ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.17, No.3, March 2006, pp.587−601
DOI: 10.1360/jos170587

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

视点相关的层次采样:一种硬件加速体光线投射算法*

陈 为+, 彭群生, 鲍虎军

(浙江大学 CAD&CG 国家重点实验室,浙江 杭州 310027)

View Dependent Layer Sampling: An Approach to Hardware Implementation of Volume Ray Casting

CHEN Wei+, PENG Qun-Sheng, BAO Hu-Jun

(State Key Laboratory of CAD&CG, Zhejiang University, Hangzhou 310027, China)

+ Corresponding author: Phn: +86-571-88206681 ext 522, E-mail: wchen@cad.zju.edu.cn, http://www.cad.zju.edu.cn/home/chenwei

Chen W, Peng QS, Bao HJ. View dependent layer sampling: An approach to hardware implementation of volume ray casting. *Journal of Software*, 2006,17(3):587−601. http://www.jos.org.cn/1000-9825/17/587.htm

Abstract: Ray casting is a widely recognized method for high quality volume rendering. It traverses and samples the volume data ray by ray in image space. Traditionally, the algorithm is implemented in CPU on PC platform, resulting in slow speed and poor interactivity. This paper introduces a new technique named View Dependent Layer Sampling (VDLS), which supports a hardware implementation of ray casting by Graphics Processing Unit (GPU). VDLS organizes the ray sampling points into a set of layers which can be efficiently represented by two view dependent geometric buffers as two dynamic textures. Based on the structure of VDLS, the six steps involved in the ray casting algorithm including ray generation, ray traversal, interpolation, classification, shading and composition can be fully accomplished in GPU, taking advantage of its programmability and flexibility. In addition, two speedup techniques exploiting object space and image space coherence are proposed for fast culling of the lapsed rays. Several advanced features regarding illumination and composition are further discussed, with which VDLS is capable of reconfiguring the well-known geometric hardware engine for volume ray casting. The novel approach of GPU supported ray casting can render up to 150 million interpolated, post shaded and composed ray samples per second for perspective view. Experimental results suggest that the proposed framework can be regarded as an alternative for on-the-fly visualization and exploitation of discrete scalar data in medical visualization, physical phenomena simulation and material testing applications.

Key words: direct volume rendering; ray casting; view dependent layer sampling; hardware acceleration; GPU (graphics processing unit)

* Supported by the National Natural Science Foundation of China under Grant Nos.60503056, 60303028 (国家自然科学基金); the National Grand Fundamental Research 973 Program of China under Grant No.2002CB312100 (国家重点基础研究发展规划(973)); the National Natural Science Foundation of China for Innovative Research Groups under Grant No.60021201 (国家自然科学基金创新群体基金); the Natural Science Special Fund for Youth Scientists' Cultivation of Zhejiang Provincial of China under Grant No.R603046 (浙江省青年人才基金)

Received 2004-09-15; Accepted 2005-02-03

摘 要: 光线投射是一种高质量的体绘制方法.它以图像空间为序,逐根光线遍历和采样体数据.因此,传统上,它只能在 CPU 上实现,因而速度慢,交互性不好.提出了一个新的视点相关的层次采样 VDLS (view dependent layer sampling)结构,VDLS 将光线上的所有采样点重新组织成一系列层,并简化为两个视点相关的几何缓冲器,进而在 GPU (graphics processing unit)中用两个动态纹理表示.利用 GPU 的可编程性,光线投射算法的 6 个步骤(光线生成、光线遍历、插值、分类、着色和颜色合成)得以完全在 GPU 中实现.在此基础上,提出两个基于体空间和图像空间连贯性的加速技巧,快速剔除无效的光线.结合其他与渲染和颜色合成有关的技巧,VDLS 将面向多边形绘制的图形引擎转化为体光线投射算法引擎,在透视投影方式下,每秒能处理 1.5 亿个插值、后分类与着色的光线采样点.实验结果表明,提出的方法能用于医学可视化、真实物理现象模拟、材质检测中灰度体数据快速交互的可视化与漫游.

关键词: 直接体绘制;光线投射;视点相关层次采样;硬件加速;图形处理单元
中图法分类号: TP391 文献标识码: A

Direct volume rendering (DVR)[1−5] is a popular method for visualizing three-dimensional scalar field. It has the ability to show different regions translucently and reveal inner structures without intermediate representations. This makes DVR an important visualization technique widely used in numerous areas, including medical imaging, material testing, physical phenomena simulation, *etc*.

DVR algorithms can be classified into two categories according to their processing orders. The image space methods include the well-known ray casting[1,4] and ray tracing[6]. The object space methods cover V-buffer[7], cell projection[8], splatting[9] and shearwarp factorization[10]. While the object space methods can achieve interactive speed by means of pre-integration footprint table, ray casting algorithm has been evidenced to produce the best image quality, but with a relative slower speed and poorer interactivity. In view of this, several acceleration techniques, including space leaping[11−15], adjusting sampling steps[16−18], hierarchical data structure[10,11,19,20], and coherence reuse[21−25] are proposed for improving the rendering speed of ray casting algorithm. However, these earlier works are solely based on software implementations, and do not take the increasing capability of consumer graphics hardware[26,27]. Recently, much effort has been made on efficient hardware acceleration. The proposed techniques include 2D and 3D texture slicing mapping[28−35], large parallel computing[36,37], CPU optimization[38,39] and special-purpose hardware[40−42]. Among them, the most efficient and popular method to date is the hardware accelerated texture slicing method combined with the pre-integration technique[32].

The fast increase in graphics hardware architectures offers rich ways to flexibly utilize the graphics pipeline. Many researchers show their interests in implementing ray tracing using commodity programmable graphics hardware[43,44]. Recently, Roettger *et al*.[45] describe a smart hardware-accelerated ray-caster which is able to adapt the sampling rate to the actual information in the data sets. It can achieve several fps (frame per second) for moderate sized volume and its image is better than that by comparable texture slicing methods. More recently, Krueger *et al*.[46] propose an acceleration algorithm with early ray termination. It can achieve interactive frame rates on the ATI Radeon 9700 graphics.

In this paper, we will restrict our focus on rendering the rectilinear volume that has orthogonal grid structure, whose voxels are represented by 8bits. We introduce an enabling technique, View Dependent Layer Sampling (VDLS) which organizes the sampling points lying on different rays into a set of layers. VDLS can be regarded as an extension of the PARC (polygon assisted ray casting) method[12], but with more scalability and feasibility for GPU acceleration. Based on VDLS, a novel ray casting engine is proposed. The steps involved in ray casting algorithm are then reconfigured and implemented entirely in GPU.

The remainder of the paper is organized as follows. In Section 1, we will outline the related work, the custom

ray casting architectures as well as some terminologies. We introduce VDLS and our new ray casting pipeline in Section 2. The ray casting algorithm based on VDLS is described in Section 3. Several speedup techniques are proposed in Section 4. Section 5 presents the experimental results on both the time performance and image quality of the new ray casting engine; Followed by the conclusions of this paper and highlights for future work.

1 Preliminaries

Typically, ray casting algorithm consists of six steps: ray generation, ray traversal, interpolation, classification, shading and composition. Existing methods on the speedup of ray casting can be divided into three forms: sample reduction, coherence reuse and hardware assisted acceleration.

1.1 Sample reduction

Sample reduction methods try to avoid unnecessary sampling and processing overhead. For instance, early ray termination technique[11,46] is commonly used in FTB traversal mode to terminate ray traversal when the accumulated opacity approaches 1.0. For empty regions within the volume, space-leaping methods will simply skip them by some preprocess efforts. Common space-leaping techniques include C-buffer[14], PARC[12], Proximity Cloud[13], Distance transform[15], *etc*. In addition, auxiliary hierarchical data structures[10,11,19,20,47] are proposed to avoid sampling in data regions having uniform or similar values.

However, three drawbacks exist for the general sampling reduction methods. First, the definition of homogeneous regions will change when the data undergo successive classification. Second, it takes more time to access hierarchical data structure than the regular data structure. Third, the special handling on data structure is data-dependent and can hardly be integrated into hardware implementation of ray casting.

1.2 Coherence reuse

Following the spirit of sampling reduction, the ways of adjusting sampling steps[16−18] are proposed to decrease the sampling rate in both empty and homogeneous regions. To explore the coherence among pixels, Ref.[21] proposes an interval ray casting algorithm by which rays are generated conditionally within the interval between two pixels. Ray template method, presented in Ref.[22], exploits the inter-ray coherence for orthogonal ray casting. On the other hand, Coordinate-buffer method[14] stores the coordinate of the first non-transparent voxel encountered by each ray. The buffer is then transformed to the next frame for space-leaping, yielding inter-frame reuse. In view of the coherence between right sight and left sight, Adelson *et al*.[24] present a stereoscopic speedup technique. To investigate the temporal coherence, Ref.[48] utilizes image caches and *isomap* to maintain the intermediate results for rendering the next images. Other object-space reuse methods try to exploit sample-memory efficiency for large sample-throughput, depicted in Refs.[19,25].

1.3 Hardware assisted acceleration

Real-time ray casting makes great demands on data storage, computation and data communication. Several ray casting architectures, including Refs.[26,36−39,41,42,49], have been proposed to overcome the above limits. However, they are designed for special purpose and are more expensive compared with commodity graphics accelerators. Recently, promoted by the fast increase in the performance of graphics hardware, hardware accelerated 2D/3D texture slicing methods[26,28−35] have been proposed and they attract lots of attentions. The efforts to hardware accelerated ray casting[45] have also been started though the results reported are not comparable to those by texture slicing methods yet.

2 View Dependent Layer Sampling

The current ray casting algorithm processes the volume data set ray by ray, and the computation of each ray is of $O(mn^2)$ complexity, where n corresponds to the data resolution and m is the traversing steps. Each ray accesses the volume data independently, making no use of the coherence of data references between adjacent rays during data sampling and interpolation. The memory bandwidth demanded by the complexity of the conventional ray casting algorithm thus puts an upper limitation on its performance. For a typical 256^3 volume data, the required memory access is 1.92 GB/s for 30Hz frame rates, which can hardly be sustained on most modern PCs. Note that, the new graphics accelerator in PC-ATI Radeon X800 XT possesses 35.8GB/s peak memory bandwidth. This fact encourages us to exploit GPU for hardware acceleration besides the known ray casting architectures[49].

Current consumer graphics hardware (see Fig.1) is designed to facilitate triangle rendering, offering no direct support to image space based ray casting pipeline. The difference between them is illustrated in Fig.1 (Top: a typical ray casting pipeline consists of six steps. Bottom: a common geometric pipeline is regarded as a streaming processor for vertices/pixels).

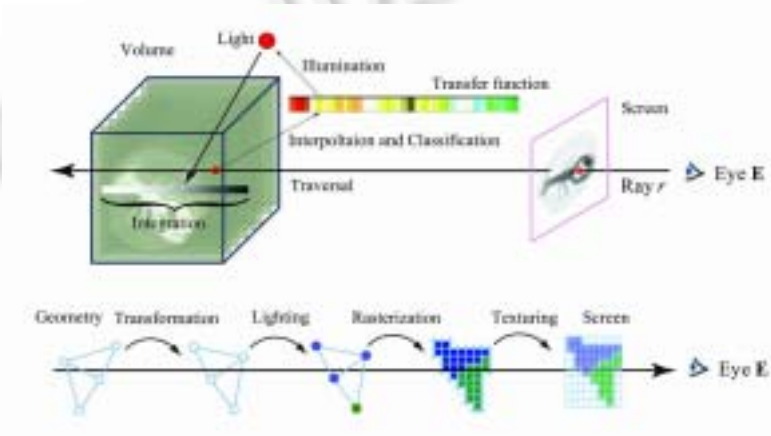


Fig.1 Ray casting pipeline versus geometric pipeline

Suppose that the bounding volume of the data set is **B**, given viewpoint **E** and a ray r, there are normally two intersection points between r and **B**. For FTB mode, they are denoted as entry point and exiting point respectively. The traversal of r begins at its entry point t_0 and ends at its exit point t_{n-1}. The sampling points t_i (i=0...n−1) constitute the ray sampling list of ray r, which contributes to the intensity of the corresponding pixel. The contributions of sampling lists of all rays compose the final image. To exploit the coherence of adjacent rays, we reorganize all sampling points into a number of layers—$\mathbf{Q}_i$, which are sorted along the viewing direction. If there are maximum n sampling points along a ray, correspondingly there are n layers $\mathbf{Q}_i$. The projection of $\mathbf{Q}_i$ (i=0...n−1) on screen overlaps within a projected geometry shape P, as shown in Fig.2. $\mathbf{Q}_i$ together with the projected shape **P** are defined as VDLS (view dependent layer sampling).

The projected shape **P** of the bounding volume **B** is a convex polygon. The number of its vertices ranges from 4 to 7 depending on its relative location to the current view frustums (See vt_i in Fig.3). When the whole volume falls inside the view frustum, **P** is totally visible. Otherwise, **P** should be clipped against the viewing window. Obviously, **P** can be partitioned into a set of triangles, and sampling points on each layer $\mathbf{Q}_i$ (i=0...n−1) are attached to the respective triangles of **P** for GPU processing and intensity composition.

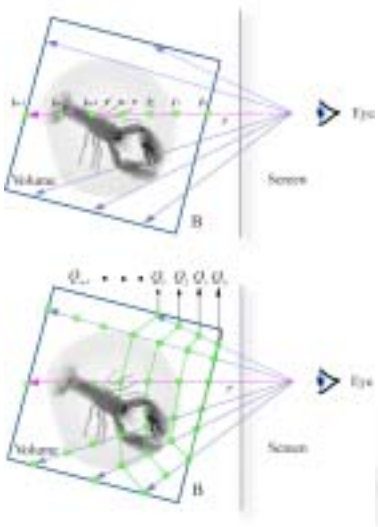


Fig.2 The definition of view dependent layer sampling (in 2D case)

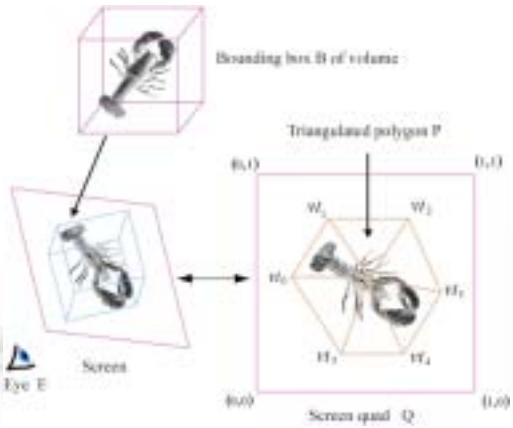


Fig.3 Projected geometry shape of the ray shells onto screen

3 VDLS Based Ray Casting Algorithm

Since the triangulated polygon **P** can be efficiently processed in geometric pipeline, the data structure of VDLS enables efficient mapping from ray casting pipeline to programmable geometric pipeline.

Here, six steps involved in classic ray casting are grouped into three main tasks, i.e, geometric transformation, traversing in vertex shader and illumination in pixel shader, as shown in Fig.4.

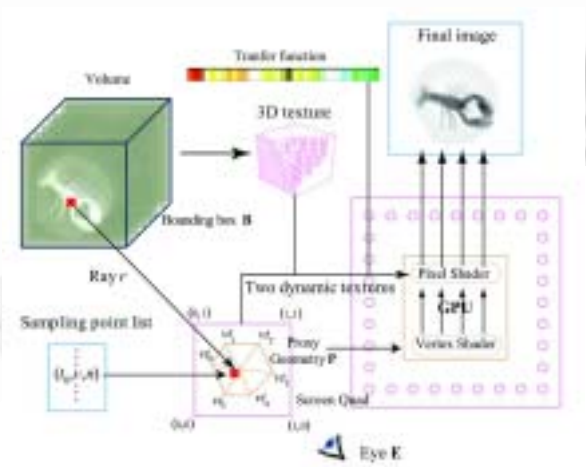


Fig.4 Mapping ray casting pipeline to programmable geometric pipeline (in 2D case)

3.1 Geometric transformation of volume

The volume data are initially embedded in the bounding box **B**. We then encode all layers $\mathbf{Q}_i$ (i=0...n–1) by two textures: one corresponds to the normalized coordinates of all sample points on the current layer with respect to the 3D volume, and the other records the incremental vector of parameter between two adjacent sampling points along

each ray *r*. We refer to these two textures as layer sample texture and delta vector texture respectively. The layer sample texture is initialized by finding the entry points of all rays with respect to the boundary surface of **B**. This can be accomplished by rendering the bounding box into textures with the render-to-texture functions and by applying the appropriate depth comparison functions. The delta vector texture can be set up by finding the exit point of each ray, then dividing the segment of the ray within the bounding box evenly into an appropriate number of intervals so that any pair of the adjacent layers inside the data volume is separated by a uniform distance in any viewing direction. The number of sample points along each ray can then be determined. This number is kept in the second texture to facilitate the early outside-volume testing. With the delta vector texture, the sample points on the next layer can be easily obtained by adding the delta vector of each ray to the sample points on the current layer, and the layer sample texture is updated dynamically. Note that, both textures are floating point textures.

Further, we rasterize each face of **B** by an optimized scan line algorithm in CPU. It is expensive to implement it in GPU because the access operations from frame buffer is time-consuming. In contrast, the cost for a software implementation is negligible compared with the performance enhancement of the integration of VDLS. Note that the geometric transformation is performed only if the viewing parameters or transfer functions are changed.

3.2 Volume traversing in vertex shader

With the information stored in the two textures, the volume data can be traversed and sampled in either FTB or BTF mode. Figure 4 illustrates the mapping from ray casting to GPU pipeline. Sample points are processed layer by layer (See $\mathbf{Q}_i$ in Fig.2). This is entirely different from the approach of CPU hosted ray casting by which traversal and intensity composition is conducted ray by ray. Note that all data are commonly represented as textures in GPU. Let *L* be layer sample texture of the current layer and Dv be the delta vector texture regarding the current viewing direction, then the sample points on the next layer can be obtained as follows:

$$L[i,j]=L[i,j]+\Delta v[i,j] \tag{1}$$

where $L[i,j]$ on the right side of Eq.(1) denotes the 3D parametric coordinates of a corresponding sample point on the current layer with a potential intensity contribution to $pixel[i,j]$, and $\Delta v[i,j]$ represents the incremental vector of the respective ray passing through $pixel[i,j]$.

Eq.(1) can be calculated conveniently in programmable pixel shader.

3.3 Volume illumination in pixel shader

With the 3D parametric coordinates of a sample point, we can easily find the 8 nearest neighboring points surrounding the sample point and get access to their density values in the 3D volume data array. Tri-linear interpolations are performed by GPU to determine the density value of current sample point. Then the next step is to map the density value to a color value for illumination calculation. We adopt a 1D look up table to fulfill the purpose. For illumination models incorporating a surface normal of each sample point, we pre-compute the gradient at each voxel, scale and bias the value to unsigned integers, and store them with another 3D array. With the lookup table and the gradient volume, the calculation of intensity of each sample point takes the following general formula:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = k_d I_d \begin{pmatrix} R(P) \\ G(P) \\ B(P) \end{pmatrix} + k_a \begin{pmatrix} R(I_a) \\ G(I_a) \\ B(I_a) \end{pmatrix}, \alpha = \alpha(P) \tag{2}$$

where *P* denotes the density at the current sample point; $R(P)$, $G(P)$, $B(P)$ and $\alpha(P)$ are picked from the lookup table, k_a and k_d denote the ambient and diffuse coefficients respectively, and I_d is the diffuse intensity.

4 Extensions of the Basic Algorithm

In this section, we describe our efforts on algorithm optimization by exploiting empty space leaping technique and image space coherence. We also propose a technique to accomplish early outside-volume testing. By means of segmented ray traversal technique, multi-stage sampling and blending can be implemented in pixel shader. Furthermore, six post-shaded filter modes are developed in our ray casting engine.

4.1 Homo-Regions for space leaping

To skip empty space conveniently, we perform a pre-process before constructing VDLS. Since the density value ranges from 0 to 255, we sort all voxels in the data volume into 256 iso-value lists by bucket sorting, adjacent voxels in the same bucket form homogenous 3D regions which are called homo-regions. Note that homo-regions are view independent. During the process of classification, the homo-regions whose density values are off interests are recognized as transparent while others are treated as semi-transparent or opaque. The union of non-transparent homo-regions determines the significant projected area of proxy shape **P** on screen. By replacing the entire data volume with a number of non-transparent homo-regions, we can reduce the computation cost by efficiently skipping regions of little interests during traversal.

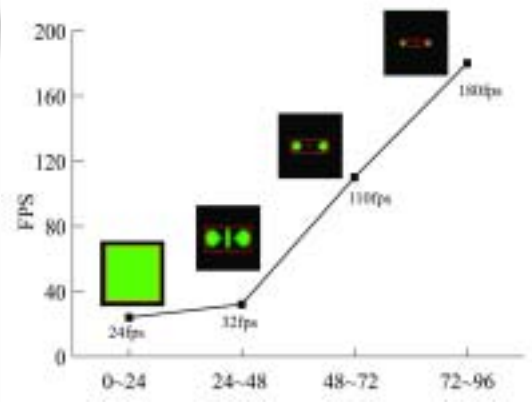


Fig.5 Rendering performance of different iso-value intervals for a hydrogenatom data set (128×128×128)

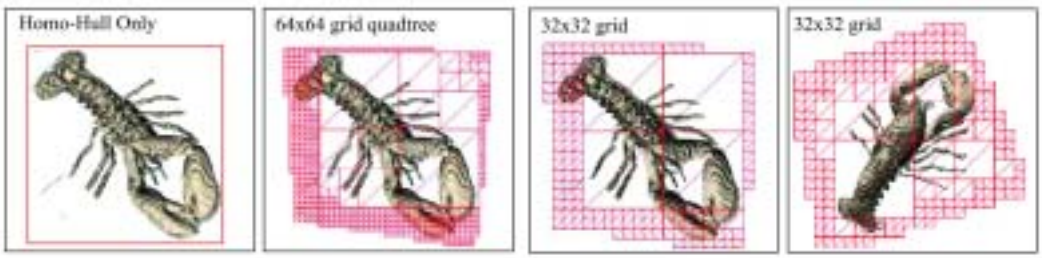
Typically, the efficiency of the empty space leaping technique depends on the density distribution of the raw volume data. We adopt the bounding box of each non-transparent homo-region as a space leaping contour. The operation for finding the union of the non-transparent homo-regions can be accomplished by at most 256×6 floating point comparison operations. An example of the performance enhancement is shown in Fig.5 (Red rectangles in images outline the valid projected ranges. Image resolution: 512×512), where four adjacent iso-value intervals are selected as opaque respectively.

4.2 Dynamic screen quadtree

Note that the number of sampling points along different rays are not the same, it is less efficient to process all regions of the proxy shape **P** at the same time which corresponds to the global maximal sampling number among all rays passing through the screen. To save the unnecessary processing time we construct the VDLS adaptively and dynamically based on a screen quadtree structure. Considering an $n \times n$ frame buffer where n is assumed to be a power of 2. The region of the screen is initially divided into a quadtree with all leaf nodes of the same size. For example, a screen at 512×512 can be partitioned into a 32×32 squares. During the scan-conversion of the non-transparent homo-regions, we record the local maximum number of sample points (LMSP for short) regarding all rays passing through each leaf node. We then reconfigure the quadtree by recursively combining the brother leafs whose LMSPs differ from each other by some threshold and assign the maximum sampling number to LMSP of the father node. The construction of the dynamic screen quadtree is adaptive and adjustable. The root of the dynamic image quadtree represents the whole screen. If we take the leaf node of one pixel size, then at the finest level, our algorithm is equivalent to the standard ray casting algorithm.

The construction of dynamic screen quadtree is conducted in CPU. It substitutes the proxy shape **P** by active

nodes. During GPU assisted volume rendering, each active node of quadtree accounts for local areas on a limited number of layers. Since most active nodes relate to the reduced number of layers, redundant traversal across the entire **P** is avoided all the time. It is found that adaptive selection of tree level will optimize the efficiency to a great extent. Figure 6 demonstrates the setup of two dynamic screen quadtrees with different resolutions (Fig.6(a): Homo-hull only(Red rectangle outlines the valid projected range): 6.5 fps; Fig.6(b): Dynamic screen quadtree with initial status of 64×64 uniform grid: 8.0 fps; Fig.6(c): Dynamic screen quadtree with initial status of 32×32 uniform grid: 8.4 fps; Fig.6(d): Another view of the case Fig.6(c). Testing with a CT lobster data set (301×324×56): 8.2 fps. Image resolution: 512×512).



(a) (b) (c) (d)

Fig.6 A comparison of w/o dynamic screen quadtree

4.3 Segmented ray traversal

To further improve efficiency, we propose two techniques, i.e., early outside-volume testing and segmented ray traversal. Notice that screen quadtree approximates the projected area of each non-transparent homo-region with a number of squares. For some leaf nodes at the lowest level, the number of effective sample points along different rays may still vary greatly, some rays may lie completely outside the homo-region, contributing no sample points to the pixel shader. To account for this, we perform an early outside-volume testing for each ray in the leaf node by comparing the pre-determined number of sample points of each ray (kept in the delta vector texture) with the number of layers that have been processed.

The second technique (segmented ray traversal) tries to reduce the processing times of each homo-region in virtue of segmented ray traversal. We then divide the segment of a ray inside the current homo-region into several sub-segments accordingly so that the adjacent sample points located within each sub-segment can be processed conjunctively. The contributions from all sample points on a sub-segment are calculated by composition operation in pixel shader and blended for output. Currently, maximal six dependent texture loading is available in ATI 9800 Pro video card and hence we can process two layers during a single turn of scanning.

4.4 Post-Shaded filters

The proposed algorithm supports different post-shaded filters for both of the volume shading and surface-like rendering (see Table 1). These filters map the density value of each sample point to intensity and opacity values for post-shading classification. Typically, they are represented by transfer function lookup tables. Users can configure these filters interactively and apply them to interpolated density in pixel shader. A frequently used filter is the threshold filter for iso-surface like rendering in conjunction with gradient information. Another color filter assigns values of color and transparency corresponding to density values. Both of them can be stored as 1D textures and applied to the interpolated raw volume data, resulting in post-shading effect. Besides the illumination model for surface-like rendering, our algorithm provides four volume-like illumination models: MIP (maximum intensity

projection), MIP (minimum intensity projection), MVP (mean value projection) and semitransparent rendering. They can be implemented fully in pixel shader.

Table 1 Posted shaded filters

Filter mode	Functions
Iso-Surface filter	if ($\min_d$<D<$\max_d$): $R(D)=G(D)=B(D)=\alpha(D)=1.0$; else: $R(D)=G(D)=B(D)=\alpha(D)=0.0$;
Arbitrary filter	$R(D)=AnyValue$, $G(D)=AnyValue$, $B(D)=AnyValue$, $\alpha(D)=AnyValue$;
Maximum filter	if ($\max D<D$): $R(D)=G(D)=B(D)=\max D=D$;
Minimum filter	if ($\min D>D$): $R(D)=G(D)=B(D)=\min D=D$;
Mean value filter	$R(D)=G(D)=B(D)=\Sigma D/SamplingNumber$;
Semi-Transparent	$R(D)=G(D)=B(D)=\alpha(D)=D$;

5 Results and Discussions

Our algorithm has been implemented on a PC equipped with one Pentium 4 CPU at 2.4G Hz with 2G RAM, and Radeon 9800 Pro graphics accelerator with 256M RAM. We try to eliminate their effect by collecting the performance under the same experimental setting, e.g., testing platform, window size, viewing parameters and ray sampling distance. In all experiments, the zooming factors are so determined that the bounding box of the whole volume covers the screen. Uniform sampling at distance of one voxel is used throughout the experiments. The fps shown in tables is collected by rotating the volume around its centroid randomly for hundreds of frames and averaging the total time.

5.1 Performance

We demonstrate the performance by rendering six volume data sets with gradient-based surface illumination model. Homo-regions (HR), dynamic screen quadtree (DVQ) and segmented ray traversal (SRT) techniques are enabled for acceleration. Table 2 lists the data size, video memory consumption, performance of the basic VDLS method and the enhanced versions, e.g., without/with those speedup techniques. In each item of performance, the number of frames and sampling points processed per second during interactive rendering are shown. The latter includes operations of tri-linear interpolation, post-shading and intensity composition and the number is in unit of million. It is clearly shown from Table 2 that our algorithm can support interactive rendering of moderate sized volume data up to 256^3. Corresponding images of the tested data sets are shown in Fig.8, and all rendered with gradient based surface illumination model.

Table 2 Performance statistics in fps for a sequence of data sets

Data	Size	Memory	Basic	Enhanced
Harmonic	32^3	128 KB	20.9	26.8
Function	64^3	1 MB	16.3	20.1
Earth Crust	$128^2\times64$	4 MB	15.1	18.1
MRI Head	128^3	8 MB	10.0	14.3
Bonsai Tree	$256^2\times128$	32 MB	4.5	8.0
CT head	$256^2\times225$	60 MB	4.0	7.0

To further investigate the effect regarding image size and data size on rendering speed, we list the performance with respect to volume data sizes (Fig.7, image resolution: 512×512; illumination model: gradient-based surface), image resolutions (Fig.8 left: comparison of performance under different image resolutions and constant sampling rates (2× sampling)) and sampling rate (Fig.8 right: comparison of performance under different sampling rates and constant image resolutions (512×512)). The experimental data sets in Fig.8 (Testing model: MRI head data sets

shown in Fig.10(d); Volume illumination mode: semitransparent) are a set of medical CT data while the left and right parts of Fig.8 apply respectively a 128×128×128 MRI head data set and a 41×41×41 Marschner function data set[52]. It can be concluded that our new approach is basically an image-space algorithm and its performance is proportional to the data size, image resolution and sampling rate. The classification is independent of the sampling procedure since the post-shading is accomplished by employing a lookup table in pixel shader. In addition, the empty space leaping and cache coherence are not data-sensitive and work well regardless of the applied illumination models.

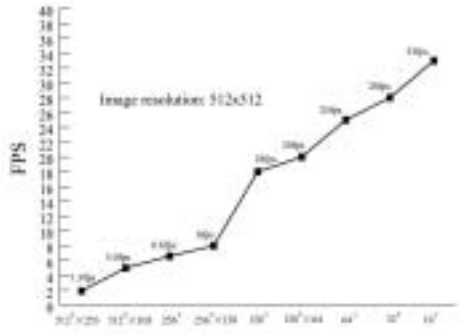


Fig.7 Comparison of performance under different data sizes

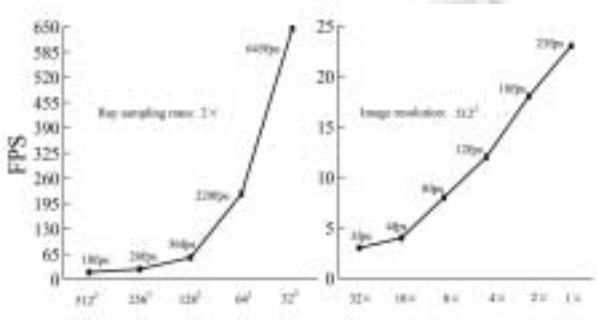


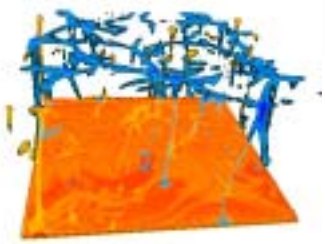
Fig.8 Comparison of performance under different image resolutions and sampling rates

5.2 Image quality

We applied the new approach to a CT Engine data set (see Fig.10) with six different illumination models. The rendering speed is about 8~10 fps, and on the image the features are faithfully reserved and visualized.

(a) CT harmonic function(32×32×32)

(b) CT math function (64×64×64)



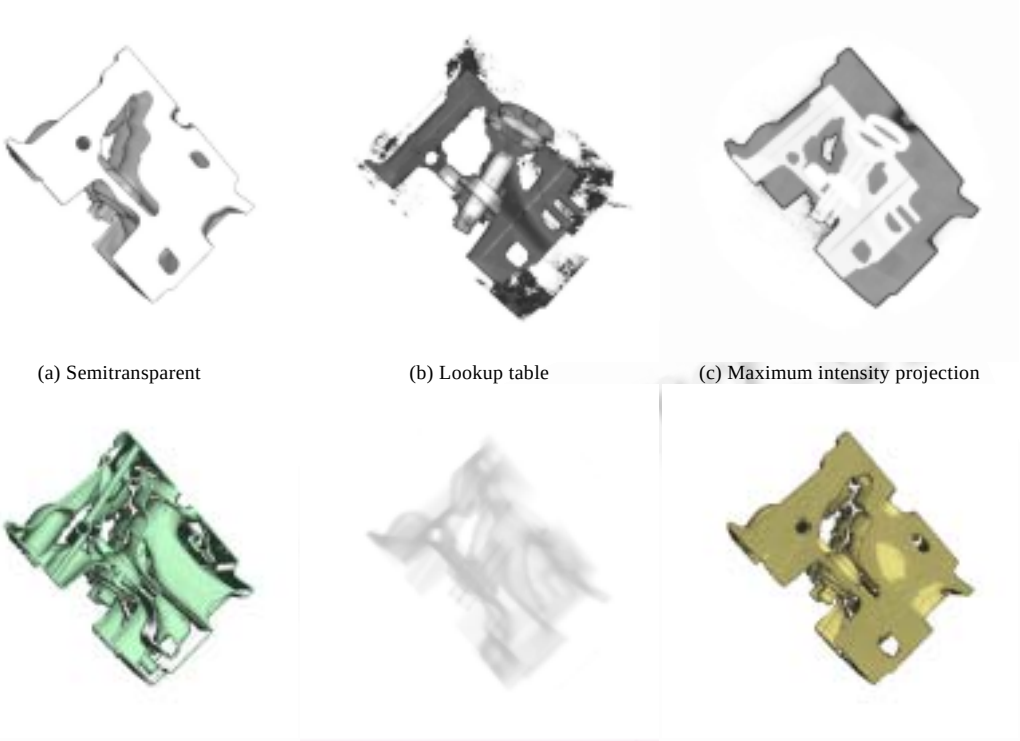
(c) CT earth crust (128×128×64)

(d) MRI head (128×128×128)

(e) CT Bonsai tree (256×256×128)

(f) CT head (256×256×225)

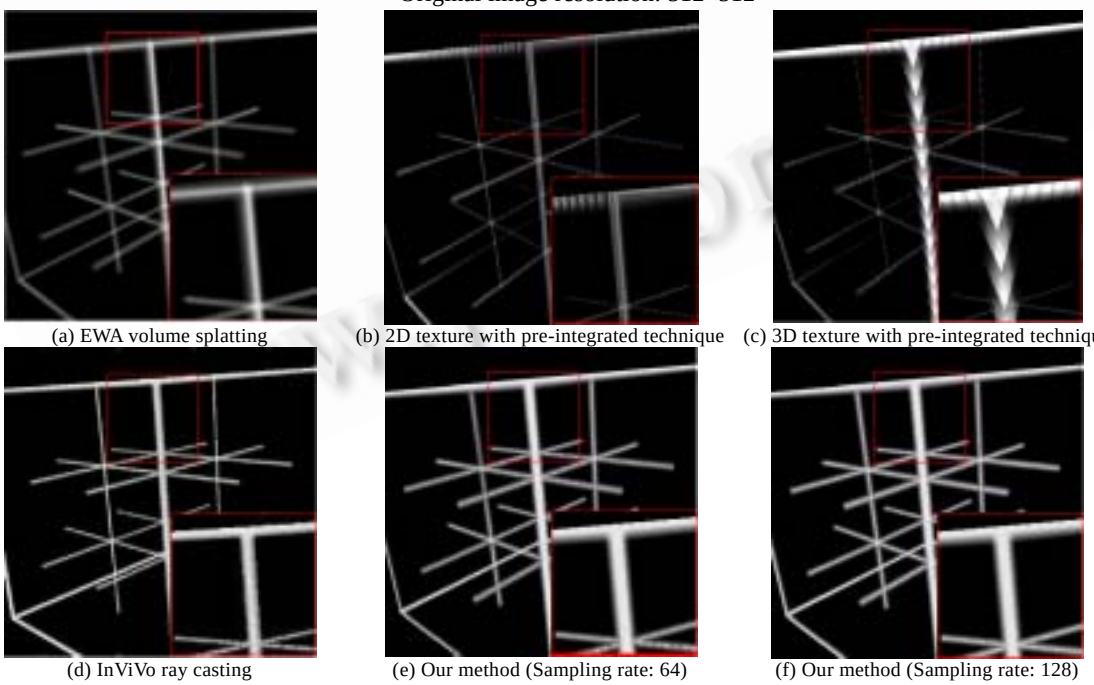
Fig.9 Rendering results of different data sets. Image resolution: 512×512



(a) Semitransparent (b) Lookup table (c) Maximum intensity projection

(d) Iso-Surface (e) Mean value projection (f) Gradient-Based surface

Fig.10 Rendering a 256×256×110 CT engine data with different illumination models. Original image resolution: 512×512



(a) EWA volume splatting (b) 2D texture with pre-integrated technique (c) 3D texture with pre-integrated technique

(d) InViVo ray casting (e) Our method (Sampling rate: 64) (f) Our method (Sampling rate: 128)

Fig.11 Snapshots of a phantom with different rendering methods. Note the artifacts shown in borders with 2D/3D texture slicing methods. Testing phantom data size: 64×64×64. Original image resolution: 512×512

5.3 Discussions

Representative but not complete, we compare the images (See Fig.11) produced by EWA volume splatting[53], 2D/3D texture slicing with pre-integrated techniques[32], InViVo ray casting[23] and our algorithm for a 64×64×64 phantom data set. We implemented a hardware accelerated EWA volume splatting algorithm, while the executable program of 2D/3D texture slicing methods were obtained from Dr.Klaus Engel. The InViVo ray casting testing environment was kindly afforded by Prof.Georgios Sakas, Fraunhofer Institute for computer graphics, Darmstadt, Germany. From Figure 11, it is observable that the texture slicing methods exhibit artifacts on the volume boundary. In addition, as pointed out in Ref.[45], 2D/3D texture slicing methods typically neglect the tri-linear interpolation and the non-linear behavior of lighting. The result of EWA volume splatting is a little bit blurring since it pre-integrates the 3D reconstruction kernel while ray casting is in spirit a 3D point sampling algorithm, which usually produces sharp image. Pre-integration technique[32] is a renowned technique that produces faithful image quality even if the sampling rate of the raw data is not adequate. It requires 4-time supersampling for good image quality as reported in Ref.[45]. In Ref.[45], adaptive pre-integration, a form of space leaping is introduced for volume ray casting on graphics hardware. However, a so-called importance volume has to be pre-computed according to the transfer function, which imposes a heavy overhead on the real-time system. This mechanism makes it infeasible for interactive transfer function changes, which is typically a mandatory requirement on visualization application. The techniques presented in Ref.[46] make use of the opacity comparison for early ray termination, which brings overhead for volumetric illumination models including semi-transparent and MIP rendering *etc*. In contrast, as depicted in our performance report, our new approach overcomes the above two obstacles in some extent by exploiting the object space and image space coherence. It seems to be a more general framework supporting arbitrary transfer functions and illumination models at a constant rendering speed.

6 Conclusions and Future Work

In this paper, we present a new approach: GPU supported view dependent layer sampling to hardware implementation of volume ray casting which has the following advantages:

- Low volume-memory bandwidth requirements.
- Increased scalability over image-space algorithms.
- Support of interactive classification.
- Less dependence of performance on viewing parameters, classification mappings, data set type and illumination models.
- High adaptability to application-oriented visualization tasks.

Nevertheless, two problems still exist as GPU-related overhead: the slow speed for dynamic screen quadtree configuration and the limitation of texture accessing number in pixel shader. Hopefully future hardware development will eliminate these bottlenecks soon. Future work should extend our algorithm to support arbitrary volume clipping and mixing surface/volume rendering. Designing more reasonable filter and incorporating pre-integration technique are also in our study schedule. For speedup issues, although not presented in this paper, we have obtained some initial results on facilitating the early ray termination by means of occlusion map combined with dynamic screen quadtree.

References:

[1] Kajiya T, Berzen B. Ray tracing volume densities. In: Glassner A, ed. Proc. of the ACM SIGGRAPH'84. New York: ACM Press, 1984. 165−174.

[2] Sabella P. A rendering algorithm for visualisation of 3D scalar fields. In: Dill J, ed. Proc. of the ACM SIGGRAPH'88. New York: ACM Press, 1998. 51−57.

[3] Drebin R, Carpenter L, Hanrahan P. Volume rendering. In: Dill J, ed. Proc. of the ACM SIGGRAPH'88. New York: ACM Press, 1998. 65−74.

[4] Levoy M. Display of surfaces from volume data. IEEE Computer Graphics and Applications, 1998,8(3):29−37.

[5] Meissner M, Huang J, Bartz D, Mueller K, Crawfis R. A practical evaluation of four popular volume rendering algorithms. In: Lorensen B, ed. ACM Symp. on Volume Visualization 2000. New York: ACM Press, 2000. 81−90.

[6] Sobierajski L, Kaufman A. Volumetric ray tracing. In: Kaufman A, ed. Proc. of the ACM Volume Visualization Symp. New York: ACM Press, 1994. 11−18.

[7] Upson C, Keller M. V-Buffer: Visible volume rendering. In: Dill J, ed. Proc. of the ACM SIGGRAPH'88. New York: ACM Press, 1998. 59−64.

[8] Wilhelms J, Gelder AV. A coherent projection approach for direct volume rendering. In: Proc. of the ACM SIGGRAPH'91. New York: ACM Press, 1991. 275−284.

[9] Westover L. Footprint evaluation for volume rendering. In: Baskett F, ed. Proc. of the ACM SIGGRAPH'90. New York: ACM Press, 1990. 367−376.

[10] Lacroute P, Levoy M. Fast volume rendering using a shear-warp factorization of the viewing transformation. In: Glassner A, ed. Proc. of the ACM SIGGRAPH'94. Orlando: ACM Press, 1994. 451−458.

[11] Levoy M. Efficient ray tracing for volume data. ACM Trans. on Graphics, 1990,9(3):245−261.

[12] Avlia R, Sobierajski L, Kaufman A. Towards a comprehensive volume visualisation system. In: Kaufman AE, Nielson GM, eds. Proc. of the IEEE Visualization'92. New York: IEEE Press, 1992. 13−20.

[13] Cohen D, Shefer Z. Proximity clouds—An acceleration technique for 3D grid traversal. Technical Report, TR-CS-92-11, Department of Computer Science, Australian National University, ACT, 1992.

[14] Yagel R, Shi Z. Accelerating volume animation by space-leaping. In: Nielson GM, ed. Proc. of the IEEE Visualization 1993. San Jose: IEEE Press, 1993. 62−69.

[15] Sramek M, Kaufman AE. Fast ray-tracing of rectilinear volume data using distance transforms. IEEE Trans. on Visualization and Computer Graphics, 2000,6(3):236−252.

[16] Brady M, Jung K, Nguyen HT, Nguyen T. Two-Phase perspective ray casting for interactive volume navigation. In: Hagen H, ed. Proc. of the IEEE Visualization'97. Phoenix: IEEE Press, 1997. 183−191.

[17] Parker S, Shirley P, Livnat Y, Hansen C, Sloan P. Interactive ray tracing for isosurface rendering. In: Ebert D, ed. Proc. of the IEEE Visualization'98. Washington: IEEE Press, 1998. 233−238.

[18] Neubauer A, Mroz L, Hauser H, Wegenkittl R. Cell-Based first-hit ray casting. In: Crawfis R, ed. Proc. of the 4th Joint IEEE TCVG—EUROGRAPHICS Symp. on Visualization 2002. Barcelona: IEEE Press, 2002. 77−86.

[19] Freund J, Sloan K. Accelerated volume rendering using homogeneous region encoding. In: Hagen H, ed. Proc. of the IEEE Visualization'97. Washington: IEEE Press, 1997. 191−196.

[20] Wan M, Kaufman AE, Bryson S. High performance presence-accelerated ray casting. In: Ebert D, ed. Proc. of the IEEE Visualization'99. Washington: IEEE Press, 1999. 379−386.

[21] Levoy M. Volume rendering by adaptive refinement. The Visual Computer, 1990,6(1):2−7.

[22] Yagel R, Kaufman A. Template-Based volume viewing. Computer Graphics Forum, 1992,11(3):153−167.

[23] Sakas G, Hartig J. Interactive visualization of large scalar voxel fields. In: Kaufman A, ed. Proc. of the IEEE Visualization'92. Washington: IEEE Press, 1992. 29−36.

[24] Adelson S, Hansen C. Fast stereoscopic images with ray—Traced volume rendering. In: Mueller K, ed. Proc. of the ACM Symp. on Volume Visualization'94. New York: ACM Press, 1994. 3−9.

[25] Mora B, Jessel J, Caubet R. A new object-order ray-casting algorithm. In: Proc. of the IEEE Visualization 2002. Washington: IEEE Press, 2002. 203−210.

[26] Eckel G. OpenGL Volumizer Programmer's Guide. CA: SGI Developer Bookshelf, 1998.

[27] Lindholm G, Kilgard M, Moreton H. A user-programmable vertex engine. In: Fiume E, ed. Proc. of the ACM SIGGRAPH 2001. Los Angeles, 2001. 149−158.

[28] Cabral B, Cam N, Foran J. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In: Yagel R, ed. Proc. of the ACM Symp. on Volume Visualization'94. New York: ACM Press, 1994. 91−98.

[29] Rezk-Salama C, Engel K, Bauer M, Greiner G, Ertl T. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In: Pfister H, ed. Eurographics/SIGGRAPH Workshop on Graphics Hardware. Interlaken, 2000. 109−118.

[30] Westermann R, Ertl T. Efficiently using graphics hardware in volume rendering applications. In: Levoy M, ed. Proc. of the ACM SIGGRAPH'98. New York: ACM Press, 1998. 169−178.

[31] Westermann R, Sevenich B. Accelerated volume raycasting using texture mapping. In: Ertl T, ed. Proc. of the IEEE Visualization 2001. Washington: IEEE Press, 2001. 363−371.

[32] Engel K, Kraus M, Ertl T. High quality pre-integrated volume rendering using hardware accelerated pixel shading. In: Ertl T, ed. Proc. of the Eurographics/SIGGRAPH Workshop on Graphics Hardware 2001. New York: ACM Press, 2001. 9−16.

[33] Guthe S, Wand M, Gonser J, Strasser W. Interactive rendering of large volume data sets. In: Pfister H, ed. Proc. of the IEEE Visualization 2002. Washington: IEEE Press, 2002. 53−60.

[34] Li W, Kaufman AE. Accelerating volume rendering with texture hulls. In: Mueller K, ed. Proc. of the 2002 IEEE Symp. on Volume Visualization. Washington: IEEE Press, 2002. 115−122.

[35] Li W, Mueller K, Kaufman AE. Empty space skipping and occlusion clipping for texture-based volume rendering. In: Bell G, ed. Proc. of the IEEE Visualization 2003. Washington: IEEE Press, 2003. 317−324.

[36] Kniss J, McCormick P, *et al*. Interactive texture-based volume rendering for large data sets. IEEE Computer Graphics and Applications, 2001,21(4):52−61.

[37] Ray H, Silver D. A memory efficient architecture for real-time parallel and perspective direct volume rendering. Technical Report, CAIP-TR-237, Rutgers University, 1999.

[38] Knittel G. The ULTRAVIS system. In: Ertl T, ed. Proc. of the IEEE Volume Visualization and Graphics Symp. 2000. Washington: IEEE Press, 2000. 71−79.

[39] Knittel G. High-Speed software raycasting on a dual-CPU PC using cache optimizations, MMX, streaming SIMD extensions, multi-threading and Directx. In: Strasser W, ed. Proc. of the SPIE Visual Data Exploration and Analysis VII 2000. San Jose: SPIE Press, 2000. 164−174.

[40] Pfister H, Kaufman AE. Cube-4: A scalable architecture for real-time volume rendering. In: Knittel J, ed. Proc. of the ACM Symp. on Volume Visualization'96. New York: ACM Press, 1996. 47−54.

[41] Meissner M, Kanus U, Strasser W. VIZARD II: A PCICard for real-time volume rendering. In: Ertl T, ed. Proc. of the Siggraph/Eurographics Workshop on Graphics Hardware'98. New York: ACM Press, 1998. 61−67.

[42] Pfister H, Hardenbergh J, Knittel J, Lauer H. The VolumePro real-time ray-casting system. In: Proc. of the ACM SIGGRAPH'99. New York: ACM Press, 1999. 251−260.

[43] Purcell T, Buck I, Mark WR, Hanrahan P. Ray tracing on programmable graphics hardware. ACM Trans. on Graphics, 2002,21(3): 703−712.

[44] Carr N, Hall J, Hart J. The ray engine. In: Olano M, ed. Proc. of the Eurographics-ACM Workshop on Graphics Hardware 2002, Sarrbruecken. New York: ACM Press, 2002. 37−46.

[45] Roettger S, Guthe S, Weiskopf D, Ertl T, Strasser W. Smart hardware-accelerated volume rendering. In: Bonneau GP, Hahmann S, eds. Joint EUROGRAPHICS-IEEE TCVG Symp. on Visualization 2003. Grenoble: Eurographics Assoicaition, 2003. 231−238.

[46] Krueger J, Westermann R. Acceleration techniques for GPU-based volume rendering. In: Bell G, ed. Proc. of the IEEE Visualization 2003 Seattle. Washington: IEEE Press, 2003. 38−45.

[47] Knittel G. High-Speed volume rendering using redundant block compression. In: Nielson GM, Silver D, eds. Proc. of the IEEE Visualization 1995. Atlanta: IEEE Press, 1995. 176−183.

[48] Yoon I, Demers J, Kim T, Neumann U. Accelerating volume visualization by exploiting temporal coherence. In: Hagen H, ed. Proc. of the IEEE Visualization 1997 Late Breaking Hot Topics, Phoenix. Washington: IEEE Press, 1997. 62−69.

[49] Ray H, Pfister H, Silver D, Cook TA. Ray casting architectures for volume visualization. IEEE Trans. on Visualization and Computer Graphics, 1999,5(3):210−223.

[50] Porter T, Duff T. Compositing digital images. In: Christiansen H, ed. Proc. of the ACM SIGGRAPH'84. New York: ACM Press, 1984. 253−259.

[51] Wittenbrink C, Malzbender T, Goss M. Opacity-Weighted color interpolation for volume sampling. In: Kaufman A, ed. Proc. of the ACM Symp. on Volume Visualization'98, Research Triangle Park, NC. New York: ACM Press, 1998. 135−142.

[52] Marschner S, Lobb RJ. An evaluation of reconstruction filters for volume rendering. In: Bergeron D, Kaufman AE, eds. Proc. of the IEEE Visualization'94. Washington: IEEE Press, 1994. 100−107.

[53] Zwicker M, Pfister H, van Baar J, Gross M. EWA volume splatting. In: Ertl T, ed. Proc. of the IEEE Visualization 2001. San Diego: IEEE Press, 2001. 29−36.

CHEN Wei was born in 1976. He received his Ph.D. degree from the Applied Mathematics Department of Zhejiang University in 2002 and is currently an associate professor at the State Key Laboratory of CAD&CG, Zhejiang University. His current research areas are visual computing and real-time graphics.

BAO Hu-Jun was born in 1966. He received his Ph.D. degree from the Applied Mathematics Department of Zhejiang University in 1993 and is currently a professor and doctoral supervisor at the State Key Laboratory of CAD&CG, Zhejiang University. His research areas are virtual reality, computer animation and visualization.

PENG Qun-Sheng was born in 1947. He received his Ph.D. degree from the School of Computing Studies of East Anglia University, UK in 1983. Now he is a professor and doctoral supervisor at the State Key Laboratory of CAD&CG, Zhejiang University, and a CCF senior member. His research areas are virtual reality, computer animation, visualization and infrared simulation.