Efficiently Rendering Large Volume Data Using Texture Mapping Hardware

Xin Tong[1], Wenping Wang[2], Waiwan Tsang[2], Zesheng Tang[1]

[1] CAD Laboratory, Department of Computer Science, Tsinghua University,
Beijing, 100084, P. R. China
ztang@tsinghua.edu.cn
[2] Department of Computer Science, University of Hong Kong,
Pokfulam Road, Hong Kong
{wenping, tsang}@cs.hku.hk

Abstract. Volume rendering with texture mapping hardware is a fast volume rendering method available on high-end workstations. However, limited texture memory often prevents the method from being used to render large volume data efficiently. In this paper, we propose a new approach to fast rendering of large volume data with texture mapping hardware. Based on a new volume-loading pipeline, the volume data is preprocessed in such a way that only the volume data that contains object voxels are loaded into texture memory and resampled for rendering. Moreover, if classification threshold is changed, our algorithm classifies and processes the raw volume data accordingly nearly in real time. Our tests show that about 40% to 60% rendering time is saved in our method for large volume data.

1 Introduction

In scientific visualization, direct volume rendering is used to generate high quality semi-transparent images with details. As millions of voxels are usually involved in rendering, the amount of computation is enormous. Although much effort has been made to shorten the rendering time [3][5][7][10], the real-time direct rendering of the volume data still cannot be achieved by software.

The rapid progress of graphics hardware development offers new possibility for real time volume rendering [1]. Carbral [2] first proposed to use 3D texture mapping hardware to render volume data in 1994. Since then, several improved algorithms and implementations have been reported [4][11]. In these methods, the volume data is rendered via the texture mapping hardware in two steps. As shown in Fig. 1, in the first step for volume loading, volume data are read from a disk file (i.e., the first level) into volume buffer (i.e., the second level) in main memory. Then the data are loaded into texture cache (i.e., texture memory, the last level) and defined as 3D texture maps. In the second step for volume rendering, a set of polygons are used to resample the volume data by texture mapping and then these texture mapped polygons are blended from back to front to generate the final image. For volume data that can reside entirely into the texture cache, the volume loading step is done only once. In the following rendering

step, the texture mapping and composition operations are performed by hardware very quickly, about ten frames per second or higher. However, due to limited texture memory supported by hardware, large volume data must be divided into several blocks and rendered one by one. For such large volume data, the increased time on resampling texture I/O will reduce the frame rate greatly.

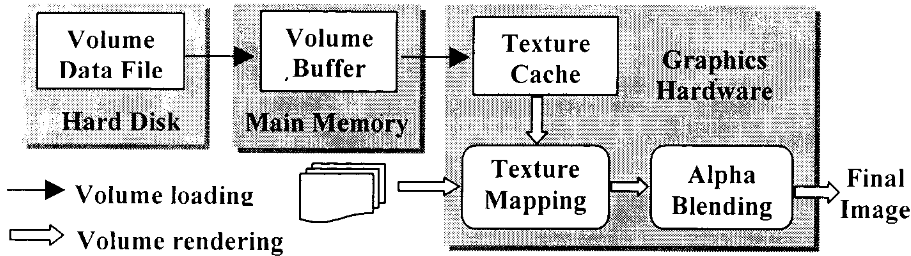


Fig. 1. The diagram of the conventional method

In fact, in many visualization applications, lots of voxels in volume data are so-called *empty voxels*, which belong to the background or parts of no interest. As the empty voxels are often treated as transparent (alpha=0) in rendering, skipping the empty regions may speed up volume rendering significantly, while not affecting image quality at all [12]. Unfortunately, in most existing methods [2][4][11], classification is performed by a hardware look-up table in the volume rendering step so that all the volume data must be loaded into the texture memory for rendering. In addition, since the texture mapping hardware requires that data in volume buffer be stored as a 3D array, volume data can not be processed efficiently using the three-level volume loading pipeline.

In this paper we propose an algorithm to speed up the volume rendering with 3D texture mapping hardware. Unlike conventional method, a four-level volume-loading pipeline is exploited to classify and preprocess volume data efficiently. After the volume data is classified into empty voxels and object voxels, only the parts that contain the object voxels are loaded into the texture memory for rendering, while the empty voxels will be kept in volume buffer but not loaded into texture memory. In this way, the volume rendering step is also accelerated by avoiding the unnecessary resampling operations.

The rest of the paper is organized as follows. In section 2, the related work is reviewed. Our algorithm is described in detail in section 3. Section 4 gives the experimental results. The paper is concluded in section 5.

2 Related Work

Skipping empty regions is a well-known acceleration technique that has been extensively exploited by ray casting type volume rendering algorithms. Marc Levoy [8] employed an octree of the binary volumes to enumerate the presence of objects in volume data. Thus for the ray casting method, the ray can skip the largest empty space

containing the current sampling point by traversing the octree. However, the octree data structure of binary volumes not only occupies auxiliary memory space but also needs to be reconstructed whenever the classification threshold is altered. Moreover, since object voxels are represented by a set of octree nodes with different sizes, it is then difficult to render these nodes by texture mapping hardware.

In proximity-clouds methods [3][14], an auxiliary distance volume is constructed from original volume data. In the distance volume each voxel value is the distance from the voxel to the closest object voxel. If the current resample point on a ray is at a voxel, then based on the distance value of the voxel the next resample point can leap to object voxels quickly, so that many unnecessary resampling can be avoided. Although traversing the distance volume is more efficient than traversing the octree, the proximity-clouds methods need large additional memory to store the distance volume. In addition, it takes long time to compute the distance volume when the classification is modified.

The PARC approach [10] uses graphics hardware to identify the segments of rays that contribute to the final image. Given a classification threshold, the outer faces of object cells are presented by a set of rectangular polygons. Then for each view direction, the polygons are projected into the zbuffer twice to obtain the ray segments that possibly contribute to the final image and only these ray segments are traversed and resampled in ray casting. Unfortunately, because the depth test occurs after the texture mapping in the graphics pipeline, the PARC method cannot be used for volume rendering with texture mapping hardware. Although the envelope of object voxels can be used to clip the resampling polygons in volume rendering with texture mapping hardware, only resampling operations in the rendering step can be reduced and the time spent on volume loading step is not reduced at all.

The shear-warp method [5] takes advantage of spatial coherence of voxels to skip the empty regions in volume data. The method also cannot be applied in the volume rendering with texture mapping hardware, as the volume must be resampled by hardware slice by slice.

3 Efficient Rendering of Large Volume Data

3.1 System Overview

As shown in Fig. 2, our method also contains two steps: volume loading and rendering. However, a four-level volume-loading pipeline is adopted, where a new texture buffer is added between volume buffer and texture cache. In the volume loading step, the volume data is divided into several volume blocks and loaded into volume buffer first. Three min-max arrays are then built for each volume block. Given a classification threshold, empty voxels in each volume block are identified and trimmed off by the bounding boxes of object voxels. These trimmed volume blocks are transferred into a texture buffer and defined as texture blocks. Finally the texture blocks in the texture buffer are merged into bigger texture chunks to reduce the overload of texture memory I/O. In the volume rendering step, after the texture chunks are sorted according to a

given view direction, they are loaded into texture cache from back to front. Then the texture blocks are resampled by a set of polygons that are clipped by the bounding boxes. After blending all the texture-mapped polygons to the frame buffer, a new image is generated for display.

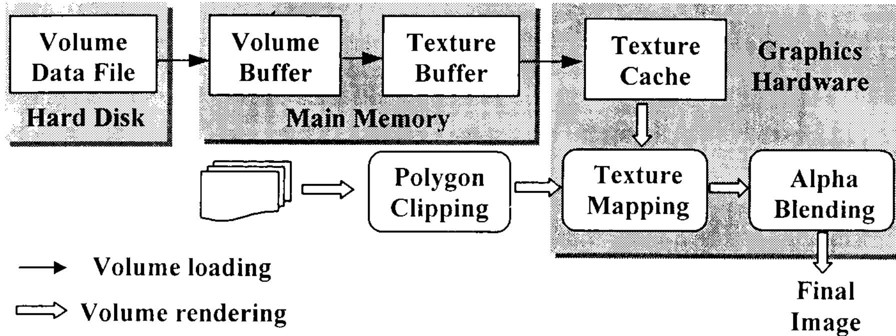


Fig. 2. The diagram of our new volume rendering method

In our method the parameters of volume-loading pipeline are determined by the configuration of texture cache, while texture cache that lies in the texture memory is configured by the operating system and special hardware implementation. In general, the following rules are compatible with texture cache, where the n, m, s and k are constant determined by the configuration of texture memory [9]:

- The texture data that will be loaded in the texture cache must be stored in an array.
- The texture maps in the texture cache is stored in a tiled page mode.
- The size of texture page is $2^n \times 2^m$ for 2D texture and $2^n \times 2^m \times 2^s$ for 3D texture.
- A texture page is the minimum unit for 3D texture I/O.
- At most 2^k texture pages can be resident in texture cache at same time.

We suppose that a volume data is axis-aligned and stored as an $l \times m \times n$ 3D array. The length of the volume (in voxels) in X, Y, Z direction is l, m, n respectively. Suppose $V_{i,j,k}$ denoted the voxel value of voxel (i, j, k) as in the 3D array.

3.2 Volume Loading

The volume loading step can be divided into three phases. The whole volume data is uniformly subdivided into several blocks and loaded into the volume buffer first (see section 3.2.1). Next the volume block is classified and trimmed by the bounding boxes of object voxels, and the trimmed blocks are then moved into texture buffer (see section 3.2.2). Finally, in order to optimize the I/O performance of texture cache, the texture blocks are merged (see section 3.2.3).

3.2.1 Volume Buffer loading and min-max array generation

In this stage, volume data is subdivided into several volume blocks. In order to avoid artifacts caused by seams between adjacent blocks in the final image, each block shares

its boundary voxel data with its neighboring blocks. Two factors are considered in selecting the block size. On one hand, the smaller is the block size, the more empty regions can be excluded. On the other hand, smaller blocks would result in more redundant data on the overlapping boundaries of the blocks and in fact the block size cannot be less than the size of the texture page. Moreover, the large number of texture fragments with irregular sizes after trimming would cause the overhead of graphics hardware, and it is also difficult to merge these small blocks into bigger blocks to optimize I/O of texture memory. On that account, the size of volume block is chosen in our method to be the size of the smallest block that satisfies the following two rules:

- The volume block must contain at least one texture page.
- The volume block must have equal length along each axis. This rule is used to simplify the following merging operation.

Given the size 2^d of volume blocks, volume data is loaded from a disk file into the volume buffer block by block. The number of blocks is

$$\left\lceil \frac{l-1}{2^d-1} \right\rceil \times \left\lceil \frac{m-1}{2^d-1} \right\rceil \times \left\lceil \frac{n-1}{2^d-1} \right\rceil \qquad (1)$$

If the last block along an axis cannot be fully filled with the volume data, the remaining part of the block is supposed to be filled with empty voxels. An array is created for the 2^d data slices of the block along one axis; each element of the array records the minimum and maximum data value of the corresponding slice. Three arrays that correspond to three coordinate axes are set up for each volume block.

3.2.2 Data classification and volume block trimming

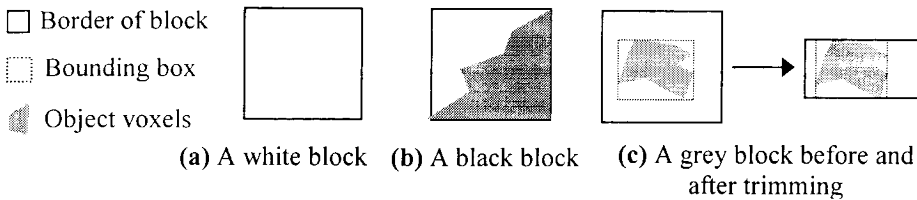
A block list is generated after the above preprocessing step. Each node of the list contains a pointer to a volume block, the min-max arrays of the block and the position of the volume block in world space. At this stage, most empty voxels in the volume block are excluded so that the block will be trimmed to a smaller size and moved into the texture buffer.

After a classification threshold is specified by the user, the three min-max arrays of the block are traversed to find three intervals $[L_x, R_x]$, $[L_y, R_y]$ and $[L_z, R_z]$ that are defined on X, Y, Z axes respectively, such that:

- $V_{i,j,k}$ are empty voxels for all $i < R_x$ and all $i > R_x$
- $V_{i,j,k}$ are empty voxels for all $j < R_y$ and all $j > R_y$
- $V_{i,j,k}$ are empty voxels for all $k < R_z$ and all $k > R_z$

The three intervals listed above bound the object voxels of the volume block in a box. According to the volume of this bounding box, there are three cases for one block. In the first case, the volume of the bounding box is zero. Obviously, all the voxels in the block are empty voxels in this case, so the block is marked as a white block. In the second case, the volume of the bounding box is the same as that of the block, which means that the block cannot be trimmed. The block is marked as black block in this case, and all the data in the volume block are moved into the texture buffer as a texture block. In the last case, part of the block can be trimmed off, and the block is marked as a grey block first. Then we choose the trimming axis to be the one along which the

interval length is the shortest. After that, the block is trimmed along the trimming axis. The remaining part that contains object voxels are moved into the texture buffer as a texture block. A 2D example is shown in Fig. 3, where the trimming axis is the Y axis. Only the empty voxels $V_{i,j,k}$ with $i<L_y$ and $i>R_y$ are trimmed off. Although the empty voxels that lie out of the other four faces of the bounding box are retained, they will be skipped to avoid resampling in rendering step. Hence after being trimmed along the trimming axis s, the size of texture block is $2^d x 2^d x d_s$, where $d_s = R_s - L_s + 1$. Finally, the information about the bounding box is saved in the node of the block list.



(a) A white block **(b)** A black block **(c)** A grey block before and after trimming

Fig. 3. The 2D illustration of the block trimming

After trimmed blocks are moved into the texture buffer, the volume data is divided into two parts. The texture blocks that contain object voxels are moved into the texture buffer, while the empty parts are trimmed off and stored in the volume buffer.

3.2.3 Texture block merging

Although the texture blocks could be directly loaded into texture memory for rendering now, the frequent texture I/O and binding operations would degrade the performance of the algorithm seriously. So we will try to merge the texture blocks into bigger texture chunks before rendering.

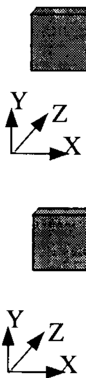
In this stage, the texture blocks are grouped into clusters first. A cluster contains $u \times v$ adjacent texture blocks with the same index along Z. If all of the texture blocks are black blocks, the size of the cluster is the same as the size of the texture cache. That is, the texture blocks in a cluster can reside in the texture cache at the same time. The constants u and v can be computed from the size of the volume block easily. If necessary, some white blocks are added into the last clusters along the axis. The position of the texture block in the texture cache (i.e. the texture coordinates) is also recorded in the block node.

After grouping, the texture blocks are merged in two steps: merging in cluster and merging between clusters. In each step, two basic merge operations are executed for a pair of texture blocks: horizontal merge and vertical merge. The merge operation and the procedure of merging blocks are described below.

3.2.3.1Merge Operation

- Horizontal merge

As shown in Fig. 4(a), two blocks with the smallest difference of their thickness (block length along Z axis) are selected and merged into a $2^{d+1} x 2^d x d_m$ chunk, where $d_m = \max(d_{block1}, d_{block2})$. Thus for a thinner data block, the extended part will be filled with empty



voxels. The original thickness of the block is saved in the block node. Although some extra empty voxels have to be added into the texture chunk, this would speed up texture memory I/O and the empty voxels can be skipped without resampling at the rendering step.

- Vertical merge

As shown in Fig. 4(b) and 4(c), If the thickness of the two blocks satisfy $d_{block1}+d_{block2}\le 2^d$, the two blocks can be piled up together. The two original texture blocks will be directly merged into a chunk of size $2^d\times 2^d\times (d_{block1}+d_{block2})$. If one of source blocks has been horizontally merged into a chunk of size $2^{d+1}\times 2^d\times d_m$, the extended part in this chunk is excluded first. Then the other original block is piled up to it. The result is a chunk of size $2^{d+1}\times 2^d\times d_{new}$, where $d_{new}=\max(d_m, d_{block1}+d_{block2})$. If necessary, some empty voxels are used to fill in the extended part.

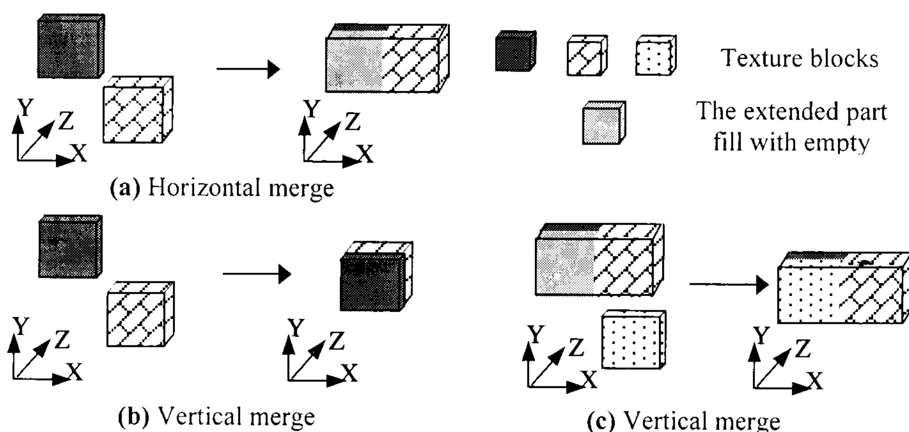


Fig. 4. The horizontal and vertical merge operation

3.2.3.2 Merging in cluster

The texture blocks in a cluster are checked first. If the trimming axis of a data block is not *Z*, the texture block is rotated and reorganized in the texture buffer so that the trimming axis is changed to *Z*. Two kinds of merge operations are executed until no pair of blocks in the cluster can be merged. In horizontal merge, pairs of blocks will be merged according to the ascending order of their thickness difference. That is, the pair of texture blocks with the smallest thickness difference will be merged first.

3.2.3.3 Merging Between clusters

After the above merge operations, the texture blocks that have not been merged in the cluster may be merged further if the merging between clusters is considered. For parallel projection of the volume data, the texture clusters are traversed and rendered along the *Z* axis first, then along the *Y* axis, at last along the *X* axis. For each axis, whether ascending or descending order of the indices is used for traversing is

determined by the view direction. Hence, a sequence of clusters $\{B_{i,j,k} \mid 1 \le k \le N_z, N_z$ is the number of clusters along Z axis$\}$ are loaded into the texture memory sequentially. If two texture blocks from adjacent clusters along Z axis are merged, after loading the merged texture chunk into texture memory for rendering one cluster, the other texture block in the adjacent cluster is also resident in texture memory. Thus the latter block needs not to be loaded again when rendering the adjacent cluster.

In this step, for each row of clusters along Z direction (that is, the clusters with same X and Y indices), if there are texture blocks that have not been merged in pair of adjacent clusters, the two merge operations are executed for them. Fig. 5 shows two examples of the merge between clusters, where the cluster contains 4 texture blocks.

Fig. 5. Two examples of merging between clusters

Note that in Fig. 5, although the merged chunk is drawn twice in resulting clusters, only one copy of the merged texture chunk is stored in the texture buffer, and referenced by the pointers of block nodes in two clusters. This guarantees that the merged data can always be loaded and resident in texture memory for rendering the two clusters with different traversing orders.

3.2.3.4 Notes about the texture block merging

In the above merging procedure, the texture blocks are rotated, moved and merged in the texture buffer. In order to keep the correct texture mapping, the block position in the world space is reserved in the block node, while the texture coordinates of the block need to be updated accordingly. Thus during rendering, the trimmed blocks will "stay" in the original position in world space so that the volume blocks can be rendered correctly for a given view direction. A 2D example is illustrated in Fig. 6.

In addition, for a set of texture blocks, different orderings of the texture merging operations would lead to different merging results. For a special configuration of the texture hardware, texture blocks with different sizes should be loaded with the different speeds. Adjusting the order of the merging operation properly may produce texture chunks that lead to higher texture I/O performance.

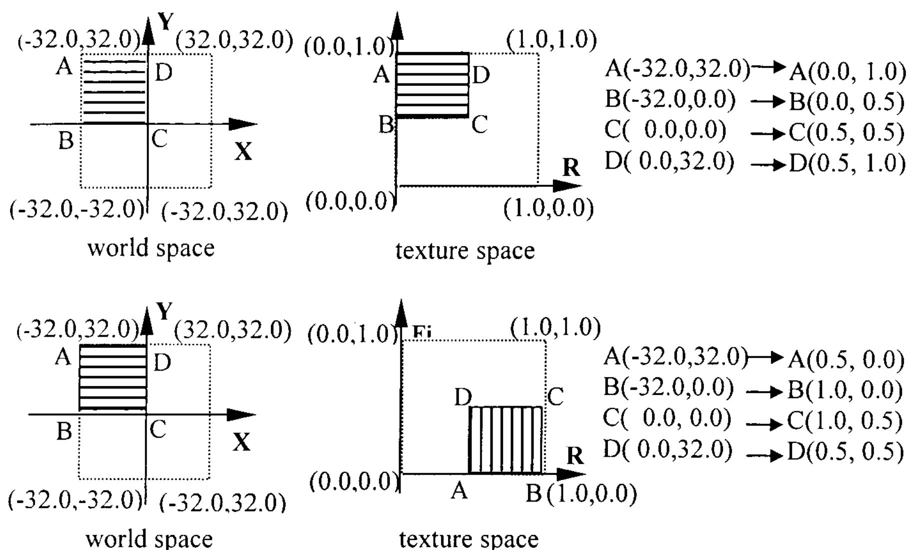


Fig. 6. The top figure shows a block ABCD and its corresponding data (i.e. texture) stored in the texture space. After moving and rotating the data in texture space (bottom figure), the texture coordinates of the block is updated accordingly, while the world coordinate is preserved So the block can be rendered with the same texture.

3.3 Volume Data Rendering

For any given view direction, the traverse order (ascending or descending) of the texture clusters along each axis is determined first. Then according to the viewing order, the texture clusters are traversed from back to front, and rendered cluster by cluster. If all blocks in a cluster are white blocks, the cluster is skipped. Otherwise, the texture chunks that do not reside in the texture cache are now loaded into the texture cache. The texture blocks in the cluster are also rendered from back to front. If a block is white, it is skipped without any further operation. Otherwise, the block is resampled by a set of parallel polygons from back to front. The bounding box of the object voxels in each texture block is used to clip the resampling polygons so that only the voxels inside the bounding box are resampled. After composing all resampling slices from back to front, the final image can be displayed on the screen.

4 Experimental Results

Our method has been implemented on SGI Octane/MXI, configured with R10000/195MHZ CPU, 128MB memory and 4MB texture memory. On this system the size of the texture page is 4k(32×64×2)[10] and the size of texture cache is *128×128×64*. Thus the size of the volume block is *64×64×64*. The texture cluster is

composed of 2×2 texture blocks. Since in our platform the texture block with longer length along X axis is loaded faster than other texture blocks, the horizontal merge is used before the vertical merge in texture merging step.

We have compared our algorithm with three other algorithms. Algorithm I is an implementation of the conventional method [2]. In Algorithm II, the resample polygons are clipped by the envelope of object voxels. To show the effect of the merge operation, an implementation of our method without the merge operation is tested as Algorithm III. The full version of our algorithm is implemented as Algorithm IV. Six data sets from [6][8][13] are used for the test, where the size of the test image is 400×400. The experimental results are listed in Table I, where

- Data Processing Time = $\text{Time}_{\text{data classification}} + \text{Time}_{\text{block trimming}} + \text{Time}_{\text{block merging}}$
- $\text{Empty} = \dfrac{\text{number of empty voxl es}}{\text{number of voxels}} \times 100\%$
- $S = \dfrac{(\text{RenderingTimeI} - \text{RenderingTimeIV})}{\text{RenderingTimeI}} \times 100\%$

Table 1. The experiment results of the four algorithms

Data	Data Size	Empty (%)	Algorithm I	Algorithm II	Algorithm III	Algorithm IV		
			Rendering Time (s)	Rendering Time (s)	Rendering Time (s)	Rendering Time (s)	S(%)	Data Processing Time (s)
Head I	256×256×128	74.1	0.700	0.499	0.406	0.287	59.0	0.156
Head II	512×512×56	68.4	0.963	0.740	0.787	0.510	47.0	0.140
Body I	512×512×78	70.7	1.329	1.032	1.022	0.652	50.9	0.242
Frog	500×470×176	83.4	1.963	1.446	1.235	0.786	59.9	0.374
Foot	250×512×352	76.6	2.552	2.056	1.578	0.978	61.7	0.535
Head III	512×512×256	67.0	3.575	3.166	2.792	1.604	55.1	2.624

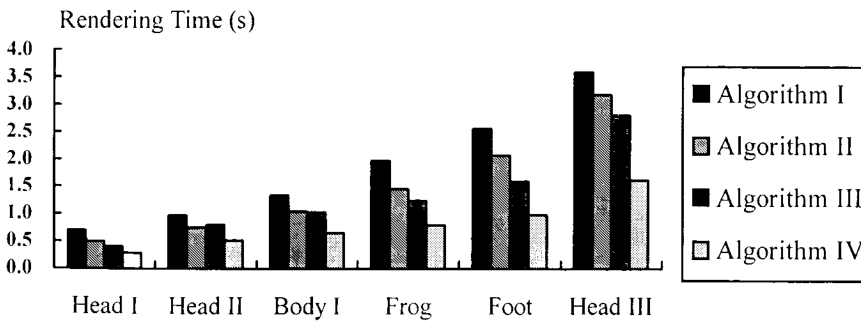


Fig. 7. The rendering time of four algorithms

As shown in Fig. 7, compared with the conventional method, about 40% to 60% rendering time can be saved by our method for the test data sets. Our method is also significantly faster than the Algorithm II and Algorithm III.

Fig.8 is the rendering result of the Head I data set by the conventional texture mapping hardware assisted volume rendering. Fig. 9 is the rendering result of the same data set by our new method. There is no difference between the visual quality of two images because the excluded and skipped empty voxels contribute nothing to the final image. Fig. 10 is the rendering result of the Frog data set by our algorithm. Fig. 11 is the rendering result of the Foot data set.

In our method, after the classification threshold is modified by the user, the volume blocks are classified again, and then only the texture blocks that contain object voxels are moved into the texture buffer for rendering. As listed in Table 1, although the volume data cannot be processed in real time, the time spent for processing is approximately on the same order of the rendering time.

5 Conclusion and Further Work

To summarize, we proposed an acceleration method to render large volume data with texture mapping hardware. Unlike the conventional method, our method uses a four-level volume-loading pipeline, through which the volume data can be processed efficiently before rendering. In the volume loading step, the empty voxels in the volume blocks are identified and trimmed off by the bounding boxes of the object voxels. Then the trimmed blocks are loaded into the texture buffer and become texture blocks. Finally, the texture blocks are merged into texture chunks to optimize the I/O performance of texture cache. During rendering, the resampling slices are clipped by the bounding box. As a result, both texture memory I/O and the texture resampling operations are reduced effectively. Experiments show that our method boosts the speed of rendering large volume data with texture mapping hardware approximately by a factor of 2. Moreover, the classification of the volume data is also accelerated by the min-max arrays of the volume blocks. Thus the raw volume data can be classified and processed quickly for the modified classification threshold.

In our method, if the volume data cannot be entirely stored in the main memory (e.g., the HEAD III data set), the frequent memory swapping operation would result in long preprocessing time. Fortunately, the volume data can be compressed and stored in the volume buffer. After the volume data is classified, only the texture blocks that contain object voxels are reconstructed from the compressed data on the fly and then moved to the texture buffer. In our further work different kinds of compression methods will be investigated to find the best compression scheme for our method. In addition, the possibility of using our method to render multiresolution volume data or the volume data on remote machine will also be explored in the future.

References

1 Akeley, K.,: RealityEngine Graphics. In Proceedings of ACM SIGGRAPH'93. (1993) 109-116

2 Cabral, B., Cam, N., and Foran. J,: Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. ACM Symposium on Volume Visualization. (1994) 91-98
3 Cohen, D. and Shefer, Z.,: Proximity Clouds - An Acceleration Technique for 3D Grid Traversal. Technical Report FC93-01, Ben Gurion University of the Negev. (1993)
4 Gelder, A. V. and Kim, K.,: Direct Volume Rendering with Shading via 3D Textures. In Proceedings of ACM Symposium on Volume Visualization. (1996) 23-30
5 Lacroute, P. and Levoy, M.,: Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In Proceedings of ACM SIGGRAPH'94. (1994) 451-458
6 LBL.,: Whole Frog Project. From http://www-itg.lbl.gov/. (1994)
7 Levoy, M.,: Efficient Ray Tracing of Volume Data. ACM Transactions on Graphics. Vol. 9, No. 3, (1990) 245-261
8 NLM.,: Http://www.nlm.nih.gov/research/visible/visible_human.html. (1997)
9 Silicon Graphics.: OpenGL on Silicon Graphics System. From http://trant.sgi.com/opengl/docs/docs.html. (1997)
10 Sobierajski, L. M., and Avila, R. S.,: A Hardware Acceleration Method for Volumetric Ray Tracing. In Proceedings of IEEE Visualization'95. (1995) 27-34
11 Westermann, R. and Ertl, T.,: Efficiently Using Graphics Hardware in Volume Rendering Applications. In Proceedings of ACM SIGGRAPH'98. (1998) 169-178
12 Yagel, R.,: Towards Real Time Volume Rendering. Proceedings of Graphicon'96. Vol. 1, July, (1996) 230-241
13 Zubal, I.G., Harrell, C.R., Smith E. O., Rattner, Z., Gindi, G. R. and Hoffer, P. B.,: Computerized Three-dimensional Segmented Human Anatomy. Medical Physics, Vol. 21, No. 2, (1994) 299-302
14 Zuiderveld, K., Koning, A. H. J., and Viergever, M. A.,: Acceleration of Ray Casting Using 3D Distance Transforms. In Proceedings of Visualization in Biomedical Computing 1992. October, (1992) 324-335,.

Editors' Note: see Appendix, p. 323 for colored figures of this paper