Computer-Aided Civil and Infrastructure Engineering **18** (2003) 121–131

Fast Volume Rendering and Cutting for Finite Element Model

Xiaosong Yang

Department of Computer Science and Technology, Tsinghua University, Beijing 100084, People's Republic of China

&

Yuanxian Gu, Yunpeng Li & Zhenqun Guan

State Key Laboratory of Structure Analysis for Industry Equipment, Department of Engineering Mechanics, Dalian University of Technology, Dalian 116024, People's Republic of China

Abstract: *Based on the incremental slicing, a direct volume rendering algorithm for finite element meshes is proposed. To improve the algorithm efficiency, the element compensation method and sliced polygon forming method are presented. The image quality is improved greatly by the hybrid rendering of both the volume data and geometric surfaces. We also integrate a volume cutting method with arbitrary convex polyhedron into the volume rendering pipeline. This offers the user more flexibility to explore the data distribution inside structures.*

1 INTRODUCTION

Existing direct volume rendering methods can be classified into the ray-tracing, the element projection, and the hybrid methods. The principal problem of the ray-tracing method (Garrity, 1990) is that it is very time-consuming in the intersection computing. The principal difficulty of the projection method (Shirley and Tuchman, 1990) is the sort operation of all elements. Even using the algorithm of Max et al. (1990) and Williams (1992), it still cannot solve the sort problem involving loops. Two major algorithms are included in the slicing-based hybrid method. The method of Giertsen (1992) and Silva and Mitchell (1997) utilizes a set of planes that is parallel to scan-line to intersect with structures. This approach suffers from low image quality because of the sampling in image space. Another method is Roni Yagel's incremental slicing (IS) (Yagel et al., 1996) algorithm. The cutting plane set is parallel to the image plane. It had better image quality as the result of sampling in object space. Because hardware is used to render polygons, it is faster than existing methods.

The new algorithm presented in this article is based on the IS method. To achieve better image quality and real-time response, four major improvements have been proposed:

1. On the basis of the element-type description table (ETDT) recording the topology of various types of elements, a fast uniform element-slicing algorithm is given out to solve the bottleneck in IS. The slice polygon forming operation in IS consumes nearly 90% of the computing time of the main loop.
2. Reduce the slicing number. The analytical graph in IS shows that the time of rendering increases linearly with the slice number. To achieve satisfactory quality at the least slices can also decrease the total rendering time. But a problem emerged during the decreasing of slices. More and more small elements are neglected by falling entirely in the gap. Sometimes these elements include important value range, such as high stress in finite element analysis,

*To whom correspondence should be addressed. E-mail: *yxs@vis.cs.tsinghua.edu.cn*.

which should have been highlighted. An important function is applied to classify these elements, and a special element compensation method is utilized to solve this problem.

3. The volume rendering is originally developed to handle rectangular grids. In this kind of data, the geometrical information is contained implicitly and is always stressed out by iso-surface or other methods. But in the finite element model, the obviously supplied geometrical information is always neglected. It is too hard to recognize the exact position of high-stress concentration from the vague image. The hybrid algorithm given in this article renders the volume data and geometrical faces together. The facets supposed to have certain transparency are integrated with volume slices in front-to-back order. The image quality is improved by means of implementing the facet rendering by the OpenGL shading models.
4. Although the volume rendering presents more information in the final image than the surface rendering, occlusion of important information in the back is still a big problem. To give the user more flexibility to explore the most important part by using volume rendering, sets are defined in the 3D space to differentiate the importance of different parts in the structure. We can define different transfer functions and even absolute transparency for each set. In this paper, a stencil-based cutting method with convex polyhedra is integrated into volume rendering pipeline. To achieve real-time requirement, several improvements are proposed.

2 ELEMENT TYPE DESCRIPTION TABLE

The large-scale data set of finite element model always needs a compact storage method to reduce the memory consumption and improve algorithm efficiency. An element set data structure shown in Figure 1 is used in this article. To reduce duplicate data, the topological information for element-face-edge-node is extracted and only a type flag is stored with a node list for each element set. The topological information of various element types is stored in ETDT, as shown in Figure 2. Because each operation is based on ETDT, the algorithm is applicable to various element types as long as the correct ETDT is supplied.

3 MODIFIED SLICING-BASED VOLUME RENDERING ALGORITHM

A set of equidistant planes parallel to the projection plane is used to slice the structure elements. The final image is generated by the integration of the projected slices in front to back order. This algorithm consists of the following steps:

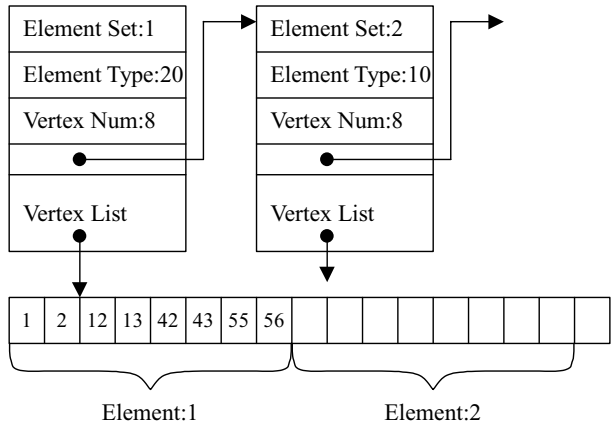


Fig. 1. Element set data structure.

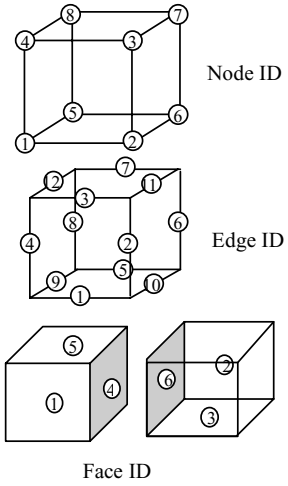
1. Initialization: Transform the grid points from object-space to image-space, slice number computation, out-faced facets' extraction
2. Do this for each slice
3. Updating active edge, element, facet list
4. Small element classification and compensation
5. Out-faced facet rendering
6. Sliced polygon forming
7. Rendering of sliced polygon and integration

The algorithm will be described in detail in the following section.

3.1 Initialization

3.1.1 Coordinate transformation. The transformation of all nodes' coordinates has several acceleration methods, as a common matrix is used for all nodes. In IS the OpenGL graphics library in feedback mode is used to implement the transformation by hardware. But due to the clipping process in OpenGL pipeline, some elements have to be neglected even when only one node is lying outside the clipping box. And another memory cache for temporary storage and an additional function call for each node make it unworthy of the acceleration. Therefore, the software implementation of coordinate transformation is used in the implementation.

3.1.2 Out-faced facet extraction. Extract all the faces of elements in the structure. In the extraction process, only three node IDs are stored for each facet, because if two facets have three nodes in common, there are inner faces that will be deleted in the following bucket sorting. The left list is the out-faced facet. To reduce memory consumption, only an element ID and face index in the




```
Element Type                    = 9
Description                     = Block element of 8 Nodes;
Element Type No                 = 10;
Element Node Number             = 8;
Element Line Number             = 12;
Element Line Node Order =
  Line  1 =      1,     2,
  Line  2 =      2,     3,
  …
Element Surface Number          = 6;
Element Surface Node Number = 4;
Element Surface Node Order      =
  Surface  1 =      1,     2,     3,     4,
  Surface  2 =      5,     8,     7,     6,
  …
```

Fig. 2. Element set data structure.

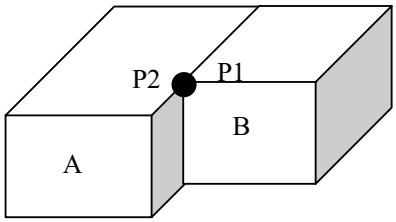


Fig. 3. Different point ID, same coordinate.

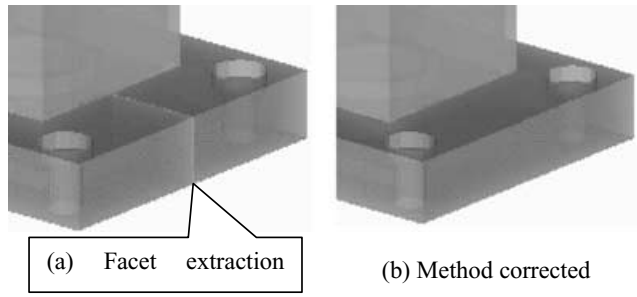


Fig. 4. Example of out-faced facet extraction.

element are stored with the facet. The topology of these facets can be regenerated from the ETDT.

This method works well in most cases. But sometimes the facets extracted are not always out-faced. During the finite element mesh generation, a structure is normally separated into several parts to create points (Figure 3). On their common face, there are always two points owning the same coordinates. Because of their different IDs, they are treated as different points in the previous method. As a result, the common face is left out to be an out-faced facet (Figure 4a).

A pretreatment step is added before the volume rendering loop to solve the problem. Combine the points with the same coordinate to have only one point ID. The result of the corrected method is shown in Figure 4b.

But this combination leads to another problem (Figure 5). When different parts of the structure have different point densities, two adjacent facets may have three points in common and the last one is different. The three-points method is invalid in this situation. This always happened in the common face of different parts.

The extraction method is modified as follows:

1. Find the point with the minimum ID from all the element faces.

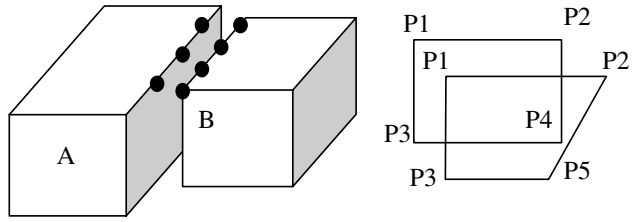


Fig. 5. Three nodes do not determine a common face.

2. Compare the first ID of various faces; if different, they are definitely not common faces. This can save a lot of unnecessary computation of full sort.
3. If the first ID is the same, go on with the second ID. If it fails, try to compare point IDs in the reversed order (i.e., the fourth ID). The third and the last ID comparison will continue with the same order as the second.

This method not only solves the extraction problem but also reduces the maximum compare number of two facets from 16 to 11. In most cases when two facets are

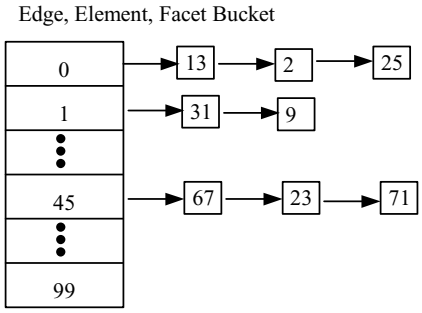


Fig. 6. Edge, element, out-faced facet bucket data structure.

not common, the compare is needed only once. But the previous method needs a full permutation of vertex IDs of every face. For each facet, the compare number reduced to half, as only the minimum ID is needed in the new method.

3.1.3 Construction of edge, element, facet bucket data structure. The bucket data structure is shown in Figure 6. Take edge bucket as an example. For each slice, only one bucket containing an edge list is stored. There are two insertions for each edge according to its two nodes, one is in-bucket, another is out-bucket.

The bucket number is given by bucket $=(Z - Zmin)/\Delta Z$. For element and facet, the bucket number is determined by the maximum and minimum buckets of its nodes. As only ID is needed by bucket, the total storage of bucket structure is given by $(EdgeNum + ElementNum + FacetNum) \cdot 2$, which will remain constant as slice increases. In the computations of edge, element, and facet buckets, lots of node calculations are redundant. Thus, the bucket ID could be computed only once for each node. The bucket number of edge, element, and facet can be deducted directly from related nodes defined in ETDT.

3.2 Updating active edge, element, facet table

The active table, just as in the scan-line polygon-filling algorithm, records the intersection information between edge (element, facet) and current slice. The slice increases with the step of ΔZ from Zmin to Zmax. All the IDs stored in current bucket will be traversed. Out-edge will remove corresponding structures from the active list. For in-edge, a new structure including interpolated coordinate and data value is inserted into the active list. Active element list only stores the element ID. The intersection between element and slice can be found from the active edge list. The processing of the active facet list is more complicated and will be described in Section 3.4.

3.3 Small element classification and compensation

In the updating of the active element list, in- and out-bucket for some elements may be the same. These elements are neglected for falling between two slices. The number increases as the sampling slices step becomes larger. This leads to less accurate images.

To solve the problem, adaptive slicing method was used in IS algorithm to keep the number of missed polyhedrons under control. Each element has the same weight of importance during the consideration. However, the frequency of the data distribution and significance of the data value in the application are not constant for all elements. This treatment equally without discrimination causes unnecessary slice computing added in favor of cutting down the total number of missed elements.

The algorithm proposed here provides users with a new mapping function—an important function to give an important weight for different elements. The element passed through the threshold will fall into a list for compensation during the slice rendering. Different mapping functions will be defined for actual physical meaning of the application. For stress value in finite element analysis, higher stress location should give more attention. A simple two-valued function is enough in this situation to emphasize unexpected high-stress concentration.

During the implementation of the compensation algorithm, the data field can be approximately assumed varying linearly on the element edge because of its small size. The rendering method used here is the traditional element projection model. The small element compensation will be integrated into the image between two slices' renderings.

3.4 Out-faced facet rendering

The facet ID, bit-based flag, and two intersection points are needed for active facet list. Different from active edge and element, intersection with both of the two adjacent slices is needed in forming the cutting part of the facet. The intersection with the previous slice should be stored in active facet structure. The intersection with current slice can be fetched from active edge list. The difficulty here is the forming of the cutting polygon by two slices. Figure 7 shows four relationships between slices and facet:

1. In- and out-bucket are the same; the facet falls entirely between two slices, just rendering the facet directly.
2. This active facet is just inserted from current bucket. There is no intersection with previous slice. Nodes can be classified as in and out on the basis of their bucket number. The intersecting node of the formed polygon can be got from the corresponding active

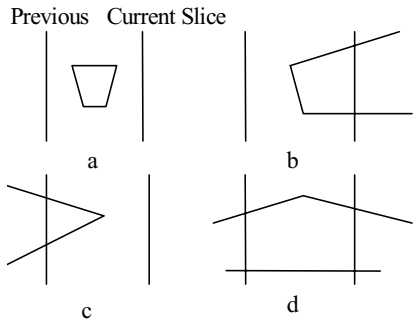


Fig. 7. Relative position between facet and two adjacent slices.

edge. All the nodes should be sorted in the order provided by ETDT to form the cutting polygon. To tackle the next slice, two intersection points should be recorded in the active facet list.

3. The current bucket is out-bucket for this facet. The processing method is same as (2) except the intersection information is stored in active facet list.
4. Facet intersect with both of the two adjacent slices. Four intersection points can be got in the same way as (2) and (3). The polygon can by generated from the intersecting point and node falling inside.

To simplify computing, an independent OpenGL shading model is used to render facets. Then the result is integrated into final image with a definite transparency.

3.5 Sliced polygon forming

To reduce memory occupation, ETDT is used to record the topology information of various element types. The algorithm of sliced polygon forming based on ETDT is applicable to all kinds of element types, providing correct ETDT is given.

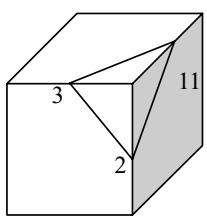
1. *Set node flag*. Allocate one bit of memory for each node, valued 1 or 0. It stands for the relative position to the slice. It can be got directly from the comparison between current bucket and node bucket number.
2. *Computation of intersecting point*. Traverse all the edges of the element; if the flags of two nodes of an edge are different, an intersection occurred between this edge and slice, and this edge must be in the active edge list. The coordinate of intersection can be fetched from the active list.
3. *Forming of polygon*. Figure 8 shows the generated ETDT with the active edge flags. The flag 1 represents the edge intersection with the current slice. Taking the first intersecting point from ETDT as the first node of the polygon, then find another intersecting edge on the same face with the previous one from the edge table of ETDT. Its intersecting point becomes the second. Search the table in the same way until returned to the first node (Figure 8). It can be proved that this method can form the correct sliced polygon, provided the element is a convex polyhedral that is guaranteed by the algorithms of finite element mesh generation.

3.6 The form of sliced polygon in case of concave element face

8-nodes block element is a commonly used element type. In some cases, the 4 nodes of some element faces cannot stay in the same plane. For the two elements sharing this face, one face must be concave (Figure 9a). To the previous algorithm, each edge of the concave face may have an intersection with the slicing plane (i.e., there are four 1s in the same row of ETDT). The algorithm cannot select the correct next face in the scan of intersected edge. A preprocessing step is needed to preclude this case in order to form the correct sliced polygon.

In fact, there are two possibilities in this situation (see Figure 9a and Figure 9d). The forming process is different for them. If wrong forming methods are used for the upper and nether element—for example Figure 9b for the upper element and Figure 9f for the nether element—a hole will exist on the sampling path. If Figure 9c and Figure 9e are used, overlap will happen at the same point.

To solve this problem, we need to analyze the two situations of Figure 9a and 9d. Actually 9a and 9d correspond to two triangulation methods of the concave face. This



Face\Edge	Edge 1	Edge 2	Edge 3	Edge 4
Face 1	1 (0)	2 (1)	3 (1)	4 (0)
Face 2	8 (0)	7 (0)	6 (0)	5 (0)
Face 3	1 (0)	9 (0)	5 (0)	10 (0)
Face 4	6 (0)	11 (1)	2 (1)	10 (0)
Face 5	7 (0)	12 (0)	3 (1)	11 (1)
Face 6	4 (0)	12 (0)	8 (0)	9 (0)

Fig. 8. Sliced polygon forming.

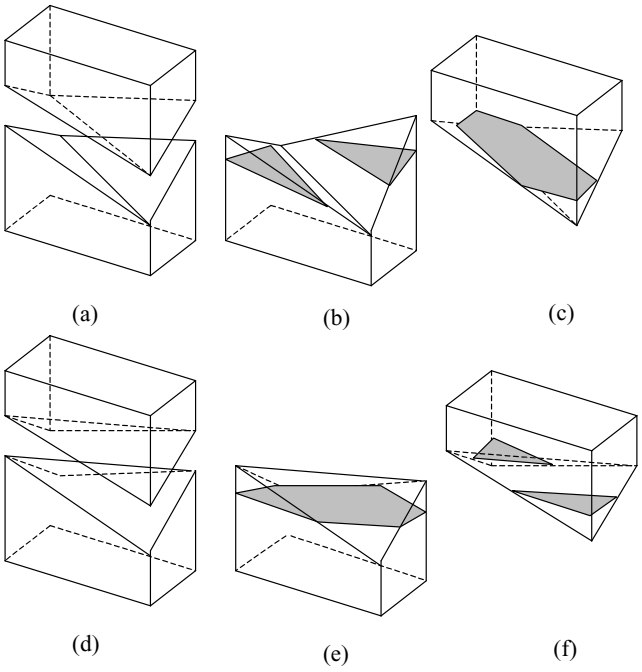


Fig. 9. Sliced polygon forming in case of concave element face.

is similar to the ambiguity problem in the iso-surface generation of marching cubes algorithm. But only one possibility is right in that case. For the problem in Figure 9, both cases can form the correct polygon. Figure 9b and 9c give the forming method in case of 9a; 9e and 9f give the correct polygon for 9d. If we concatenate the upper and nether element, the sampling on the common face is the same for both cases. Thus, the key point is that the same triangulation method is used for both the upper and the nether element.

In order to keep the consistency of triangulation, the key parameters must be the same for them. A simple method is to sum up the IDs of diagonal vertexes and select the bigger one as the segmentation diagonal. But in the mesh generation of FEA, vertex ID may be different for the same point in the upper and nether element. The merely coherent parameter is the coordinate. So the coordinate sum is used instead of the vertex ID.

Actually our objective is the separation of four edges into two groups by using the diagonal, rather than triangulation. In Figure 10, edges AB and AD are in the same group, as are edges BC and CD. Only the intersection points on the edges of the same group can form an edge of the sliced polygon (i.e., only AD and BC can be the correct edge).

In the preprocessing step, the start row cannot be the line of four 1s. If the scan stops at the concave row of ETDT, the next intersection can only be on the edge of the same group.

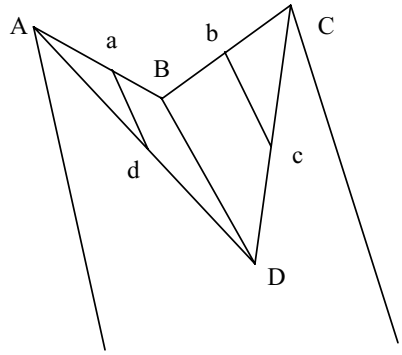


Fig. 10. Grouping of element edge on common face.

3.7 Rendering of sliced polygon

The hardware supported polygon RGBA filling is used to integrate slices in the order from front to back. The data values inside the polygon are interpolated from the node value.

3.8 Transfer function selection

The optical model used in this article is the source-attenuation model. The integral along each ray path is solved by Riemann sum. This is an approximation method on the assumption that data vary very little inside each step of the path. The error will increase with the varying of data. The transfer function is the key point to solve this problem. There are two independent transfer functions used in volume rendering, one for color mapping and one for transparency mapping. The commonly used transfer function is linear function or segment linear function, which has better accuracy. High opacity and red color are always designated for high value data, such as high stress range.

4 VOLUME CUTTING BY CONVEX POLYHEDRON

The user can define several convex polyhedra to partition the 3D space of the FEA structure and define different transfer functions for each part of them to set their contributions to the final image. Here we take the term *set* to represent a part in the partition. If one polyhedron is used in cutting the volume, the entire 3D space is partitioned into two sets. If the transfer function for the transparency of a set is constantly 100%, this is the same with a polyhedron cutting on the structure. This gives the user more flexibility to control whether a set will be rendered or how much of the set will be integrated into the image. We use the stencil function in the OpenGL pipeline to implement this function.

The stencil is a very important term in the OpenGL specification. It controls if one pixel affects the value in the frame buffer. If combined with depth test, it can finish very complicated functions. Normally the video card of PC supports 8 bits of stencil buffer. In our algorithm, each bit corresponds to a polyhedron. So at most the user can define eight polyhedra to partition the structure.

4.1 Stencil-based volume cutting

The volume rendering algorithm in this article uses the integration of sliced polygon to get the final image. So the set can be defined on the 2D slice planes. Figure 11 shows the sliced polygon between an element and the current slice layer. If the intersection between the cutting polyhedron and slice (Figure 12) can also be gotten on the slice plane, the sets to which each pixel belongs will be obvious. So if the intersection can be defined in the stencil buffer before the rendering of the sliced polygon, the volume cutting will be straightforward. Here we separate the work into two parts:

1. *Preprocess before the volume rendering*. Partition the polyhedron into front and back parts according to each facet's normal in the screen coordination. Disable stencil test, enable the write to stencil buffer, and disable depth test.
2. For each slice in the volume rendering iteration:
   a. Clear the content of stencil buffer, and set the depth buffer with current slice's z value. Enable depth test and disable the write of depth buffer and frame buffer.
   b. Set the stencil function to GL_ALWAYS and reference value to 1. Set the stencil operation to GL_INCR. Render the back part of the polyhedron.
   c. Set the stencil operation to GL_DECR. Render the front part of the polyhedron.
   d. Disable depth test and enable the write of frame buffer. Set the stencil operation to GL_KEEP. Set the stencil function to GL_EQUAL and reference value to 1. Now in the stencil buffer, the pixel with value one is just the part of the structure inside the polyhedron.

4.2 Multiple polyhedra

For each set, the second step in section 4.1 should be called once. For sets, much more time for the iteration is needed. This makes the algorithm too hard to meet the real-time requirement. Actually, the key factor that affects the number of iterations is the number of stencil reference values. Figure 13 shows eight sets defined by three cutting polyhedra. Figure 14 shows the stencil reference value for each set. If only the part inside the three polyhedra is needed, it is not necessary to render seven times for the seven different reference values. Actually, only once with the reference value 000 and stencil function GL_NOEQUAL is enough. So how to decrease

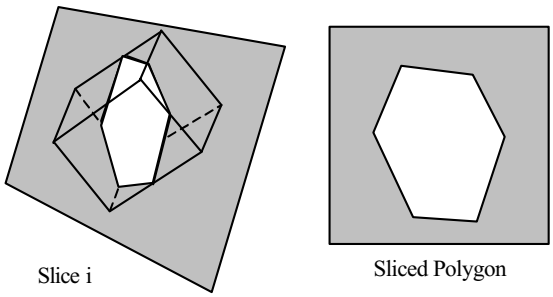


Fig. 11. The sliced polygon between the element and slices.

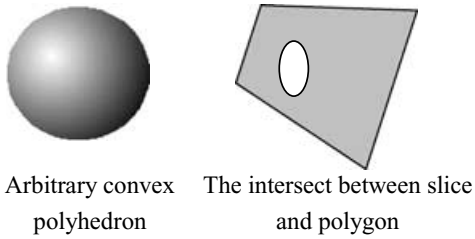


Fig. 12. The intersection between polyhedron and slices.

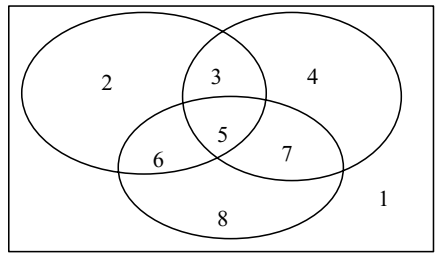


Fig. 13. Eight sets defined by three polyhedra.

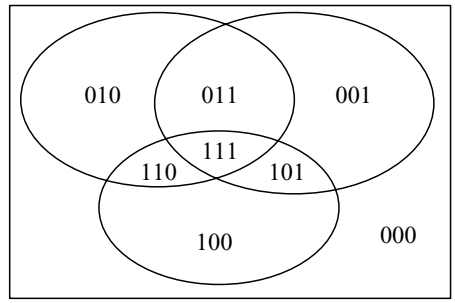


Fig. 14. Stencil reference value for each set.

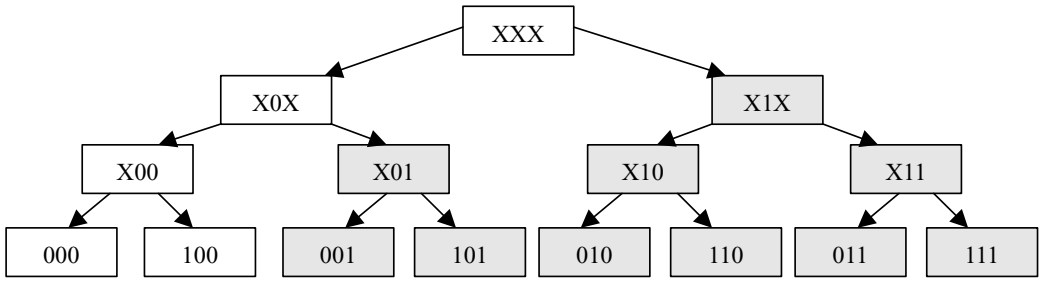


Fig. 15. Addition of stencil reference value.

the number of reference values becomes the key factor to speed up the algorithm.

In OpenGL, there is also a parameter, a mask for the stencil function. Here an additional operation of stencil reference value is defined to merge them together. Suppose A and B are two reference values to be merged. If A and B are different on only one bit, they can be added together by using XOR. The invert of the sum C can be used as the mask for stencil operation. The reference value can take either A or B. Take 010 and 011, for example; they are different on only the least bit, and the sum is 001. The mask for stencil is 110.

If there are many reference values needed for addition, the reference value and mask should be transferred to the next addition together. A marker "X" is added on the mask bit. For example,

$$\begin{aligned} &010 + 011 + 110 + 111 \\ &\quad = 01x + 11x \\ &\quad = x1x \end{aligned}$$

The number of final reference values depends on the order of addition. For example,

$$\begin{aligned} &010 + 011 + 001 + 110 \\ &\quad + 111 + 101 \\ &= 01x + 001 + 110 + 1x1 \end{aligned} \qquad \begin{aligned} &010 + 011 + 110 + 111 \\ &\quad + 001 + 101 \\ &= 01x + 11x + x01 \\ &= x1x + x01 \end{aligned}$$

The result of the addition for the same six reference values is different. The final number of reference values in the left addition is four. It takes nearly double the time to finish the rendering as the right addition. So the order of addition is very important for the algorithm efficiency.

Here we use the data structure of the binary tree to define the addition order. Each layer of the binary tree corresponds to one cutting polyhedron. We need to compute which bit the reference value has the most in common with. We put the bigger common bit at a higher position in the binary three. Figure 15 shows the tree and the addition process for the above example.

4.3 Out-faced facet cutting

In the algorithm presented in this article, out-faced facets should also be considered in the polyhedra cutting. Because the facet falling between two adjacent slices doesn't have constant depth value as sliced polygon, the cutting operation is a little bit complicated. There are two methods:

1. Assume a constant depth for all the facets falling into the same bucket and render the facets just the same way as with the sliced polygon. There will be jagged edges at the boundary of the intersection between cutting polyhedra and current slice, especially when the slice number is less than 200.
2. The first one is an approximate method that got more errors when the slice number becomes even less. To accurately render the facet, the exact depth value should be set into the depth buffer. It takes the following steps:
   a. Clear stencil buffer and disable the write of frame buffer.
   b. Rendering the facet to write the exact depth value into depth buffer.
   c. Same with section 4.1 to render cutting polyhedra to set stencil buffer.
   d. Disable depth test and the write of depth buffer. Enable the write of frame buffer. Render the facet again.

Because each facet should take all the above steps (a–d), the efficiency of the algorithm becomes worse. Ordinarily the second method is only used to get a more accurate result when the slice number is too small. But actually the best way for the user to get an accurate rendered image is to increase the sliced number, which will take the first method.

Figure 16 shows some examples of volume cutting by using simple cubes, spheres, and cones.

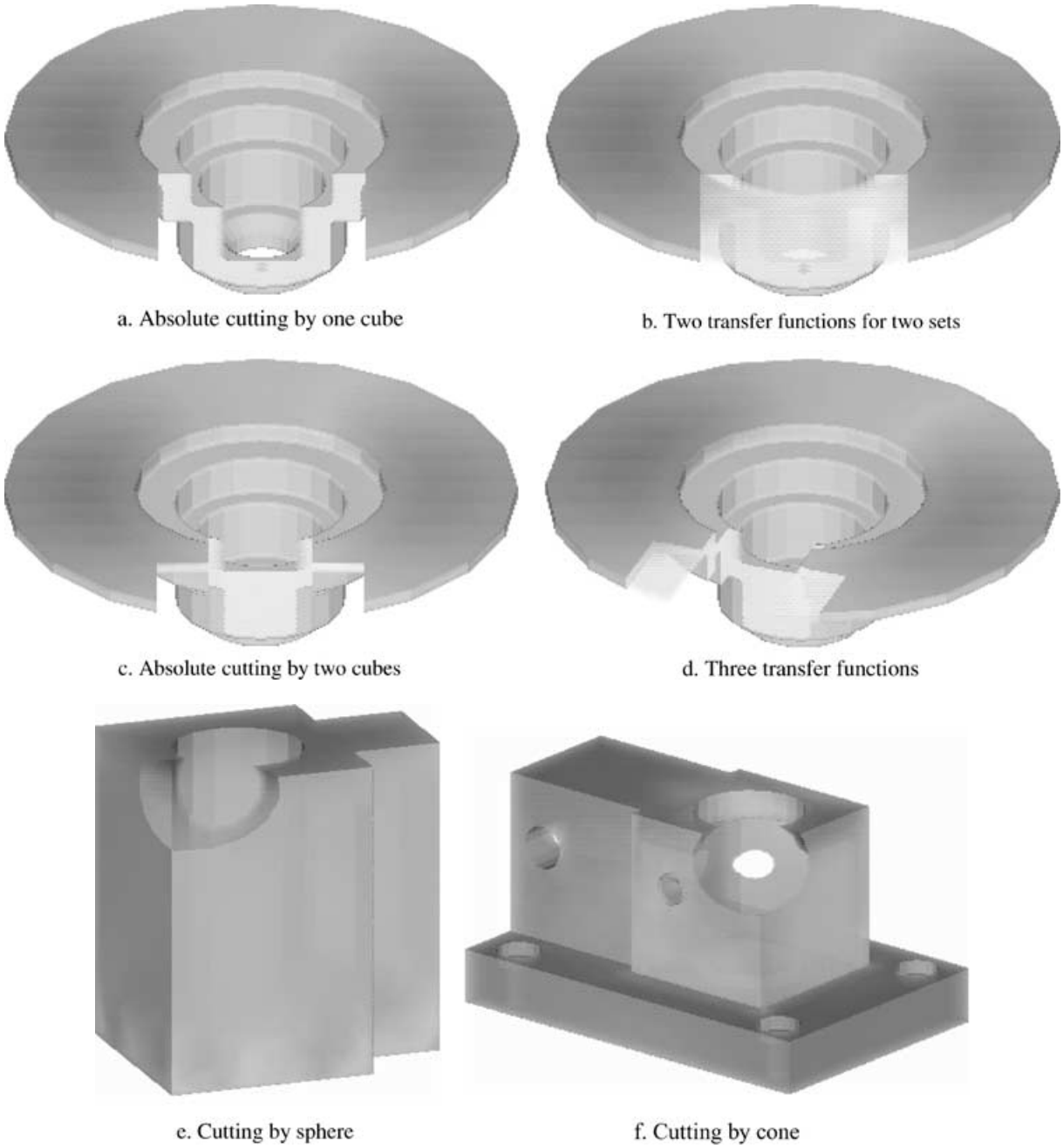


Fig. 16. Example of volume cutting by convex polyhedra.

5 RESULTS

As the result of the algorithm optimization of sliced polygon forming and the compensation of small elements, a satisfying image of volume rendering can be achieved within a very short time period. To evaluate the performance of our algorithm, the algorithm was implemented on a Pentium 350 with Riva TNT display card. Figure 17 and Figure 18 give the comparison of small element compensation. Figure 19 and Figure 20 show the result of out-faced faces rendering. It can be seen that the hybrid rendering makes the image more realistic. The time used by rendering, given in Table 1, shows that the present algorithm can achieve a speed of 1 fps for a moderate-sized data set of finite element model. It is enough to meet the real-time requirement of finite element software.

Table 1
Rendering time for five structures

Data set	*Element number*	*Node number*	*Facet number*	*50 slice (s)*	*200 slice (s)*	*300 slice (s)*
Example 1	5040	6300	17620	**0.733**	2.095	2.99
Example 2	864	1300	816	**0.26**	0.799	1.158
Example 3	192	442	440	**0.165**	0.473	0.71
Example 4	1976	3827	3166	**0.66**	1.725	2.445
Example 5	125309	135000	18966	**6.642**	11.879	15.745

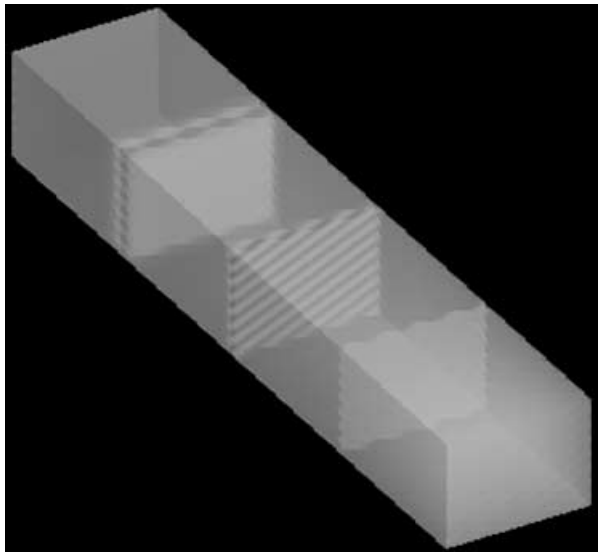


Fig. 17. Without small element compensation.

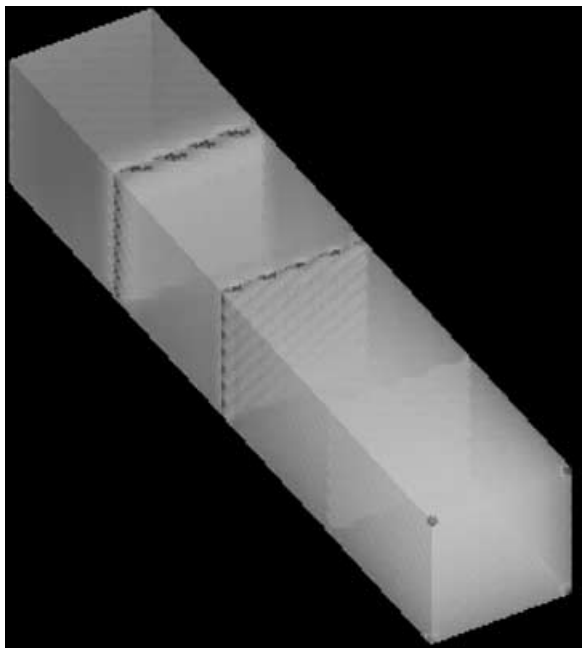


Fig. 18. With small element compensation.

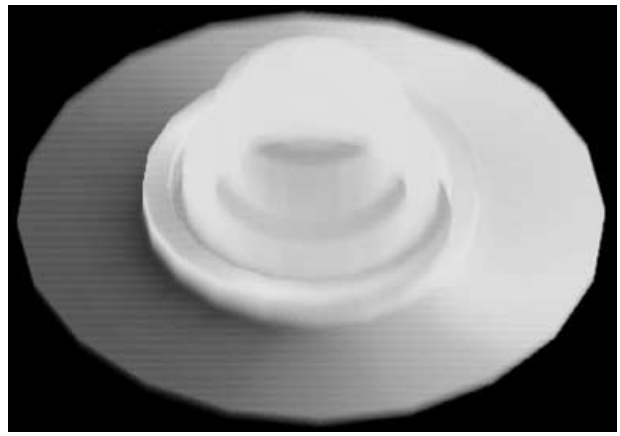


Fig. 19. Without out-faced faces.

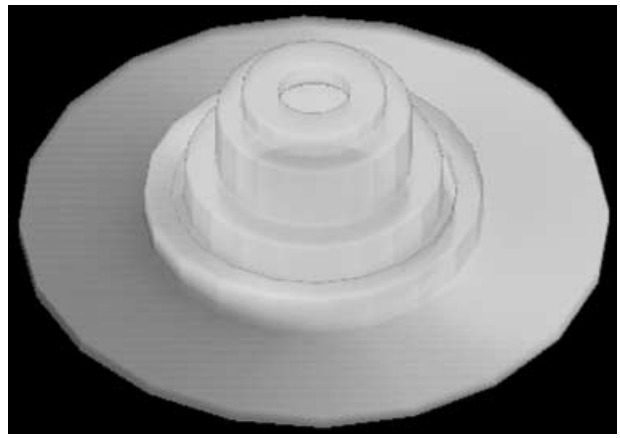


Fig. 20. Hybrid rendering of volume data and faces.

6 CONCLUSION

In this article, some technical approaches are proposed to accelerate volume rendering and improve image quality. The structure of ETDT is introduced to unify different treatments of various element types and greatly decrease the space consumption requirement of

volume rendering. Furthermore, the concept of important function gives classification to the missed elements. The compensation method to render important small elements gives a satisfying image at the least slice number. And the hybrid rendering of out-faces with the volume data improves the spatial perception of the volumetric structure. Finally, the partition on the structure defined by convex polyhedra makes the volume rendering more powerful and flexible for the user's data exploration.

ACKNOWLEDGMENTS

The project was supported by the Scientific Fund for National Outstanding Youth of China (19525206) and the Special Funds for National Key Basic Research of China (No. G1999032805).

REFERENCES

Garrity, M. (1990), Ray tracing irregular grid, *Computer Graphics*, **24**(5), 35–40.

Giertsen, C. (1992), Volume visualization of sparse irregular meshes, *IEEE Computer Graphics & Applications*, **12**(2), 40–8.

Max, N., Hanrahan, P. & Crawfis, R. (1990), Area and volume coherence for efficient visualization of 3D scalar functions, *Computer Graphics*, **24**(5), 7–33

Shirley, P. & Tuchman, A. (1990), A polygonal approximation to direct scalar volume rendering, *Computer Graphics*, **24**, 63–70.

Silva, C. T. & Mitchell, J. S. B. (1997), The lazy sweep ray casting algorithm for rendering irregular grids, *IEEE Transactions on Visualization and Computer Graphics*, **3**(2), 142–57.

Williams, P. L. (1992), Visibility ordering meshed polyhedral, *ACM Transactions on Graphics*, **11**(2), 103–26.

Yagel, R., Reed, D. M., Law, A., Shih, P. & Shareef, N. (1996), Hardware assisted volume rendering of unstructured grids by incremental slicing, *IEEE-ACM Volume Visualization Symposium*, San Francisco, CA, Nov., 55–62.