Volume Graphics (2008), pp. 1–8
H. - C. Hege, D. Laidlaw (Editors)

A Flexible and Memory Efficient Data Structure for GPU-based Polyhedral Grid Raycasting

Philipp Muigg, Markus Hadwiger, Helmut Doleisch

VRVis Research Center, Vienna, Austria

Abstract

The foundation for unstructured grid volume visualization via raycasting is a compact and easily traversable representation of the grid's topology. This data structure is used to query every cell intersecting a viewing ray in order to sample the data volume and accumulate corresponding color and opacity information. Current techniques mainly deal with tetrahedral volumes. However, with the ongoing evolution of simulation technology more complex grid structures containing not only tetrahedral but general convex polyhedral cells have been introduced. We propose a cell face-centered data structure capable of representing such polyhedral grids efficiently while still allowing for fast viewing ray propagation for GPU-based raycasting. By choosing to abandon an explicit cell representation we avoid storing redundant information and allow for a fast and straight-forward addressing scheme within the data structure. The benefits of our approach are demonstrated by comparing it to other state-of-the-art volume rendering methods.

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques

1. Introduction

Unstructured grids are an important volumetric representation that is especially common in the field of computational fluid dynamics (CFD), e.g., simulations of engineering problems computed with finite volume methods. The steady evolution of these methods and the available hardware has led to larger data sets as well as more complex grid structures. While early algorithms were based on purely tetrahedral or curvilinear meshes, state-of-the-art simulation packages such as ANSYS's Fluent [flu] or CD-adapco's Star-CD [sta] are able to cope with more general mesh structures containing arbitrary convex polyhedra. In order to directly visualize scalar data specified on such grids volume rendering can be used. However, most currently available unstructured grid volume rendering approaches are only capable of dealing with purely tetrahedral grids. In previous work, we have introduced a volume rendering framework that extended GPU-based unstructured grid volume rendering methods to convex polyhedral grids. Now, instead of adapting data structures initially used for tetrahedral grids to polyhedral cells, we propose to choose an altogether different approach to grid representation, by abandoning explicit cell representation. Our new data structure stores only minimal information per face to represent the grid topology. This allows for significant memory savings and added flexibility and simplicity, at a reasonable expense in rendering speed.

Previous grid representations used for raycasting have been mainly optimized for fast volume traversal, sacrificing memory and flexibility for speed. The most prominent exception to this observation are tetrahedral strips [WMKE04], which provide the option to save varying amounts of memory by sacrificing rendering performance. The main drawback of this approach when considering polyhedral meshes is that it is only applicable to tetrahedral cells. Our face-based data structure is capable of representing polyhedral cells, and although it also incurs higher computational costs during ray propagation, it significantly reduces both the amount of required memory, as well as the general complexity of the raycasting process. As illustrated by the pseudo code in Figure 6, only a single loop is required in order to propagate a viewing ray through the entire grid. Such kinds of trade-offs have become more important over the past years, especially when considering the evolution of modern GPUs: while raw processing power is increasing rapidly and memory bandwidth is also increasing, the amount of graphics memory is increasing much more slowly.

2. Related Work

When discussing unstructured grid volume rendering methods, two main approaches can be distinguished: image-order and object-order.

Generally speaking, object-order algorithms are based on separately processing individual elements (e.g., cells) of the overall data volume. Shirley and Tuchman introduced the Projected Tetrahedra (PT) technique [ST90], which is the basis of several subsequent approaches. In order to properly composite the grid elements, they have to be rasterized in visibility order. Thus highly efficient sorting algorithms such as Meshed Polyhedra Visibility Ordering (MPVO) [Wil92] or related methods are at the core of most object-order approaches [SBM94, SMW98]. Even though there are extensions of the PT algorithm that allow an approximated rendering of polyhedra [MWSC03], accurate implementations are still limited to tetrahedra. Contrary to this, the Hardware Assisted Visibility Sorting (HAVS) approach introduced by Callahan et al. [CICS05] is, just like our proposed data structure, solely based on face information. By avoiding explicit cell representation, flexibility is added to the grid structure, which can be exploited by fast level of detail and progressive rendering approaches, for example [CCSS05, CBPS06].

In contrast, image-order approaches such as raycasting [WKME03, HSS*05] are usually slower in pure rendering performance than cell-projection techniques, but compensate for this fact by the lack of an explicit sorting step. They are also very flexible, e.g., with respect to adaptive sampling, and are a natural choice when complex non-linear interpolation techniques such as mean value coordinates [JSW05] are desired. Most unstructured grid raycasting algorithms are based on the work of Garrity [Gar90], who uses cell connectivity information to propagate viewing rays through the volume. Weiler et al. presented the first GPU-based implementation [WKME03], which performed the ray propagation by applying a multi-pass approach. In order to reduce the excessive memory consumption, Weiler et al. introduced Tetrahedral Strips [WMKE04], which extends the notion of triangle strips to 3D and adds mesh topology information. Additionally, an approach similar to depth peeling is introduced in order to cope with concave meshes. Other GPU-based methods have been proposed by Espinha and Celes [EF05], and Bernardon et al. [BPCS06], each improving upon the memory consumption of the original algorithm presented by Weiler et al. [WMKE04], but never achieving higher memory efficiency than Tetrahedral Strips.

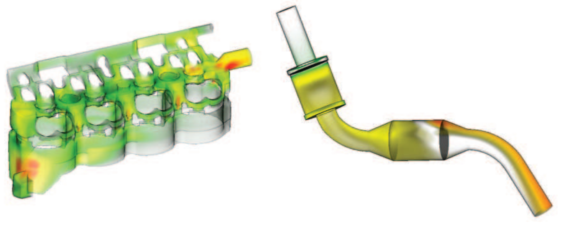


Figure 1: *Two volume renderings of real world data sets which have been used to benchmark our face-based mesh representation. A cooling jacket (denoted as Large Cooling Jacket in the remainder of this paper) is shown on the left and an exhaust system for a diesel engine on the right.*

3. Unstructured Grid Traversal for Raycasting

In order to perform volume rendering of unstructured grids, the data set has to be sampled along viewing rays. For each sample, the grid element (i.e., the cell) containing its location has to be available in order compute interpolated data values at the sample position. But since point location (i.e., the identification of a cell containing a certain point within the grid) cannot be solved analytically in unstructured grids, the ray propagation for a viewing ray through such a volumetric representation always involves the traversal of a data structure representing the topology of the underlying grid. Previous approaches dealing with tetrahedral grids are cell-based. That is, all information necessary for ray propagation through the volume is stored per cell (or tetrahedral strip [WMKE04]). Thus, per-face data such as face normals or vertices are stored redundantly by adjacent cells. Furthermore, for a data volume composed of more general cells such as convex polyhedra, storing topology information per cell introduces the additional problem of varying memory requirements per cell. This either necessitates padding and thus wasting GPU memory, or a more complex addressing scheme [MHDH07]. The data structure presented in this paper avoids these two central drawbacks of cell-based data representation by switching to a face-centered approach.

In order to perform ray propagation during raycasting, only very little information about a cell is actually necessary:

1. The plane equations of all cell faces.
2. The neighboring cells that are connected to the faces.

By using this information, a viewing ray that has been started at a cell on the mesh boundary can be intersected with each face plane of a cell. The closest intersection point (after the ray entry position into the cell) can be calculated. The next cell traversed by the viewing ray is connected through the corresponding face. We propose to abandon the explicit representation of a cell and instead store only face information explicitly. This is possible by employing a data structure that is similar in spirit to the winged edge data structure [Bau75]

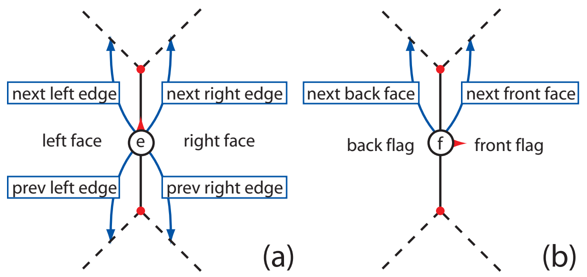


Figure 2: *The winged edge data structure (a) is similar to our face-based approach (b) since it represents an n-D object by using linked $(n-1)$-D primitives. Note that the winged edge stores four links to connected edges, the edge direction, two adjacent cells, and incident vertices in order to allow for different queries. Our approach stores only information necessary for raycasting methods: two links to faces in the adjacent cells, the face orientation (the side indicated by the red triangle is in front of the face), incident vertices, and two flags which are used during the data structure traversal.*

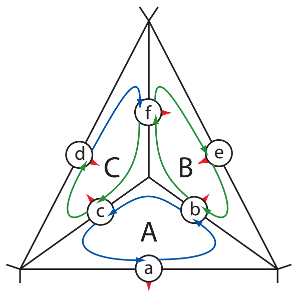
for representation of polygonal meshes. Instead of storing edges to represent 2D grids, we extend the concept to 3D by using face data to represent 3D grids. Contrary to the winged edge data structure, which has been designed to allow for a fast evaluation of different queries, our data structure has been stripped of redundant information and is mainly suited for efficiently traversing all faces of a cell and querying faces of neighboring cells. Although a cell is not represented explicitly, conceptually it consists of a linked list of its faces. Each face can be a part of at most two cells and thus at most two such linked lists. For each face, only its plane equation and two links have to be stored. The first link determines the "next" face of the cell on one side of the face, and the second link determines the "next" face of the cell on the other side. The link fields are only used to retrieve the set of faces that constitute a cell, and thus the order of faces in these lists is arbitrary. It is not possible to always use the "first" link field for retrieving all faces of one cell, and the "second" link field for the other cell. Therefore, when stepping from one face to the next in a given cell, the link to use must be determined. In order to enable this, additional annotation flags have to be introduced (Section 4).

4. A Face-Based Grid Representation

The unstructured grid representation and traversal approach proposed in this paper relies solely on a compact face representation. Faces of one cell can be traversed by following links stored at each face. Similarly, faces of an adjacent cell can be reached again by following different links. This section provides an in-depth description of the data structure itself, its construction based on per-cell topology information, and its traversal during raycasting.

4.1. Data Structure

As already mentioned in Section 3, our data structure is related to the winged edge representation of polygonal meshes. Figure 2 provides a schematic overview of the storage primitives used by both grid representations. The core of the winged edge data structure is the edge representation which stores its orientation and several links to adjacent edges and polygons. This allows for fast evaluation of different queries such as finding all edges of a polygon or all polygons connected by an edge. Our approach extends this data structure to 3D by replacing edges of a polygon with faces of a polyhedron. It should be noted that in 3D the notion of "left" and "right" with respect to an edge is replaced by the notion of "in front" or "behind" with respect to the signed distance to a face. Furthermore, we do not require a face linking to another face to share edges or vertices with that edge. The order in which cell faces are considered is irrelevant for ray propagation. Note that we do require that all faces of one cell are linked in one common circular loop. This is similar to the winged edge structure, which imposes the additional constraint that linked edges have to have a common vertex. Besides this straight-forward translation from 2D to 3D, we also propose to store only face data which is relevant for ray propagation through the unstructured mesh. Thus, instead of storing four links to following and previous faces we store only two links to faces in the adjacent cells. The *front link* connects to a face of the cell in front of the current face and



#	F	B	ϕ	β
a	-	b	-	B
b	f	c	F	B
c	d	a	F	B
d	f	-	B	-
e	b	-	F	-
f	e	c	F	F

Figure 3: *This figure shows how our data structure can be used to represent three connected cells (A, B, and C). The front side of a face is indicated by a red triangle. The links are called front or back links with respect to the side of a face from which they originate. The two flags corresponding to the links are encoded as color. Blue indicates that the next back link has to be chosen when iterating through a cells faces whereas green links imply that the next front link has to be chosen. The table on the right shows the link and flag data which is used to store the three cells. The columns labeled "F" and "B" contain the front and back links respectively. The columns labeled ϕ and β contain the flags associated with the corresponding links (ϕ for front and β for back links). A value of "F" corresponds to a green link whereas a value of "B" to a blue one in the figure on the right.*

```
function TraverseFaces(startFace, startFlag)
    currentFace ← startFace, currentFlag ← startFlag
    repeat                      ▷ This loop iterates over all faces of a cell
        Process face currentFace
        if currentFlag = "F" then
            currentFlag ← GetFrontFlag(currentFace)
            currentFace ← GetFrontLink(currentFace)
        else
            currentFlag ← GetBackFlag(currentFace)
            currentFace ← GetBackLink(currentFace)
        end if
    until currentFace = startFace
end function
```

Figure 4: *This listing demonstrate how all faces of one cell can be visited by following face links. The input parameters are a face startFace where the iteration should be started and a flag startFlag indicating whether the faces of the cell in front or behind of startFace should be traversed. In order to determine whether front or back links should be selected the corresponding flags are used. Note that currentFlag always stores the flag which is stored at the previously visited face in order to make this decision for the currentFace.*

the *back link* connects to a face of the cell behind the current face. Since we do not use explicit cell representations we omit storing links to adjacent cells. In the special case that all cells are consistently either in front or behind of all their faces, these two links are sufficient to iterate through every face of a cell. This can be observed when only considering cells A and B in Figure 3. Here cell A is always behind its faces (a, b, and c) whereas cell B is always in front of its faces (b, e, f). Thus all faces of A can be visited by always following back links (column denoted by "B" in the table in Figure 3). Similarly all faces of B can be visited by always following front links (column denoted by "F" in the table in Figure 3).

In order to cope with cells such as cell E in Figure 3 (faces c and d face towards whereas face f faces away from the cell), additional data has to be stored indicating wether to follow the front or back link of a face. We propose two one-bit flags, each related to one of the face links: ϕ to the front link and β to the back link. They indicate whether to follow the front or the back link when visiting the next face. The link coloring in Figure 3 represents these flags: The flag corresponding to a blue link indicates that the next back link has to be selected (expressed as "B" in the corresponding table). A green link indicates that the next front link has to be chosen when traversing all faces of the corresponding cell (indicated by "F" in the table in Figure 3). Thus iterating through all faces of cell C can be accomplished as follows: starting at face c we follow the front link to face d (we choose the front link since the cell C is in front of face c). The flag ϕ of face c is set to "F" which indicates that the next face of C is stored in the front link of face d. By following this link we reach face f. The flag ϕ of face d indicates that now the back link of f has to be selected. This leads back to face c. Figure 4 provides pseudo code illustrating this face traversal method.

Data Structure Construction: The face-based data structure can be constructed very efficiently if basic cell-centered topology information is available. Per cell only the number of faces, their orientation, and the neighboring cells corresponding to each face are used during the initialization process. Additionally, two temporary vectors storing one entry per cell are used to steer the data structure construction. The vector **F** stores a link to one face of a cell (if at least one face has already been created for this cell) and **G** stores whether the cells are in front or behind the faces referenced by **F**. The overall data structure is created by iterating over every cell c_i. Per cell every neighboring cell c_j is inspected. If $i < j$ we know that c_j has not been visited and that the face connecting c_i and c_j has not been initialized. Thus we have to create a new face f_{ij} (if $i > j$ we assume that all faces of j,

```
function AddFaceToCellList($f_{ij}$, $c_i$, initFlag)
    if $\mathbf{F}_i$ not set then
        $\mathbf{F}_i \leftarrow f_{ij}$, $\mathbf{G}_i \leftarrow$ initFlag
        if initFlag ="F" then
            SetFront($f_{ij}$, $f_{ij}$, "F")
        else
            SeBack($f_{ij}$, $f_{ij}$, "B")
        end if
    else
        if $\mathbf{G}_i$ = "F" then
            face ← GetFrontLink($\mathbf{F}_i$)
            flag ← GetFrontFlag($\mathbf{F}_i$)
            SetFront($\mathbf{F}_i$,$f_{ij}$, initFlag)
        else
            face ← GetBackLink($\mathbf{F}_i$)
            flag ← GetBackFlag($\mathbf{F}_i$)
            SetBack($\mathbf{F}_i$,$f_{ij}$, initFlag)
        end if
        if initFlag = "F" then
            SetFront($f_{ij}$, face, flag)
        else
            SetBack($f_{ij}$, face, flag)
        end if
    end if
end function
function AddFace($c_i$, $c_j$)
    init empty $f_{ij}$ facing towards $c_i$
    AddFaceToCellList($f_{ij}$,$c_i$, "F")
    AddFaceToCellList($f_{ij}$,$c_j$, "B")
end function
```

Figure 5: *This listing demonstrates how to set up the face links and flags when adding one face when initializing the whole grid representation.* AddFace *has to be called for every cell c_i and every neighbor c_j of this cell where $i < j$. Additionally boundary faces of cell c_i have to be initialized by creating the corresponding face f_{ix} and calling* AddFaceToCellList*(f_{ix}, c_i, "F").*

including f_{ij} have already been created) and link it to faces already created for both cells which can be retrieved from $\mathbf{F}_i$ and $\mathbf{F}_j$. The orientation of these faces f_{xi} and f_{yj} with respect to cells c_i and c_j can be queried from $\mathbf{G}_i$ and $\mathbf{G}_j$. Now the links of the new face f_{ij} can be connected to the faces succeeding f_{xi} and f_{yj} in their respective cells. Links of faces f_{xi} and f_{yj} themselves are set to f_{ij}. If $\mathbf{F}_i$ or $\mathbf{F}_j$ has not been initialized until now f_{ij} is placed into the vector. The link related to the uninitialized cell is set to directly lead back to f_{ij}. The exact process of extending the data structure by one face and the related modifications of links and flags of already existing faces is illustrated by pseudo code presented in Figure 5. Here the face insertion process is performed by calling the function ADDFACE which initializes the face data structure connecting cells c_i and c_j and inserts it into the corresponding lists by calling ADDFACETOCELL for each adjacent cell. In order to initialize boundary faces for a cell c_i (i.e., faces contained only by one cell) a new face data structure f_{ix} is initialized facing towards c_i which is then directly inserted into the mesh representation by calling ADDFACETOCELLLIST(f_{ix}, c_i, "F").

4.2. Face-Based Raycasting

Every unstructured grid raycasting approach consists of three basic steps. First, a viewing ray has to be initialized. Then, the ray has to be propagated through the unstructured volume while color and opacity are computed per ray segment. During the first stage, each viewing ray is initialized by rasterizing the volume surface and encoding face indices in multiple color channels. Determining which face is part of the surface is straight forward since the back links of such faces are not initialized. Now the ray propagation through the volume can be started at the boundary faces. As mentioned in Section 4.1 faces are initialized facing towards the cell with the lower index. That is, the back link of boundary faces is not initialized whereas the front link points towards the next face of the first volume cell which is intersected by the viewing ray. In order to determine the next adjacent cell intersected by the viewing ray we have to determine through which face it leaves the current cell. Thus we iterate through all faces of a cell by using the code presented in Figure 4. Per definition the plane equations $\mathbf{n}\cdot\mathbf{x}-d=0$ for each face are initialized such that the face normal n is directed towards a cell located in front of it (i.e., all points of a cell in front of the face have positive distances). When iterating through the faces of a cell c_i some faces might have been initialized facing towards and some away from c_i. This is indicated by the *currentFlag* during the iteration process. If it is set to "F" the face is already facing towards c_i, whereas otherwise it is facing away from the cell. In order to determine whether a viewing ray can exit through a face f_{ij} two cases have to considered:

- When the face is facing towards the cell a ray can only exit through f_{ij} if and only if $\mathbf{n_{ij}}\cdot\mathbf{v}<0$.

```
1: function PROPAGATERAY(startFace, startPos, v)
2:     entryFace ← startFace
3:     currentFace ← startFace, currentFlag ←"F"
4:     minFace ← startFace, minFlag ← "F", minDistance ← ∞
5:     P ← startPos
6:     repeat ▷ This loop iterates over all faces of cells along a ray
7:         if currentFlag = "F" then
8:             currentFlag ←GETFRONTFLAG(currentFace)
9:             currentFace ←GETFRONTLINK(currentFace)
10:        else
11:            currentFlag ←GETBACKFLAG(currentFace)
12:            currentFace ←GETBACKLINK(currentFace)
13:        end if
14:        n ← GETFACENORMAL(currentFlag,currentFace)
15:        d ← GETFACEPARAMETER(currentFlag,currentFace)
16:        if $\mathbf{n}\cdot\mathbf{v}<0$ then
17:            $s \leftarrow (\mathbf{n}\cdot\mathbf{P}-d)/(-\mathbf{n}\cdot\mathbf{v})$
18:            if s < minDistance then
19:                minDistance ← s
20:                minFlag ← currentFlag
21:                minFace ← currentFace
22:            end if
23:        end if
24:        if currentFace = entryFace then
25:            entryFace ← minFace, currentFace ← minFace
26:            if minFlag = "F" then ▷ Use other link at minFace
27:                currentFlag ← "B"
28:            else
29:                currentFlag ← "F"
30:            end if
31:            $\mathbf{P} \leftarrow \mathbf{P}+s\cdot\mathbf{v}$
32:            minDistance ← ∞
33:            Process ray segment
34:        end if
35:    until currentFace is a boundary face
36: end function
```

Figure 6: *This listing demonstrates how a ray starting at startPos with the direction v through the face startFace can be tracked through a data volume. Three distinct blocks can be distinguished: line 7 to 13 deals with loading the next face within a cell (compare to Figure 4), line 13 to 23 intersects the current face with the ray if the intersection point lies ahead along the ray, and line 24 to 34 resets the face traversal to the face for which the ray intersection is closest to the ray entry position after all faces of a cell have been visited.*

- When the face is facing away from the cell a ray can only exit through f_{ij} if and only if $\mathbf{n_{ij}}\cdot\mathbf{v}>0$.

With $\mathbf{v}$ being the ray direction and $\mathbf{n_{ij}}$ the face normal of f_{ij}. This assumption can be proven by examining the following equation. It computes the distance s of a ray entry point into a cell $\mathbf{P}$ to the intersection of the ray with a plane $\mathbf{n}\cdot\mathbf{x}-d=0$ which is part of this cell's surface and facing towards it:

$$s = \frac{\mathbf{n} \cdot \mathbf{P} - d}{-\mathbf{n} \cdot \mathbf{v}} \quad (1)$$

$\mathbf{n} \cdot \mathbf{P} - d$ is always larger or equal zero since $\mathbf{P}$ lies on the surface of the cell. Thus the sign of s is only determined by the term $\mathbf{n} \cdot \mathbf{v}$. If it is positive (i.e., s is negative) we can discard the corresponding face since the intersection with the face does not lie further along the ray. If $\mathbf{n} \cdot \mathbf{v} < 0$ we have to evaluate equation 1. The face for which s is smallest is the face through which the ray exits the current cell. When dealing with faces that are facing away from the current cell, $-s$ is the actual signed distance of the intersection. In order to traverse the faces of the corresponding neighbor cell we simply follow the link we did not choose when iterating over the faces of the current cell. This process is repeated until a volume boundary face is reached. The pseudo code presented in Figure 6 illustrates the overall ray propagation process starting at a volume boundary face *startFace*, at the position *startPos* with the direction $\mathbf{v}$. Lines 7 to 13 retrieve the next face of the current cell (compare to Figure 4). Lines 14 to 23 determine whether the intersection of the ray with the current face is the closest yet and lines 24 to 34 restart the iteration process for the next cell along the ray.

In order to sample the current cell along the viewing ray, mean value interpolation [JSW05] can be used [MHDH07]. This interpolation method is also suited for our face-based volume representation, since the interpolation weights can be computed on a face by face basis.

5. Discussion

Before showing quantitative results which are based on a prototype implementation integrated into a framework which has been presented in previous work we will discuss several considerations related to our approach.

5.1. Data Structure Flexibility

Besides the memory footprint and traversal performance of a data structure the flexibility with respect to grid topology changes is of importance when considering level of detail approaches for unstructured grids [THJW98, CL03]. Tetrahedra based methods have to guarantee that every level of detail representation is still made up of tetrahedra. This is mostly accomplished by progressively applying simplification operations such as edge collapses which remove multiple cells at once and change vertex positions and face orientations in the edge's surrounding. In order to propagate these modifications into previous grid data structures involves the modification of multiple tables: Connectivity information for multiple tetrahedra has to be adapted, face orientations have to be changed, and vertex positions have to be modified and in the case of tetrahedral strips eventually the whole strip generation process has to be repeated. Since our data structure can store polyhedral cells, other ways of merging multiple cells without the often undesirable modification of surrounding structures can be explored. For example merging two adjacent cells simply works by modifying the corresponding links and flags of two faces (i.e., removing the connecting face from both cells). Since there is no explicit cell representation this operation does not involve any additional data structure update.

Tex. Format	Usage	
4 Byte RGBA	front lnk	back lnk
2 Byte Lum. Alpha	front/back flg	front/back lnk
4 Byte RGBA	vtx id 1	vtx id 2
4 Byte RGBA	vtx id 3	vtx id 4
16 Byte F RGBA	plane equation	
Sum: 30 Byte		

Table 1: *This table shows how our face-based data structure can be packed into five textures. Note that the 20-bit face indices stored for the front and back links are split into a 16-bit part and a 4-bit part, which are stored in separate textures.*

It should be noted that the proposed ray propagation approach is currently only applicable to convex cells which means that currently cell merging would only be allowed if the resulting cell remains convex. However, an extension of our propagation method to concave cells should be possible in future work which would remove this restriction (the interpolation method currently used is already capable of interpolating data within concave polyhedra).

5.2. Implementation

In order to evaluate our face-based data structure, we have integrated it into a GPU based unstructured grid volume rendering framework. This framework currently subdivides an unstructured grid into bricks containing at most 64K cells and vertices in order to allow for memory savings when storing cell and vertex addresses. Our implementation uses 20 bits to address faces within a brick, which allows for a maximum of 1,024K faces. This restriction is feasible, since the number of overall faces n within an unstructured grid can be computed as:

$$n = \frac{c \cdot f_{avg} + f_{bnd}}{2} \quad (2)$$

With c being the number of cells, f_{avg} the average number of faces of all cells and f_{bnd} the number of boundary faces. Even in a worst case scenario where every cell within the volume has only boundary faces and $c = 64K$ and $n = 1,024K$ the average number of faces f_{avg} is as high as 16. This is sufficient especially when considering that our overall framework currently deals only with a selected number of polyhedral cell types which are limited to eight faces. Thus all data representing one face can be packed into a 30-byte structure as shown in Table 1. The 20-bit face links are split into 16-bit and 4-bit parts, which are stored in two separate textures. Note that we store four 16-bit vertex indices for every face in order to directly represent quads without having to triangulate them.

5.3. Performance

The comparison of our old cell based memory layout and the new face-based version is shown in Table 2. Here two reasonably large unstructured data sets have been used: The Large Cooling Jacket contains mostly hexahedra whereas the Generator consists of a mixture of tetrahedral and hexahedral cells. The memory requirements (rows entitled "Mem.") specified in the table represent the overall texture memory allocated to store the data necessary for volume rendering. This also includes cells and faces which are shared between bricks and thus have to be stored multiple times. Our new data structure requires roughly 40% less memory for the Large Cooling Jacket and 46% less for the Generator data set when compared to our previous speed optimized data structure. In order to relate these results to the memory requirements of other GPU based raycasting approaches we have added the number of tetrahedra which would result from two different tetrahedra decompositions. The first decomposition (shown in columns labeled "LQ") minimizes the number of resulting tetrahedra whereas the second decomposition (shown in columns labeled "HQ") provides a higher quality tetrahedralization by introducing vertices at the cell centers. This latter decomposition yields results comparable to our mean value interpolation based approach. Now memory consumption per tetrahedron can be computed and compared to other memory layouts. The original approach proposed by Weiler et al. [WKME03] uses 160 bytes per tetrahedron, whereas later cell based methods as proposed by Bernardon et al. [BPCS06] or Espinha and Celes [EF05] use 144 bytes and 96 bytes, respectively. When comparing these numbers to the results presented in Table 2 it becomes apparent that even when assuming the low quality tetrahedralization far less memory is required. The (memory optimized) tetrahedral strip method [WMKE04] is the only approach which is more efficient in most cases since it uses only roughly 14 bytes per tetrahedron.

	Generator		Large Cooling J.	
# Cells	6,730K		1,538K	
# Tets	LQ 15,406K	HQ 32,939K	LQ 7,149K	HQ 17,044K
	Cell Centered			
Mem.	1,330 MB		318 MB	
per Tet	LQ 86 bytes	HQ 40 bytes	LQ 44 bytes	HQ 19 bytes
	Face Centered			
Mem.	714 MB		192 MB	
per Tet	LQ 46 bytes	HQ 22 bytes	LQ 27 bytes	HQ 11 bytes

Table 2: *This table presents a comparison of the memory consumption of our previous cell centered data structure and our new face-based approach. In order to relate these results to other state of the art raycasting methods we have included the number of tetrahedra for two different tetrahedral decompositions of the data sets.*

In order to compare the rendering speed to our previous cell centered approach we tested several data sets which could wholly fit into GPU memory. The number of cells of these data sets ranges from 12K up to 1,538K for the Large Cooling Jacket. On average the face centered data structure performed 37% worse than the performance optimized cell centered approach. This corresponds nicely to the roughly 40% memory savings which can be achieved by using our new method. The main cause for the slower rendering performance is the need to traverse the faces of a cell by following the face links. This introduces one additional indirection per face, which can only be avoided if face data is replicated for each adjacent cell and stored tightly packed in an array (this is the case in our cell centered data layout).

6. Conclusions and Future Work

We have presented a new face-based data structure for the representation of polyhedral grids which allows for significant memory savings by sacrificing a reasonable amount of rendering speed. This is accomplished by abandoning an explicit cell representation which often introduces data redundancy as can be observed in previous tetrahedra based methods (e.g. face normals are stored twice for cells in the interior of the volume). Besides the memory savings the simplicity and flexibility of this new data structure is highly important. Because cell merging and splitting can be accomplished by modifying only a few links new level of detail approaches based on merging multiple cells into polyhedra become feasible. In order to provide quantitative results we have presented a prototype implementation which has been integrated into a brick based unstructured grid volume rendering system.

There are several interesting areas for future work. The improved flexibility of our approach can be used to research new ways to level of detail representations. For example multiple levels of detail could be easily stored within one data structure without introducing unreasonable memory overhead by allowing multiple front and back links per face. Furthermore our current ray propagation method can be extended to concave polyhedra which would allow for even more freedom when dealing with grid topology modifications. Finally we will try to further improve the traversal speed of our data structure. Here for example storing faces of some cells consecutively within our lookup textures would improve cache coherency and eventually avoid additional indirections.

References

[Bau75] BAUMGARDT H. B.: A polyhedron representation for computer vision. In *Proceedings AFIPS National Conference* (1975), vol. 44, pp. 589–596.

[BPCS06] BERNADON F. F., PAGOT C. A., COMBA J. L. D., SILVA C. T.: Gpu-based tiled ray casting using depth peeling. *journal of graphics tools 11*, 4 (2006), 1–16.

[CBPS06] CALLAHAN S. P., BAVOIL L., PASCUCCI V., SILVA C. T.: Progressive volume rendering of large unstructured grids. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1307–1314.

[CCSS05] CALLAHAN S. P., COMBA J. L. D., SHIRLEY P., SILVA C. T.: Interactive rendering of large unstructured grids using dynamic level-of-detail. In *IEEE Visualization '05* (2005), pp. 199–206.

[CICS05] CALLAHAN S. P., IKITS M., COMBA J. L. D., SILVA C. T.: Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3 (2005), 285–295.

[CL03] CHIANG Y.-J., LU X.: Progressive simplification of tetrahedral meshes preserving all isosurface topologies. *Computer Graphics Forum 22*, 3 (Sept. 2003), 493–504.

[EF05] ESPINHA R., FILHO W. C.: High-quality hardware-based ray-casting volume rendering using partial pre-integration. In *SIBGRAPI* (2005), pp. 273–280.

[flu] Fluent. See URL: http://www.fluent.com.

[Gar90] GARRITY M. P.: Raytracing irregular volume data. *ACM Computer Graphics 24*, 5 (1990), 35–40.

[HSS*05] HADWIGER M., SIGG C., SCHARSACH H., BUHLER K., GROSS M.: Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum 24*, 3 (2005), 303–312.

[JSW05] JU T., SCHAEFER S., WARREN J.: Mean value coordinates for closed triangular meshes. In *Proceedings of SIGGRAPH 2005* (2005), pp. 561–566.

[MHDH07] MUIGG P., HADWIGER M., DOLEISCH H., HAUSER H.: Scalable hybrid unstructured and structured grid raycasting. *IEEE Trans. Vis. Comput. Graph 13*, 6 (2007), 1592–1599.

[MWSC03] MAX N., WILLIAMS P., SILVA C., COOK R.: Volume rendering for curvilinear and unstructured grids. In *Proc. of Computer Graphics International* (2003), pp. 210–215.

[SBM94] STEIN C. M., BECKER B. G., MAX N. L.: Sorting and hardware assisted rendering for volume visualization. In *Proc. IEEE Symposium on Volume Visualization '94 (VolVis '94)* (1994), pp. 83–89.

[SMW98] SILVA C. T., MITCHELL J. S. B., WILLIAMS P. L.: An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *Proc. IEEE Symposium on Volume Visualization '98 (VolVis '98)* (1998), pp. 87–94.

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

[sta] Star-cd. See URL: http://www.cd-adapco.com.

[THJW98] TROTTS I. J., HAMANN B., JOY K. I., WILEY D. F.: Simplification of tetrahedral meshes. In *IEEE Visualization* (1998), pp. 287–295.

[Wil92] WILLIAMS P. L.: Visibility-ordering meshed polyhedra. *ACM Trans. Graph. 11*, 2 (1992), 103–126.

[WKME03] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-based ray casting for tetrahedral meshes. In *Proceedings IEEE Visualization 2003* (2003), pp. 333–340.

[WMKE04] WEILER M., MALLÓN P. N., KRAUS M., ERTL T.: Texture-encoded tetrahedral strips. In *Proc. IEEE Symposium on Volume Visualization 2004 (VolVis 2004)* (2004), pp. 71–78.