, a given tetrahedralization is not unique and introduces linearization artifacts when interpolation is used, since the original cells are split up into separate, piecewise linear constituents.

Existing visualization systems that are not constrained to just tetrahedra still only support a small number of convex cell types [32], or essentially perform point splatting [48]. In this paper, we introduce the first interactive visualization approach for extremely complex unstructured grids, which takes into account all of the properties of state-of-the-art volume meshes outlined above.

A major practical problem of complex meshes composed of general polyhedral cells is that their representation requires very flexible data structures. In order to be able to traverse these data structures for interactive visualization, e.g., during ray-casting, a lot of additional information is usually stored. Commonly, efficient traversal of mesh topology is facilitated by using both per-cell information, e.g., a list of faces a cell is composed of, as well as per-face information, e.g., pointers to the cells a given face connects. All this information consumes a significant amount of memory and book-keeping overhead for meshes with general polyhedral cells. In principle, either one of these two types of information is redundant. Nevertheless, it is usually incorporated for performance reasons. In contrast, as a basis for a variety of interactive visualization algorithms, we propose a very compact representation for such grids, which is purely face-based while still allowing for efficient traversal. Building on this data structure, we have developed a very flexible, interactive GPU ray-casting method. We note that, for similar reasons, state-of-the-art volume meshers have also switched to

- *Philipp Muigg and Eduard Gröller are with Vienna University of Technology, Vienna, Austria. E-mail: {muigg|groeller}@cg.tuwien.ac.at.*
- *Markus Hadwiger is with King Abdullah University of Science and Technology, Saudi Arabia. E-mail: markus.hadwiger@kaust.edu.sa.*
- *Helmut Doleisch and Philipp Muigg are with SimVis GmbH, Vienna, Austria. E-mail: {doleisch|muigg}@simvis.at.*

Manuscript received 31 March 2011; accepted 1 August 2011; posted online 23 October 2011; mailed on 14 October 2011.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

purely face-based grid representations [2, 3]. In this regard, our contribution is the development of an augmented face-based data structure that is almost as compact as these original mesh representations, but whose goal is efficient visualization instead of raw data storage. In summary, the main contributions of this paper are:

- A very compact, purely face-based data structure for complex unstructured grids composed of general polyhedral cells. This *TSFSL representation* specifically targets direct visualization algorithms, such as ray-casting, not only raw data storage.
- A very efficient GPU ray-casting approach that operates directly on the proposed TSFSL representation, with full support for domain decomposition (in this work, using a kD-tree and bricking).

2 Related Work

We discuss previous work in two distinct fields of research. First, our novel volume rendering approach for arbitrary unstructured grids is related to a multitude of volume visualization methods. Second, our method is based on a novel grid representation, which in turn is related to research conducted in the corresponding research areas.

Unstructured-grid volume-rendering: Unstructured grid rendering approaches can be categorized as either object-order methods that iterate over the cells of a mesh, or image-order methods that accumulate data for each image pixel. Both classes require different mesh representations. The Projected Tetrahedra algorithm [36], which is the basis of many object-order techniques, does not require connectivity information between neighboring cells. However, the sorting step necessary to composite individual cells in visibility order is very costly. This has resulted in several approaches that improve sorting performance by utilizing cell-to-cell connectivity information [37, 38, 47]. Object-order methods such as HAVS [11] instead utilize a hybrid CPU/GPU sorting scheme, or avoid sorting by using special order-independent optical models [43, 48], and thus do not require storing cell connectivity. With the exception of complex approximation techniques [30], all these methods are limited to tetrahedral grids. Image-order approaches such as ray-casting [16, 42] compute the final image pixel by pixel, by casting viewing rays through the mesh. For rendering performance, cell-to-cell connectivity information is used to traverse the grid. Thus, no sorting step is necessary, which compensates for the slower rendering performance of ray-casting. However, since cell-to-cell connectivity is required to traverse the mesh, memory consumption has always been a limiting factor of these methods. Therefore, different data-organization schemes have been developed to minimize the memory footprint [13, 31, 44]. Marmitt et al. [28] provide a more detailed overview of ray-casting techniques. Since some ray-casting and cell-projection approaches require a convex volume boundary, convexification methods have been proposed [22, 34]. We avoid the need for this by employing a method similar to depth peeling [6, 14, 44].

All mentioned rendering methods target tetrahedral grids and cannot easily be generalized to polyhedral cells, which have become increasingly relevant in state-of-the-art simulation packages [1, 3]. Thus, several approaches have been developed to deal with more complex cell types. Callahan et al. [10] propose a level-of-detail extension to HAVS that selectively removes triangles from a tetrahedral grid. The resulting mesh comprises polyhedral cells and can be rendered by using piecewise linear interpolation. Contrary to the proposed approach, HAVS does not support more complex interpolation methods because only face-to-vertex connectivity information is stored. Lévy et al. [26] extend the half-edge data structure by incorporating additional links for representing general polyhedral cell complexes in a *Circular Incident Edge List (CIEL)*. They propose isosurface extraction, as well as slice-based volume rendering algorithms, using CIEL. The main drawback of the CIEL data structure is the high memory consumption of storing half-edges with additional links, as stated by the authors. Additionally, parallelizing CIEL-based volume rendering is difficult, because a single global active edge list has to be maintained during rendering. Adaptive sampling of grid cells of different sizes is also not possible. This is a significant limitation, since the cell sizes in unstructured grids used for simulation can vary by several orders of magnitude. Muigg et al. [32] have introduced the first GPU ray-caster that renders more general cells directly. However, for reasons of memory addressing and alignment, their overall data layout targets grids with only a small number of different cell types. Cell faces are also limited to triangles or planar quadrangles, and only convex cells are supported. Space-time discontinuous Galerkin simulations are based on even more complex grids containing non-convex curvilinear cells. Üffinger et al. [39] have proposed a volume rendering algorithm for such grids and the corresponding simulations, which not only contain a simple scalar value but a polynomial of varying degree per cell. This polynomial representation allows for using fewer cells to resolve spatially small features. However, that work focuses mainly on solving problems related to the polynomial representation of the scalar data volume and uses a very basic grid representation. Resampling techniques create a regular [46], or semi-regular [35], grid representation from the original unstructured volume that can be rendered efficiently.

In order to tackle large data volumes, and for parallel rendering, many approaches developed for structured grids employ a spatial subdivision scheme such as octrees [7, 24, 45]. Such bricking methods have also been applied in the context of unstructured meshes [32, 33, 40]. Progressive rendering and streaming is also used to cope with large grids [9]. Besides GPU-based unstructured-grid rendering-approaches, highly optimized CPU algorithms have also been developed for volume rendering of scalar data defined on tetrahedral [17] and hexahedral grids [29], as well as point clouds [12].

Unstructured-grid representations: Data structures for representing 2D regular complexes, such as the winged-edge data-structure [5], and the half-edge data-structure [41], are the foundation of many volumetric mesh representations (see Kettner [21] for an overview). Both CIEL [26] and the data structure proposed by Bru and Teillaud [8] are direct extensions of half-edges to 3D grids. The former is highly optimized for isosurface extraction and slice-based volume rendering, and stores redundant linking information in order to speed up traversal. The latter is an extensible general purpose data structure, which stores only one additional link per half edge in a minimal configuration. However, if necessary, additional linking information can be stored per face, vertex, cell, and half facet. There are two major drawbacks of using such data structures for GPU ray-casting. First, the use of an explicit half-edge representation increases the memory footprint, because $2n+m$ half-edges per edge have to be stored and addressed (with n and m being the number of internal and external faces incident to an edge, respectively). This increased memory consumption does allow for greater flexibility with regard to edge-based queries. However, these are not required for ray-casting methods. Instead, for ray-casting one needs fast access to all faces of each cell. This is the second drawback of half-edge representations: In order to sequentially access all faces of one cell, without adding additional data, all half edges of that cell must be traversed. This requires using a stack or queue, as well as a way of marking already visited half-edges (e.g., a hash table), since the underlying half-edge graph contains an arbitrary number of cycles. This is especially problematic for GPU-based visualization. There are also grid representations that are specifically targeting meshes from numerical simulations (e.g., CFD or FEM), which often contain only a limited number of cell types. For example, Alumbaugh and Jiao [4] propose an array-based half-face data structure (AHF), which is capable of representing such volumetric meshes. Similarly to our work, half faces corresponding to one cell are packed in memory in order to reduce the number of necessary links. However, the lack of flexibility with respect to cell types (only a predefined set with small variation in face count should be used), and the need to utilize additional cell-to-vertex connectivity-data, makes the AHF mesh representation unsuitable for coping with more general polyhedral grids. The simplest type of unstructured mesh contains only tetrahedral cells. Mesh data structures optimized for this case are highly efficient with respect to memory used per tetrahedron. Examples include tetrahedral strips [44], the Compact Half Face (CHF) data structure [23], and its extension proposed by Gurung and Rossignac [18]. Since such algorithms exploit topological properties of tetrahedral cells, a straightforward generalization to arbitrary polyhedra is not possible.

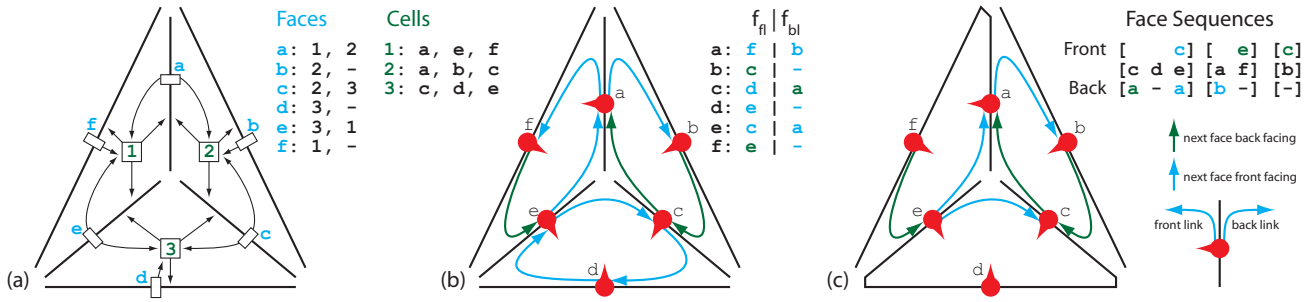


Fig. 2. Comparison of the most common representation of unstructured grids (a), with our basic TSFL (b) and extended TSFSL (c) data structures. In (a), for every cell a list of faces is stored, and every face stores which two cells it connects. In contrast, our data structure (b) stores only faces and two links per face, completely avoiding explicit per-cell storage. These two links reference the next face of the cell in front of the face (f_{fl}), and behind the face (f_{bl}), respectively. The face normals are indicated by red arrows in the figure. Additionally, each link requires a boolean flag (f_{ff}, and f_{bf}, respectively) to indicate whether the cell is in front or behind of the next face that it is linked to. Here, blue and green colors indicate the front and back half spaces, respectively. Figure 6 gives pseudo code for efficient traversal of the TSFSL data structure, shown in (c), during ray-casting.

3 Polyhedral Grid Representation

In order to develop an efficient representation and corresponding data structure for arbitrarily complex polyhedral grids, we first consider the basic requirements that such a representation has to support. The most important operation is efficient traversal of the grid topology, for example marching from cell to cell along a given ray in ray-casting. For this, one has to be able to determine *which cells* are intersected by a ray in *what order*. To achieve this efficiently, all state-of-the-art unstructured grid ray-casters rely on an adjacency graph of *cell-face-cell connectivity* (see Figure 2(a)). One must also be able to obtain sample values at arbitrary positions within each cell via *interpolation*. In tetrahedral meshes, this can naturally be done via barycentric interpolation. For more complex polyhedra, mean value interpolation [15, 20] can be used. This, however, requires data from all faces of the enclosing cell for every sample. Finally, determining the *entry face* through which a ray enters the mesh must be efficient. Current unstructured-grid ray-casting-methods usually achieve this by using GPU rasterization of the mesh boundary, storing a face ID in each pixel.

Thus, our representation must support the following operations:

1. *Query all faces of an individual cell:* Provide efficient means for enumerating all faces of a given cell.
2. *Query neighboring cell across a face:* Given a cell and one of its faces, enable fast access to the neighboring cell across this face.

These fundamental operations are not only sufficient for ray-casting, but for all algorithms that require traversing the grid from cell to cell in some order. Examples include the computation of stream, path, and streak lines or surfaces, vortex-core extraction-techniques, and many other visualization algorithms. With the exception of tetrahedral strips and CIEL, all data structures used by previous ray-casting approaches employ an explicit cell representation. This is simple and efficient for tetrahedral grids, where each cell has four vertices and four faces. When the number of faces per cell is not constant, however, a more complex representation must be used in order to allow each cell to contain an arbitrary number of links to faces comprising its boundary.

3.1 Standard Unstructured-Grid Representation

Figure 2(a) illustrates the most common data structure for representing unstructured grid topology. This example depicts three cells composed of six faces in total, where each face knows which two cells it connects (*face-to-cell connectivity*, column "Faces"), and each cell knows all of its comprising faces (*cell-to-face connectivity*, column "Cells"). With this information, querying all faces of an individual cell, as well as traversing from one cell through one of its faces to its adjacent cell, is trivial. Note, however, that only one of the two types of connectivity is really required to represent the entire mesh and to allow reconstruction of the other type. In principle, it is trivial to reconstruct face-to-cell information from cell-to-face information, and vice versa. The advantage of storing redundant connectivity is that it allows for fast *cell-face-cell traversal*, which is crucial for many visualization algorithms. Thus, the trade-off is sacrificing memory for higher traversal performance. Naturally, for complex cells the memory footprint of redundant topology information rises rapidly.

3.2 TSFL/TSFSL Unstructured-Grid Representation

In contrast to the standard representation outlined above, the flexible data structure we introduce in this paper does not store any redundant connectivity information at all. However, it still allows for very efficient grid traversal and supports the two required fundamental operations described above. This is achieved by representing all connectivity through lists of faces comprising the mesh, without any explicit cell representation. Furthermore, we have separated the actual face-geometry data, such as how many and which vertices make up a face, from the representation of mesh topology. This enables a very compact representation of polyhedral meshes generated by different meshing approaches that employ a wide variety of cell characteristics.

3.2.1 Two-sided face lists (TSFL)

We only store mesh faces, and exploit the fact that each face has at most two adjacent cells. Therefore, if we think of linking all faces of a given cell together, then any given face can be a part of at most two separate face lists. That is, a face is part of two face lists when it connects two cells, or is part of only a single face list when it is a boundary face. Therefore, it is sufficient to reserve only two link fields per face, including some additional information per link as described below. Because of the two sides of each face and their respective lists, we call the resulting data structure *Two-Sided Face Lists (TSFL)*.

Face links: Consider Figure 2(b), which depicts six faces, each of which has two links to other faces. The *front link* f_{fl} links to another face of the cell located in its front half space; the *back link* f_{bl} links to another face of the cell in its back half space. Each cell is then only represented implicitly as one cycle in the directed graph whose vertices are the cell faces, and whose edges are these front and back links, respectively. This representation has obvious similarities to the winged edge data structure for surface meshes [5]. However, instead of linking edges in adjacency order, we link the faces comprising a cell in *arbitrary order*, storing only minimal linking information that is sufficient to support the two queries required above. This arbitrary linking differentiates TSFL from other half-edge/face data structures, where each link represents a specific geometric relation.

Face link flags: In order to enumerate all faces of a cell, the front and back links alone are not sufficient. For example, consider enumerating the faces of the cell in front of face f in Figure 2(b). At every step of following a face link, one has to know whether to follow the front link or the back link, respectively. It is not possible to orient face normals consistently with respect to every cell. This problem can be solved by augmenting every link field with a boolean flag that indicates whether the cell whose faces are being enumerated is in the front or back half space, respectively, of the face that the link refers to.

Cell-face-cell traversal: The TSFL representation, consisting of the per-face front link (f_{fl}) with its boolean flag (f_{ff}), and the per-face back link (f_{bl}) with its boolean flag (f_{bf}), is sufficient for complete

enumeration of all faces of any cell in the entire mesh, and for traversing from any cell across any of its faces to the respective adjacent cell. That is, cell-face-cell traversal is performed by intentionally following the link corresponding to the opposite half space of a face, thus stepping from the cell on one side of the face to the cell on its other side. In the remainder of this paper, we will further use the notation that the two values that the two boolean flags (f_{ff}, f_{bf}) can assume are $+$ and $-$, to indicate the front and back half space of a face, respectively.

3.2.2 Two-sided face sequence lists (TSFSL)

We now introduce an important refinement of the TSFL representation, which results in the construction of the *Two-Sided Face Sequence Lists* (TSFSL) data structure that we use in the remainder of this work. A TSFSL is constructed from a TSFL by storing selected sequences of linked faces in adjacent memory locations without storing actual links. This significantly optimizes access and traversal speed, and reduces the overall memory footprint. One of the main advantages of storing a mesh via both cell and face arrays, as in Figure 2(a), is that references can be stored sequentially in memory and thus allow for fast access. For example, enumerating all faces of cell 3 in the figure simply scans the array $[c,d,e]$. The same enumeration in the TSFL data structure involves more effort, because for each face additional linking information must be read from memory. This introduces a large number of random memory accesses that reduce performance on architectures that prefer coherent memory accesses, such as GPUs. It also inherently incurs memory access latency, because the next face can only be visited after the corresponding link has been fetched from memory.

The TSFSL data structure overcomes these problems to a large extent. Considering the mesh shown in Figure 2(b), we can observe sequences of faces that are connected by arrows of the same color. Note especially the cell with faces (c,d,e). Here, within the cell only front links have to be followed since all faces are facing toward the cell. The occurrence of such *face sequences* can be exploited by storing their faces sequentially in an array without links. If this sequence is stored consecutively in memory as $[c,d,e]$, we can drop the *front* links and flags entirely. This does not result in any loss of information, since the storage location itself now implicitly encodes these links. If face sequences are constructed using front links, which we have chosen to do, this immediately implies that the *back* links and flags have to be retained, because they are then required to connect to adjacent cells. This also implies that all face sequences can be constructed by collecting all the faces of each cell that are facing toward it.

We guarantee *at most one* face sequence per cell, by putting all faces that are facing toward it into the same sequence. This is possible because the order in which the faces of a cell are linked is arbitrary. Guaranteeing *at least one* face sequence per cell is also possible, because simply flipping a face's normal allows removing it from one sequence and adding it to the sequence of the cell on the opposite side. Together, this allows guaranteeing *exactly one* face sequence per cell. We use the following two-phase *face sequence generation algorithm*: The first phase starts with an initially empty face sequence s_i for each cell c_i. Then, the number of faces n_i assigned to each s_i is tracked, while assigning each face to the sequence with the lower n_i (outside the mesh, $n_i = \infty$). For all meshes that we tested, creating sequences this way was already sufficient. However, in general this may produce empty sequences with $n_i = 0$. In order to guarantee $n_i > 0$ everywhere, we process each empty s_i in a second phase. For each such s_i, we pick a neighboring cell c_j based on two conditions: c_i has to be connected to c_j through a face that is not flagged as *used*, and n_j has to be the maximum among the neighbors that fulfill the first condition. Because $n_i = 0$, the face connecting c_i to c_j is assigned to s_j. By re-assigning this connecting face to s_i, n_i is increased from zero to one, and n_j is decreased by one. The face is then flagged as *used*. This face re-assignment is repeated as long as $n_j = 0$. If $n_j \neq 0$, all flags that identify faces as *used* are cleared. After each iteration of this algorithm, the number of empty face sequences is reduced by one.

See Figure 2(c) for an illustration of how face sequences can be defined in this example. Faces within a sequence are shown connected without arrows. Note that the last link e_{ft} and flag e_{ff} have to be retained, in order to connect the end of the face sequence with subsequent faces of the cell. In the example of sequence $[c,d,e]$, all faces of the cell are in one face sequence. This is not true in general, as can be seen in the face sequence $[a,f]$. Also shown in Figure 2(c) is the face b, which is considered as a face sequence of length one, since it cannot be connected to a longer face sequence.

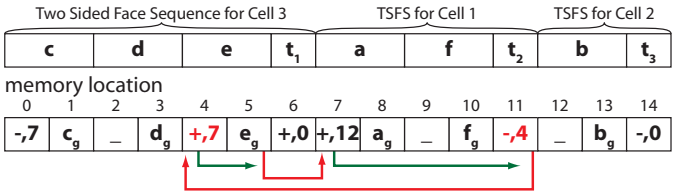


Fig. 3. All face sequence lists of the mesh from Fig. 2 are stored in a single linear 1D array. The colored arrows illustrate the traversal of the faces of cell 1. Green arrows indicate sequential memory reads, whereas red arrows denote jumps to different memory locations. The $+$ and $-$ signs indicate flag values of front- and back-facing, respectively.

In the following, we summarize four important properties of the TSFSL representation that result from its construction:

1. The face list corresponding to any given cell contains *exactly one* face sequence, which contains all the faces that are facing *toward* the cell. This sequence can, however, be of length one.
2. The *front* links/flags for faces in the sequence are *not stored*, with the exception of the last face in the sequence.
3. The *back* links/flags must be stored for *all* faces in the sequence.
4. All faces facing *away* from a cell are not part of this cell's face sequence, but of the sequence of the cell they are facing toward.

A crucial consequence of this construction is that *all* face sequence lists of an entire mesh can be stored back-to-back in a single 1D array.

Memory savings of the TSFSL representation: In order to quantify the memory savings resulting from face sequences, we count the number of links that can be discarded, in comparison with the TSFL structure. For a sequence containing n faces, $n-1$ links can be removed. For a mesh containing c cells, with f internal and b mesh boundary faces, the average face sequence length is $(f+b)/c$, because each face belongs to exactly one face sequence, which in turn corresponds to exactly one cell. Thus, on average $(f+b)/c-1$ links can be omitted per cell, which amounts to omitting $f+b-c$ links in total over the entire mesh. Note that with an increasing average number of faces per cell, $(b+2f)/c$, the number of links that can be discarded increases. Overall, the number of links stored in the TSFL representation is equal to $b+2f$, whereas the TSFSL representation thus contains only $f+c$ links. For comparison, the standard approach depicted in Figure 2(a) must store $2(f+b)$ links from faces to cells, and $b+2f$ links from cells to faces. It is interesting to note that the number of links stored in our data structure depends on the number of cells, whereas this is not the case in the standard approach. In order to directly compare both numbers, we can use the property that the number of faces per cell is at least 4 (i.e., cells are at least tetrahedra), and thus: $(b+2f)/c \geq 4$. From this follows that $(b+2f)/4 \geq c$. An upper bound for TSFSL can be defined as $f+c \leq 3f/2+b/4$. For the "worst" case of a tetrahedral mesh this is still smaller than either the number of face-to-cell or cell-to-face links required in the standard representation. Relative memory savings increase as the average number of faces per cell goes up. Actual memory consumption numbers and comparisons for real-world meshes are reported in Section 5.3.

3.3 Face Geometry and TSFSL Storage

In addition to storing mesh topology in the TSFSL data structure, the geometry of each face must also be stored. Conceptually, we completely separate mesh topology from face geometry. This enables adapting the geometry representation to a variety of characteristics of real-world meshes and the corresponding meshing algorithms and tools, which can also vary between different mesh regions or zones. Despite the important conceptual separation, we can interleave the actual storage of face geometry in memory (denoted below abstractly as f_g for face f) with the TSFSL records of the corresponding faces. This

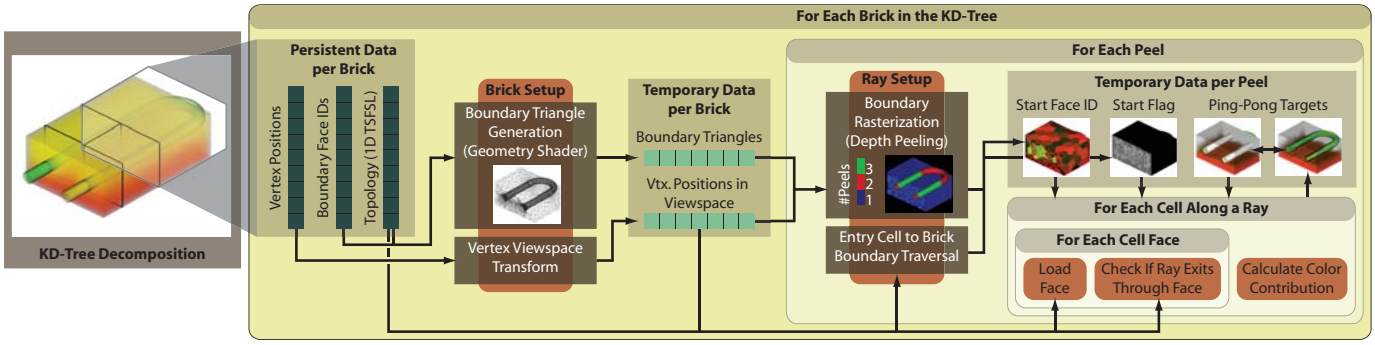


Fig. 4. The pipeline for ray-casting unstructured meshes given in the TSFSL representation. A large mesh is subdivided into bricks that correspond to the leaves of a kD-tree. Rendering proceeds from brick to brick in front to back order. Each brick contains the corresponding 1D TSFSL array, as well as vertex positions and IDs of the faces comprising the mesh boundary. In order to obtain ray-start positions, the mesh boundary is rasterized. Non-convex cells are treated implicitly in each rendering pass, whereas non-convex areas of the mesh are handled using depth peeling.

improves memory-access coherency. The TSFSL representation stores $n+1$ face links/flags for a face sequence of n faces. Interleaving this storage with actual face geometry, a face sequence of faces $[a,b,c]$ is stored in memory as $[(a_{bf},a_{bl}),a_g,(b_{bf},b_{bl}),b_g,(c_{bf},c_{bl}),c_g,(c_{ff},c_{fl})]$. Note that the front link/flag of the last face in the sequence is stored after the corresponding face geometry. We will call it the *face sequence terminator* (denoted as t_i in Figure 3). Given this memory layout, the face sequence can be traversed by alternately reading a link/flag tuple and face geometry, until the sequence terminator is read. As above, one of the most important observations to make here is that *all* faces of a mesh are stored as part of a face sequence. This is possible because every face of the mesh is part of exactly one face sequence. This sequence is the one corresponding to the cell in the front half space of the face. The faces' back links, also stored in the sequence, reference faces at non-sequential storage locations, which in turn are part of other face sequences. Therefore, the entire mesh can be stored in a single 1D array, as illustrated in Figure 3.

Face geometry records: The actual geometry storage format of f_g can be arbitrary, with the restriction that its size can either be inferred implicitly (e.g., if all faces use an equal amount of memory), or can be derived from a small header at the start of f_g. Our current implementation supports two fixed-size face geometry representations, which can store three or four vertex indices. A variable-sized version stores the number of vertices of each face in addition to its vertex IDs. The former are used for meshes (or bricks, see below) that only contain triangle and quad faces. The latter is used for more complex polyhedral meshes, for example the mesh shown in Figure 1. A more detailed description is given in the context of ray-face intersection in Section 4.3.

4 Ray-Casting of TSFSL Meshes

For efficient ray-casting of large TSFSL meshes, the grid is subdivided into bricks that are the leaves of a kD-tree (Section 4.1). Bricks are rendered individually, and composited in front to back order (Sections 4.2 and 4.3). The mesh in each brick is represented by a single 1D TSFSL array. Figure 4 gives an overview of the resulting ray-casting pipeline.

4.1 Bricking

We employ a domain decomposition of the entire mesh into bricks. Reasons are: scalability, memory management, adaptivity with respect to characteristics of the contained cells, coarse-grained culling, and the potential for direct parallelization across multiple GPUs. The brick geometry corresponds to the leaves of a kD tree, which is built using criteria similar to previous work [32]. However, we avoid explicitly clipping cells against brick boundaries. Instead, each brick is assigned a *submesh* that comprises all cells that are entirely or *partially* inside the brick boundary. Ray-casting is performed brick by brick in front-to-back visibility order, determined using the kD tree. In the following, the ray-casting procedure is described considering a single brick.

4.2 Ray-Setup Stage

The first step of ray-casting is to determine the positions where rays enter each brick. The most important part of this step is determining the ID of the face through which a given ray enters the first cell. This can be done very efficiently by encoding IDs in colors and rasterizing the grid boundary accordingly. In previous GPU ray-casting approaches [13, 32, 42, 44], the triangle mesh representing the volume boundary had to be stored explicitly in a format suitable for fast rasterization (e.g., in OpenGL vertex buffer objects). We avoid permanently storing this redundant representation for the whole mesh. Instead, each brick's submesh boundary is extracted on-the-fly from the TSFSL data structure before a brick is rendered, and immediately discarded afterward. The size of the buffer for storing this temporary boundary geometry needs only be large enough to accommodate the largest brick of the mesh. The submesh boundary geometry is generated on-the-fly using a GPU geometry shader. The faces of the triangulated boundary are computed from an array of boundary face IDs stored with the brick data (see Figure 4). Details on using a geometry shader for computing this triangulation are provided in Section 5.1.

Special care has to be taken when bricking is used. The spatial subdivision requires that the submesh in each brick must only contribute to the volume rendering integral inside the brick boundary for correct compositing. Previous options for doing this include either explicitly clipping cells against the brick boundary, or rasterizing one quad per cell that is large enough to cover the projection of the cell/clipping plane intersection in screen space [32]. The former approach requires storing explicit geometry for the resulting cuts, which is very costly for general polyhedral cells and usually involves triangulating the cut. The latter approach requires discarding fragments outside of the cell using face geometry information. However, for a complex cell and face geometry, the inside/outside test for discarding fragments becomes very expensive. Given these drawbacks of previous approaches for complex polyhedral cells, we use a different approach. Before the actual rendering passes for each brick, we perform a setup-rasterization step that traverses rays in order to find the entry positions of rays into cells inside the brick. This setup step is illustrated in Figure 5. Instead of rasterizing the volume boundary and the cell/brick-boundary intersections separately, we rasterize the unmodified volume boundary of the submesh that intersects a given brick, without clipping it. This rasterizes faces completely within the brick boundary, faces intersected by the brick boundary, and faces completely outside the brick boundary.

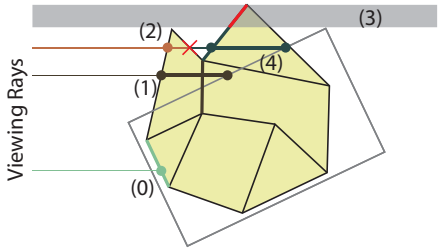


Fig. 5. Instead of explicitly clipping cells against brick boundaries (grey rectangle), the ray-setup stage of each brick determines ray-start positions by traversing rays from boundary-cell faces to the brick boundary.

During this rasterization, a setup ray is cast for each fragment. If the start position is already in the brick (ray (0) in Figure 5), the fragment immediately writes out the ID of the face that is being rasterized. For all other rays, the ray start position is outside the brick boundary, where we distinguish the following cases:

1. Rays may enter the brick without first leaving the corresponding submesh (ray (1) in Figure 5). At that point, these rays terminate and write out the face ID of the last face they intersected.
2. Rays may leave the submesh before they enter the brick (ray (2) in Figure 5). These rays terminate without writing a face ID.
3. Rays that do not intersect the brick boundary at all (in the area marked with (3) in Figure 5). These rays terminate immediately.

Note that since the entire submesh boundary is rasterized, this procedure also determines correct entry positions in non-convex areas, with respect to both non-convex mesh and cell boundaries. For example, ray (4) in Figure 5 determines and writes out the correct intersection and start position at the brick boundary, whereas ray (2) is correctly discarded, as described above.

The output of the ray-setup stage is a *ray-setup image* that for every pixel stores a face ID and flag that identify the cell where the corresponding ray must start for volume ray-casting (Section 4.3). The face ID is encoded as a color, and the alpha channel is used for the flag. We deal with rays exiting and re-entering non-convex meshes by utilizing an approach that is similar to depth peeling [32, 44]. The grid is processed in depth layers corresponding to successive front-facing mesh boundary faces. For each depth layer, a ray-setup image is generated. However, the full ray-setup stage is only required for the first depth layer, since for all subsequent layers the ray start positions are already guaranteed to lie inside the brick. For these layers, simply clipping the submesh boundary to the brick's bounding box is sufficient.

4.3 Ray-Casting

After the ray-setup stage described above, the main volume ray-casting pass is executed. In this pass, a ray is generated for every pixel where the flag in the ray-setup image is $\neq 0$, i.e., where the ray has been determined to intersect the mesh, and the pixel's face ID is valid. Volume ray-casting proceeds in a loop from sample to sample along each ray, repeatedly carrying out the following three main tasks:

1. *Mesh traversal*, for traversing the mesh topology along a ray cell by cell, using the TSFSL data structure (Section 4.3.1).
2. *Ray-face intersection*, for determining the next face intersected by the ray, also for non-convex faces and cells (Section 4.3.2).
3. *Sampling and interpolation*, for determining sample values via interpolation within the current cell (Section 4.3.3).

4.3.1 Mesh traversal

The algorithm is illustrated in pseudo code in Figure 6. The outer loop (lines 3-27) traverses the mesh along a ray by stepping from cell to cell through shared faces. The inner loop (lines 6-23) iterates over all faces of the current cell using the TSFSL data structure, and computes the next intersection of a ray with a cell face. Because we do not store explicit cell information, both of these tasks are performed by following face links: Either by stepping from one side of a face to the other side by following the opposite-side front/back link, or by iterating over all faces of a cell, in turn invoking the ray-face intersection (Section 4.3.2) for every encountered face. Mesh traversal is started at the face specified in the color channel of the ray-setup image: *startFace*. The flag stored in the alpha channel is used to decide whether to initially follow front or back links: *cellInFrontOfStartFace*. The result of the inner loop is the face through which the ray leaves the cell, as well as the flag indicating whether the next cell is in front or behind this face. Note that non-convex cells can be entered and exited multiple times.

Before traversal is continued in the outer loop, the entire ray segment between the entry and exit faces of the inner loop is sampled and composited. The exit face and negated flag resulting from the inner loop completion then become the input to the next iteration of the outer loop. The simple operation of logically negating the flag before the next loop iteration is what performs the step from one cell to the next. The outer loop terminates when the ray has exited the brick.

4.3.2 Ray-face intersection

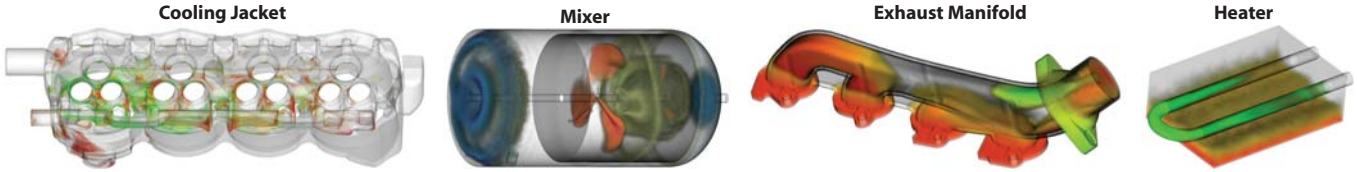
The inputs to the ray-face intersection algorithm are: a face ID, a ray direction $\mathbf{v}$ and origin $\mathbf{o}$, and a boolean flag s that indicates whether the face normal should be flipped for the intersection test as described below. The output is the distance to the intersection, or ∞ if there is no intersection. An intersection with a face is only reported when the local face normal $\mathbf{n}_x$, at an intersection point x, points in the same direction as $\mathbf{v}$, i.e., $\mathbf{n}_x \cdot \mathbf{v} > 0$. If s is *true*, the face normal is flipped for this comparison. This test, together with a check whether the intersection is further down the ray than the cell entry position, correctly handles faces that are viewed edge-on. It also avoids erroneous detection of intersection with the face through which the ray has entered the cell. In the case of orthogonal projection, we further simplify the computations for ray-face intersection by transforming all vertices of a brick from world space to view space, and performing intersection in the latter. The ray direction then is $(0,0,-1)$, which reduces the required 3D ray-triangle intersections to computing 2D barycentric coordinates for $\mathbf{o}$ transformed and projected into view space. The involved dot product reduces to a single multiplication. For further optimization, our system integrates three different types of face geometry representations at a per-brick granularity (see also Section 3.3). Each type employs its own optimized ray-face intersection:

1. For purely tetrahedral bricks, the three vertex indices of each triangle are stored, which makes the ray-face intersection trivial.
2. For bricks with cells where all faces are either triangles or quads (e.g., a mixture of tetrahedral, hexahedral, and octrahedral cells), quads are decomposed on-the-fly into two triangles. The ray-face intersection is then carried out separately for these two triangles. Note that this case includes non-planar faces.
3. For bricks containing cell faces with more than four vertices, i.e., an arbitrary number of vertices describing possibly highly non-planar faces, we use the general strategy described below.

For efficiency reasons, our system triangulates cell faces with more than four vertices when a mesh is loaded, instead of triangulating them

```
1: function MeshTraversal
2:     Read startFace and cellInFrontOfStartFace from ray-setup images
3:     repeat                          ▷ This loop traverses all cells along a view ray
4:         currentFace ← startFace, d_min ← ∞
5:         cellInFrontOfFace ← cellInFrontOfStartFace
6:         repeat                      ▷ This loop iterates over all faces of a cell
7:             d ← RayFaceIntersection(currentFace, cellInFrontOfFace)
8:             if d < d_min then
9:                 d_min ← d
10:                minFace ← currentFace
11:                cellInFrontOfMinFace ← cellInFrontOfFace
12:            end if
13:            if cellInFrontOfFace then          ▷ Go to next face of cell
14:                advance currentFace along face sequence
15:                if sequence terminator reached then
16:                    cellInFrontOfFace ←GetFrontFlag(currentFace)
17:                    currentFace ←GetFrontLink(currentFace)
18:                end if
19:            else
20:                cellInFrontOfFace ←GetBackFlag(currentFace)
21:                currentFace ←GetBackLink(currentFace)
22:            end if
23:        until currentFace = startFace
24:        Perform sampling between startFace and minFace
25:        startFace ← minFace
26:        cellInFrontOfStartFace ← ¬cellInFrontOfMinFace
27:    until startFace is boundary face or ray exits brick
28: end function
```

Fig. 6. Pseudo code for mesh traversal during ray-casting using the TSFSL data structure (see Section 4.3). Note that a face in this context simply denotes a face reference (ID), not actual face geometry data.



	Cooling Jacket	Mixer	Exhaust Manifold	Heater
Cells/Vertices/Faces:	1,538K / 1,631K / 4,707K	1,362K / 7,432K / 8,869K	82K / 324K / 441K	17K / 68K / 91K
Tetrahedra:	17,044K (~8.5 byte/tet)	89,417K (~7.5 byte/tet)	4,095K (~7.0 byte/tet)	851K (~7.0 byte/tet)
Celltypes:	tets/pyramids/wedges/hexas	general (non-convex) polyhedra	general (non-convex) polyhedra	general (non-conv.) polyh.
Bricks/Cell Overhead:	4/1.7%	10/8.6%	1/0%	1/0%
TSFSL Creation Time:	4.0s	9.0s	1.7s	1.0s

Fig. 7. The data sets used to evaluate our TSFSL ray-casting approach, with mesh statistics. Timings for ray-casting are given in Table 2.

on-the-fly during ray-casting. For this purpose, arbitrary triangulations can easily be integrated into our system. We have done extensive experiments with triangulations based on either a triangle strip or a triangle fan. Only the latter has turned out to work very well in practice. A very important consideration for face triangulations is that in complex real-world meshes the faces are often non-planar, ranging from slightly non-planar to extremely non-planar. In the meshes that we have examined, a triangle strip decomposition of each face has turned out to be infeasible. These meshes contain at least some faces that are so highly deformed that they cannot be decomposed into triangle strips without introducing mesh inconsistencies. Our current approach therefore decomposes each face into a triangle fan around the centroid of all face vertices. Such a triangulation is symmetric, while the triangle strip decomposition is not. Ray-face intersection during ray-casting is then performed by intersecting the ray with each triangle of the fan individually. A drawback of triangle fans around the centroid is that an additional centroid vertex has to be stored per face. Naturally, general ray-face intersections are costly. Therefore, for each face we also store the squared radius r^2 of a bounding sphere centered at the centroid. Full ray-face intersection is only performed when the ray's intersection with the face's bounding sphere is non-empty. This test is also performed in screen space which reduces the ray-sphere intersection to simply computing the squared distance between the ray position and face centroid in 2D screen space. The proposed triangle-fan face-decomposition is not feasible for faces which do not contain their centroid or are not star-shaped. To cope with such faces, a more general face-geometry representation, such as an actual triangle mesh, could be used. However, none of the real-world meshes that we have evaluated actually contained such faces.

4.3.3 Sampling and interpolation

The final component of ray-casting is obtaining values at sample positions by using different interpolation schemes. We switch between two different interpolation methods on a per-brick basis:

1. For vertex-centered data, we employ mean value interpolation in the interior of complex cells [32], and barycentric interpolation in purely tetrahedral cells, as well as on all triangular faces.
2. For cell-centered data with a constant scalar value per cell, we employ piecewise constant interpolation. This is very useful in practice, because in many cases real simulation data from state-of-the-art packages [1, 2, 3] can be cell-centered. Thus, this simple approach generates visualizations that are an adequate representation of the simulation results.

For mean value interpolation, we allow the user to enable two different optimizations during interaction that slightly reduce image quality while increasing rendering performance significantly. First, we can fall back to *linear interpolation* along a ray segment between cell-ray entry and exit positions. Second, we can use an *adaptive scheme*, which alternatively uses either linear interpolation along a ray segment or inserts one mean-value sample at the segment's center. Subsequently, linear interpolation between entry, center, and exit points is used. The additional mean-value sample is only used when the ray-segment length exceeds a user defined threshold. Even for high quality visualizations, we found it to be sufficient to use one mean-value sample at the center of each ray segment within a cell. This can be attributed to the nature of mean value interpolation, which deviates most strongly from linear behavior in the cell interior, whereas it quickly converges toward barycentric interpolation at the cell faces. Note that in the case of non-convex polyhedra, mean value interpolation can generate negative weights for faces whose exterior side is facing toward the sample position. In order to solve this, positive mean value interpolation could be used instead [27]. However, this would come at the expense of smoothness, since positive mean value coordinates are not guaranteed to remain smooth within non-convex polytopes.

As an additional consideration, using complex interpolation methods for cells that are smaller than one image pixel in the final visualization is a waste of processing resources. Thus, we automatically enable piecewise constant interpolation for bricks where all cells project to one image pixel or less. These bricks are determined by storing their maximum cell size and calculating the corresponding projected size at the distance of the brick corner closest to the view plane.

5 Results

We discuss important implementation details, give results for complex real-world meshes, and compare our approach with previous work.

5.1 Implementation

This section discusses details of how the TSFSL representation and additional mesh data are stored in memory, as well as our TSFSL ray-casting implementation.

Memory layout: Our ray-casting system is implemented in OpenGL and GLSL. The entire TSFSL data structure for each brick is stored in a single 1D 32-bit integer texture. Each face is represented by a 32-bit int header followed by the vertex index data. The header contains a 24-bit address for the face's back link, 6 bits for the size of the following vertex data, and two additional flag bits. One is the back flag for the back link, and the second flag indicates whether the face is the last in the sequence. If the latter bit is set, the integer value following the face vertex data is the face sequence terminator. As in the face header, the int of the face-sequence terminator is split into a 24-bit address for the front link, and one bit front flag. In the case of cell-centered data, the terminator is followed by a 32-bit int addressing the cell corresponding to the face sequence that is terminated. If triangle-fan face-decomposition is used, we store all vertex indices of a face, starting with the center vertex, as 32-bit int indices. For triangle and quad faces, each vertex is a single 32-bit int.

Ray-casting: Two different kinds of temporarily derived data are generated before ray-casting each brick (see Section 4): First, all vertex positions are transformed into view space by utilizing the transform feedback capabilities of current GPUs. Then, the mesh boundary containing b faces is triangulated on-the-fly by a geometry shader. The input primitives for this shader are b points, for each of which only a single integer index is stored. This index is the address of a face in the TSFSL data structure that is subsequently triangulated according to the current face-triangulation settings. On current GPUs, we are limited to generate triangle strips in the geometry shader. We therefore generate the triangle-fan decompositions of complex cell faces by inserting degenerate triangles. This approach is preferable to restarting a new strip after each triangle since fewer vertices have to be emitted

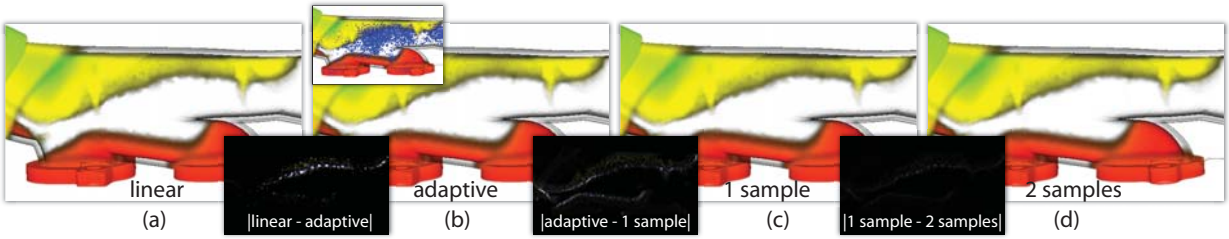


Fig. 8. Comparison of the influence of different sampling strategies on image quality. The three bottom-row inlays depict the differences between two neighboring images, respectively, with difference-color values scaled by a factor of 16. The additional inset at the top of the image labeled "adaptive" depicts the regions using mean value interpolation (colored in blue).

by the shader. After depth peeling has been performed for a brick, the triangulation data as well as the transformed vertex positions are discarded. This approach is feasible because the overall mesh is processed brick by brick, each of which contains only a fixed maximum of cells/faces/vertices. The shading effects visible in Figure 7 are generated by applying limb darkening [19].

5.2 Data Sets

In order to demonstrate the usefulness of our new volume-visualization algorithms we have selected the four representative data sets depicted in Figure 7, which range from fairly small to reasonably large in size. Among these four data sets, two different configurations of our ray-casting approach have to be employed in order to cope with different cell characteristics. In order to compare memory requirements and rendering performance, we provide the number of tetrahedra into which the mesh can be decomposed. The tetrahedralization scheme taken to derive this number is selected based on the face-triangulation approach used for the respective mesh. Each sub-triangle of an interior face results in two tetrahedra made up of the triangle vertices and the cell centroid of either the cell in front or behind the triangle. Sub-triangles of boundary faces only generate one tetrahedron. Although this is not the tetrahedralization resulting in the minimum number of cells, we have used it to obtain result images that are comparable with our results, which use mean value interpolation.

The *heater*, *manifold*, and *mixer* data sets have been created by the same simulation package [3] and therefore share common mesh characteristics. All meshes comprise general polyhedral cells with often highly non-planar faces. We cope with this by utilizing a triangle fan decomposition of each face. Note that as soon as more complex cells are present, the number of tetrahedra generated by tetrahedralization can increase between one to two orders of magnitude when compared to the original cell decomposition. The *cooling jacket* data set [25] contains only a fixed set of cell types (tetrahedra, quadratic pyramids, triangular prisms, and hexahedra), which again can be non-convex. The mixer and the cooling jacket data sets are decomposed into four and ten bricks, respectively, due to their size. The increase of the number of cells that have to be stored due to bricking is shown in Figure 7.

5.3 Evaluation and Discussion

The evaluation of the methods proposed in this work is complex and will be performed in multiple parts. First, we compare the memory requirements of the TSFSL mesh representation to other data structures that have been used to perform volume rendering. Next, we evaluate the rendering performance of different quality and sampling settings, and discuss their impact on the resulting image quality.

5.3.1 Memory Consumption

This comparison is difficult since to our knowledge there exist only four other GPU volume renderers capable of directly rendering convex polyhedral cells [10, 26, 32, 39]. Three of those [10, 26, 39] can process non-convex cells without requiring subdivision. The HAVS level-of-detail extension has been included in this list since it can cope with polyhedral cells on a basic level. It is, however, limited to piecewise linear interpolation along ray segments within a cell because only face-to-vertex connectivity data is available to the GPU. We compare the memory requirements of our TSFSL structure with different state-of-the-art representations by giving the number of bytes per tetrahedron in Table 1. For non-tetrahedral meshes, this is computed by dividing the overall memory consumption by the number of tetrahedra in an equivalent tetrahedralization. The memory overhead that is introduced by our technique due to bricking is included in these figures.

The hybrid ray-casting algorithm [32] is optimized for meshes containing only few cell types and requires convex cells composed of triangular or planar quadrangular faces. Slicing-based volume rendering on CIEL is mainly executed on the CPU, because only the slice rasterization is performed on the GPU, and thus requires basically no GPU memory. The per-tetrahedra memory consumption for CIEL has been estimated based on the overall memory consumption stated in the corresponding publication. Scalar data are represented as polynomials in the case of the higher-order finite-elements visualization technique (HOFEV) [39]. Therefore a direct comparison to the other techniques in Table 1 would be biased toward techniques dealing with conventional CFD data. In order to avoid this biasing, we have compared the memory required to store only topology information. In the case of data sets such as the heater or the manifold, the TSFSL representation uses 34% less memory than the topology data structure employed by HOFEV. For simpler data sets, such as the cooling jacket, this difference is 24%. Similarly to TSFSL, HAVS uses face data to represent a volume mesh. Only three vertex indices per face as well as vertex positions and scalar data are stored in GPU memory. This results in very competitive memory requirements on the GPU side. However, a significant amount of memory is used to perform efficient face sorting on the CPU. The per-tetrahedra memory figures are based on our four test data sets. In summary, Table 1 clearly shows that our GPU-based data structure is superior with respect to memory consumption to most state-of-the-art grid representations for volume rendering. Only HAVS utilizes a data representation on the GPU that is equally efficient.

5.3.2 Rendering Performance

Besides the memory footprint, the actual rendering performance of our ray-casting approach has to be evaluated. However, we note that the main goal of this work is to reduce the memory that is necessary to represent an unstructured mesh, while still allowing for efficient ray-casting. A major reason for this rationale is that recent developments in graphics hardware indicate that computing speed is growing far more rapidly than on board memory size. Table 2 compares the performance of an object-order volume-rendering-method (HAVS) and a state-of-the-art GPU-based tiled ray-casting method (TRC) to TSFSL volume rendering. All tests have been performed by rendering into a 1024x768 view port on a Core 2 Quad CPU at 2.8 GHz with 8 GB of RAM and an NVIDIA Geforce 480 GTX with 1.5 GB of memory. The different results listed for ray-casting with TSFSL are due to different quality and sampling settings. The rows labeled high quality (HQ) represent results generated by using the full 1024x768 render target. HAVS and TRC create output images with the same resolution. The low quality (LQ) figures were measured when rendering only 512x384 images that were subsequently scaled to 1024x768 while also only performing ray-casting for one depth peel per brick. The disproportionately strong performance increase between the HQ and the LQ modes for the manifold data set is caused by the fact that the data set consists of a thin layer of geometry representing solid parts surrounding fluid

Algorithm	Cell Types	Byte/Tet
TSFSL	general (non-convex) cells	$\sim$ 7 - 9
HARC [42]	tetrahedra only	160
TRC [6]	tetrahedra only	144
HARC-Partial [13]	tetrahedra only	96
Hybrid Rayc. [32]	convex cells (few celltypes)	$\sim$ 45 - 80
CIEL [26]	general (non-convex) cells	$\sim$50 CPU
VF-Ray-GPU [31]	tetrahedra only	38
Tet. Strips [44]	tetrahedra only	$\sim$15
HAVS [11]	tetrahedra only	$\sim$ 7 - 9 GPU $\sim$118-149 CPU

Table 1. Memory footprint comparison of TSFSL with state-of-the-art unstructured grid rendering approaches (measured in bytes/tetrahedron).

Algorithm		Heater	Manifold	Cooling J.	Mixer
TSFSL	HQ	124*ms*	156*ms*	172*ms*	1.8*s*
(cell-centered)	LQ	48*ms*	49*ms*	65*ms*	512*ms*
TSFSL	HQ	145*ms*	181*ms*	192*ms*	2.2*s*
(linear)	LQ	50*ms*	49*ms*	68*ms*	579*ms*
TSFSL	HQ	1.3*s*	1.7*s*	1.2*s*	28.1*s*
(mean value)	LQ	343*ms*	226*ms*	325*ms*	6.4*s*
TSFSL	HQ	261*ms*	303*ms*	222*ms*	2.9*s*
(adaptive)	LQ	91*ms*	56*ms*	81*ms*	742*ms*
HAVS [11]		125*ms*	604*ms*	2.8*s*	-
TRC [6]		150*ms*	170*ms*	-	-

Table 2. Performance measures and comparison with state-of-the-art unstructured grid rendering approaches. Rows labeled HQ correspond to high rendering quality (1024x768), and rows labeled LQ correspond to low rendering quality (512x384).

cells. Thus, terminating the ray-casting after one depth peel skips a considerable part of the overall data volume. When working with the mixer data set, the memory requirements for HAVS exceeded the 8 GB of RAM installed in the workstation used for benchmarking. Therefore, no conclusive results were measured due to constant swapping operations by the operating system. When assuming linear scaling in the number of grid triangles, rendering the mixer data set with HAVS should require around 16 seconds.

5.3.3 Interpolation and Image Quality

We have evaluated four different sampling strategies. The fastest is piecewise-constant sampling using cell-centered data. We note that all four test data sets originally contain cell-centered data, and therefore this sampling approach properly represents the original simulation results. However, if smoothed results are desired for presentation purposes, vertex-centered data can be visualized. Our rendering system uses barycentric interpolation at cell faces. Results stated in the rows labeled as "linear" have been generated by using linear interpolation between ray entry and exit positions of a cell. In order to generate higher quality images, one mean-value sample is used at the midpoint between ray entry and exit position. Performance figures for this sampling strategy are given in the rows denoted as "mean value". As can be expected, the complex computations necessary per cell face incur a severe performance hit on the overall rendering time. This is especially prominent for data sets containing general polyhedral cells (heater, manifold, and mixer). The main differences between images generated by the fast "linear" quality setting and the "mean value" option lie in regions where large cells are located (see image quality comparison in Figure 8). Therefore, we propose a fourth "adaptive" sampling strategy that is controlled by a user defined threshold θ. Reconstruction using mean value interpolation is only used if the length of a viewing-ray segment within a cell is greater than θ. Otherwise the much faster linear reconstruction is used within a cell. The user has the option to choose two out of the four sampling strategies in their HQ or LQ configurations for rendering preview images during interaction, and presentation quality visualizations when interaction has stopped.

When compared to HAVS, our high-quality TSFSL methods are only faster for reasonably large data sets such as the cooling jacket. Here, the main limiting factor for HAVS is the number of triangles ($\sim$53Mio), which have to be sorted by the CPU. Also note that the HAVS algorithm generates rendering artifacts on current generation graphics hardware since it performs reading from and writing to the same texture target within one GPU program. This operation results in undefined behavior in OpenGL and in DirectX. The tiled ray-casting approach scales similarly to our low quality volume rendering implementations. However, it requires much more GPU memory (see Table 1), and the publicly available open source implementation has not been able to load the cooling jacket nor the mixer data set (even though a 64 bit binary has been used).

Image Quality: Figure 8 compares the image quality of the different sampling strategies that are supported by our rendering framework. The transfer function used throughout the comparison represents a worst case scenario, which highlights the differences between the sampling methods. This is because it comprises a nearly opaque region and a completely transparent region without a smooth transition in between. Figure 8(a) depicts the fastest sampling strategy for vertex-centered data, which interpolates linearly between ray-cell entry and exit positions. The adaptive sampling method is used in Figure 8(b). The linearization artifacts introduced by the first approach are reduced by incorporating one mean-value sample for ray segments that are longer than a user defined threshold θ. If θ is chosen to be larger than the diameter of the largest cell, then only the linear approximation is used, whereas if θ is set to zero, a mean-value sample is selected for every ray segment. The viewing rays for which mean-value samples have been computed are highlighted in the inset as blue regions. Figure 8(c) shows a result image where one mean-value sample has been used per ray segment within a cell. In order to qualify differences between visualizations we have included difference images that have been scaled up by a factor of 16. They clearly show how linear interpolation and the adaptive sampling strategy differ in regions where mean-value samples have been included. Naturally, the largest differences between the adaptive sampling scheme and always using one mean-value sample show up in regions composed of small cells. The number of mean-value samples per ray segment within a cell is limited to one in all our sampling strategies. Additional samples only marginally add to the overall image quality. This is illustrated in Figure 8(d). Here, two mean-value samples have been computed per ray segment. The difference image between Figures 8 (c) and (d) barely shows any distinguishable features.

6 Conclusions

We have presented a compact, face-based representation for general unstructured grids that does not store redundant information. Our *two-sided face sequence list* (TSFSL) representation is flexible enough to support the efficient traversal of rays from cell to cell, as well as enabling efficient access to all information required for high-quality interpolation, such as mean value schemes. These are the two crucial components required for volume visualization via ray-casting, as well as other important visualization algorithms. In order to demonstrate the feasibility of the TSFSL representation in practice, we have presented a complete GPU-based unstructured-grid ray-casting pipeline. Our system can handle large data sets via bricking, and is able to efficiently cope with both non-convex mesh boundaries and non-convex polyhedral cells, even in the presence of highly non-planar faces.

Acknowledgments

The authors thank Dr. Eberhard Schreck and Dipl.-Ing. Julian Gänz from CD-Adapco for providing us with the polyhedral data sets. The cooling jacket data set is courtesy of AVL List GmbH, Graz, Austria. Parts of this work were funded by the Austrian Research Funding Agency (FFG) in the scope of the project "AutARG" (No. 819352), as well as by the Vienna Science and Technology Fund (WWTF) project "Scale-VS" (No. ICT08-040).

References

[1] Fluent by Ansys. See URL: http://www.fluent.com, last visited 2011.

[2] OpenFOAM. See URL: http://www.openfoam.com, last visited 2011.

[3] Star-CCM+ by CD-adapco. See URL: http://www.cd-adapco.com/products/star_ccm_plus, last visited 2011.

[4] T. J. Alumbaugh and X. Jiao. Compact array-based mesh data structures. In *Proc. of the 14th International Meshing Roundtable, IMR 2005, September 11-14, 2005, San Diego, CA, USA*, pages 485–503, 2005.

[5] H. B. Baumgardt. A polyhedron representation for computer vision. In *Proc. of AFIPS National Conference*, pages 589–596, 1975.

[6] F. F. Bernardon, C. A. Pagot, J. L. D. Comba, and C. T. Silva. GPU-based tiled ray casting using depth peeling. *J. Graphics Tools*, 11(4):1–16, 2006.

[7] I. Boada, I. Navazo, and R. Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17(3):185–197, 2001.

[8] A. Bru and M. Teillaud. Generic implementation of a data structure for 3d regular complexes. In *Abstracts of 24th European Workshop on Computational Geometry*, pages 95–98, 2008.

[9] S. P. Callahan, L. Bavoil, V. Pascucci, and C. T. Silva. Progressive volume rendering of large unstructured grids. *IEEE TVCG*, 12(5):1307–1314, 2006.

[10] S. P. Callahan, J. L. D. Comba, P. Shirley, and C. T. Silva. Interactive rendering of large unstructured grids using dynamic level-of-detail. In *Proc. of IEEE Visualization 2005*, pages 199 – 206, 2005.

[11] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE TVCG*, 11(3):285–295, 2005.

[12] H. Childs, M. Duchaineau, and K.-L. Ma. A scalable, hybrid scheme for volume rendering massive data sets. In B. Raffin, A. Heirich, and L. P. Santos, editors, *Proc. of Eurographics Symposium on Parallel Graphics and Visualization*, pages 153–161, 2006.

[13] R. Espinha and W. C. Filho. High-quality hardware-based ray-casting volume rendering using partial pre-integration. In *Proc. of SIBGRAPI*, pages 273–280, 2005.

[14] C. Everitt. Interactive order-independent transparency, Sept. 01 2001.

[15] M. Floater. Mean value coordinates. *Computer Aided Geometric Design*, 20(1):19–27, 2003.

[16] M. P. Garrity. Raytracing irregular volume data. *ACM Computer Graphics*, 24(5):35–40, 1990.

[17] M. Gross, H. Hagen, and F.-J. Pfreund. Interactive simd ray tracing for large deformable tetrahedral meshes. In *Proc. of IEEE Symposion on Interactive Ray Tracing*, pages 147 –154, 2008.

[18] T. Gurung and J. Rossignac. Sot: compact representation for tetrahedral meshes. In *Proc. of 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pages 79–88, 2009.

[19] A. Helgeland and O. Andreassen. Visualization of vector fields using seed LIC and volume rendering. *IEEE TVCG*, 10(6):673–682, 2004.

[20] T. Ju, S. Schaefer, and J. Warren. Mean value coordinates for closed triangular meshes. In *Proc. of SIGGRAPH 2005*, pages 561–566, 2005.

[21] L. Kettner. Designing a data structure for polyhedral surfaces. In *Proc. of Symposium on Computational Geometry*, pages 146–154, 1998.

[22] M. Kraus and T. Ertl. Cell-projection of cyclic meshes. In *Proc. of IEEE Visualization 2001*, pages 215–222, 2001.

[23] M. Lage, T. Lewiner, H. Lopes, and L. Velho. CHF: A scalable topological data structure for tetrahedral meshes. In *Proc. of SIBGRAPI*, pages 349–356, 2005.

[24] E. LaMar, B. Hamann, and K. I. Joy. Multiresolution techniques for interactive texture-based volume visualization. In *Proc. of IEEE Visualization '99*, pages 355–361, 1999.

[25] R. S. Laramee, C. Garth, H. Doleisch, J. Schneider, H. Hauser, and H. Hagen. Visual Analysis and Exploration of Fluid Flow in a Cooling Jacket. In *Proc. of IEEE Visualization 2005*, pages 623–630, 2005.

[26] B. Lévy, G. Caumon, S. Conreaux, and X. Cavin. Circular incident edge lists: a data structure for rendering complex unstructured grids. In *Proc. of IEEE Visualization 2001*, pages 191–198, 2001.

[27] Y. Lipman, J. Kopf, D. Cohen-Or, and D. Levin. GPU-assisted positive mean value coordinates for mesh deformations. In *Proc. of the Fifth Eurographics Symposium on Geometry Processing*, pages 117–123, 2007.

[28] G. Marmitt, H. Friedrich, and P. Slusallek. Interactive Volume Rendering with Ray Tracing. In *Eurographics State of the Art Reports*, 2006.

[29] G. Marmitt and P. Slusallek. Fast ray traversal of tetrahedral and hexahedral meshes for direct volume rendering. In B. S. Santos, T. Ertl, and K. Joy, editors, *Proc. of the 8th Joint IEEE TCVG - EUROGRAPHICS Symposium on Visualization (VisSym 2006)*, pages 235–242, 2006.

[30] N. Max, P. Williams, C. T. Silva, and R. Cook. Volume rendering for curvilinear and unstructured grids. In *Proc. of Computer Graphics International*, pages 210–215, 2003.

[31] A. Maximo, S. Ribeiro, C. Bentes, A. Oliveira, and R. Farias. Memory Efficient GPU-Based Ray Casting for Unstructured Volume Rendering. In *Proc. of Volume Graphics*, pages 155–162, 2008.

[32] P. Muigg, M. Hadwiger, H. Doleisch, and H. Hauser. Scalable hybrid unstructured and structured grid raycasting. *IEEE TVCG*, 13(6):1592–1599, 2007.

[33] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE TVCG*, 5(3):238–250, 1999.

[34] S. Röttger, S. Guthe, A. Schieber, and T. Ertl. Convexification of unstructured grids. In *Proc. of Workshop on Vision, Modeling and Visualization (VMV 2004)*, pages 283–292, 2004.

[35] N. Shareef, T.-Y. Lee, H.-W. Shen, and K. Mueller. An image-based modelling approach to gpu-based rendering of unstructured grids. In *Volume Graphics*, pages 31–38, 2006.

[36] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Computer Graphics*, 24(5):63–70, 1990.

[37] C. T. Silva, J. S. B. Mitchell, and P. L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *Proc. of VolVis '98*, pages 87–94, 1998.

[38] C. M. Stein, B. G. Becker, and N. L. Max. Sorting and hardware assisted rendering for volume visualization. In *Proc. of VolVis '94*, pages 83–89, 1994.

[39] M. Üffinger, S. Frey, and T. Ertl. Interactive high-quality visualization of higher-order finite elements. *Computer Graphics Forum*, 29(2):337–346, 2010.

[40] H. T. Vo, S. P. Callahan, N. Smith, C. T. Silva, W. Martin, D. Owen, and D. Weinstein. iRun: Interactive rendering of large unstructured grids. In *Proc. of Eurographics Symposium on Parallel Graphics and Visualization*, pages 93–100, 2007.

[41] K. Weiler. Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments. *IEEE Computer Graphics and Applications*, 5(1):21–40, Jan. 1985.

[42] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proc. of IEEE Visualization 2003*, pages 333–340, 2003.

[43] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based view-independent cell projection. *IEEE TVCG*, 9(2):163–175, 2003.

[44] M. Weiler, P. N. Mallón, M. Kraus, and T. Ertl. Texture-encoded tetrahedral strips. In *Proc. of VolVis 2004*, pages 71–78, 2004.

[45] M. Weiler, R. Westermann, C. D. Hansen, K. Zimmerman, and T. Ertl. Level-of-detail volume rendering via 3D textures. In *Proc. of VolVis 2000*, pages 7–13, 2000.

[46] R. Westermann. The rendering of unstructured grids revisited. In *Proc. of the 3rd Joint IEEE TCVG - EUROGRAPHICS Symposium on Visualization (VisSym 2001)*, pages 65–74, 2001.

[47] P. L. Williams. Visibility-ordering meshed polyhedra. *ACM Trans. Graph.*, 11(2):103–126, 1992.

[48] Y. Zhou and M. Garland. Interactive point-based rendering of higher-order tetrahedral data. *IEEE TVCG*, 12(5):1229–1236, 2006.