Optica Applicata, Vol. XXXIX, No. 2, 2009

Real time multiple planar volume clipping based on the programmable graphics process unit

FEINIU YUAN

School of Information Technology, Jiangxi University of Finance and Economics,
Nanchang 330013, Jiangxi, China; e-mail: yfn@ustc.edu

We propose a real time volume clipping method which is capable of using several analytical planes for virtual clipping, in order to display internal anatomical structures within volumetric data sets. A single proxy plane is used for computation of the direction of a ray that is cast from the viewpoint. Intersections between the rays and the planes are computed on graphics process unit (GPU). The start and end points for each ray are determined by analyzing relationships with the ray direction, intersections and the normal of planes. Then the volume integral is computed along the ray from the start point to the end point. To obtain immediate visual feedback of clipping effects, we implement translation and rotation of planes on GPU to interactively change the shape of clip object. At last, several experiments were performed on a standard PC with a GeForce FX8600 graphics card. Experimental results show that the method can freely clip and clearly visualize volumetric data sets at real time frame rates.

Keywords: planar volume clipping, graphics process unit (GPU) ray casting, single proxy plane.

1. Introduction

Visualization techniques include surface rendering and volume rendering. In surface rendering techniques, intermediate triangles of iso-surfaces must be extracted from 3D volumetric data sets and then the triangles are rendered using traditional computer graphics hardware. It can achieve interactive frame rates. However, the quality of the rendered images is not high, due to the loss of details during the extraction process. While in volume rendering techniques, extraction of polygons is not required and the volumetric data set is directly rendered according to a transfer function specified by users, so it can produce high quality images. But it is memory and time consuming. Ray casting algorithm is one of the image space volume rendering techniques. It can generate high quality images and often be used in many applications. It can achieve high rendering frame rates on high end workstations and specialized volume rendering hardware (such as VolumePro [1], *etc*.). However, it is unable to obtain interactive frame rates on the popular PC platform without graphics hardware.

Many acceleration techniques were proposed to speed up the brute force ray casting algorithm. One kind of these techniques is the acceleration technique based on space

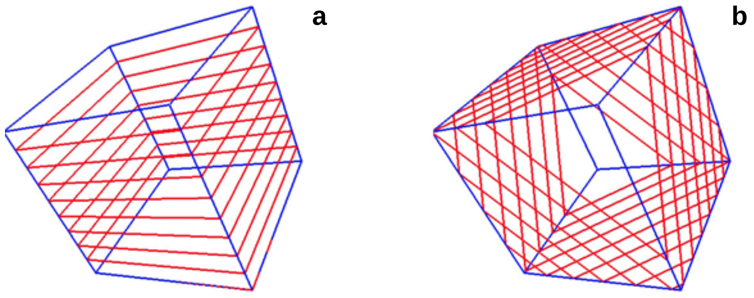

Fig. 1. Axis-aligned (**a**) and view-aligned resampling slices (**b**).

leaping, which can avoid many empty resamplings without loss of image quality, such as cylindrical approximation of tubular organs presented by VILANOVA *et al.* [2], spherical approximation of tubular organs proposed by SHARGHI and RICKETTS [3]. But these methods require a complicated or time-consuming preprocessing. Another kind of these techniques is based on a tradeoff between image quality and rendering speed, such as two-phase perspective ray casting for interactive volume navigation presented by BRADY *et al.* [4], screen and object adaptive sampling, *etc*. The third kind of acceleration techniques is based on graphics hardware.

With the rapid development of computer game industry, consumer level graphics cards have huge computation performance. There are two typical approaches based on graphics cards, including texture based volume rendering and graphics process unit (GPU) based volume rendering. The texture based volume rendering was originally presented by CULLIP and NEUMANN [5] and further developed by CABRAL *et al.* [6]. The algorithm can directly utilize the texture mapping capabilities of graphics hardware by proxy resampling planes, which can be either axis-aligned [7] with three sets of 2D texture stacks or view-aligned [8] with one 3D texture, as shown in Fig. 1. It can achieve interactive frame rates, but it produces relatively low image quality, especially in the cases of close views. As shown in Fig. 2, we can clearly observe circular artifacts when the viewpoint is located within the human trachea. The quality of rendered image depends mainly on the number of proxy surfaces. If the number of

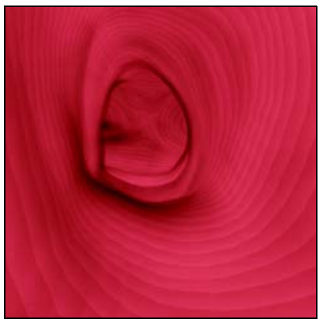

Fig. 2. Circular artifacts with close views.

proxy surfaces increases, the rendering speed becomes reduced. Up to now, most of the modern graphics cards have flexible capabilities of programmability on GPU. These capabilities lead to the rapid development of GPU based volume rendering. And graphics hardware has become preferred technique for the standard implementation of ray casting. KRÜGER and WESTERMANN [9] proposed a GPU based ray casting. BENTOUMI *et al.* [10] proposed a GPU based shear-warp algorithm. GPU based volume rendering algorithm can generate high quality images at interactive frame rates. GPU can also be used in traditional graphics rendering tasks. REIS *et al.* [11] presented high--quality rendering of quartic spline surfaces on the GPU. KIM *et al.* [12] presented vertex transformation streams based on GPU. Flexibility of GPU improves parallel computation performance in many time-critical applications [13, 14].

Transfer function is very important for volume rendering. However, rapid specification of an appropriate transfer function is usually difficult in practice. Therefore, volume clipping becomes an important compensatory tool for difficulty in designing transfer function. Clip planes are frequently used in texture based volume rendering. For instance, VAN GELDER and KIM [15] used clip planes to specify the boundaries of the data set in 3D texture-based volume rendering, thus planar volume clipping was implemented for texture based volume rendering. WESTERMANN and ERTL [8] presented a volume clipping method using a stencil buffer. The clip object has to be rendered for each slice to set the stencil buffer correctly. WEISKOPF *et al.* [16] presented clipping techniques based on a volumetric description of clip objects. Clip objects must be first voxelized and represented by a 3D volumetric texture. DIEPSTRATEN *et al.* [17] proposed another depth-based clipping method for depth sorting semi-transparent surfaces. The method is related to virtual pixel maps [18] and dual depth buffers [19]. TIEDE *et al.* [20] used a similar method to visualize attributed volumes by ray casting. WILLIAMS *et al.* [21] presented a volumetric curved planar reformation for virtual endoscopy.

In this paper, we propose a real time visualization method based on GPU ray casting, which is capable of using multiple planes for convex volume clipping. This paper is organized as follows. Section 2 introduces traditional GPU based ray casting. In Section 3, the single proxy plane based GPU ray casting is presented. Section 4 discusses geometrical transformation of clip planes. In Section 5, some experiments are described. At last, conclusions are given.

2. Traditional GPU ray casting

The algorithm presented by STEGMAIER *et al.* [22] is the classical GPU ray casting, within which the data set is stored as a 3D texture to take advantage of the built-in tri-linear interpolation in graphics hardware. In their algorithm, a bounding box for the data set is created, and coordinates of start and end points for each ray are encoded in the color channel of the rendered surfaces of the box, as shown in Figs. 3**a** and 3**b**,

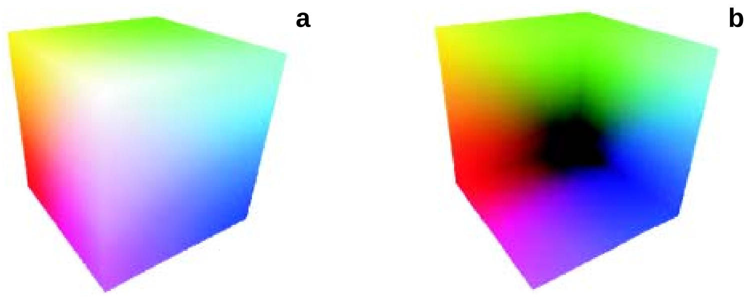

Fig. 3. Rendered colors of front (**a**) and back (**b**) surfaces of the bounding box of the 3D texture which encode the start and end points, respectively.

respectively. The color of the front surfaces is regarded as the start point for the ray casting process. And the color of the back surfaces is regarded as the end point. The ray direction at any given pixel can be computed by subtracting the color of front surfaces at the pixel from the color of back surfaces in the same position. Two-pass rendering is performed [23], that is, one pass for the front surface and another for the back.

3. Single proxy plane based GPU ray casting

In the GPU ray casting algorithm proposed by CHU *et al*. [24], which is different from the classic GPU ray casting, only a single proxy plane is used to compute ray equations and ray-plane intersections are computed by analytical geometry on GPU, instead of the two-pass rendering. In their algorithm, six planes of the bounding box are used. We extend six planes of the bounding box to an arbitrary number of planes, and these planes are not required to be parallel to one of *xy*, *yz* and *xz* coordinate planes. Therefore, our method is more flexible and more useful for volume clipping.

3.1. Single proxy plane

In the classic GPU based ray casting algorithm, six faces of the bounding box are rendered first to get the end point for ray termination while front face culling is enabled, and six faces of the same box are rendered again to generate the start point for ray casting. In other words, it has six proxy planes for encoding information of each ray, as shown in Fig. 4. Computation of start and end points is automatically completed by rendering six proxy faces twice.

Ray-plane intersections can also be calculated by mathematical analytic geometry on GPU. A ray direction must be given before computing intersections of rays and planes. As shown in Fig. 5, a plane specified by OpenGL's GL_QUADS function is first rendered to generate a ray direction. But in the Chu's algorithm [24], the six planes must be parallel to one of *xy*, *yz* or *xz* coordinate surfaces. Because the orientation and the number of planes are fixed, it has no clipping function and that limits the technique to be widely applied. Our algorithm allows users to specify arbitrary orientation

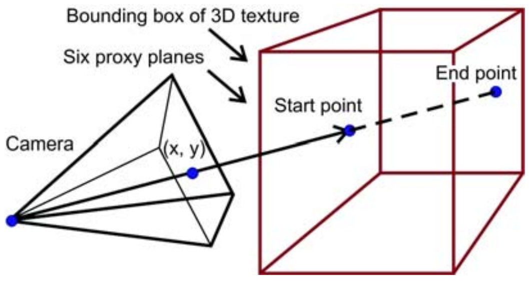

Fig. 4. Six proxy planes.

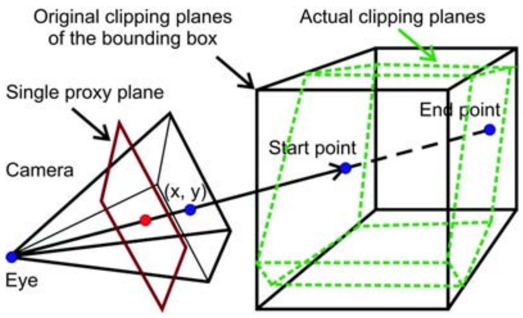

Fig. 5. Single proxy plane.

and number of planes by adding some actual clip planes, as shown in Fig. 5. And all the original planes of the bounding box and actual clip planes can be regarded as ordinary clip planes.

3.2. Computation of ray directions

Suppose that $\mathbf{M_{mv}}$ is the model view matrix consisting of a translation matrix $\mathbf{M_T}$ and a rotation matrix $\mathbf{M_R}$, $\mathbf{P_w}$ and $\mathbf{P_e}$ are the same vertex in the world and eye coordinate systems, respectively. We have

$$\mathbf{P_e} = \mathbf{M_{mv}} \mathbf{P_w} = \mathbf{M_T} \mathbf{M_R} \mathbf{P_w} \tag{1}$$

As shown in Fig. 6, the left-handed camera coordinate system has the same origin, *y* and *z* axes as the right-handed eye coordinate system, but its *x* axis is just contrary to *x* axis in the eye coordinate system. A fragment vertex $\mathbf{P_e}$ in the proxy plane can be

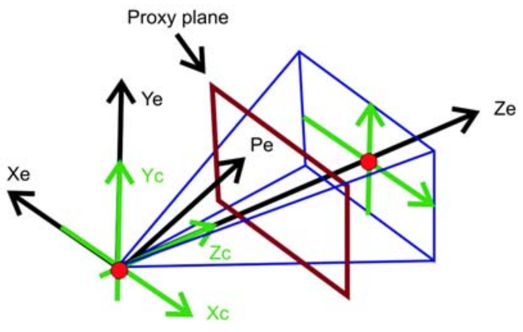

Fig. 6. Camera location in the eye coordinate system.

used to generate the normalized direction of the ray in the eye coordinate system for that pixel. The ray direction $\mathbf{D_e}$ can be computed by

$$\mathbf{D_e} = \text{normalize}(\mathbf{P_e}.xyz) \tag{2}$$

In Equation (2), we adopt the symbol denotation used by OpenGL shading language. $\mathbf{P_e}.xyz$ stands for a new vector consisting of the first three components of the 4D homogeneous coordinates.

3.3. Computation of ray-plane intersections

Since the ray direction is computed in the eye coordinate system and its bounding box is defined in the world coordinates, so we must transform the ray direction back to the world system and compute the intersections in the world coordinate system. So, we have

$$\mathbf{D_w} = \text{normalize}\left[(\mathbf{M_R^{-1}}\mathbf{P_e}).xyz\right] \tag{3}$$

According to the analytic geometry, the parameterized equation of a ray is defined as

$$\mathbf{P_w}.xyz = \mathbf{P_{eye}}.xyz + \mathbf{D_w} \cdot t \tag{4}$$

where $\mathbf{P_w}$ is the 4D homogeneous coordinates $(x, y, z, 1)$, $\mathbf{P_{eye}}$ is the eye position (viewpoint), t is the distance from the eye position. According to the analytic

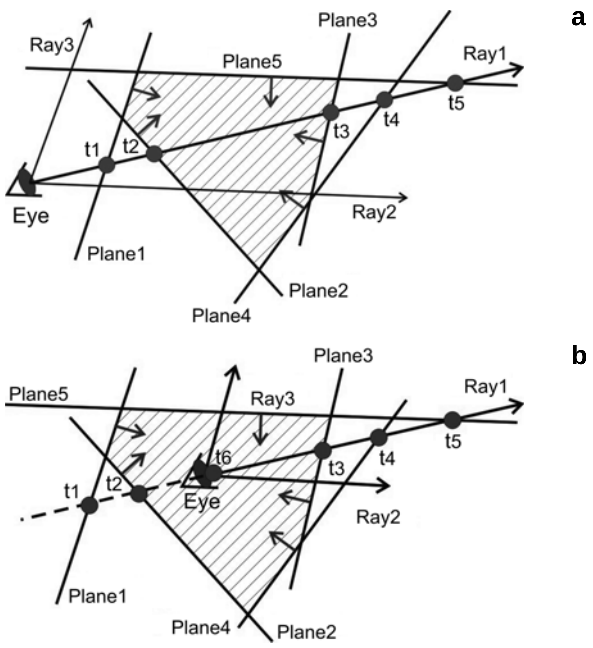

Fig. 7. Determination of start and end points. The viewpoint outside (**a**) and inside (**b**) the clipped volume.

geometry, positive t stands for points having the distance t along the ray direction $\mathbf{D_w}$. And negative t denotes points with the distance t along the contrary ray direction $-\mathbf{D_w}$.

A plane is usually defined as

$$Ax + By + Cz + D = \mathbf{S} \cdot \mathbf{P} = 0 \tag{5}$$

where (A, B, C) is the normalized normal of the plane, $\mathbf{S} = (A, B, C, D)$ and $\mathbf{P_w} = (x, y, z, 1)$ are homogeneous coordinates. Therefore, a plane equation can be defined as the dot product of two vectors $\mathbf{S}$ and $\mathbf{P_w}$. Combining Eqs. (4) and (5), we obtain the intersection between a ray and a plane by

$$t = -\frac{Ax_{\text{eye}} + By_{\text{eye}} + Cz_{\text{eye}} + D}{Ax_d + By_d + Cz_d} = -\frac{\mathbf{S} \cdot \mathbf{P_{eye}}}{\mathbf{S}.xyz \cdot \mathbf{D_w}} \tag{6}$$

where $\mathbf{P_{eye}}$ is the homogeneous coordinates of the eye $(x_{\text{eye}}, y_{\text{eye}}, z_{\text{eye}}, 1)$ and $\mathbf{D_w}$ is the 3D vector of the ray direction (x_d, y_d, z_d).

In Equation (6), when the denominator $d = \mathbf{S}.xyz\mathbf{D_w}$ is equal to zero, t becomes infinite, and it means that there is no intersection between the ray and the plane. In other words, the ray is parallel to the plane.

3.4. Determination of start and end points

In our method, a plane splits the 3D space into two parts which are called saved part and discarded part. We keep the saved part that the normal of the plane points to, whilst the discarded part is deleted. Several planes are often used for convex volume clipping and each ray may have several intersections with clip planes. How to select two points as start and end points from these intersections is a key problem to correctly implement convex clipping.

3.4.1. A ray is parallel to a plane

If a ray is parallel to a plane and the eye is located in the discarded part, current ray should be discarded as there is no intersection. In the case of Fig. 7**a**, Ray3 is parallel to Plane1 and the eye is in the discarded part of Plane1, so we can discard Ray3 immediately. If a ray is parallel to a plane and the eye is in the saved part, we need to compute possible intersections with other planes. In the case of Fig 7**b**, because Ray2 is parallel to Plane5 and the eye is in the saved part of Plane5, Ray2 cannot be discarded and continue to be processed to compute intersections with other clip planes.

The criterion which can determine if the eye $\mathbf{P_{eye}}$ is in the saved part of plane $\mathbf{S}$ is defined as

$$Ax_{\text{eye}} + By_{\text{eye}} + Cz_{\text{eye}} + D = \mathbf{S} \cdot \mathbf{P_{eye}} > 0 \tag{7}$$

If Equation (7) is satisfied, we think that the eye $\mathbf{P_{eye}}$ is located in the saved part of plane $\mathbf{S}$. Otherwise, it is in the discarded part. Equation (7) is just the numerator in Equation (6).

3.4.2. Sorting classified intersections

If a ray is not parallel to a plane, there must be an intersection. Ray-plane intersections are classified into two categories. One category is composed of such intersections with an angle less than 90 degrees which the ray and the plane normal form, and these intersections are denoted by parameters $tn(1), tn(2), \ldots, tn(M)$. Another category consists of intersections with an angle greater than 90 degrees, represented as $tf(1), tf(2), \ldots, tf(N)$. The criterion for judging the angle less than 90 degrees is defined as

$$Ax_d + By_d + Cz_d \; = \; \mathbf{S}.xyz \cdot \mathbf{D_w} > 0 \tag{8}$$

Equation (8) is just the denominator $d = \mathbf{S}.xyz\,\mathbf{D_w}$ in Eq. (6). Then we compute the maximum value of parameter t in the first category of intersections and the minimum value of parameter t in the second one. The process can be formulated as

$$tn \; = \; \max\left\{ tn(1), tn(2), \ldots, tn(M) \right\} \tag{9}$$

$$tf \; = \; \min\left\{ tf(1), tf(2), \ldots, tf(N) \right\} \tag{10}$$

At last, we must decide whether or not the two parameters are valid. If $tf < tn$ or $tf < 0$, the ray has no intersection with the convex clipped geometry. In other words, the two parameters are invalid. When the eye is inside the clipped volume, the parameter tn for the start point is usually less than 0. In this case, the parameter tn is set to 0, in order to ensure that the start point is not behind the eye along the ray direction. After the aforementioned processing, tn and tf are the parameters for the valid start and end points, respectively. Now, we can trace each ray to compute volume integral from the start point to the end point on the GPU. As shown in Fig. 7**a**, as for Ray1, the start and end points are determined by parameters t_2 and t_3, respectively. Figure 7**b** illustrates the eye inside the clipped volume. We obtain the parameters for start and end points which are t_2 and t_3, respectively. But t_2 is less than 0, so the start point parameter is modified to 0. So, the parameters for start and end points are t_6 and t_3, respectively.

Figure 8 lists GPU codes for computing start and end points. In our implementation, vector operations are widely applied to take full advantage of intrinsic parallel hardware computation contained on GPU. We pass all the parameters of clip planes from CPU to GPU using uniform variables defined by OpenGL Shading Language. The uniform qualifier is used to declare global variables whose values are the same across the entire primitive being processed. All uniform variables are read-only and are initialized externally either at link time or through the API. The uniform variable

```
vec3 Dw=normalize(FragVertex.xyz);
int i=0;
tn=-1000.0;
tf=1000.0;
while(i<ClipPlaneNum)
{/*Compute intersections between rays and planes S[i]*/
   n=dot(S[i], Peye);
   d=dot(S[i].xyz, Dw);
   if(abs(d)<=0.00001)
{/*A ray is parallel to a plane*/
   if(n<=0.0)
   { /*The eye is located in the discard part of plane S[i]*/
      NoIter=1;//Discard this ray
      break;//Stop searching
   }
}else
   {//A ray is not parallel to a plane
      t=-n/d;
      if(d>0.0)
      {//an angle less than 90 degrees
       //calculate maximum values for first category
         tn=max(tn,t);
      }else
         {//Calculate minimum values for second category
            tf=min(tf,t);
         }
   }
i++;
}
if(tf<tn||tf<0.0)
NoIter=1;// No intersection
else
   { if(tn<0.0)
      { /*Set the start point to the eye when the eye inside the clipped volume.*/
         tn=0.0;
      }
   }
```

Fig. 8. GPU codes for computing start and end points.

S specifies the parameters of clip planes and the uniform variable *ClipPlaneNum* stores the number of clip planes. The detailed declarations are listed as follows: uniform vec4 S[64]; uniform int ClipPlaneNum.

In our implementation, the number of clip planes varies from 6 to 64. We can see that the flexibility of our method is obvious.

4. Translation and rotation of clip plane

Translation and rotation of clip planes can change the shape of convex clip objects. According to the definition of a plane [25], the parameter *D* is just a negative distance to the origin from the plane. As shown in Fig. 9, the translation vector is **T**, so the equation of moved plane becomes

$$Ax + By + Cz + D' = 0 \tag{11}$$

where

$$D' = D - A\cdot\mathbf{T}.x - \mathrm{B}\cdot\mathbf{T}.y - C\cdot\mathbf{T}.z \tag{12}$$

Rotation of a plane is more complex than translation. To implement plane rotation, the standard equation of a plane in 3D space (Eq. (5)) is converted to the point-normal equation. The point-normal equation of a plane is defined as

$$A(x - x_0) + B(y - y_0) + C(z - z_0) = 0 \tag{13}$$

where (A, B, C) is the normalized normal of the plane, (x_0, y_0, z_0) is a point $\mathbf{P_0}$ on the plane. For convenience, $\mathbf{P_0}$ is defined as an intersection between the plane and a ray which is cast from the origin and has the normalized direction $\mathbf{N} = (A, B, C, 1)$. So, the ray parameterized equation is

$$\mathbf{P}.xyz = \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \cdot t \tag{14}$$

Combining Eqs. (5) and (14), we can obtain the parameter of intersection $t = -D$. So, the point $\mathbf{P_0}$ is equal to

$$\mathbf{P_0}.xyz = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} = -\begin{bmatrix} A \cdot D \\ B \cdot D \\ C \cdot D \end{bmatrix} \tag{15}$$

Now, we rotate the plane normal around the point $\mathbf{P_0}$. As shown in Fig. 10, given a rotation matrix $\mathbf{M_R}$, the new normal $\mathbf{N'} = (A', B', C', 1)$ is computed by

$$\mathbf{N'} = \mathbf{M_R N} \tag{16}$$

Because the standard equation is used for computing ray-plane intersections on GPU, the point-normal equation should be converted back to the standard equation. Combining Eqs. (13), (14) and (16), we have

$$D' = -\mathbf{P_0}.xyz\cdot\mathbf{N'}.xyz \tag{17}$$

Thus, we obtain the standard equation of the rotated plane as follows

$$A'x + B'y + C'z + D' = 0 \tag{18}$$

where $A' = \mathbf{N'}.x$, $B' = \mathbf{N'}.y$, $C' = \mathbf{N'}.z$.

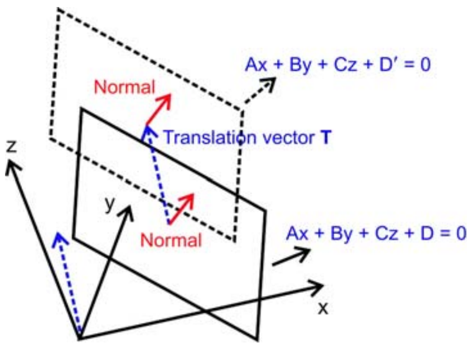

Fig. 9. Plane translation.

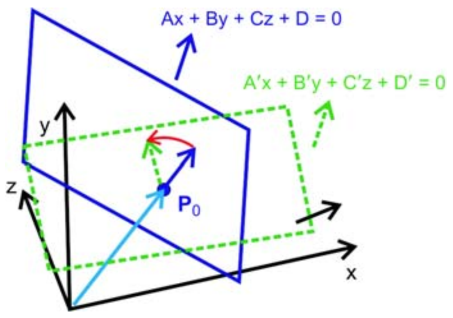

Fig. 10. Plane rotation.

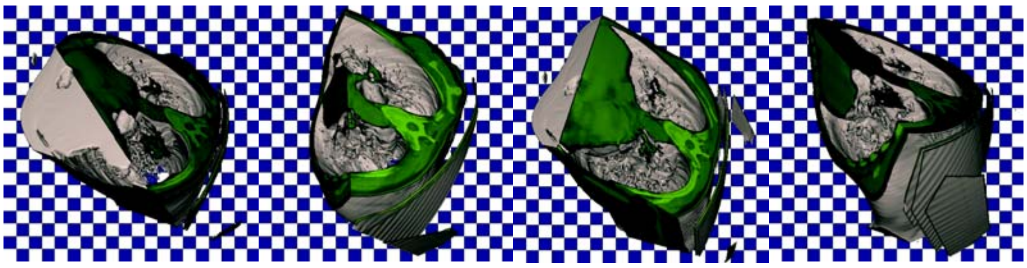

Fig. 11. Translation and rotation of two clip planes.

Figure 11 gives experimental results of our convex volume clipping by interactively translating and rotating two clip planes.

5. Experiments

We implemented the real time GPU ray casting with our volume clipping techniques using Visual C++ and OpenGL Shading Language, and then several experiments were performed on a standard PC with an Intel E2160 dual core 1.8 GHz processor with a 2 Gbyte RAM. And a GeForce FX8600 graphics card was installed.

First, we test our methods on a volumetric data set with the viewpoint outside the clipped volume. The data set was acquired by a CT scanner from a human in

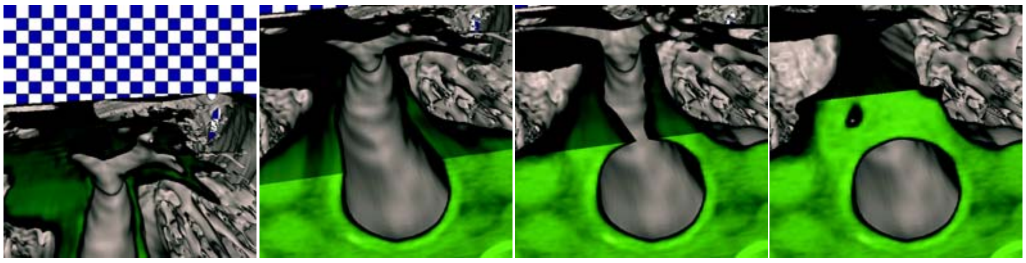

Fig. 12. Convex volume clipping with the viewpoint outside the clipped volume.

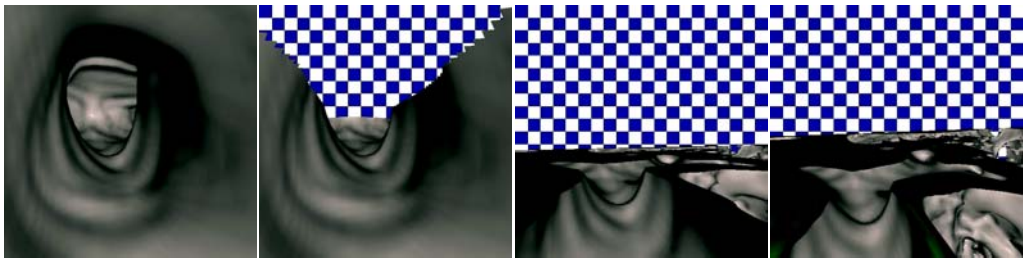

Fig. 13. Convex volume clipping with the viewpoint inside the volume.

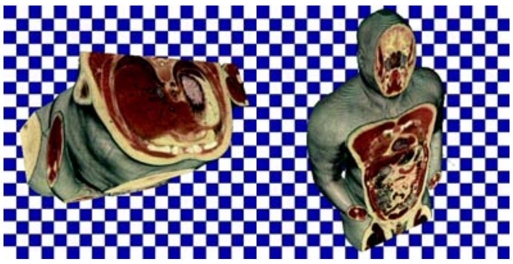

Fig. 14. Clipping the visible human.

a hospital. The data set is composed of 112 DICOM images with 2D cross-section image size 512×512, so the 3D data set size is 512×512×112. An iso-surface ray casting is used for rendering. As shown in Fig. 12, we can find the volume to be correctly clipped and the rendered images to have high quality. We can interactively alter the transfer function by simply re-generating the 2D lookup texture or directly specify different ambient, diffuse and specular colors for the illumination model. So, our algorithm is very convenient in cases where users need to frequently modify material optical property.

Second, we navigate into the human trachea of the volumetric data set, in order to make the viewpoint located inside the volume. As shown in Fig. 13, we can find that the volume is also correctly clipped and the rendered images have high quality. Our method can also process RGB color volumetric data sets by modifying the ray

T a b l e. Frame rates with different viewport sizes and different number of clip planes.

Number of clip planes	Viewport size	Frame rates [fps]
6	512×512	59.4
9	512×512	58.6
12	512×512	57.9
12	600×600	39.7
12	800×800	22.3

casting algorithm and replacing the color transfer function with direct resamplings of RGB volumes. Figure 14 shows the resulting images with our method for the visible human color data set using three clip planes.

The size of data sets does not influence the rendering speed with our method, as long as the data set can be fully loaded into video memory of graphics card. But the size of viewport obviously influences rendering speed. The number of clip planes has a little influence on the rendering speed. The Table illustrates rendering speed under different viewport sizes and different number of clip planes. Experimental results show that our method can render volumetric data sets with an iso-surface ray casting in real time. The method provides users with immediate visual feedback.

6. Conclusions

We propose a real time clipping method based on GPU ray casting, which is capable of using multiple planes for convex volume clipping. The traditional GPU based ray casting methods often use two-pass rendering of front and back proxy surfaces. Different from the classic GPU based ray casting, one proxy plane is used and ray--plane intersections are computed on GPU. Fragment coordinates encode the direction of a ray that is cast from the viewpoint. And we presented an approach to compute intersections between the ray and clip planes, and then the start and end points of the ray are thus obtained by analyzing relationships between the ray direction, intersections and the normal of each plane. Then the volume integral is computed along the ray from the start point to the end point. To obtain immediate visual feedback of volume clipping effects, we implement translation and rotation of planes on GPU to interactively change the shape of clip object. This implementation of GPU based ray casting has good rendering performance and clipping capability. At last, several experiments were performed on a standard PC with an Intel dual core 1.8 GHz processor and a GeForce FX8600 graphics card. Experimental results show that the method can clip and visualize volumetric data sets clearly at real time frame rates.

Acknowledgements – This project was supported by the Natural Science Foundation of Jiangxi Province (2007GQS0076), Foundation of Education Department of Jiangxi Province (GJJ09297) and Key Project of Jiangxi University of Finance and Economics.

References

[1] Pfister H., *Architectures for real-time volume rendering*, Future Generation Computer Systems **15**(1), 1999, pp. 1–9.

[2] Vilanova A., Groller E., Kong A., *Cylindrical approximation of tubular organs for virtual endoscopy*, Technical Report TR-186-2-00-02, February 2000.

[3] Sharghi M., Ricketts I.W., *A novel method for accelerating the visualization process used in virtual colonoscopy*, Information Visualization 2001, San Diego, California, October 22–23, 2001, pp. 167–172.

[4] Brady M., Jung K., Nguyen H.T., Nguyen T., *Two-phase perspective ray casting for interactive volume navigation*, Proceedings of the IEEE Visualization Conference, Phoenix, Arizona, October 19–24, 1997, pp. 183–189.

[5] 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.

[6] Cabral B., Cam N., Foran J., *Accelerated volume rendering and tomographic reconstruction using texture mapping hardware*, Proceedings of IEEE Symposium on Volume Visualization, 1994, pp. 91–98.

[7] 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] *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2000, pp. 109–118.

[8] Westermann R., Ertl T., *Efficiently using graphics hardware in volume rendering applications*, [In] *Proceedings of the ACM SIGGRAPH Conference on Computer Graphics*, 1998, pp. 169–178.

[9] Krüger J., Westermann R., *Acceleration techniques for GPU-based volume rendering*, Proceedings of the IEEE Visualization Conference, 2003, pp. 287–292.

[10] Hadjira Bentoumi, Pascal Gautron, Kadi Bouatouch, *GPU-based volume rendering for medical imagery*, International journal of computer systems science and engineering **1**(1), 2007, pp. 36–42.

[11] 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 **14**(5), 2008, pp. 1126–1139.

[12] Youngmin Kim, Chang Ha Lee, Amitabh Varshney, *Vertex-transformation streams*, Graphical Models **68**(4), 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] Krüger J., Westermann R., *Linear algebra operators for gpu implementation of numerical algorithms*, ACM Transactions on Graphics **22**(3), 2003, pp. 908–916.

[15] Van Gelder A., Kim K., *Direct volume rendering with shading via three-dimensional textures*, Proceedings of the Symposium on Volume Visualization, 1996, pp. 23–30.

[16] Weiskopf D., Engel K., Ertl T., *Interactive clipping techniques for texture-based volume visualization and volume shading*, IEEE Transactions on Visualization and Computer Graphics **9**(3), 2003, pp. 298–312.

[17] Diepstraten J., Weiskopf D., Ertl T., *Transparency in Interactive technical illustrations*, Computer Graphics Forum **21**(3), 2002, pp. 317–325.

[18] Mammen A., *Transparency and antialiasing algorithms implemented with the virtual pixel maps technique*, IEEE Computer Graphics and Applications **9**(4), 1989, pp. 43–55.

[19] Diefenbach P.J., *Pipeline Rendering, Interaction and Realism through Hardware-Based Multi-Pass Rendering*, PhD Thesis, University of Pennsylvania, 1996.

[20] Tiede U., Schiemann T., Hohne K.H., *High quality rendering of attributed volume data*, Proceeding on IEEE Visualization 1998, 1998, pp. 255–262.

[21] WILLIAMS D., GRIMM S., COTO E., ROUDSARI A., HATZAKIS H., *Volumetric curved planar reformation for virtual endoscopy*, IEEE Transactions on Visualization and Computer Graphics **14**(1), 2008, pp. 109–119.

[22] STEGMAIER S., STRENGERT M., KLEIN T., ERTL T., *A simple and flexible volume rendering framework for graphics-hardware based raycasting*, Volume Graphics 2005 Eurographics/IEEE VGTC Workshop Proceedings – Fourth International Workshop on Volume Graphics, 2005, pp. 187–195.

[23] TRIERS P., *GPU ray casting tutorial*, http://www.daimi.au.dk/~trier/?page_id=98.

[24] CHU JINGJUN, YANG XIN, GAO YAN, *Ray-casting-based volume rendering algorithm using GPU programming*, Journal of Computer-Aided Design and Computer Graphics **119**(2), 2007, pp. 257–262 (in Chinese).

[25] YANG WENMAO, *Spacial Analytical Geometry*, Wuhan University Publish House, Beijing, China, 2006, pp. 121–140 (in Chinese).

Received October 5, 2008
in revised form December 11, 2008