Computing and Informatics, Vol. 31, 2012, 551–571

AN INTERACTIVE CONCAVE VOLUME CLIPPING METHOD BASED ON GPU RAY CASTING WITH BOOLEAN OPERATION

Feiniu YUAN

School of information technology
Jiangxi University of Finance and Economics
Nanchang 330032, Jiangxi, China
e-mail: yfn@ustc.edu

Communicated by Dieter Kranzlmüller

Abstract. Volume clipping techniques can display inner structures and avoid difficulties in specifying an appropriate transfer function. We present an interactive concave volume clipping method by implementing both rendering and Boolean operation on GPU. Common analytical convex objects, such as polyhedrons and spheres, are determined by parameters. So it consumes very little video memory to implement concave volume clipping with Boolean operations on GPU. The intersection, subtraction and union operations are implemented on GPU by converting 3D Boolean operation into 1D Boolean operation. To enhance visual effects, a pseudo color based rendering model is proposed and the Phong illumination model is enabled on the clipped surfaces. Users are allowed to select a color scheme from several schemes that are pre-defined or specified by users, to obtain clear views of inner anatomical structures. At last, several experiments were performed on a standard PC with a GeForce FX8600 graphics card. Experimental results show that the three basic Boolean operations are correctly performed, and our approach can freely clip and visualize volumetric datasets at interactive frame rates.

Keywords: Medical image processing, volume clipping, GPU ray casting, Boolean operation

1 INTRODUCTION

Consumer level graphics cards have powerful computation performance. With the rapid development of the game industry, modern graphics cards already provide a flexible degree of programmability on graphics processing unit (GPU) that opens up a wide field of time-consuming applications. There are two main categories of volume rendering based on graphics cards. The first category originally presented by Cullip and Neumann [1], and further developed by Cabral et al. [2] is directly exploiting the texture mapping capabilities of graphics hardware by proxy re-sampling planes. The second one is to implement volume rendering on GPU. For example, Kruger and Westermann [3] proposed a GPU based ray casting using a two-pass rendering approach. Rler et al. [4] presented a GPU based multi-volume ray casting algorithm. Bentoumi et al. [5] proposed a GPU implementation of the shear-warp algorithm. Marroquim et al. [6] proposed a method for volume and iso-surface rendering with GPU-accelerated cell projection. They also used two-pass rendering techniques. Furthermore, GPU is widely applied in surface graphics. Gerd Reis et al. [7] presented high-quality rendering of quartic spline surfaces on the GPU. Charles and Blinn [8] proposed a real-time GPU algorithm for rendering piecewise algebraic surfaces. Reimers and Seland [9] presented an algorithm for interactive ray-casting of algebraic surfaces of high degree. Michael et al. [10] presented GPU-based trimming and tessellation of NURBS and T-Spline surfaces. Dyken et al. [11] implemented an approach for Marching Cubes on graphics hardware, which currently outperforms all other known GPU based iso-surface extraction algorithms in direct rendering for sparse or large volumes. Kim et al. [12] presented vertex transformation streams based on GPU, which addressed the input geometry bandwidth bottleneck for interactive 3D graphics applications. Flexibility of GPU improves parallel computation performance in many time-critical applications. For example, Fialka and Cadk [13] implemented FFT and convolution on GPU, Kruger and Westermann [14] presented linear algebra operators for GPU implementation of numerical algorithms, Jin et al. [15] used GPU to design marbling textures.

Although GPU based volume rendering techniques can provide users with immediate visual feedback, rapidly specifying an appropriate transfer function is still a very difficult task. Therefore, volume clipping plays an important role in exploring 3D datasets because it allows users to cut away undesired parts of the volume. So the volume clipping technique can be regarded as a good complementary tool to avoid difficulties in transfer function design. Clip planes are frequently used in texture-based volume rendering by simply enabling the clip functionality of the OpenGL. Van Gelder and Kim [16] used clip planes to specify the boundaries of the dataset in 3D texture-based volume rendering; but clip planes only produce convex geometries. Westermann and Ertl [17] presented stencil based volume clipping. The clip object has to be rendered for each slice to correctly set the stencil buffer. Xie et al. [18] proposed CFD based volume rendering to implement Boolean operations using voxelized objects; but the clipping object consumes a large video memory due to voxelization. Weiskopf et al. [19] presented volume clipping techniques based on

a volumetric description of clip objects. A depth based clipping technique analyzes the depth structure of the clip geometry to decide which regions of the volume have to be clipped. Another approach allows a clip object to be voxelized and represented by a 3D volumetric texture. The approach allows users to specify arbitrarily structured clip objects and it is a very fast technique for volume clipping with complex clip geometries. Another depth-based method is used for depth sorting semi-transparent surfaces [20]. The technique is related to virtual pixel maps [21] and dual depth buffers [22]. Tiede et al. [23] used a similar method to visualize attributed volumes by ray casting. More recently, Williams et al. [24] presented a volumetric curved planar reformation for virtual endoscope and fully analyzed projected modes. It belongs to one kind of complex shaped clip object because a centerline of tubular object must be extracted before the planar reformation.

In this paper, an interactive concave volume clipping method is proposed by implementing both rendering and Boolean operations on GPU. It is fast and useful for viewing inner structures without careful specification of the transfer function. Common analytical convex objects are used to implement volume clipping on GPU, due to their compact representation and little memory consumptions. These analytical convex objects with Boolean operations can form complicated structures. To obtain clear views of inner structures within medical datasets, a pseudo color based rendering model is proposed and the Phong illumination model is enabled on the clipped surfaces. The main contribution of this paper is an interactive concave volume clipping approach by implementing Boolean operation, rendering, and pseudo color mapping together on GPU.

This paper is organized as follows. Section 2 describes the concept of rendering segments for rendering a convex object. Section 3 describes the Boolean operation method based on 1D rendering segment. Section 4 briefly presents a pseudo color based rendering model. The detailed GPU implementation of the Boolean operation based volume clipping techniques is followed in Section 5. At last, some experiments are performed and conclusions are drawn.

2 RENDERING SEGMENTS

In this section, in order to easily understand our method, the classical GPU ray casting algorithm is reviewed first, and then the algorithm is modified thus to introduce the concept of rendering segment.

Stegmaier et al. [25] presented the classical GPU ray casting. In their algorithm, a bounding box is used to assume that each component of texture coordinates is always within the normalized range $[0, 1]$. The color of the front surface is regarded as coordinates of a start point for the ray casting process and the color of the back surface stands for coordinates of an end point. The ray direction at any pixel can be computed by simply subtracting the front color from the back color. In their algorithm, two pass rendering is required. To render the back faces, front face culling is enabled. An OpenGL frame buffer object is used to store the result from

the rendering of the back faces, and the front faces are rendered to generate the fragments that start the ray casting process until the ray encounters the fragments of back faces. Thus, the ray casting can achieve interactive frame rates.

As an alternative, the start and end points can also be computed by solving intersections of the ray casting from a proxy plane and the analytic geometry on GPU. It is called single plane based method by us. In both the classical algorithm and single plane based method, the ray direction D_w, the start point P_s and the end point P_e must be computed first. According to the analytic geometry, the parameterized equation of a ray is defined as

$$P_w.xyz = P_s.xyz + Dw \times t \tag{1}$$

where P_w is the 4D homogeneous coordinates $(x, y, z, 1)$, P_s is the start point, t is a signed distance from the start point P_s. Positive t implies that the point P_w is in the front of the point P_s along the direction D_w and has distance t from it, and if t is negative, the point P_w is in the back of the start point P_s with distance t. We adopt the symbol denotation used by the OpenGL shading language, that is, $P_w.xyz$ stands for a new vector consisting of the first three components of the 4D homogeneous coordinates.

According to a definition of convex object, a ray has no more than two points of intersection with any convex object. As shown in Figure 1, a ray Ray1 has two points of intersection with a convex object, which are called a start point and an end point represented by the distance parameters t_s and t_e, respectively. The ray Ray2 can be considered to have two points of intersection which are overlapping and ray Ray3 has no intersection with the convex object. We call the ordinal pair of two values (t_s, t_e) a rendering segment. The absolute value of $|t_e - t_s|$ is called its length. The GPU ray casting algorithm performs volume integral from the start point t_s to the end point t_e to render the desired part of volume within the convex clip object. If there is no intersection, we think that the rendering segment is empty. If the rendering segment is empty or has a length less than the re-sampling interval, the rendering segment can be discarded immediately.

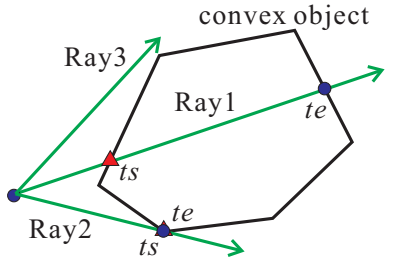


Fig. 1. Points of intersection between a ray and a convex object

In both the classical and single plane based GPU ray casting methods, the bounding box which is a convex clip object is required to prevent texture coordinates

out of the normalized range. The bounding box is a special polyhedron which is defined by six planes. In other words, convex polyhedron volume clipping must be performed first in our algorithm. Since any polyhedron can be represented as a series of planes, polyhedron volume clipping is actually the same as multiple planar clipping.

3 CONCAVE VOLUME CLIPPING USING BOOLEAN OPERATION

Methods with multiple clip planes can only implement convex clipping, but cannot perform concave clipping. So we make use of Boolean operations of analytical convex objects in order to perform concave volume clipping on GPU.

We implement three basic Boolean operations, which are *intersection*, *union* and *subtraction*. The intersection operation removes everything except the overlapping areas of the two objects. The subtraction operation removes the overlapping portion of the second object from the first object. The union operation joins all selected objects into a single object. The *intersection*, *subtraction* and *union* operations are represented as mathematical multiplication, subtraction and addition, respectively.

In our implementation of concave volume clipping, multiple planar volume clipping must be first performed to limit texture coordinates to the normalized range, and then some additional convex objects, such as polyhedrons and spheres, are used to actually cut away and keep parts of volume using different combination of Boolean operations.

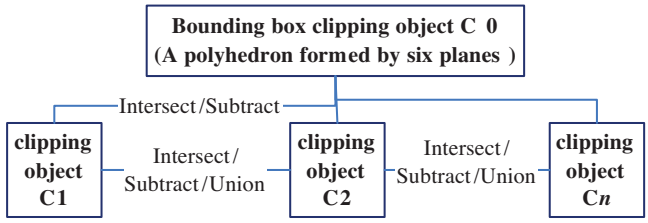


Fig. 2. Hierarchical structure of clipping objects

Figure 2 shows the hierarchical structure of clip objects in our method. The clip object of the bounding box **C0** lies in the top of the hierarchical structure. It is responsible for preventing 3D texture coordinates out of the bounding box. In our method, we do not consider arbitrarily shaped objects, such as voxelized objects, to reduce memory consumption. These common analytical convex objects have compact representation and very little memory consumptions; so they can be easily implemented on GPU. Since Boolean operations of convex objects can approximate arbitrary objects, convex objects are used for concave volume clipping to simplify and speed up our method. All the additional clip objects from **C1** to **Cn** can be polyhedrons, spheres and other convex objects. Additional clip objects from **C1**

to **Cn** can only perform intersection and subtraction operations with the top clip object **C0**. And any of additional clip objects can perform intersection, union and subtraction with another one of additional clip objects. Equation 2 describes this hierarchical structure of Boolean operations in mathematical form.

$$\mathbf{C0}\left\langle \begin{matrix} \times \\ - \end{matrix} \right\rangle \left(\mathbf{C1}\left\langle \begin{matrix} \times \\ - \\ + \end{matrix} \right\rangle \mathbf{C2}\left\langle \begin{matrix} \times \\ - \\ + \end{matrix} \right\rangle \ldots \mathbf{Cn}\left\langle \begin{matrix} \times \\ - \\ + \end{matrix} \right\rangle \right) \tag{2}$$

For the sake of simplicity, analysis of priorities of Boolean operations is not taken into account. Latter Boolean operations are performed just on results of former operations. For example, Equation (3) describes that **C2** is subtracted from **C1** to produce a result **R1**, and an intersection Boolean operation of the result R1 and **C3** is then performed, resulting in a result **R2**, at last intersection operation of the result **R2** and **C0** is performed. In fact, Equation (3) is just the same as Equation (4). Our method does not analyze priorities of Boolean operations, thus it can efficiently improve the performance of Boolean operation on GPU.

$$(\mathbf{C1} - \mathbf{C2} \times \mathbf{C3}) \times \mathbf{C0} \tag{3}$$

$$((\mathbf{C1} - \mathbf{C2}) \times \mathbf{C3}) \times \mathbf{C0} \tag{4}$$

Each clip object used in our method is convex, so each ray has no more than two points of intersection with any convex clip object and a rendering segment can fully encode information about rendering. By making use of rendering segments, we can convert the three basic Boolean operations in 3D object space into 1D ray space, and implement an interactive concave volume clipping on GPU. After several Boolean operations are performed, each ray may be divided into several separate rendering segments for volume integral. Each rendering segment has start and end points determined by parameters t_s and t_e, respectively. As shown in Figure 3, subtraction operations are performed between a polyhedron and two spheres, ray R1 has three rendering segments, including (t_1, t_2), (t_3, t_4) and (t_5, t_6), ray **R2** has only one rendering segment (t_7, t_8), while ray R3 has no rendering segment.

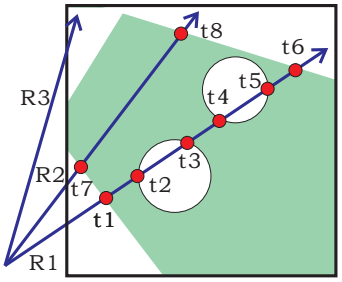


Fig. 3. Rendering segments

For each ray, a table **seg** of rendering segments is created to save rendering segments. Before we analyze a new 1D Boolean operation of current object, let us suppose that the current ray already has n rendering segments in the table **seg**. The i^{th} entry in the rendering segment table is represented by a 2D vector $\mathbf{seg}[i]$, and $\mathbf{seg}[i].x$ and $\mathbf{seg}[i].y$ denote the parameters of start and end points, respectively.

3.1 Intersection Operation

If the current ray does not intersect the current clip object, we immediately set the number n of rendering segments in the table **seg** to zero. Thus all the rendering segments in the table **seg** are quickly deleted to early terminate subsequent processing. Otherwise, we must compute two points of intersection between the ray and the clip object. The two points can be determined by scalar parameters t_s and t_e, respectively. The value pair (t_s, t_e) is called an incoming rendering segment. According to Figure 4, we consider all possible cases of the incoming rendering segment (t_s, t_e) and i^{th} rendering segment $(\mathbf{seg}[i].x, \mathbf{seg}[i].y)$. The intersection operation can be represented by Equation (5).

In case 1, the incoming rendering segment is empty due to no intersection between the current ray and current clip object, i.e. $(t_s, t_e) = \varphi$. All the rendering segments for the current ray are deleted by simply setting the number n of rendering segments to zero.

In case 2, if the incoming segment meets $t_e \leq \mathbf{seg}[i].x$ or $t_s \geq \mathbf{seg}[i].y$, i.e. $(t_s, t_e) \cap \mathbf{seg}[i] = \varphi$, we directly delete the i^{th} rendering segment, but still need to process the remainder rendering segments in the table seg. In case 3, the two rendering segments must have common portion, i.e. $(t_s, t_e) \cap \mathbf{seg}[i] \neq \varphi$. The i^{th} rendering segment is modified as follows.
If $t_e < \mathbf{seg}[i].y$ then $\mathbf{seg}[i].y = t_e$
If $t_s > \mathbf{seg}[i].x$ then $\mathbf{seg}[i].x = t_s$

$$\begin{cases} \text{Delete } \mathbf{seg}[i], \forall i & \text{if } (t_s, t_e) = \\ \text{Delete } \mathbf{seg}[i] & \text{if } (t_s, t_e) \cap \mathbf{seg}[i] = \\ \mathbf{seg}[i] \leftarrow (t_s, t_e) \cap \mathbf{seg}[i] & \text{if } (t_s, t_e) \cap \mathbf{seg}[i] \neq \end{cases} \tag{5}$$

3.2 Subtraction Operation

Subtraction operations are more complicated than intersection operations. Figure 5 illustrates all possible cases for the incoming rendering segment (t_s, t_e) and the i^{th} rendering segment $(\mathbf{seg}[i].x, \mathbf{seg}[i].y)$. It can also be represented by Equation (6).

In case 1, the current ray does not intersect the current clip object, i.e. $(t_s, t_e) = \varphi$, so the incoming rendering segment is empty. In this case, we ignore the clip object with subtraction operations to immediately terminate searching the table **seg**. All the rendering segments for the current ray are kept unchanged.

In case 2, both the start point t_s and the end point t_e of the incoming segment are located in the same outside of the i^{th} rendering segment $\mathbf{seg}[i]$, i.e. $(t_s, t_e) \cap \mathbf{seg}[i] =$

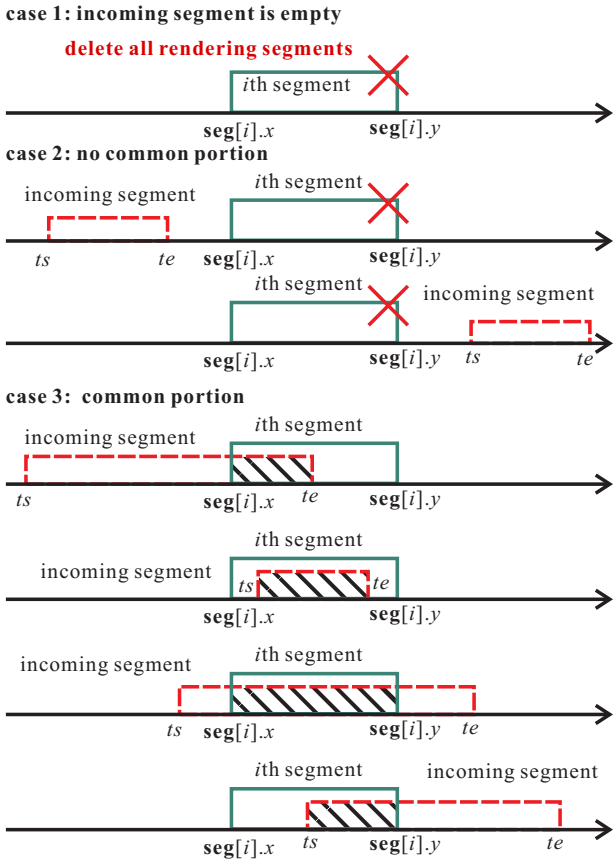


Fig. 4. Possible cases for intersection operation

φ. In this case, we directly ignore the i^{th} entry of the table **seg** to keep the i^{th} rendering segment unchanged, but need to continue to search the remainder entries of the table.

In case 3, both the start point t_s and the end point t_e of incoming segment are located inside the i^{th} segment $(\mathbf{seg}[i].x, \mathbf{seg}[i].y)$, i.e. $(t_s, t_e) \subseteq \mathbf{seg}[i]$. In this case, the i^{th} segment of $(\mathbf{seg}[i].x, \mathbf{seg}[i].y)$ is replaced with $(\mathbf{seg}[i].x, t_s)$, and at the same time a new rendering segment $(t_e, \mathbf{seg}[i].y)$ is created and inserted into the rendering segment table.

In case 4, one of points t_s and t_e is inside the i^{th} segment $\mathbf{seg}[i]$, whilst another point is outside, i.e. $t_s \notin \mathbf{seg}[i] \cap t_e \in \mathbf{seg}[i] \cup t_s \in \mathbf{seg}[i] \cap t_e \notin \mathbf{seg}[i]$. In this case, the current segment is modified by the following codes.
If $t_e < \mathbf{seg}[i].y$ then $\mathbf{seg}[i].x = t_e$
If $t_s > \mathbf{seg}[i].x$ then $\mathbf{seg}[i].y = t_s$

In case 5, both the start point t_s and the end point t_e of the incoming segment are located outside the i^{th} segment $(\mathbf{seg}[i].x, \mathbf{seg}[i].y)$, but they are in different sides, i.e. $(t_s, t_e) \supseteq \mathbf{seg}[i]$. One is in the side where its value is less than two values of the i^{th} segment. Another is located in the side where its value is greater than the two values. The i^{th} segment should be deleted from the table.

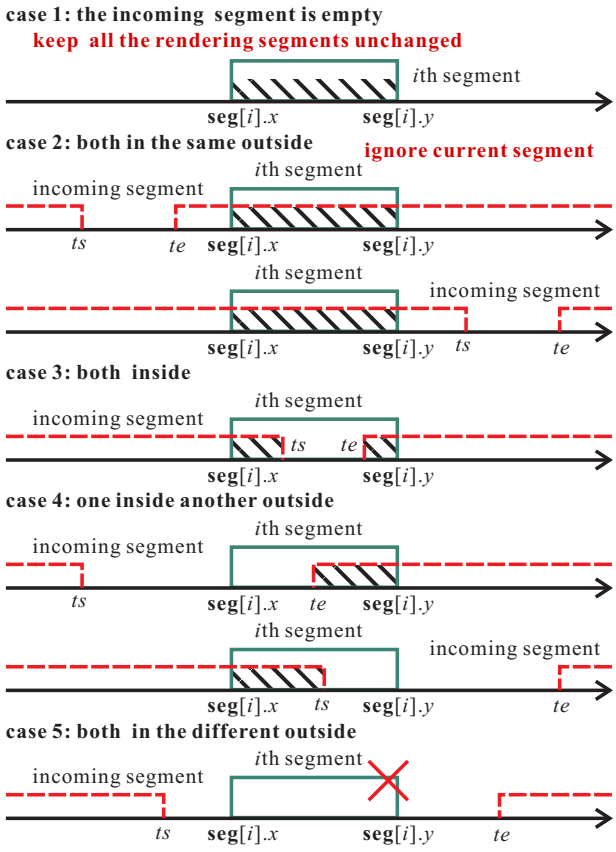


Fig. 5. Possible cases for subtraction operation

$$\begin{cases} \text{Ignore } \mathbf{seg}[i], \forall i & \text{if } (t_s, t_e) = \\ \text{Ignore } \mathbf{seg}[i] & \text{if } (t_s, t_e) \cap \mathbf{seg}[i] = \\ \mathbf{seg}[i] \leftarrow (\mathbf{seg}[i].x, t_s), & \text{if } t_s \notin \mathbf{seg}_i \cap t_e \in \mathbf{seg}[i] \\ \quad (t_e, \mathbf{seg}[i].y) \text{ is added} & \quad \cup\, t_s \in \mathbf{seg}[i] \cap t_e \notin \mathbf{seg}[i] \\ \text{Delete } \mathbf{seg}[i] & \text{if } (t_s, t_e) \supseteq \mathbf{seg}[i] \end{cases} \tag{6}$$

3.3 Union Operation

The bounding box must be used to prevent re-sampling coordinates out of the normalized range and the corresponding mandatory rendering segment can be represented as a 2D vector **seg0**. The **union** operation combines the incoming segment (t_s, t_e) with n rendering segments in the table **seg**. As described in Equation (2), an intersection operation of the union operation result and the mandatory rendering segment **seg0** must be performed at last, in order to prevent final results out of the range $[0, 1]$.

First, parameterized values of start and end points of all the rendering segments and the incoming segment are sorted ascendingly. As shown in Figure 6, each segment has a start point marked by a triangle and an end point marked by a circle. After ascending sorting, a sorted table **ST** is created. In Figure 6, sorted values are **seg**$[0].x$, t_s, **seg**$[0].y$, **seg**$[1].x$, **seg**$[1].y$, **seg**$[2].x$, t_e, **seg**$[2].y$, **seg**$[3].x$ and **seg**$[3].y$. In order to compute resulting segments of union operation, flag table **FT** is created. At the beginning, a flag value is set to zero. If a start point is encountered when searching the sorted table **ST**, 1 is added to the flag value and the flag value is stored in the corresponding entry of the flag table **FT**. -1 is added to the flag value for an end point. In this way, a flag table **FT** is thus generated. In the case of Figure 6, values in the flag table **FT** are 1, 2, 1, 2, 1, 2, 1, 0, 1, 0.

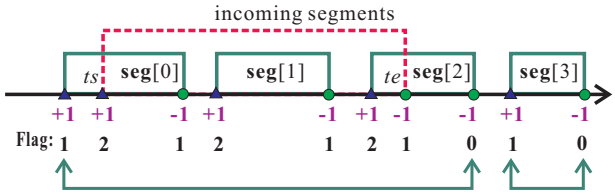


Fig. 6. Union operation

Before searching the flag table **FT**, a flag *newseg* is set to 0. If the value in the flag table **FT** is equal to 1 and *newseg* equals 0, a new segment is created, its start point is set to the value in the corresponding entry of the sorted table **ST** and the flag *newseg* is set to 1. Then we continue to search next value in the flag table **FT**. If the value is equal to 0, the end point for the new segment is set to the value in the corresponding entry of the sorted table **ST** and *newseg* is set to 0. The 1D **union** operation is thus implemented in this way. In Figure 6, resulting segments are $(\mathbf{seg}[0].x, \mathbf{seg}[2].y)$ and $(\mathbf{seg}[3].x, \mathbf{seg}[3].y)$.

Figure 7 a) was rendered without additional clip objects, except a bounding box to limit texture coordinates to the normalized range. Figure 7b shows a resulting image produced by the **intersection** Boolean operation between a polyhedron and a sphere. Figure 7c gives the result with the **subtraction** operation, after two small spheres are subtracted from a big sphere. Figure 7d illustrates experimental results with the **union** operation between a polyhedron and a sphere.

a) b) c) d)

Fig. 7. Results by Boolean operation

4 PSEUDO COLOR BASED RENDERING MODEL

Rendering models should allow users to easily recognize the orientation and shape of clip objects. Therefore, an appropriate rendering model is very important for enhancement of visual effects of volume clipping.

In our method, complex 3D Boolean operation is converted into 1D Boolean operation. So each ray may have several separate rendering segments due to Boolean operation. Each segment has two terminal points which are start and end points.

As for non-terminal points, regular transfer function and the Phong illumination model are used, and the gradient of the 3D volume is estimated to approximate the normal of re-samplings. The ray casting algorithm can produce translucent images according to certain transfer functions.

Weiskopf et al. [19] used a surrounding layer of finite thickness to enable shading on the boundary of the clip object in the texture based volume rendering. Their approach has finite re-sampling intervals at the terminal points of each rendering segment and can produce good results. Due to characteristics of the ray casting algorithm, our approach also has finite equidistance intervals through all the rendering segments, of course, including terminal points. For the sake of simplicity, we can use a binary transfer function for iso-surface based volume rendering to focus on concave volume clipping with Boolean operation. Shading at terminal points is very important for easily recognizing shapes of clip objects. So we pay more attention to rendering method at terminal points in the following subsections.

4.1 Re-Samplings at Terminal Points

Color of re-samplings at the terminal points is determined by the re-sampled value. If the value v is less than a threshold v0, we think that the re-sampling is empty for the iso-surface based ray casting and so the ray goes across the terminal points with regular rendering method. If the re-sampled value v is greater than the threshold v0, the ray casting processing is terminated immediately. The intensity of color for this re-sampling is computed by the traditional window-leveling function. That is, a 2D texturing method on iso-surfaces is performed. The window-leveling function is

shown in Equation (7), where v is the re-sampled value, g_{min} and g_{max} are the minimum and maximum intensities of color to be mapped, W is the window width and L is the location of window center.

$$g = \begin{cases} g_{min} & \text{if } v < (L - W/2) \\ g_{min} + \frac{v-(L-W/2)}{W} \times (g_{max} - g_{min}) & \text{if } (L - W/2) \leq v \leq (L + W/2) \\ g_{max} & \text{if } v > (L + W/2) \end{cases} \quad (7)$$

To enhance 3D effects, the Phong illumination model is also applied. The normal for lighting is the real surface normal of analytical clip objects at corresponding positions, instead of the volume gradient. In order to implement illumination effects, the real normal of clip objects must be computed and stored for terminal points in the Boolean operation stage.

4.2 Pseudo Color Mapping

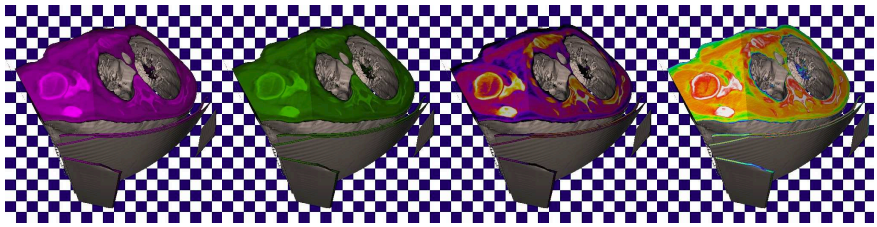
The value computed by the window leveling function is scalar, and it can be directly regarded as a grayscale color for illumination. To further enhance visual effects of anatomical structures, pseudo color mapping is used. As we know, artificially coloring an image can reveal textures and qualities within the image that may not have been apparent in the original gray scale images. To clearly reveal and distinguish inner structures, the pseudo coloring method is used to enhance visual effects. The pseudo color method colorizes the scalar value which maps to a full RGB color range. Pseudo color can help improve image qualities that would not be readily visible within the image's true color.

Figure 8 shows rendered images using the pseudo color mapping method. Figure 8 a) gives results by mapping gray scale intensity values, which are computed by the window-leveling function with $W = 782$ and $L = 976$, to different intensity of a purple color scheme as the diffuse color. Figure 8 b) is the case of a pure green scheme with $W = 782$ and $L = 976$. Figure 8c uses a hot metal color scheme for the pseudo color mapping with $W = 556$ and $L = 1\,131$. In Figure 8 d), a rainbow color scheme is used for the pseudo color mapping with $W = 782$ and $L = 976$.

Our method provides several pre-defined color schemes. The method allows users to select an appropriate color scheme from the pre-defined schemes for rendering, in order to obtain satisfying effects. It even allows users to generate a new color scheme by specifying colors along a curve in the HSV chroma circle.

5 IMPLEMENTATION ON GPU

In order to speed up volume clipping, our method was implemented on GPU using the OpenGL shading language. Most operations, including computation of points of intersection, Boolean operations and pseudo color mapping, are performed on GPU. Pseudo codes of the fragment shaders are listed as shown in Algorithm 1.



a) b) c) d)

Fig. 8. Results with pseudo color mapping

Concave volume clipping on GPU

```
uniform sampler1D MyColorTable;
uniform sampler3D MyVolume;
uniform vec4 EyePos;
varying vec4 LightDir;
uniform ivec3 ObjectFlag[32];
uniform vec4 ObjectPara[256];
uniform float v0;
void main()
{
    vec2 seg0, seg[34];
    Compute points of intersection between the current ray
      and the bounding box to obtain the mandatory rendering
      segment seg0;
    If(seg0.x>seg0.y)
    {/*There is no intersection between current ray
      and the bounding box*/
        Discard;
    }
    int n=0, i=0,j=0
    int m, isFirstHit;
    float ts,te;
    vec4 pos;
    seg[j].x= seg0.x;
    seg[j].x= seg0.x;
    j++;
    //*************Boolean operation*************
    while(n<=32)
    {//Maximum number of clip objects is 32.
        if(ObjectFlag[n].x==0)
        {/*If the object type identifier ObjectFlag[n].x is equal
          to 0, the nth clip objects in the array are empty;
```

```
        In other words, the tail of the object list is reached.*/
          break;
      }
      According to the object type identifier ObjectFlag[n].x
        of the nth clip object, the number ObjectFlag[n].y
        of the nth clip object parameters, which are stored
        in the ith to (i+ObjectFlag[n].y-1)th entries
        of the uniform array ObjectPara, are used to calculate
        the incoming rendering segment (ts, te);
      m=0;
      while(m<j)
      {//Searching the rendering segment table seg
          if(ObjectFlag[n].z==1)
          {//Intersection Boolean operation
              According to approaches described in subsection 3.1,
                intersection operation between the mth rendering
                segment seg[m] and the incoming rendering segment
                (ts, te) are performed to update the parameters
                and entry number j of the rendering segment
                table seg;
          }else if(ObjectFlag[n].z==2)
          {//Subtraction Boolean operation
              According to approaches described in subsection 3.2,
                subtraction operation between current mth rendering
                segment seg[m] and the incoming rendering segment
                (ts, te) are performed, resulting in updating
                the parameters and entry number j of the rendering
                segment table seg;
          }else if(ObjectFlag[n].z==3)
          {//Union Boolean operation
              According to approaches described in subsection 3.3,
                Subtraction operation between current mth rendering
                segment seg[m] and the incoming rendering segment
                (ts, te) are performed, resulting in updating
                the parameters and entry number j of the rendering
                segment table seg;
          }
      }
      n++;
      i=i+ObjectFlag[n].y;
  }
  //********Rendering all the rendering segments********
  if(j==0)
  {/*The number of entries in the rendering segment table is zero.
```

```
      In other words, the table is empty*/
        Discard;
    }else
    {
        m=0;
        while(m<j)
        {//Rendering the mth segment
            ts=seg[m].x;
            te=seg[m].y;
            isFirstHit=0;
            while(ts<te)
            {
                pos.xyz=EyePos.xyz+ts* LightDir;
                d=texture3D(MyVolume, pos.xyz);
                if(d.a<v0)
                {//Voxel intensity is stored in the alpha channel
                    ts=ts+0.005;
                }else
                {
                    isFirstHit++;
                    if(isFirstHit==0)
                    {
                        Render the re-sampling using a selected
                          pseudo color scheme stored in the uniform
                          MyColorTable;
                        Compute light effect using the Phong model;
                        Composite shaded color and opacity;
                    }else
                    {
                        Render the re-sampling using the regular
                          ray casting method;
                        Compute light effect using the Phong model;
                        Composite shaded color and opacity;
                    }
                }
            }
        }
    }
}
```

In our GPU implementation, a special data structure is used as shown in Figure 9. Fixed length arrays are used since GPU shaders do not support dynamic memory allocation. We used two fixed length arrays, which are ObjectFlag and

ObjectPara, to pass parameters of common analytical convex clip object from CPU to GPU using the OpenGL shading language qualifier Uniform.

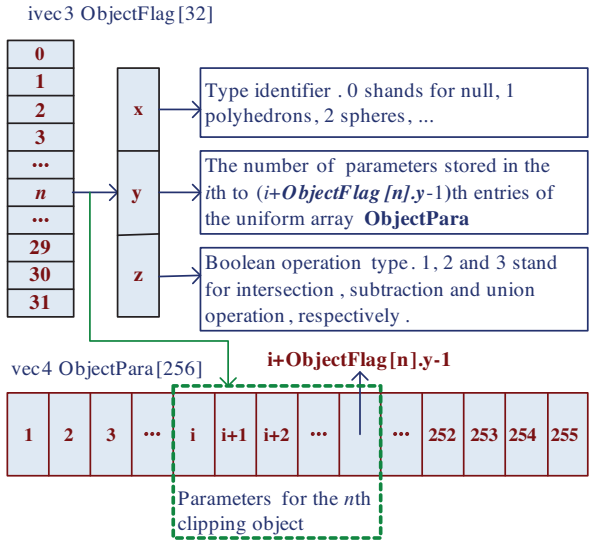


Fig. 9. Data structures for GPU implementation

The array **ObjectFlag** stores three flag values which include object type, number of parameters and types of Boolean operations with other objects for all the clip objects. These three flag values can be stored in a 3D vector. The data type of the array is defined as a 3D integer vector. Variables **ObjectFlag**$[n].x$, **ObjectFlag**$[n].y$ and **ObjectFlag**$[n].z$ stand for the object type identifier, the number of parameters and types of Boolean operations of the n^{th} clip object, respectively. If **ObjectFlag**$[n].x$ is set to 0, it means that the tail of the object flag table is reached and the $\geq n^{\text{th}}$ entries are empty. Different values of **ObjectFlag**$[n].x$, which are greater than 0, stand for different types of analytical objects. Fox example, if **ObjectFlag**$[n].x$ is equal to 1 and 2, the n^{th} clip object is a polyhedron and a sphere, respectively. In our method, the maximum number of clip objects is 32. It is enough for complicated concave volume clipping.

Different types of clip objects have different parameters. We use a uniform array **ObjectPara** to store parameters for all the clip objects, and the data type of array **ObjectPara** is defined as 4D float vectors. For example, a box can be represented as 6 planes. As we know, each plane equation requires only 4 parameters; so the box can be stored in successive 6 entries of the array **ObjectPara**. But for a sphere, the center position and radius of the sphere require only one entry of the array. **ObjectPara**$[n].xyz$ stores the position of the sphere center and **ObjectPara**$[n].w$ is the radius.

As shown in Figure 9, the two arrays must cooperate with each other. The array **ObjectFlag** stores the three object flag values and **ObjectPara** store corresponding parameters of each clip object specified by the array **ObjectFlag**. According to the object type identifier **ObjectFlag**$[n].x$ of the n^{th} clip object, **ObjectFlag**$[n].y$ determines the number of parameters for the n^{th} clip object, which are stored in the i^{th} to $(i + \textbf{ObjectFlag}[n].y - 1)^{\text{th}}$ entries of the array **ObjectPara**. For example, parameters of the first object must be stored in the 0^{th} to $(\textbf{ObjectFlag}[0].y - 1)^{\text{th}}$ entries of the array **ObjectPara**. Parameters for the 2^{nd} object should be stored in the $(\textbf{ObjectFlag}[0].y)^{\text{th}}$ to $(\textbf{ObjectFlag}[0].y + \textbf{ObjectFlag}[1].y - 1)^{\text{th}}$ entries of the array **ObjectPara**.

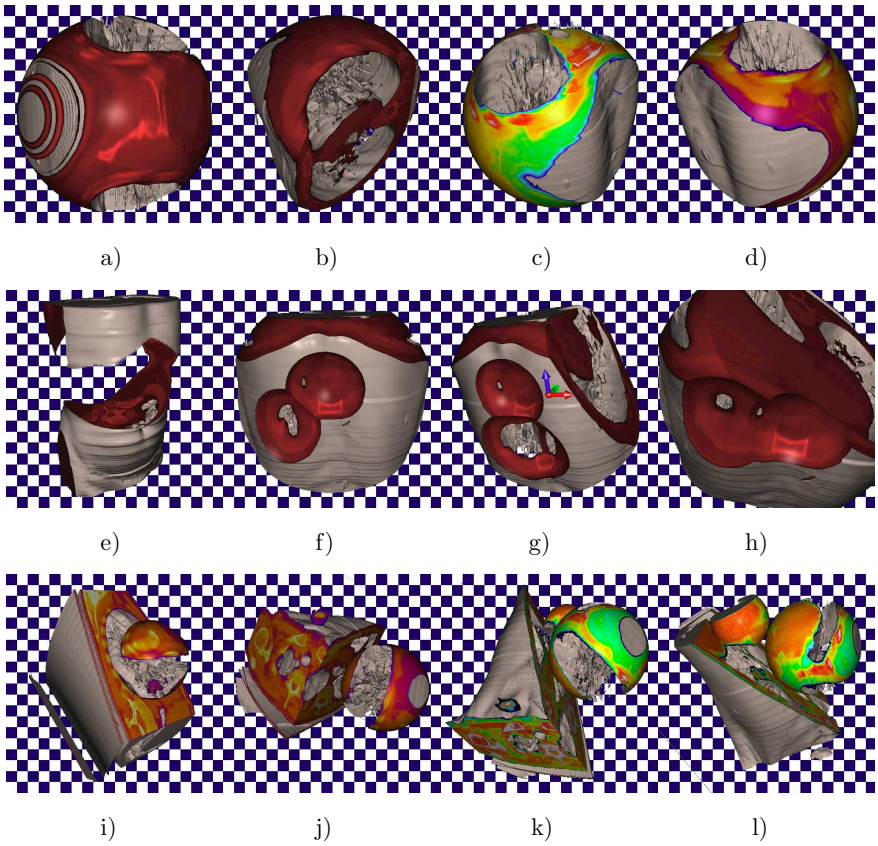


Fig. 10. Boolean operations. a), b), c) and d) resulting images by intersection operation. e), f), g) and h) resulting images by subtraction operation. i), j), k) and l) resulting images by union operation

6 EXPERIMENTS

We implemented the concave volume clipping method based on the GPU ray casting using Visual C++ and OpenGL Shading Language. The experimental platform is a desktop personal computer with a GeForce FX8600 graphics card. One of datasets is a CT scalar dataset with the size of $512 \times 512 \times 112$ which was scanned from a human in a hospital. Each voxel of the human dataset has 16 bits. Another dataset is a publicly available engine dataset downloaded from a web site [26]. The engine dataset has $256 \times 256 \times 110$ voxels. Each voxel has only 8 bits.

First, we performed the intersection operations using different clip objects. In Figure 10 a), a resulting image was produced by the intersection Boolean operation between a polyhedron and a sphere. The polyhedron is specified by 6 planes. The sphere is located inside the polyhedron. By changing the position of sphere, different results were generated as shown in Figures 10 b), c) and d).

Second, we test the subtraction operation with different clip objects. Figure 10 e) shows the rendered image when a big sphere was subtracted from a polyhedron. Figure 10 f) gives the resulting image when the intersection operation was performed between a polyhedron and a big sphere, and then two small spheres were subtracted from the result of intersection operation just performed. We changed the shape of the polyhedron by rotating and moving clip planes, also moved the position of two small spheres, and another resulting image was produced as shown in Figure 10 g). Figure 10 h) also illustrates the result of intersection and **subtraction** combined operations with different parameters.

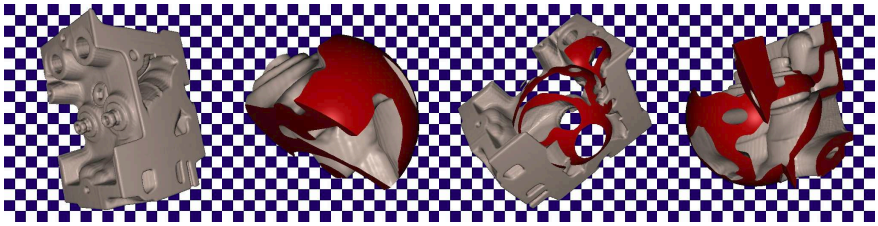
Third, we test the union operation using a polyhedron and a sphere, as shown in Figures 10 i) and j). Two resulting images are different due to different shapes and relative positions of the polyhedron and the sphere. Figures 10 k) and l) show experimental results by union operations of a polyhedron and two spheres. The difference between the two rendered images is only due to different viewing parameters.

We also test our pseudo color mapping. Both Figures 10 i) and j) show results of pseudo color mapping by using the hot metal color scheme. Both Figure 10 k) and l) show resulting images using the pseudo color mapping with the rainbow color scheme. Users can freely select a color scheme from pre-defined color schemes or a user-defined color scheme.

We test our Boolean operation clipping approach on an engine dataset publicly available for scientific research purpose. Figure 11 a) shows the resulting image of the original engine dataset without volume clipping. In Figure 11 b), a resulting image was produced by intersection Boolean operation between a polyhedron and a sphere. Figure 11 c) gives the resulting image when a small sphere and a big sphere were subtracted from a polyhedron. Figure 11 d) shows resulting image of union operations using a polyhedron and a big sphere.

Table 1 lists timings with the same viewport of 512×512. As we can see, the rendering speed decreases when the number of analytical clip objects increases. When one polyhedron defined by 6 planes is used, our method can render the human CT and the engine datasets at about 20 frames per second. When the number of

clipping objects increases, the method can obtain about 18 to 19 frames per second. If the original GPU ray casting is applied, the two datasets can be visualized at about 59.9 fps due to removal of Boolean operations.



a) b) c) d)

Fig. 11. Concave volume clipping of the engine dataset. a) Original dataset, b) Intersection, c) Subtraction, d) Union

Datasets	Clipping objects	Frame rates
The human CT dataset with size of $512 \times 512 \times 112$	1 polyhedron	20.9 fps
	2 polyhedrons	19.6 fps
	2 polyhedrons, 1 sphere	19.2 fps
	2 polyhedrons, 2 spheres	18.7 fps
	No clipping	59.9 fps
The engine dataset with size of $256 \times 256 \times 110$	1 polyhedron	21.2 fps
	2 polyhedrons	20.1 fps
	2 polyhedrons, 1 sphere	19.7 fps
	2 polyhedrons, 2 spheres	19.2 fps
	No clipping	59.9 fps

Table 1. Timing results with our GPU based volume clipping method

7 CONCLUSIONS

In this paper, we present a concave volume clipping approach based on GPU ray casting. Some common analytical convex objects, such as polyhedrons and spheres, are used to implement concave volume clipping using Boolean operations. Since the volume integral path for convex object can be represented by an 1D rendering segment, the complex 3D Boolean operation is easily converted into 1D Boolean operation. To enhance visual effects, a pseudo color based rendering model is proposed and the Phong illumination model is enabled on the clipped surfaces. Experimental results show that the three basic Boolean operations are correctly performed and our approach can freely clip and clearly visualize volumetric datasets at interactive frame rates. Our method can provide users an alternative method to avoid the

difficulty of designing transfer functions and speed up exploration of 3D datasets on GPU. In the future, we focus on modeling of the complex clip objects.

Acknowledgement

This project was supported by Natural Science Foundation of China (61063034) and Science Technology Project of Education Department of Jiangxi province (GJJ11-416). Special thanks are given to anonymous reviewers for their helpful suggestions.

REFERENCES

[1] Cullip, T.—Neumann, U.: Accelerating Volume Reconstruction with 3D Texture Mapping Hardware. Technical Report TR93-027, Department of Computer Science, University of North Carolina, Chapel Hill 1993.

[2] Cabral, B.—Cam, N.—Foran, J.: Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In Proceedings of IEEE Symposium on Volume Visualization, 1994, pp. 91–98.

[3] Kruger, J.—Westermann, R.: Acceleration Techniques for GPU-Based Volume Rendering. In Proceedings of IEEE Visualization 2003.

[4] Rössler, F.—Botchen, B. P.—Ertl, T.: Dynamic Shader Generation for GPU-Based Multi-Volume Ray Casting. IEEE Computer Graphics and Applications, Vol. 28, 2008, No. 5, pp. 66–77.

[5] Bentoumi, H.–Gautron, P.—Bouatouch, K.: GPU-Based Volume Rendering for Medical Imagery. International journal of computer systems science and engineering, Vol. 1, 2007, No. 1, pp. 36–42.

[6] Marroquim, R.—Maximo, A.—Farias, R.—Esperanca, C.: Volume and Isosurface Rendering with GPU-Accelerated Cell Projection. Computer Graphics Forum, Vol. 27, 2008, No. 1, pp. 24–35.

[7] Reis, G.—Zeilfelder, F.—Hering-Bertram, M.—Farin, G.—Hagen, H.: High-Quality Rendering of Quartic Spline Surfaces on the GPU. IEEE Transactions on Visualization and Computer Graphics, Vol. 14, 2008, No. 5, pp. 1126–1139.

[8] Loop, C.—Blinn, J.: Real-Time GPU Rendering of Piecewise Algebraic Surfaces. ACM Transactions on Graphics, July 2006, Volume 25, Issue 3.

[9] Reimers, M.—Seland, J.: Ray Casting Algebraic Surfaces using the Frustum Form. Computer Graphics Forum, Vol. 27, 2008, No. 2, pp. 361–370.

[10] Guthe, M.—Balázs, Á.—Klein, R.: GPU-Based Trimming and Tessellation of NURBS and T-Spline Surfaces. ACM Transactions on Graphics Volume 24, 2005, No. 3.

[11] Dyken, C.—Ziegler, G.—Theobalt, C.—Seidel, H.-P.: High-Speed Marching Cubes using HistoPyramids. Computer Graphics Forum, Vol. 27, 2008, No. 8, pp. 2028–2039.

[12] Kim, Y.—Lee, C. H.—Varshney, A.: Vertex-Transformation Streams. Graphical Models, Vol. 68, 2006, pp. 371–383.

[13] Fialka, O.—Cadk, M.: FFT and Convolution Performance in Image Filtering on GPU. Proceedings of the Information Visualization, 2006.

[14] Kruger, J.—Westermannm, R.: Linear Algebra Operators for GPU Implementation of Numerical Algorithms. SIGGRAPH 2003.

[15] Jin, X.—Chen, S.—Mao, K.: Computer-Generated Marbling Textures: A GPU-Based Design System. IEEE Computer Graphics and Applications, Vol. 27, No. 2, pp. 78–84.

[16] Van Gelder, A.—Kim, K.: Direct Volume Rendering with Shading via Three-Dimensional Textures. Proceeding of Symp. Volume Visualization, 1996, pp. 23–30.

[17] Westermann, R.—Ertl, T.: Efficiently Using Graphics Hardware in Volume Rendering Applications. SIGGRAPH 1998, 1998, pp. 169–179.

[18] Xie, K.—Sun, G.–Yang, J.—Zhu, Y. M.: Interactive Volume Cutting of Medical Data. Computers in Biology and Medicine, Vol. 37, 2007, pp. 1155–1159.

[19] Weiskopf, D.—Engel, K.—Ertl, T.: Interactive Clipping Techniques for Texture-Based Volume Visualization and Volume Shading. IEEE Transactions on Visualization and Computer Graphics, Vol. 9, 2003, No. 3, pp. 298–312.

[20] Diepstraten, J.—Weiskopf, D.—Ertl, T.: Transparency in Interactive Technical Illustrations. Eurographics 2002, 2002, pp. 317–325.

[21] Mammen, A.: Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. IEEE Computer Graphics and Applications, Vol. 9, 1989, No. 4, pp. 43–55.

[22] Diefenbach, P. J.: Pipeline Rendering: Interaction and Realism through Hardware-Based Multi-Pass Rendering. Ph. D. thesis, University of Pennsylvania 1996.

[23] Tiede, U.—Schiemann, T.—Hohne, K. H.: High Quality Rendering of Attributed Volume Data. Proceedings of IEEE Visualization 1998, 1998, pp. 255–262.

[24] Williams, D.—Grimm, S.—Coto, E.—Roudsari, A.—Hatzakis, H.: Volumetric Curved Planar Reformation for Virtual Endoscopy. IEEE Transactions on Visualization and Computer Graphics, Vol. 14, 2008, No. 1, pp. 109–119.

[25] Stegmaier, S.—Strengert, M.—Klein, T. et al.: A Simple and Flexible Volume Rendering Framework for Graphics-Hardware Based Raycasting. Volume Graphics (2005).

[26] `http://www9.informatik.uni-erlangen.de`.

Feiniu Yuan received his B. E. and M. E. degrees in mechanical engineering from Hefei University of Technology in 1998 and 2001, respectively. He received his Ph. D. degree in pattern recognition and intelligence system from University of Science and Technology of China (USTC) in 2004. From 2004 to 2006, he worked as a post-doctor at the State Key Laboratory of Fire Science, USTC. His research interests are in video fire detection, pattern recognition, medical image processing, and 3D visualization. He is currently a Professor with Jiangxi University of Finance and Economics.