Anti-Aliased and Real-Time Rendering of Scenes with Light Scattering Effects

Takashi Imagire[1 2], **Henry Johan**[3], **Naoki Tamura**[1], **Tomoyuki Nishita**[1]

[1] Department of Complexity Science and Engineering, The University of Tokyo, Tokyo, Japan
[2] Namco Bandai Games Inc., Tokyo, Japan
[3] School of Computer Engineering, Nanyang Technological University, Singapore

Abstract Recently, for real-time applications such as games, the rendering of scenes with light scattering effects in the presence of volumetric objects such as smoke, mist, etc, has gained much attention. Slice-based methods are well known techniques for achieving fast rendering of these effects. However, for real-time applications, it is necessary to reduce the number of slice planes that are used. As a result, aliasing (striped patterns) can appear in the rendered images. In this paper, we propose a real-time rendering method for scenes containing volumetric objects that does not generate aliasing in the rendered images. When a scene consists of volumetric and polygonal objects, the proposed method also does not generate aliasing at the boundaries between the polygonal and the volumetric objects. Moreover, we are able to reduce aliasing at shadows inside a volumetric object that are cast by polygonal objects by interpolating the occlusion rates of light at several locations. The proposed method can be efficiently implemented on a GPU.

Key words volumetric object – real-time rendering – anti-aliasing – GPU

1 Introduction

With the advancement in the processing capabilities of GPU technology, more and more expressions can be realized for real-time applications such as games and virtual reality. One of the remaining challenges is the real-time rendering of scenes that include volumetric objects, such as smoke, mist, etc. To achieve fast rendering of these objects, billboard-based methods have been developed. In these methods, the texture of the volumetric object is mapped onto simple rectangular polygons (billboards) and the object is displayed by rendering the billboards; thus fast rendering can be realized. However, when the billboards intersect other objects in the scene, artifacts can be observed at the intersection regions. Therefore, in order to realize effects such as shafts of light, methods that sample the scene using a set of slice planes and compute the light scattering at the slice planes have also developed. However, in a similar way to billboards, these methods will still produce artifacts at the intersections between the slice planes and other objects in the scene.

Since smoke and mist are transparent objects that can be described using their density distributions, we can also use volume rendering methods for high-quality rendering of these objects. Many methods have been proposed for volume rendering, which can be roughly classified into two categories. The first category involves indirect methods, for example the Marching cube method [1] which generates polygons for the iso-surfaces and then displays the polygons. In this technique, some additional storage and processing are required to create the surface polygons. The second category covers direct methods, which integrate intensities along a viewing ray; for example, the ray tracing method and the ray casting [2] method. When using these methods, the integration is computed approximately by sampling the intensities at several points along the viewing ray.

With the recent advancement of GPU technology, several methods for accelerating volume rendering using a GPU have been proposed [7–9,3,10,11,4,13,5,12,6,14]. Most of these involve slice-based methods, which work by slicing the volume data into several planes which are represented using polygons and then visualizing the volume data by rendering these planes from back-to-front. Fast rendering can be achieved by implementing the slice-based methods using a GPU. By controlling the number of slice planes, it is possible to control the trade-off between rendering speed and image quality. To achieve real-time rendering, it is necessary to reduce the number of slice planes. However, this can cause aliasing patterns to appear in the rendered images. Figure 1 shows rendered images that include aliasing of the type that may be produced when we using a slice-based methods. These aliasing are easy to be noticed especially when we move the objects or the viewpoint. In interactive ap-

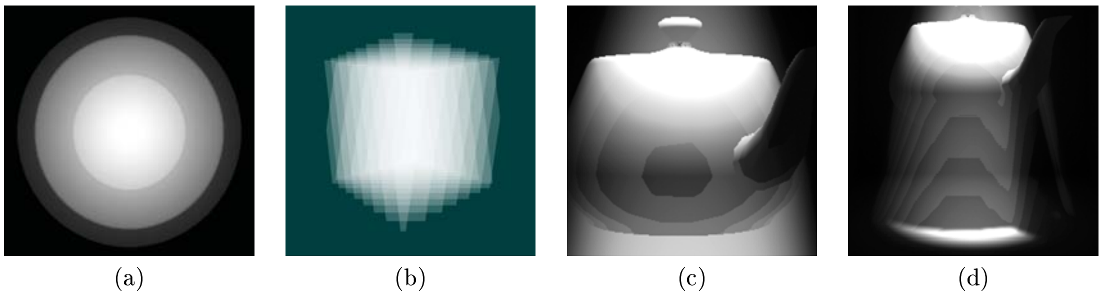


Fig. 1 Aliasing in images that are the results of rendering a volumetric object using a slice-based method; (a) shows aliasing when the number of slices is very small compared with the resolution of the volumetric object (in this example, the volumetric object is a uniform density sphere defined using voxels with a resolution of 128 × 128 × 128, and the number of slices for rendering is four); (b) shows aliasing at the edges of the slice planes; (c) shows aliasing at the intersections between an opaque polygonal object and the slice planes; (d) shows aliasing at the shadow cast by an opaque polygonal object.

plications where the viewpoint is often moved, it is very crucial to eliminate these aliasing.

Up to now, several methods for removing aliasing have been proposed. However, no effective method for removing all four of the types of aliasing that are shown in Figure 1 has been proposed as yet. In this paper, we propose a real-time rendering method for scenes containing volumetric objects such as smoke and mist that does not generate aliasing in the rendered images. For each pixel in the rendered image, we determine the nearest intersection point from a viewpoint between the viewing ray and the opaque objects in the scene. Based on this information, we compute a valid sampling path along the viewing ray that does not intrude on the opaque objects in order to correctly sample the volumetric object for intensity computation. The proposed method can be implemented efficiently by using a GPU. Our method utilizes the different processing parts of the GPU in a balanced manner. As a result, we are able to speed up the rendering process. Moreover, to reduce aliasing in the shadow regions inside a volumetric object that are cast by opaque objects, we compute the occlusion rate of light at a particular location by averaging the occlusion rates computed at several nearby locations.

The remainder of this paper is organized as follows. Section 2 outlines related work. Section 3 presents the details of our method. Section 4 describes the essential points for implementation. Rendering results will be demonstrated in Section 5. Finally, we summarize the paper in Section 6.

2 Related Work

In this section, we review some of the related work in real-time volume rendering.

2.1 Volume rendering using a GPU

The slice-based volume rendering method [8] is performed by arranging several slice planes at regular intervals, sampling volumetric objects at the slice planes, and then rendering the slice planes using a GPU. A slice plane is a polygon generated by calculating the intersections between the bounding box of the volumetric object and a plane. There are two methods for generating the slice planes. The first of these is a method that slices volumetric objects parallel to the screen (image-aligned slices). The second method slices volumetric objects along the axes of their volume local coordinates (object-aligned slices) [7].

Rendering using the slice-based method is performed as follows. First, we compute the slice planes, then, for each vertex of the slice planes we assign the volume local coordinates that are necessary for sampling the volumetric object for color and opacity information. Finally, the slice planes are rendered in 'back-to-front' order from the viewpoint and the results are composited, taking into account the opacity. In the rendering process, the volume local coordinates stored in the vertices of the slice plane are interpolated to compute the volume local coordinates for pixels inside the slice plane. To enable real-time processing using a slice-based method, it is necessary to limit the number of slice planes. However, this can result in aliasing appearing in the rendered images.

2.2 Anti-aliasing

Interleaved sampling [4] reduces aliasing by synthesizing two or more pre-rendered images and then blending these images while changing the weights of blending for each image in each pixel. The pre-rendered images are generated by rendering the volumetric object while shifting the location of the slice planes. However, this method creates periodic noise because a dither-pattern is used periodically to calculate the weight of blending. Moreover, it is necessary to generate small-sized pre-rendered images in order to achieve real-time rendering. This results in some deterioration of the quality of the final image.

Engel et al. [11] and Guthe et al. [5] proposed a method that pre-computes the integral of the transfer function along the viewing ray and stores the results as a texture. In the rendering process, the image is generated by referring to the values stored in the texture. However, this method cannot remove aliasing at the intersection regions between a volumetric object and opaque objects.

The spherical billboard [15] technique eliminates aliasing by calculating the length of the path traveled by a viewing-ray inside a volumetric object. The limitations of this method are that it assumes that the volumetric object is approximately spherical in shape and it cannot remove the aliasing that is derived from shadows.

Kajihara et al. [14] proposed a method for removing aliasing around the intersections between a volumetric object and opaque objects by using a Ray-Volume Buffer. A Ray-Volume Buffer is a 3D buffer that stores the light intensity and total opacity at each voxel. However, this method requires a great deal of memory to store the Ray-Volume Buffer. Furthermore, it takes several seconds to perform the rendering process when using this method.

Dobashi et al. [13] proposed a method for reducing aliasing in shadow regions by detecting the shadow regions and then adding a number of slice planes in these regions. However, this method requires a large number of slice planes in order to remove aliasing at the intersections between the volumetric object and opaque objects. As a result, it is difficult to achieve real-time rendering in such cases.

In this paper, we introduce the concept of **sampling hulls** for the anti-aliased rendering of volumetric objects. We divide the bounding box of the volumetric object into several sampling hulls, render the hulls, and then composite the results. A volumetric object inside a sampling hull is rendered by computing the sampling points inside the hull and integrating the light intensities at the sampling points. Aliasing due to the presence of opaque objects is removed by taking into account the position of the opaque objects when performing sampling inside the sampling hulls. Moreover, aliasing at shadows is reduced by interpolating the occlusion rates of light at several locations.

3 Anti-Aliased Volume Rendering

3.1 Basic idea for removing aliasing

In this paper, a volumetric object is represented using a set of voxels. The outermost part of the voxel region defines the bounding box of the volumetric object. A density value is defined for each voxel. Each voxel is accessed through a volume local coordinate. We assume that there is one volumetric object inside a scene consisting of opaque polygonal objects.

There are two causes of aliasing in slice-based methods. During the rendering process, slice-based methods

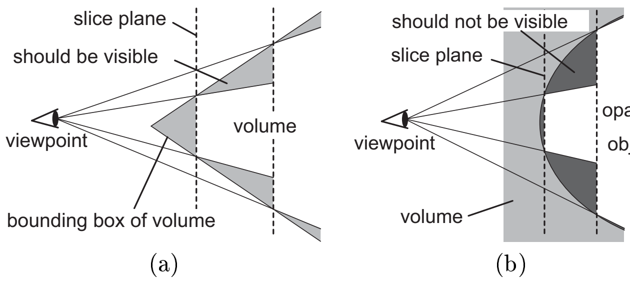


Fig. 2 Causes of aliasing in slice-based methods; (a) shows the shaded regions where a volumetric object exists but is not considered during rendering; (b) shows dark shaded regions that are mistakenly considered as if a volumetric object exists in these regions.

consider the intersection regions between the slice planes and the bounding box of a volumetric object that is swept along the viewing rays (for instance, the unshaded region inside the bounding box of the volume in Figure 2(a)). The density in the space between two slices is assumed to be uniform according to the density of one of the slices.

The first cause of aliasing is that there are regions inside the volumetric objects that are visible from the viewpoint but are not considered during the rendering process. The shaded regions in Figure 2(a) are such regions, and this can cause aliasing at the boundaries of the volumetric object. The second cause of aliasing occurs when, in the presence of opaque objects, there are regions of volumetric objects that are inside the opaque objects that should not be considered but are mistakenly rendered. As a result, aliasing can appear at the boundaries between the volumetric and the opaque objects. Figure 2(b) shows an example where there is an opaque sphere inside a volumetric object. The dark shaded regions in Figure 2(b) are regions where no volumetric object exists, but, when using a slice-based method, they are actually treated as if a volumetric object does exist inside them.

To solve these two causes of aliasing, we compute a valid sampling path along the viewing ray and perform sampling along this path for intensity computation. The valid sampling path will be explained in the next section. In the presence of opaque objects, we make sure that the path will not intrude on the opaque objects.

3.2 Rendering a volumetric object using sampling hulls

A GPU consists of multiple computation units that make it possible to process several polygons and pixels in parallel [16]. In addition, computations such as shading, alpha blending, etc, can be performed simultaneously. The processing speed of a GPU depends on the slowest process among those processes that are performed simultaneously. In a typical GPU, memory access is considered to be a slow process. In the slice-based methods,

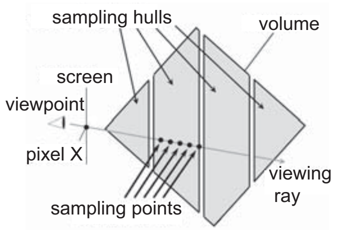


Fig. 3 The sampling hulls. In this figure, we display a space between neighboring sampling hulls to make the sampling hulls easy to recognize (actually, there is no space between sampling hulls).

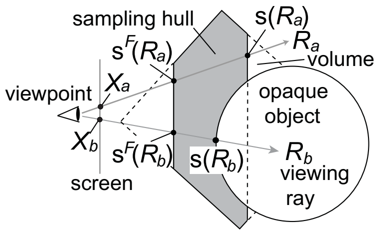


Fig. 4 The valid paths from $\mathbf{s}(R)$ to $\mathbf{s}^F(R)$ for sampling inside a sampling hull.

rendering a slice plane involves alpha blending; that is, it involves reading and writing to memory. Therefore, when the number of slice planes is large, the slice-based methods have high computational cost.

Our method is designed to take into account the performance capabilities of the GPU. In our method, we reduce the requirement for alpha blending by introducing the concept of **sampling hulls**. A sampling hull is defined as a closed convex polyhedron resulting from cutting the bounding box of a volumetric object parallel to the screen at two locations (see Figure 3). When we render the volumetric object, we perform multiple samplings inside the sampling hull, integrate the scattering light intensities at the sampling points, and perform alpha blending only once for each sampling hull. As a result, by using the same number of samplings, our method can reduce the number of alpha blending operations that are required compared to slice-based methods.

When using the proposed method, we have to decide the number of sampling hulls and also the number of sampling points inside each hull. If there are too many sampling hulls, then the number of alpha blending operations is large, and thus the rendering speed will be slow. If the number of sampling hulls is too small, then we have to increase the number of sampling points inside the hulls in order to generate high-quality images. However, in this case, the computational cost for sampling inside hulls is bigger than the cost for alpha blending. As a result, this will slow down the rendering process. In this paper, to realize real-time rendering, the optimal number of sampling hulls and the number of sampling points inside each hull are determined through experiments (see Section 5 for details).

Efficient rendering of volumetric objects using sampling hulls is implemented as follows. Assume that the final rendered image is stored inside the frame buffer. First, as mentioned above, we divide the bounding box of the volumetric object into a set of sampling hulls (see Figure 3). Next, we render the opaque objects in the scene into the frame buffer. Then, we process the sampling hulls in 'back-to-front' order with respect to the screen. For each sampling hull, we render the volumetric object inside the hull and composite the resulting image with the image in the frame buffer using alpha blending.

We render a volumetric object inside a sampling hull as follows. Assume R is the viewing ray from the viewpoint through pixel X on the screen. Let $\mathbf{s}(R)$ be the intersection point between the viewing ray R and the back face of the sampling hull or the surface of an opaque object, whichever is nearest to the viewpoint. $\mathbf{s}(R)$ will then be the location of the first sampling point. We also define $\mathbf{s}^F(R)$ as the intersection point between the viewing ray R and the front face of the sampling hull or the surface of an opaque object, whichever is nearest to the viewpoint. Thus, the path from $\mathbf{s}(R)$ to $\mathbf{s}^F(R)$ is the valid sampling path inside the sampling hull and we perform the sampling along this valid path. Figure 4 shows some examples.

Let N be the number of sampling points on each valid path. The location $\mathbf{s}_i(R)$, $i = 0, 1, \cdots, N-1$ of the i-th sampling point is calculated as follows.

$$\mathbf{s}_i(R) = \mathbf{s}(R) + i \times \Delta\mathbf{s}(R), \tag{1}$$

$$\Delta\mathbf{s}(R) = (\mathbf{s}^F(R) - \mathbf{s}(R))/N. \tag{2}$$

When the opacities at the voxels are given, the opacity at each sampling point is computed as the corrected opacity [17] (see Appendix A.1) taking into account the sampling interval. For each sampling point, we compute the intensity of the scattering light. The intensity at pixel X is computed by integrating the intensities of the scattering light at all of the sampling points.

It is obvious that the lengths of the valid sampling paths can be different for each pixel. At first glance, it seems that changing the number of sampling points according to the length of the valid path will increase the rendering efficiency. However, to implement this, we have to use a branching command. Unfortunately, the execution of branching commands is relatively slow when using a GPU, and this decreases the rendering speed. Therefore, in our method, we use the same number of sampling points for all of the valid sampling paths.

3.3 Anti-aliased shadows

When the volumetric object is a medium such as smoke, it is necessary to consider the volumetric shadow in space

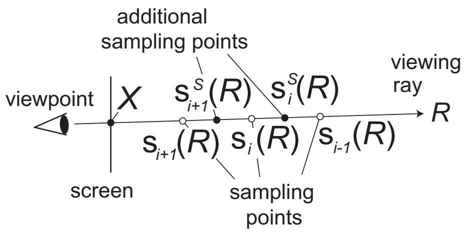


Fig. 5 Additional sampling points for shadow computations.

that is caused by opaque objects. We propose the following method for generating anti-aliased shadows. When we compute the occlusion rate of light at a particular sampling point, we reduce aliasing by interpolating the occlusion rates at several nearby locations.

The occlusion rate at sampling point $\mathbf{s}_i(R)$ is computed as follows. First, we set additional sampling points along the viewing ray R at the middle of $\mathbf{s}_i(R)$ and its neighbor sampling points (Figure 5). The additional sampling points $\mathbf{s}^S_j(R), j = i, i+1$ are set at the following locations.

$$\mathbf{s}^S_j(R) = 0.5(\mathbf{s}_j(R) + \mathbf{s}_{j-1}(R)). \tag{3}$$

Then, the occlusion rate at sampling point $\mathbf{s}_i(R)$ is determined by the weighted average using the following equation.

$$I_i(R) = (1/4)I^S_i(R) + (1/2)I_i(R) + (1/4)I^S_{i+1}(R), \tag{4}$$

where $I_i(R)$ is the occlusion rate at $\mathbf{s}_i(R)$ and $I^S_j(R), j = i, i+1$ are the occlusion rates at $\mathbf{s}^S_j(R)$. In this paper, the occlusion rates are computed using the variance shadow map method [18]. We adopt the variance shadow map method because it can generate soft shadows.

4 GPU Implementation

We assume that the volumetric object is stored inside the texture memory. When the viewpoint or the volumetric object moves, we divide the bounding box of the volumetric object into M pieces of sampling hulls by cutting the bounding box parallel to the screen. This computation is performed in the CPU. The final rendered image is stored in a frame buffer which we call the **back buffer**. We also prepare a frame buffer that has the same size as the back buffer, and we call this the **local coordinates map**. The local coordinates map is used to store the volume local coordinates. It is necessary to provide a buffer with sufficient precision for the local coordinates map. If the voxel resolution of the volumetric object is less than 256, then we use an 8 bits integer buffer; otherwise we use a floating point buffer.

The rendering process is performed in $2 + 2M$ steps (Figure 6). We assume that we have already created a shadow map using the variance shadow map method for each light source [18]. In the first step, the opaque objects are rendered into the local coordinates map (Figure 6(a)). In this rendering process, the coordinates of the objects are transformed into the volume local coordinates and these local coordinates (x, y, z) are used as the color (r, g, b) of the objects. In the second step, we render the opaque objects into the back buffer using the usual shading computation (Figure 6(b)).

The rest of the $2M$ steps are performed to render the sampling hulls in back-to-front order from the screen. We perform the following two steps for each sampling hull. First, we render the back faces of the sampling hull into the local coordinates map using the volume local coordinates as colors (Figure 6(c)). For the depth test, we use the z-buffer result from the previous step. As a result of the depth test, the volume local coordinates of opaque objects will be retained if the sampling hull is occluded by opaque objects.

Next, we switch the rendering destination to the back buffer. For each pixel X, we perform the following computations. We use the volume local coordinates stored in pixel X of the local coordinates map as the starting point $\mathbf{s}(R)$, where R is the viewing ray from the viewpoint through pixel X, for sampling inside the sampling hull. The end point $\mathbf{s}^F(R)$ is computed by rendering the front faces of the sampling hull using the volume local coordinate as the attribute. Based on $\mathbf{s}(R)$ and $\mathbf{s}^F(R)$, we compute N sampling points (Equation 2), compute the scattering light intensity at each sampling point by considering the occlusion rates of the light, integrate the scattering light intensities at the sampling points, and composite the result with the value previously stored inside the back buffer. The pseudo-code of our algorithm is shown in Figure 7.

The optimal number of sampling hulls M and the number of sampling points inside each sampling hull N are decided based on the intended image quality and the performance of the GPU. The image quality can be increased by increasing the total number of sampling points ($M \times N$) per viewing ray. Even when we use the same total number of sampling points, varying the values of M and N affects the rendering speed. For instance, from our experiments, using a total of 32 sampling points, setting $M = 1$ and $N = 32$ will produce a slower rendering performance compared to setting $M = 8$, $N = 4$. The reason for this is that when we use only one sampling hull, most of the computation is carried out in the ALU (Arithmetic and Logical Unit) of the fragment shader, while the ROP (Rasterizing OPeration unit), which blends a color to a frame buffer, is virtually unused. As a result, there is an unbalanced utilization of the processing units of the GPU, resulting in a decrease in performance. In this paper, the optimal values of M and N for each scene are determined by experiments (see Section 5 for details).

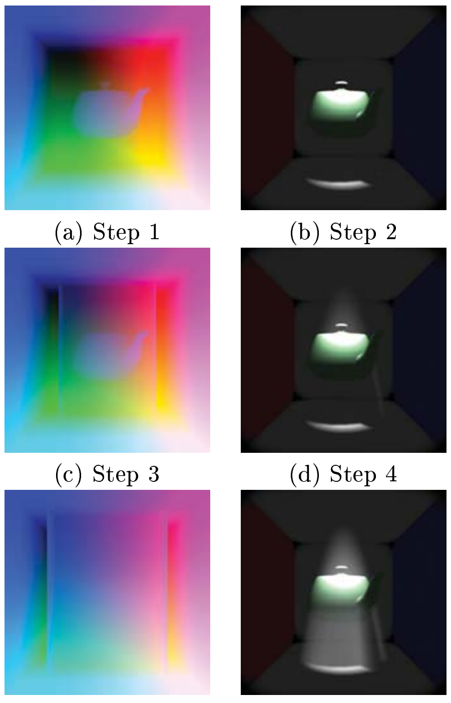

(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4 (e) Step 2M+1 (f) Step 2M+2

Fig. 6 Result of each step of our rendering method. The scene is a floating teapot inside a box. The light source is a spotlight. (a),(c),(e) show the contents of the local coordinates map and (b),(d),(f) show the contents of the back buffer during the rendering.

5 Results

Figures 8-11 show various rendering results and overall performance when using the proposed method. We use fp16 (16 bits floating point) buffers during the rendering processes. The rendering results are then converted to the 8 bits format to produce the rendered images. The sizes of the images are 720×540 pixels. In our experiments, we use a desktop PC with an Intel Pentium D 3.2GHz CPU and an nVIDIA GeForce 7800 GTX GPU. We implemented our method on the DirectX 9 platform.

For each scene, the optimal number of sampling hulls and sampling points in each sampling hull are decided as follows. First, we create reference images using a slice-based method with 1024 slice planes. Then, we perform experiments using the proposed method by varying the number of sampling hulls and sampling points, and look for the optimal values that yield the maximum rendering performance without producing any artifacts in the rendered images compared to the reference images.

Figure 8 shows the results of rendering a foggy forest scene. The fog is represented using voxels with a resolution of $128 \times 128 \times 128$. Figures 8(a) and (b) show the results obtained by the proposed method when setting $M = 2$, $N = 4$ and $M = 4$, $N = 8$ respectively. There is no aliasing in either images. However, due to an insufficient number of sampling points, the result in Figure 8(a)

```
// ← : store the results in a frame buffer specified
//       on the left hand side

LocalCoordinatesMap ← transform the coordinates of
      objects into volume local coordinates and render the
      objects using the volume local coordinates as colors;

BackBuffer ← render objects using the usual shading;

For $i = 1$ to $M$ (from back to front order)
      $B$ = back surface of hull $i$;
      LocalCoordinatesMap ← render $B$ using the
            volume local coordinates as colors;

      $F$ = front surface of hull $i$;
      BackBuffer ← render $F$ using volume local
            coordinates as colors and compute
            ($A \times$BackBuffer$+C$);
            // A and C are computed using
            // the subroutine described below
EndFor
```

```
Subroutine: compute $C$ and $A$ for each pixel
      $C = 0.0$;
      $A = 1.0$;
      For $j = 1$ to $N$
            compute sampling point from volume local
                  coordinates of $F$ and LocalCoordinatesMap;
            sampling color $C_0$ and opacity $A_0$;
            $A_0$ = opacity correction of $A_0$;
            $C = A_0 \times C_0 + (1.0 - A_0) \times C$;
            $A = (1.0 - A_0) \times A$;
      EndFor
EndSubroutine
```

Fig. 7 Pseudo-code of our algorithm.

shows an unnatural color contrast. Figure 8(c) shows the result of using a slice-based method, in which we set the number of slices to 128. Using this method, striped patterns can be seen at the intersections between the slice planes of the fog and the polygonal model of the ground (see Figure 8(d)). Although we can eliminate aliasing such as this if we use 512 slices, the rendering performance then drops to 26 fps. Table 1 shows the rendering performance when we change the number of sampling hulls M and the number of sampling points N.

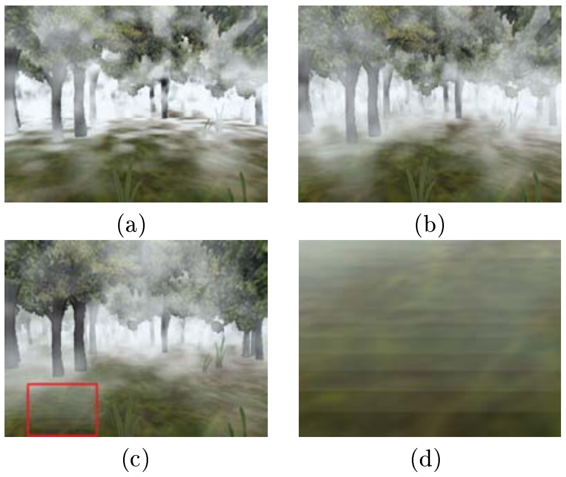
Figure 9 shows the results of a water tank scene. The water is modeled as a volumetric object with a voxel resolution of $128 \times 64 \times 64$. Figures 9(a) and (b) show the results using our method. We rendered the scene from two different viewpoints. The numbers of sampling hulls and sampling points are set to 2 and 8, respectively. Figure 9(c) shows the result of using a slice-based method with 64 slices, where the viewpoint is set to be the same as that in Figure 9(b). Using the above-mentioned parameters, both methods can render the scene at around 96 fps. However, when using the slice-based method, aliasing can be seen at the surface of the object inside the water and also at the water tank (see Figure 9(d)).

Table 1 The changes of the frame rate for the foggy forest scene when the number of sampling hulls and the number of sampling points are changed when the total sampling count is 32.

Number of hulls	1	2	4	8	16	32
Number of samples	32	16	8	4	2	1
fps	50	59	64	62	55	47

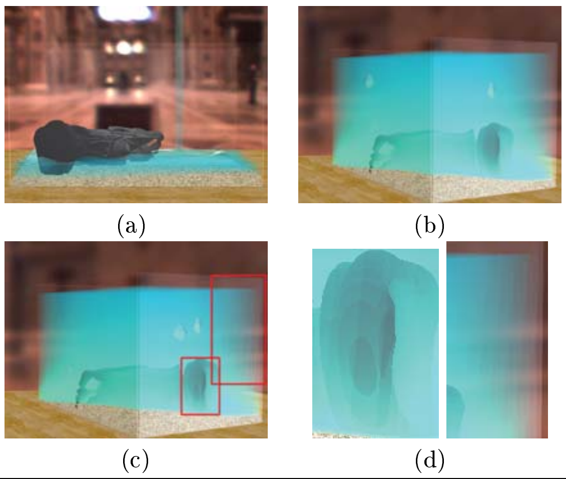
Figure 10 shows a scene that includes a teapot inside a room with two light sources. The space inside the room is modeled as a volumetric object. For each voxel, we compute the intensity of the scattering light, taking into account the locations and the directions of the light sources. We neglect the phase function when computing the intensity of the scattering light. To compute shadows, we create shadow maps with a resolution of 256 × 256. The directions of the light sources change with time. Figures 10(a) and (b) show the results of the proposed method using 4 sampling hulls and 4 sampling points inside each hull. The light directions are different in the two images. Figure 10(c) shows the result of the slice-based method using 32 slices. Figure 10(d) shows the aliasing in Figure 10(c). For comparison, our method does not generate aliasing in the rendered image (Figure 10(e)) and the rendering process is faster than it is for the slice-based method.

Figure 11 shows a scene containing a Buddha with one moving spotlight. We store the intensity of the scattering light due to the spotlight as a volumetric object that is defined in the local coordinate system of the spotlight with a resolution of 64 × 64 × 128. Thus, when the spotlight moves, the volumetric object also moves accordingly. This way of representing the light distribution of a particular light source is useful for accelerating the rendering process in games applications. We create a shadow map with a resolution of 256 × 256 for generating shadows. Figures 11(a) and (c) show the results of the proposed method (8 sampling hulls and 8 sampling points) whereas Figures 11(b) and (d) are the results of a slice-based method with 64 slices. Figure 11(c) shows that the proposed method can reduce aliasing in volumetric shadows when compared to a slice-based method (Figure 11(d)).



Method	fps
Our method (2 hulls, 4 samples)	80 fps
Our method (4 hulls, 8 samples)	64 fps
Slice-based method (128 slices)	70 fps (has aliasing)
Slice-based method (512 slices)	26 fps

Fig. 8 A foggy forest scene. (a) and (b) are the results of the proposed method using 2 hulls with 4 samples and 4 hulls with 8 samples, respectively. (c) is the result of a slice-based method using 128 slices. (d) shows the aliasing in (c).

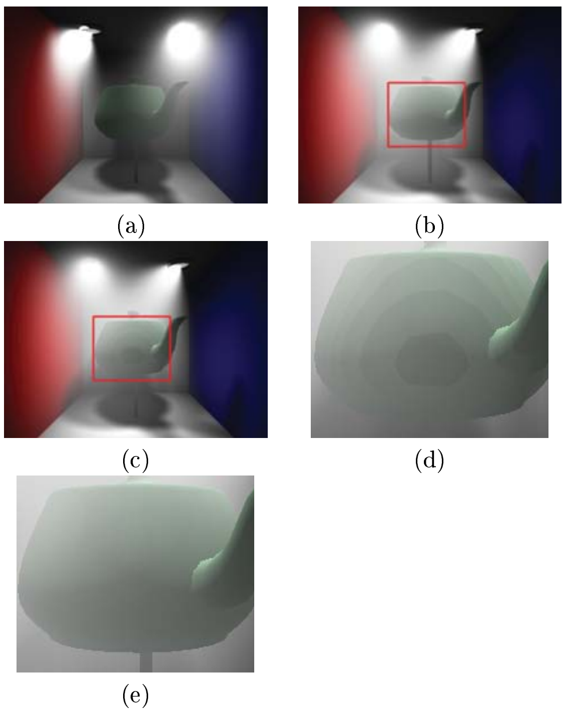


Method	fps
Our method (2 hulls, 8 samples)	96 fps
Slice-based method (64 slices)	96 fps (has aliasing)

Fig. 9 A water tank scene. There is sand at the bottom of the water tank and a Buddha statue lying on the sand. (a) and (b) are the results of the proposed method (2 hulls, 8 samples). (c) is the result of a slice-based method with 64 slices. (b) and (c) are rendered from the same viewpoint. The rendering performances of both methods are 96 fps. However, (c) has aliasing that is shown in (d).

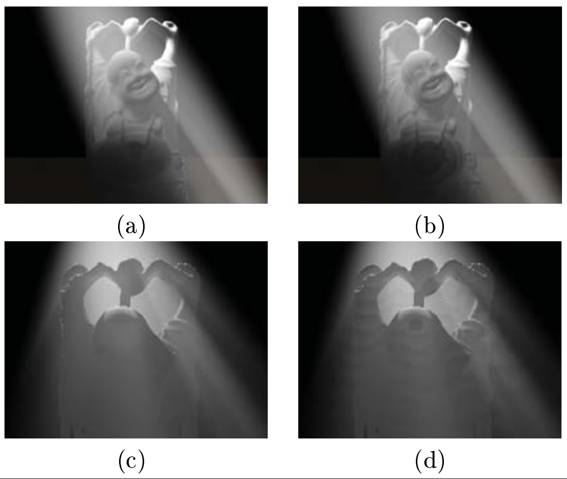
6 Conclusion and Future Work

In this paper, we have proposed a method for anti-aliased volume rendering. For each viewing ray, we compute a valid sampling path that takes into account intersections between the viewing ray and opaque objects to correctly sample the volumetric object. To efficiently execute our method on a GPU, we introduce the concept of sampling hulls. Most of the processing steps in the proposed method can be implemented on a GPU. In addition, our method utilizes the various processing units inside the GPU in a balanced manner. As a result, it is possible to perform high-quality volume rendering in real-time using the proposed method. We also reduce the aliasing at shadows by interpolating the occlusion rates of light at several locations to compute the occlusion rate at a sampling point.



Our method (4 hulls, 4 samples)	82 fps
Slice-based method (32 slices)	68 fps (has aliasing)

Fig. 10 A teapot on the top of a rod in a room with two light sources. (a) and (b) are the results of the proposed method (4 hulls, 4 samples). (c) is the result using a slice-based method (32 slices). (d) shows the aliasing in (c). For comparison, (e) shows that our method does not generate aliasing.



Our method (8 hulls, 8 samples)	77 fps
Slice-based method (64 slices)	203 fps (has aliasing)

Fig. 11 A Buddha statue with a moving spotlight. (a) and (c) are the results of the proposed method (8 hulls, 8 samples). (b) and (d) are the results of a slice-based method (64 slices). We can see the aliasing on the face and the stomach of the Buddha in (b) and inside the volumetric shadow in (d).

In our method, the image quality and the rendering performance depend on the number of sampling hulls and the number of sampling points inside a hull. Currently, we determine these values experimentally. For future work, we want to determine the optimal values of these parameters automatically by taking into consideration the performance of the hardware, the resolution of the volume data, and the details of the geometry of the opaque objects.

References

1. W. E. Lorensen, H. E. Cline: Marching cube: a high resolution 3D surface construction algorithm. In: Computer Graphics (Proc. SIGGRAPH 1978), pp. 163–169 (1978).
2. H. Tuy and L. Tuy: Direct 2D display of 3D objects. In: IEEE Computer Graphics and Applications, 4(10), pp. 29–33 (1984).
3. R. Westermann, T. Ertl: Efficiently using graphics hardware in volume rendering applications. In: Computer Graphics (Proc. SIGGRAPH 1998), pp. 291–294 (1998).
4. A. Keller, W. Heidrich: Interleaved sampling. In: Proc. of the 12th Eurographics Workshop on Rendering Techniques, pp. 269–276 (2001).
5. S Guthe, S, Roettger, A. Schieber, W. Strasser, T, Ertl: High-quality unstructured volume rendering on the PC platform. In: SIGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 119–126 (2002).
6. J. Kruger, R. Westermann: Acceleration techniques for GPU-based volume rendering. In: IEEE Visualization 2003, pp. 287–292 (2003).
7. L.A. Westover: Splatting: a parallel, feed-forward volume rendering algorithm. In: doctoral thesis, Dept. of Computer Science, Univ. of North Carolina at Chapel Hill, Chapel Hill, N.C., (1991).
8. T. J. Cullip, U. Neumann: Accelerating volume reconstruction with 3D texture hardware. In: Tech. Rep. TR93-027, University of North Carolina, Chapel Hill N.C. (1994).
9. M. Brady, K. Jung, H.T. Nguyen, T Nguyen: Two-phase perspective ray casting for interactive volume navigation. In: Visualization '97, pp. 183–190 (1997).
10. C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, T. Ertl: Interactive volume rendering on standard PC graphics hardware using multitextures and multi-stage rasterization. In: SIGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 109–119 (2000).
11. K. Engel, M. Kraus, T. Ertl: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In: Eurographics/SIGGRAPH Workshop on Graphics Hardware, pp. 9–16 (2001).
12. J. Kniss, S. Premoze, C. Hansen, D. Ebert: Interactive translucent volume rendering and procedural modeling. In: IEEE Visualization 2002, pp. 168–176 (2002).
13. Y. Dobashi, T. Yamamoto, T. Nishita: Interactive rendering of atmospheric scattering effects using graphics hardware. In: Proc. of Graphics Hardware 2002, pp. 99–108 (2002).
14. Y. Kajihara, H. Takahashi, M. Nakajima: A method of rendering scenes including volumetric objects using ray-

volume buffers. In: Proc. of Computer Graphics International 2003, pp. 930–235 (2003).
15. T. Umenhoffer, L. Szirmay-Kalos, G. Szijarto: Spherical billboards and their application to rendering explosions. In: Proc. of the 2006 Conference on Graphics Interface, pp. 57–63 (2006).
16. J. Montrym, H. Moreton: The GeForce 6800. In: IEEE Micro, Vol 25, No 2, 41–51 (2005)
17. D. Laur, P. Hanrahan: Hierarchical splatting: A progressive refinement algorithm for volume rendering. In: Computer Graphics (Proc. SIGGRAPH 1991), pp. 285–288 (1991).
18. W. Donnelly, A. Lauritzen: Variance shadow maps. In: Proc. of the Symposium on Interactive 3D Graphics and Games, pp. 161–165 (2006).

A Appendix

A.1 Corrected opacity

Let Δd_0 be the voxel interval. Assume that the opacity α that is stored in a voxel is computed by using the opacity transfer function with Δd_0 as the sampling interval.

$$\alpha = 1 - e^{-\phi \Delta d_0}. \tag{5}$$

Here, ϕ is an extinction coefficient. Then, for another sampling interval Δd, the corrected opacity can be computed as follows.

$$\alpha_{corrected} = 1 - [1 - \alpha]^{\frac{\Delta d}{\Delta d_0}}. \tag{6}$$