# The VSBUFFER: Visibility Ordering of Unstructured Volume Primitives by Polygon Drawing

Rüdiger Westermann, Thomas Ertl

Computer Graphics Group
Universität Erlangen-Nürnberg, Germany*

## Abstract

Different techniques have been proposed for rendering volumetric scalar data sets. Usually these approaches are focusing on orthogonal cartesian grids, but in the last years research did also concentrate on arbitrary structured or even unstructured topologies. In particular, direct volume rendering of these data types is numerically complex and mostly requires sorting the whole data base. In this paper we present a new approach to direct rendering of convex, voluminous polyhedra on arbitrary grid topologies, which efficiently use hardware assisted polygon drawing to support the sorting procedure. The key idea of this technique lies in a two-pass rendering approach. First, the volume primitives are drawn in polygon mode to obtain their cross-sections in the VSBUFFER orthogonal to the viewing plane. Second, this buffer is traversed in front-to-back order and the volume integration is performed. Thus, the complexity of the sorting procedure is reduced. Furthermore, any connectivity information can be completely neglected, which allows for the rendering of arbitrary scattered, convex polyhedra.

## 1 Introduction

Scalar volume data can be visualized by many different methods which strongly depend on the intended application and on the topology of the grid on which the data samples are given. In direct volume rendering techniques the object is supposed to be filled with a semi-transparent gel which is rendered according to the physics of light transport [12, 5, 6]. A variety of techniques have been developed for the direct rendering of 3D scalar data fields on regular cartesian grids [9, 10]. Apparently, the underlying theory is well understood and several approaches offering combinations of rendering speed and image quality have been established [20, 2, 8, 16, 21, 7].

Recently, hardware assisted direct rendering methods benefiting from real-time 3D texture interpolation [1, 19] produce remarkable frame rates. However, these approaches are restricted to rectilinear grids, and it seems questionable whether they can be adapted to other grid topologies. On the other hand, in order to directly render volume data generated by numerical simulations or adaptive refinement strategies the visualization of irregular or even unstructured grids becomes a major challenge. Since the emerging grid types and cell primitives can be of arbitrary topology the development of rendering algorithms which are not restricted to a certain class of application is an important goal. Simultaneously, the overhead spent in such a universal framework should not dominate the overall rendering times.

In this paper we present a new approach to the visualization of scalar data fields which are available on unstructured grid topologies. Our method has the following basic advantages:

- ***Arbitrary Topologies:*** All kinds of convex polyhedra are possible candidates to be rendered. This even holds if the primitives are available on disconnected or non-convex grids.
- ***Efficient Visibility Ordering:*** The ordering of primitives with respect to the present viewing definition is accelerated by hardware assisted polygon drawing, thus reducing the complexity of the entire rendering process.
- ***Memory Optimization:*** An intermediate representation, the **VSBUFFER**, is generated during rendering. This data structure allows direct inspection of primitives along a line of sight. Connectivity information, often needed by traditional methods, can be completely neglected. On the other hand, if connectivity information is available it can be easily integrated in order to optimize the rendering algorithm.

This is achieved by a two-pass rendering approach which takes advantage of hardware assisted polygon drawing available on a variety of different platforms. First, the data base is successively rendered into a buffer orthogonal to the viewing plane and parallel to the active scanline. Second, this buffer, which now holds the cross-sections between the primitives and the plane, is traversed. Since the cross-sections are already in correct order, the sorting of primitives within each scanline can be completely avoided. The final rendering pass is performed by casting rays along each line of sight through the detected data cells. Consequently, arbitrary integration rules can be applied, and these rules can also be adapted to the underlying cell structure.

We should note here that only the sorting of primitives according to the actual viewing definition and the determination of intersection points between rays and cells is accelerated by this approach. No optimization of the integration process itself will be achieved.

Our method is very similar to the ones proposed by Giertsen [4], Silva [14], Wilhelms [22], and Yagel [24]. All three approaches utilize the coherence within cutting planes in object space and thus reduce the 3D problem to a 2D problem within each plane. Furthermore, the sorting procedure within each plane can be reduced to the sorting of primitives along each line of sight. However, the fundamental difference to our method lies in the way in which we obtain the relevant information within each plane.

Definitely the most critical part when rendering unstructured grids is to determine the visible ordering of the available primitives. In general, for each modification of the viewers line of sight the whole data base has to be sorted to obtain the sequence of elements which have to be inspected in correct order. Since the underlying sorting procedure immensely effects the complexity of the complete rendering process, it is not astonishing that much work has been spent on efficiently integrating or extending available sorting algorithms [11, 23, 3]. Additionally, specialized graphics hardware has been used to improved the rendering process [13, 15, 24]. Drawing each primitive as transparent polygon structure results in a very fast but less accurate approximation of realistic volume effects.

In the following sections we first discuss the new sorting procedure and the way in which we take advantage of hardware assisted

*Lehrstuhl für Graphische Datenverarbeitung (IMMD9), Universität Erlangen-Nürnberg, Am Weichselgarten 9, 91054 Erlangen, Germany, Email: `wester@informatik.uni-erlangen.de` polygon drawing. The basic concept of the VSBUFFER and the resulting rendering strategy are described. Then, results are given and the run times are analyzed. We conclude with some ideas for future work.

## 2 The VSBUFFER

Basically, there are two possible strategies which can be used to optimize the complexity of rendering algorithms for unstructured grids. First, one can try to optimize known sorting algorithms or the emerging data structures in order to accelerate the sorting procedure [11, 23]. Second, explicitly sorting the whole data base can be avoided at the expense of less costly and possibly hardware supported operations [13, 15, 18]. We decided to choose the second strategy which will be outlined in the following.

### Sweep-Planes

A well known approach for the visualization of unstructured grids is the use of **sweep-planes**. A sweep-plane belongs to a certain scanline of the screen. It is a plane in object space that is perpendicular to the viewing plane and parallel to that scanline (see Figure 1). According to this definition a sweep-plane is uniquely described for each scanline.

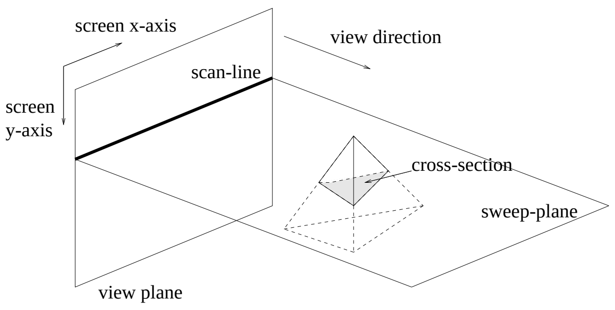

Figure 1: Overview of sweep-plane approach.

Probably the first approach to the application of sweep-planes in direct volume rendering came from Upson et al. [17]. They introduced the V-Buffer to efficiently render volume cells. Although this approach was not specifically designed for the rendering of arbitrary grid types, cutting planes in object space were introduced to improve scanline based cell projections. In this respect the V-Buffer might be seen as a forerunner to a whole number of sweep-plane based approaches for the visualization of arbitrary grid topologies.

The key idea in rendering unstructured grids with sweep-planes is to reduce the complexity of the sorting procedure by restricting the number of primitives which have to be sorted for each scanline [4, 14]. This is achieved by first determining those primitives which are potential candidates for an intersection with the actual plane. Thereupon these primitives are sorted with respect to the chosen viewing transformation. Thus, the 3D problem is reduced to a 2D problem within one sweep-plane.

Now the critical part is to find the right order of objects with respect to their intersections with these planes. Computing the sequence of primitives which are hit consecutively by arbitrary rays emanating from a scanline is a non-trivial task.

Note, that the intrinsic problem is to find the cross-sections between the primitives and the sweep-planes in correct order. Assuming that the coverage of a sweep-plane with all cross-sections has been determined, then the problem collapses to the traversal of the 2D plane for each ray of sight emanating from the corresponding scanline [4]. This is outlined in Figure 2 where the coverage of a sweep-plane according to a number of primitives is shown. In the following we will call the buffer that holds all the cross-sections for a certain scanline the VSBUFFER.

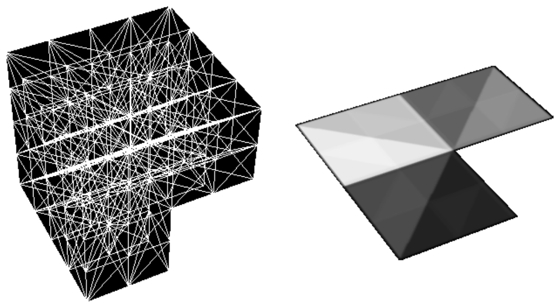

Figure 2: Coverage of sweep-plane with cross-sections.

But it is not yet clear how to obtain this information without sorting and projecting the whole data base. This will be demonstrated in the next section, where we introduce an elegant method to compute the coverage of a sweep-plane with the cross-sections of primitives. Once the coverage is computed and stored in a buffer the shooting and traversal of rays in object space collapses to a simple walk through the regular domain of that buffer.

### Buffered Polygon Drawing

In the following we will make two assumptions. First, the polyhedra are build up of several triangles and stored in a linear list. Second, an orthographic projection is chosen to generate the final images. Thus all scanlines can be processed using the same viewing direction, and consequently all sweep-planes have the same orientation. In case of perspective projections the orientation of sweep-planes changes with each scanline. Although perspective projections can be handled by our approach, we will focus on orthographic views to simplify the discussion.

In order to process one scanline after another we temporarily change the viewing parameters, thereby performing an orthographic projection perpendicular to the sweep-plane. The position of the observer moves above the scene, thus looking down onto the sweep-plane with a viewing direction parallel to the former screen y-axis. Of course, this can also be managed by rotating the objects in the opposite direction. Since we are in global orthographic mode, the intermediate projection with respect to the sweep-plane orientation does not change while processing arbitrary scanlines.

Our goal is to determine the coverage of the VSBUFFER with all cross-sections of primitives which intersect the buffer. This is similar to rendering the whole scene into the buffer with a viewing direction orthogonal to it, and with a front and back clipping plane set to the actual sweep-plane. This process can be repeated simultaneously for each scanline. What should remain visible in each buffer are the cross-sections with the rendered objects. However, because of two reasons this can not be achieved in general. First, standard graphics libraries as OpenGL would produce incorrect or probably empty results due to rounding errors. Second, even if it would be possible to choose the same front and back clipping plane only the surrounding polygons of each cross-section remain visible in the buffer. Thus a different method needs to be developed which solves this problem.

The key idea is to render the primitives twice into different buffers. Both times only one clipping plane is enabled which is set to the orientation of the actual sweep-plane. We are always looking

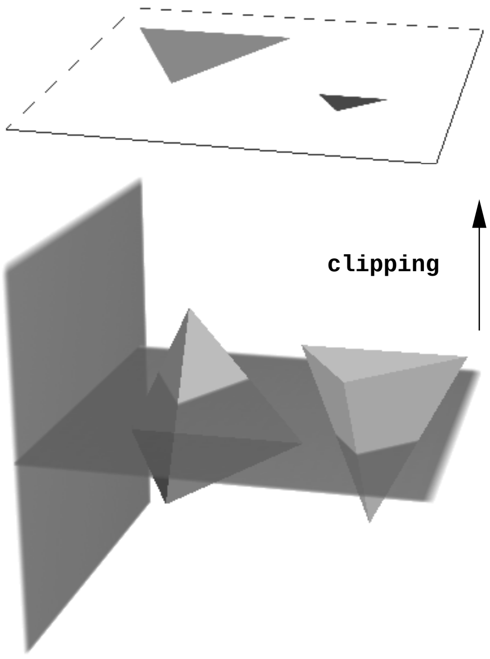

Figure 3: First rendering pass: We are looking from above. Objects above the sweep-plane are clipped. Back faces are drawn.

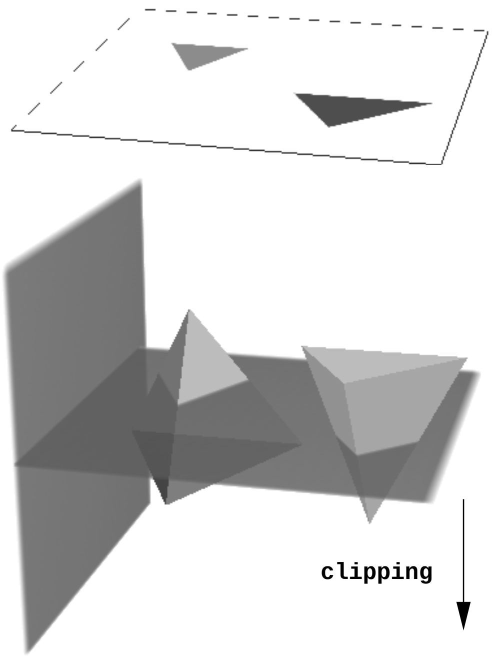

Figure 4: Second rendering pass: We are looking from above. Objects below the sweep-plane are clipped. Front faces are drawn.

from above down to the sweep-plane, and we classify each face of a polyhedra whether it is a front or a back face with respect to this intermediate viewing direction. In the first pass only **back facing** polygons of the convex polyhedra are drawn and we clip everything away that is in front of the clipping plane. Depth test is set to **less**. This means that those objects behind the clipping plane which are nearest to the plane are visible. In the second pass the normal of the clipping plane is reversed. Now everything behind the plane is clipped. Additionally, only **front facing** polygons are drawn and depth test is set to **greater**. Again, only those objects in front of the plane are drawn which are closest to it. Both passes are performed with disabled lighting, flat shading, and no anti-aliasing to avoid averaging pixel values. Both steps are outlined in Figures 3 and 4.

Since the polyhedra are stored in a linear list each of them can be assigned an ID which uniquely defines that cell within the entire data base. This ID is used as the color of the faces of each primitive which is drawn. For example, if we have a RGB-visual with 8 Bits per color channel we can code $2^{24}$ primitives in the RGB color values. Coding the IDs of objects in this way allows their direct identification from the pixel values already drawn.

After both rendering passes the footprints of the drawn faces are uniquely coded by the color values available in the used buffers. Unfortunately it is impossible to obtain the correct cross-sections by considering only one buffer. Depending on the orientation of primitives the cross-sections in either of both buffers will cover more area than is really occupied. In order to find corresponding pixel values which have the same color, i.e. where the same object was drawn to that location, both buffers have to be compared to each other. The result of the comparison is stored in the VSBUFFER. In Figure 5 the intermediate buffers after both rendering passes and also the resulting VSBUFFER are shown.

At this time the reader might wonder whether all primitives have to be drawn in both rendering passes although they possibly do not have an intersection with a certain sweep-plane. It will be described below how this can be avoided.

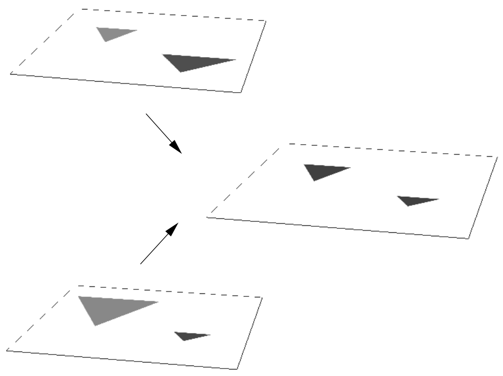

Figure 5: Result of buffer comparison.

The basic idea is once more demonstrated in Figure 6 along one line of sight. A 2D slice through the 3D scene is simulated that shows the generated intervals on a ray passing through the primitives. As a result the ray is broken down into distinct segments each of which indicate that a certain cell was hit.

## Building the buffer

The set of pixels which have the same color value in both buffers exactly determines all cross-sections between a sweep-plane and the polyhedra. The correct result is obtained by comparing pixel values at the same location in both buffers. This is completely done in software by walking through the regular domain of the buffers.

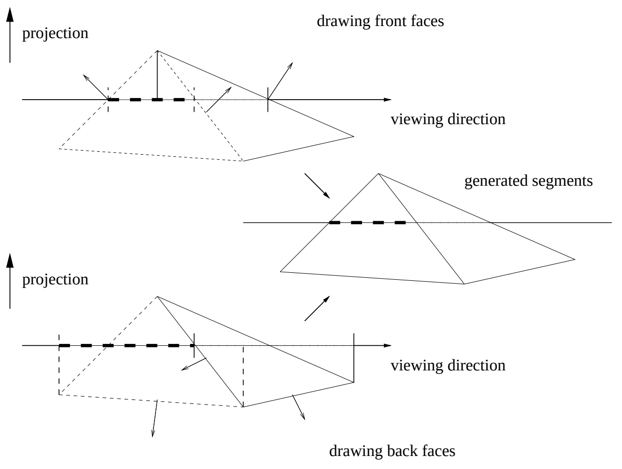

Figure 6: Clipping along one ray of sight.

All that remains to be specified is how to access the buffer values. After each rendering pass the framebuffer extent that was affected by the actual intermediate orthographic projection is read into main memory. Prior to processing a new sweep-plane the comparison of pixel values starts and the result is stored in the VSBUFFER. An unsigned integer value is allocated for each pixel. Either the pixel is set, then the color value uniquely identifies a tetrahedron that was hit, or the pixel value is zero.

Since the viewing transformation that was applied to render the objects twice into distinct buffers was carefully adjusted to the original coordinate system, the IDs of cells which are hit along one ray of sight emanating from a scanline is now exactly given by one pixel column in the VSBUFFER. In order to traverse a ray and to determine all intersections between that ray and the cell primitives we just have to walk through the discrete buffer thereby testing its values.

Note the basic idea we propose. Sorting the data base is traded in against successive rendering of parts of the data base into the VSBUFFER. Sorting is given up at the expense of multiple polygon drawings and framebuffer read operations. However, the result is the same: A unique ordering of primitives within each sweep-plane according to the viewing specification.

Due to the fact that the sorting of primitives is completely neglected the complexity of our approach is reduced. All that needs to be done is to successively render the primitives into the used buffers. Furthermore, the algorithm neither depends on the grid topology nor on the connectivity between the available objects. Each cell is treated as an separate atom processed independently of all others. All we require is that the objects are convex. Additionally, the grid structure can be disconnected or even concave. This will not influence the algorithm.

Of course, in order to minimize the amount of polygons passed through the graphics pipeline we have to determine the set of primitives contributing to a certain scanline in advance. Otherwise we will draw the whole data base for every sweep-plane. We address this problem by building a tree structure over all primitives which allows us to efficiently determine those cells which have an intersection with the actual scanline. This will be described in more detail below.

## 3 Volume Integration

Once the VSBUFFER for a certain scanline is build the rendering of volume primitives starts. In order to integrate volume effects like emission and absorption within the voluminous primitives the volume rendering integral

$$I = \int_{t_0}^{t_1} q(t) e^{-\int_{t_0}^{t} \alpha(s) ds} dt$$

is evaluated for each ray that intersects that primitive. Thereby, the volume source term $q(t)$ and the absorption $\alpha(s)$ are obtained by interpolating and mapping the material values given at the objects vertices via a transfer function. Traditionally the continuous integral is approximated by discretization. The intervals a certain ray passes through a cell are split into equally spaced segments for which the material properties are assumed to be constant.

Now, for each ray of sight emanating from a scanline the VSBUFFER is traversed along the corresponding pixel column. Each time a color value is determined it indicates that the ray hits a certain primitive. The values in the buffer directly correspond to the ID of the objects. Then, the buffer is traversed until a different ID is identified.

Once we find a cell that is hit by a certain ray of sight we start computing the entry and the exit points along that ray. This is managed by first determining the front and back faces of the cell and by testing the scalar product $\vec{N} \cdot \vec{D}$ between the normal of each face and the viewing direction. The entry point is the furthermost intersection point to the viewpoint of all front faces, while the exit point is the closest one of all back faces.

The intersection points are computed in barycentric coordinates of each triangle that is hit. We assume that each polyhedra is build from a number of triangles. Given three vertices $\vec{U}, \vec{V}, \vec{W}$ that define a triangle, a point within the triangle can be expressed by $P(u, v, w) = u\vec{U} + v\vec{V} + w\vec{W}$, with $u + v + w = 1$ and $0 \leq u, v, w \leq 1$.

The positions of intersection points along the ray are computed by clipping the ray against each face of a cell that is hit. The scalar product $\vec{N} \cdot (\vec{S} + t\vec{D} - \vec{T})$ has to be zero, where $\vec{S}$ is the start point of the ray and $\vec{T}$ an arbitrary vertex of the triangle to be clipped. Since we know that the ray has an intersection with the primitive, otherwise we would not have determined the ID of the object, no testing is performed whether the point is inside or outside the primitive. Thus, we obtain the intersection point $\vec{P}$ and solve the resultant system of equations to get its representation in barycentric coordinates:

$$\begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} P_x \\ P_y \\ P_z \\ 1 \end{bmatrix}$$

In order to be on the safe side ($det(\vec{U}\vec{V}\vec{W}) \neq 0$) we first translate $\vec{P}$ and the triangle vertices along a fourth vertex that does not lie in the triangle plane. For example, if the primitives are tetrahedra this point is the unused vertex. Since the system of equations is over-determined we replace the last row of the system matrix with the row that contributes the least information, delete the last row and solve the $3 \times 3$ system. This row can be found by determining the maximum component of the normal vector to the actual face, i.e. if all triangle vertices lie in the y-z plane then we replace the first (x) row in our system.

Rounding errors are avoided by accounting for the condition $0 \leq u, v, w \leq 1$. We only try to find an intersection point if the ID of a primitive was found in the VSBUFFER. Consequently, we can be sure that there really is an intersection. In the case of barycentric coordinates outside $(0, 1)$ we slightly shift the point inside the triangle by adjusting the wrong coordinate.

If the data set consists of tetrahedra we can efficiently use this representation to perform the volume traversal. Since it is also possible to express a certain point within a tetrahedron in barycentric coordinates of its four vertices, and it also holds that $u+v+w+q = 1$ and $f = u\vec{U} + v\vec{V} + w\vec{W} + q\vec{Q}$, where $\vec{Q}$ is the fourth vertex, we can easily interpolate between the entry and the exit point. Since for each point on one of the tetrahedron faces one coordinate equals zero and the others have been computed, a linear interpolation between both representations with arbitrary step size can be applied. For other types of polyhedra the same intersection routine can be used but the integration has to be adapted accordingly.

Particularly, for tetrahedron we follow the ideas in [11] assuming a linear range of material values within each cell. A trapezoidal rule where we also account for attenuation effects in the objects delivers good results. Of course, higher order integration methods can be applied.

## 4 Implementation Details

We implemented the proposed technique using the OpenGL graphics library. The basic algorithm is supplied in pseudo code notation in Figure 7. The basic code is very short and can be implemented quite easily. Only the traversal routine has to be adapted to the available grid cells. It can be clearly realized that the kernel of the algorithm neither depends on the grid topology nor on the structure of the underlying cells. As long as the volume primitives can be split into distinct triangles the same algorithm can be applied and only the traversal routine needs to be changed.

```
// rotating the objects
glRotatef(-90.0, xview[0],xview[1],xview[2]);
// set the orthographic projection with adjusted parameters
glOrtho(left,right,top,bottom,front,back);
for (each scanLine ) {
    glCullFace(GL_FRONT);
    glDepthFunc(GL_LESS);
    updateClipPlane(clipPlaneEquation);
    glClipPlane(GL_CLIP_PLANE0, clipPlaneEquation);
    drawObjects(facesList);
    glReadPixels(0,0,x,y,RGBA,BYTE,frontMem);
    glCullFace(GL_BACK);
    glDepthFunc(GL_GREATER);
    clipPlaneEquation[0,1,2] = -clipPlaneEquation[0,1,2];
    glClipPlane(GL_CLIP_PLANE0, clipPlaneEquation);
    drawObjects(facesList);
    glReadPixels(0,0,x,y,RGBA,BYTE,backMem);
    VSBUFFER = compare(frontMem,backMem);
    traverse(VSBUFFER,facesList);
}
```

Figure 7: OpenGL pseudo code for VSBUFFER generation.

In the following we will have a closer look on different optimization strategies and some of the emerging difficulties and limitations.

### Performance tuning

So far, a major limitation of the proposed method is the enormous number of triangles passed through the graphics pipeline during generation of the VSBUFFER. For each scanline the whole data base is drawn twice into the framebuffer. This requires a huge amount on hardware resources, and it is indeed the bottleneck for large scale data sets.

The problem can be attacked in two principle ways. First, a data structure is generated in which all faces are stored which contribute to the active scanline that is processed. Each time the viewing coordinate system is changed the entire list of vertices is converted from object space to screen space and sorted with respect to the actual view y-direction. From the sorted list those cells can be obtained which contribute to the actual scanline by successively updating a y-active list which holds all active vertices for the scanline. A very complete description of this procedure can be found in [4, 14, 22]. The second approach, as proposed in [22], uses a k-d tree that is build over all primitives. At every node of the tree the bounding box structure for all primitives inside the node is stored. Polygons are inserted at the node deepest in the tree that completely contains the polygon. Each time another scanline is processed the tree is traversed and the cells at each node that have an intersection with the scanline are drawn.

While in the first approach the complete list of vertices has to be transformed and ordered each time the viewing system changes, the k-d tree which is build in a pre-processing step remains unchanged for arbitrary views. The disadvantage is the additional amount of memory and computation. The whole tree has to be stored, and for each scanline the tree traversal takes place checking for possible intersections between the bounding box of the nodes and that scanlines. Only if the scanline intersects a bounding box all polygons stored at that node are drawn.

On the other hand, the tree structure allows us to choose the optimal depth of the tree. Thus we can try to find the most effective setting with respect to the computational load, the used memory, and load resulting from drawing operations. In particular, it turned out that even if we work with a rather flat tree the additional number of drawing operations does not effect the final rendering times dramatically. Consequently, we implemented a regular octree equally partitioning the underlying domain.

The tree structure allows us to completely avoid the sorting of primitives. Instead, each cell within a node that seems to have an intersection with the actual scanline is directly rendered into the intermediate buffers. The correct order is found implicitly by walking through the generated VSBUFFER.

The drawing routine was further improved by computing the facing of all primitives with respect to the actual sweep-plane orientation in advance. Thus, a certain triangle is never rendered twice for the same sweep-plane. Since each volume cell commonly overlaps many scanlines this phenomenon will probably occur quite often.

We should also mention that the use of clipping planes in the OpenGL code can be completely avoided. All we have to do is to call *glOrtho* twice for each scanline with the near and far values adapted to the actual clipping plane equation. Since we are always looking orthogonal to the sweep-plane we just have to restrict the visible region in world coordinates. One time the near value is set equal to the distance to the sweep-plane and the other time this is done for the far value. This implies that we generate a new projection matrix twice for each scanline, but on the other hand, our timings showed that almost twice the time for the drawing pass was consumed with enabled clipping planes. Additionally, restricting the distance between the near and the far values to the largest size of available primitives also optimizes the drawing pass, since faces which are completely outside the visible region are not drawn. Actually, this method is used in our implementation.

### OpenGL issues

In the present approach possible sources of error are vertices which belong to multiple cells. If the same vertex is drawn twice for different triangles we can not be sure that the color value at this location really belongs to the object we expect. The IDs of objects we find in the intermediate buffers depend on the drawing order. This is sketched in Figure 8 where the circle indicates a possible incorrect color value along a ray. However, in case of wrong drawing order the VSBUFFER at this point will be zero since the wrong object is only drawn once. Thus the problem can be easily solved be skipping over that point and traversing the ray until the next correct ID is found. During ray traversal a list is updated which stores the IDs

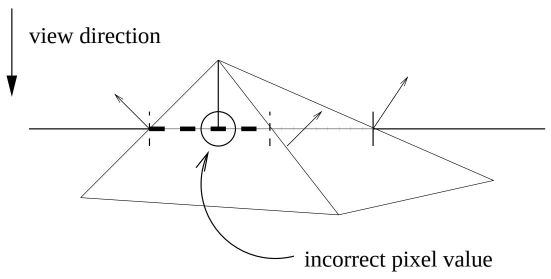

Figure 8: Clipping along one ray of sight.

already found along the actual ray. If a segment along the ray is split into two distinct segments due to incorrect drawing order we will check whether that primitive has already been traversed by inspecting the list. Therefore, multiple integrations along the same object will never occur.

### Buffer resolution and anti-aliasing

As already pointed out in [4, 14, 22] the resolution of the VSBUFFER plays a critical role. Two relevant problems should be further discussed. First, the resolution of the buffer with respect to the size of the involved primitives. Second, the handling of aliasing artifacts due to point-sampling of triangles during the drawing procedure.

The first problem is indeed difficult to handle, but two things should be mentioned here. In order to obtain accurate results the sampling frequency and thus the resolution of the used buffers has to be adapted to the coverage of the smallest available primitives. In general, this is impossible even if the cell sizes vary in the order of 1:10000. As a consequence small objects will be missed. On the other hand, we do have the same problem in all visualization methods for unstructured grids, either they are object-space or image-space driven. The accuracy of ray-tracing variants depends on the screen sampling frequency and also projection methods will definitely lead to incorrect results without multipoint-sampling.

The main problem in our approach is that the resolution of the VSBUFFER possibly not suffices to detect all cells which are hit by a certain sweep-plane. We try to solve the problem by choosing different buffer resolutions in the x- and y-direction. Of course, in the x-direction it is fixed by the actual viewport definition. However, we adjust the resolution in y with respect to the bounding box of the underlying grid and the size of details we want to be able to detect. If the resolution exceeds the available framebuffer extent we split the VSBUFFER in several distinct parts and perform the entire procedure for each of these parts separately.

Since we render multiple sweep-planes orthogonal to the active viewport, we need a separate render area in which the cell primitives can be drawn. Choosing the back buffer has several disadvantages. First, objects already drawn into the back buffer would be destroyed. Second, the actual render area could be too small or could be overlay-ed by other windows, which results in wrong color values read from the buffer. Therefore, we decided to use the `SGIX_pbuffer` extension which provides a part of the physical framebuffer which can be directly accessed by the graphics hardware, but which is not displayed on the screen. Furthermore, p-buffers can be locked exclusively by a certain application against other access.

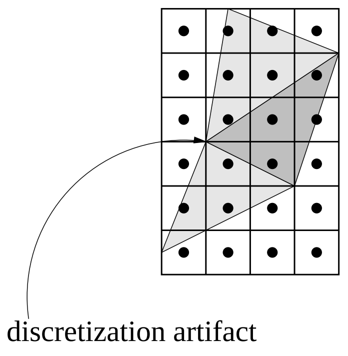

Figure 9: Possible discretization artifact.

Discretization errors as shown in Figure 9, where the dark triangle overlaps a certain pixel column but is not drawn in that column, can be solved efficiently. These kinds of errors can only occur at the border between two objects. Each time we leave a primitive we check its direct neighbors in the VSBUFFER for an intersection. Errors as outlined can be completely avoided in this way.

### Implicit use of connectivity

One important issue of our approach is the capability to handle unstructured grids even if no connectivity information is given. At first glance we loose performance since we do compute the entry and the exit intersection point for each primitive. In order to improve the procedure we always store the last exit point and it's barycentric coordinates with respect to the face that is hit along the ray. For the next cell along the ray we first check the faces belonging to that cell. If it has a face in common with the last cell and the ray left the last object on that face, then we can use the previous exit point as the new entry point. The numerical complexity is thus reduced by a factor of two. Of course, if explicit connectivity information comes with the geometric representation of the used grid, further acceleration of the final rendering can be achieved.

### Perspective projection

Our approach easily extends to perspective projections. All which has to be done is to modify the final projection matrix before primitives are drawn to the intermediate buffers. An additional rotation of 90 degrees around the screen x-axis is applied after the perspective distortion. If all vertices are in normalized device coordinates and we multiply the projection matrix with the rotation matrix we have all primitives in the right orientation. Also, the sweep-planes must be inclined according to the chosen perspective for each scanline. The angle of rotation changes incrementally from one scanline to another.

## 5 Results

All result were computed on a Silicon Graphics Indigo$^2$ IMPACT with 250 Mhz R4400 processor. The experiments were run on three tetrahedra data sets: An artificial data set (ell64) that should aid comprehension, a finite-element data set (fedata), and the tetrahedralised NASA bluntfin data set (bluntfin). The image size was 450x450 pixels. The resolution of the VSBUFFER was chosen equal to the image resolution; the long main diagonal of the data sets is mapped to 450 pixel.

Two parts of the algorithm were investigated separately. First, we measured the elapsed time that was consumed in the graphics pipeline (**GrOps**) including polygon drawing and reading the framebuffer into local memory. Second, we analyzed the CPU time spent for the calculation of intersection points and the final volume integration (**Int**). The corresponding images are shown on the color page below. For each data set we also give the total number of primitives and the total number of drawn primitives. Of course, the number of drawn primitives increases the original amount to some extent. This is due to the fact that, in general, one cell overlaps multiple scanlines. Additionally, at each node of the used tree structure some primitives which are stored might be drawn although they do not contribute to a certain scanline.

To demonstrate the complexity of the basic algorithm we processed the entire 450 scanlines for each data set, not depending on whether the object really had a cross-section with the corresponding sweep-plane or not. All times were measured without explicit use of connectivity information. Table 1 shows explicit timings and the number of involved primitives for all data sets.

**Table 1: Processed primitives and timings (seconds) for the used data sets.**

| | #Tetra | #DrawnTetra | GrOps | Int | Total |
|---|---|---|---|---|---|
| ell64 | 44689 | 954317 | 9.2 | 36.6 | 46.9 |
| fedata | 99306 | 1498032 | 12.4 | 42.1 | 56.0 |
| bluntfin | 224874 | 2102153 | 16.3 | 42.3 | 61.6 |

Note that **GrOps** exactly specifies the time that was needed to completely determine the visibility ordering of all primitives. This is a major improvement over other techniques. For example, in [14] it was reported that it took approximately 48 seconds to completely order the bluntfin data set containing 190000 primitives with respect to 200 scanlines.

A detailed comparison concerning the time used for the integration process seems to be difficult because it strongly depends on the applied intersection routine, the sampling frequency, and finally the kind of volume compositing that was performed. In [24] it took roughly 40 seconds to render the smaller bluntfin data set with hardware supported compositing of volume cells. Slicing the data set with 350 slices parallel to the view plane would yield almost the same result as with the actual resolution of the VSBUFFER in our configuration.

However, we should mention that the times used for the integration within volume cells can be improved considerably. Actually, for each primitive within a certain scanline all calculations are performed from the scratch, including normal calculations and solving the system of equations to obtain the intersection points in barycentric coordinates. By computing implicit descriptions for the available tetrahedra in advance the overall times can be accelerated to some extent.

In the total times the time needed to traverse the tree data structure scanline by scanline is included. Prior to each new rendering pass we first transform the coordinates of the bounding boxes at the nodes to screen space. In this way we can easily check whether a node has an intersection with the actual scanline or not.

Since the finite element data set and the artificial data set have much more intersections with scanlines than the bluntfin data set this reflects in the measured integration times. On the other hand, the time spent in the graphics pipeline grows almost linearly with the number of primitives, if we take into account a constant offset of 5.6 seconds that is consumed for reading and clearing the framebuffer and the depth buffer.

For the bluntfin data set we build a tree structure of depth 5. The additional memory usage was 2.6 MB. Building the tree in advance took 1.4 minutes.

## 6 Conclusion

We have presented a novel technique for hardware assisted sorting and rendering of convex non-overlapping polyhedra which are cells of arbitrary unstructured grid topologies. No connectivity information between primitives is needed, but if primitives are connected this can be used efficiently to accelerate the intersection procedure. The underlying grid types can be convex, non-convex or even disconnected.

The timing results have shown that this method significantly accelerates the rendering process by completely avoiding the sorting of the data base. In contrast to other methods the complexity of the actual approach is thus bound by the numerical intersection calculations and the volume integration.

Since the kernel of the program is a volume ray-casting variant it offers highest flexibility in the chosen visualization options and the image quality. The code is quite easy to implement and runs on standard architectures supporting OpenGL.

Different visualization modes and new types of volume primitives will be integrated in the future. As long as the cell primitives are disjunct and can be split into triangle lists we can use the same sorting procedure. Only the integration rule has to be adapted to the underlying topology.

Furthermore, we think about a parallelization of the algorithm in the style of the SGI Performer toolkit. The whole drawing of primitives is done by a specific processor, but when the intermediate buffers are generated multiple processors are used to build the VSBUFFERS for different scanlines. Of course, this implies that the processing order of the scanlines is chosen in a predefined way, avoiding multiple access of different processors to the same primitives. However, since we do not take advantage of spatial coherence between sweep-planes the processing order can be chosen arbitrarily.

## 7 Acknowledgements

Thanks to Alex Dietz for providing the p-buffer code, and Paolo Cignoni who made the tetrahedralized bluntfin data set available to us.

## References

[1] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In A. Kaufman and W. Krüger, editors, *1994 Symposium on Volume Visualization*, pages 91–98. ACM SIGGRAPH, 1994.

[2] J. Danskin and P. Hanrahan. Fast Algorithms for Volume Ray Tracing. In *1992 Workshop on Volume Visualization*, pages 91–98, 1992.

[3] M. Garrity. Ray tracing irregular volume data. In Kaufmann A., editor, *1990 Workshop on Volume Visualization*, pages 35–40. ACM SIGGRAPH, 1990.

[4] C. Giertsen. Volume visualization of sparse irregular meshes. *Computer Graphics and Applications*, 12(2):40–48, 1992.

[5] J. T. Kajiya and B. P. Von Herzen. Ray Tracing Volume Densities. *Computer Graphics*, 18(3):165–174, July 1984.

[6] Krüger, W. The Application of Transport Theory to the Visualization of 3-D Scalar Data Fields. In A. Kaufman, editor, *Visualization 90*, pages 273–280, Los Alamitos, CA, 1990. IEEE, IEEE Computer Society Press.

[7] P. Lacroute and M Levoy. Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transform. *Computer Graphics, Proc. SIGGRAPH '94*, 28(4):451–458, 1994.

[8] D. Laur and P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *Computer Graphics*, 25(4):285–288, July 1991.

[9] M. Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37, March 1988.

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

[11] N. Max, P. Hanrahn, and R. Crawfis. Area and Volume Coherence for Efficient Visualization of 3D Scalar functions. *Computer Graphics*, 24(5):27–33, November 1990.

[12] P.A. Sabella. A Rendering Algorithm for Visualizing 3D Scalar Fields. *Computer Graphics*, 22(4):51–58, August 1988.

[13] P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *San Diego Workshop on Volume Visualization, Computer Graphics*, 24(5):63–70, December 1988.

[14] C. Silva, J. Mitchell, and A. Kaufman. Fast rendering of irregular grids. In *1996 Symposium on Volume Visualization*, pages 15–23. ACM SIGGRAPH, 1996.

[15] C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In A. Kaufman and W. Krüger, editors, *1994 Symposium on Volume Visualization*, pages 83–90. ACM SIGGRAPH, 1994.

[16] T. Totsuka and M. Levoy. Frequency Domain Volume Rendering. *Computer Graphics*, 27(4):271–78, August 1993.

[17] C. Upson and Keeler M. V-BUFFER: Visible Volume Rendering. *Computer Graphics*, 22(4):59–64, August 1988.

[18] A. van Gelder and J. Wilhelms. Rapid exploration of curvilinear grids using direct volume rendering. In *Visualization 1993*, pages 70–78. IEEE, IEEE Computer Society Press, 1993.

[19] A. Van Geldern and K. Kwansik. Direct Volume Rendering with Shading via Three-Dimensional Textures. In R. Crawfis and Ch. Hansen, editors, *1996 Symposium on Volume Visualization*, pages 23–30. ACM SIGGRAPH, 1996.

[20] L. Westover. Footprint Evaluation for Volume Rendering. *Computer Graphics*, 24(4):367–376, August 1990.

[21] J. Wilhelms and A. van Gelder. Multi-dimensional trees for controlled volume rendering. In *1994 Symposium on Volume Visualization*, pages 27–35. ACM SIGGRAPH, 1994.

[22] J. Wilhelms, A. van Gelder, P. Tarantino, and J. Gibbs. Hierarchical and parallelizable direct volume rendering for irregular and multiple grids. In *Visualization 1996*, pages 57–65. IEEE, 1996.

[23] P. Williams. Visibility ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):102–126, 1992.

[24] R. Yagel, D. Reed, A. Law, P. Shih, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *ACM Symposium on Volume Visualization*, pages 55–63. ACM SIGGRAPH, 1996.

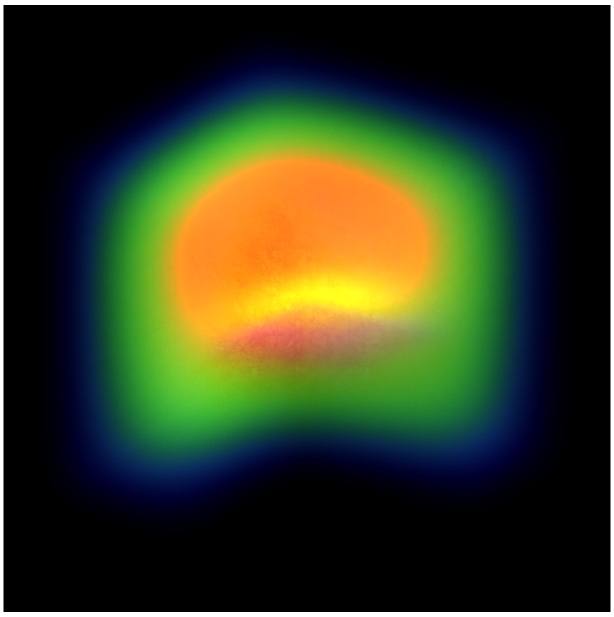

Figure 10: The concave finite-element data set.

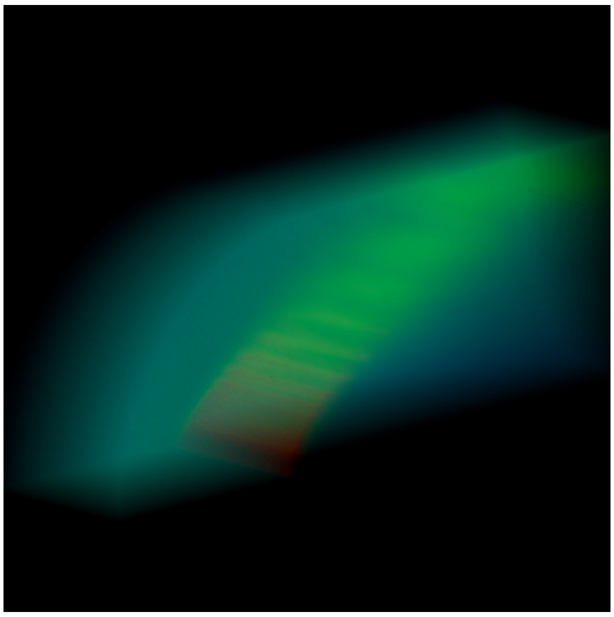

Figure 12: The NASA bluntfin data set.

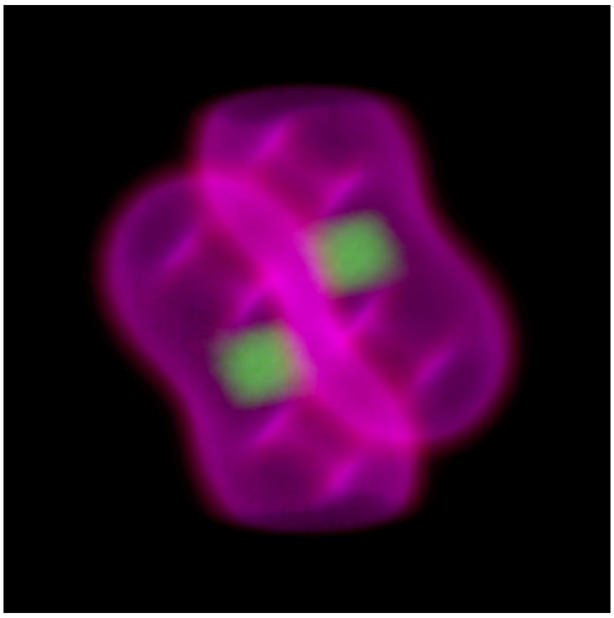

Figure 11: The concave and disconnected artificial data set.

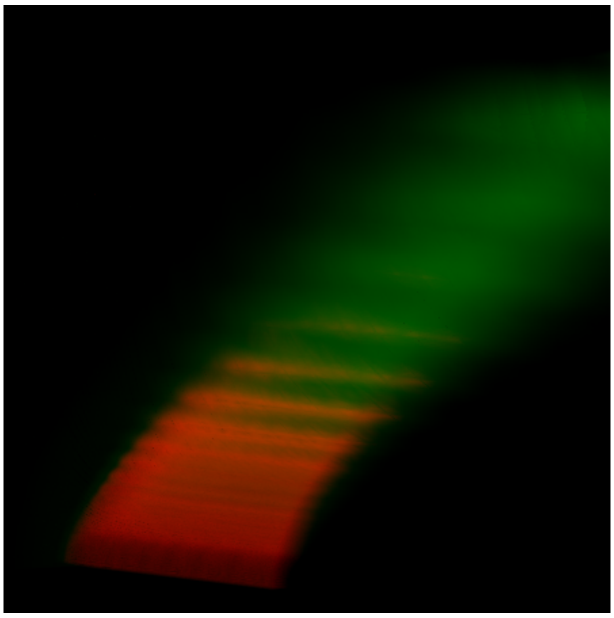

Figure 13: A detailed view of the bluntfin data set.