Technical Section

Real-time ambient occlusion and halos with Summed Area Tables

José Díaz a, Pere-Pau Vázquez b,*, Isabel Navazo a, Florent Duguet b

a MOVING Group, Universitat Politècnica de Catalunya, Barcelona, Spain
b ALTIMESH, Versailles, France

1. Introduction

One of the most important goals of visualization is to provide the adequate insights into the data being analyzed. With the increase in computational power and the improvements of capture processes, volume visualization algorithms are fed with more and more complex datasets. Understanding spacial arrangement of such complex volumetric models is a difficult task, especially with renderings of volume data acquired from CT or MRI. Rendered images of such models are difficult to interpret because they present many fine, overlapping structures. In specialized media, e.g. medicine books, illustrators often use a set of special techniques in order to obtain the depictions that reduce perception ambiguity, such as halos, contours, or cut-away views. Computer graphics researchers have borrowed some of these techniques and used them successfully in visualization. Improving perception of volumetric models has usually been addressed by two different kinds of methods: a more realistic shading model or using non-photorealistic techniques.

Volumetric data is typically illuminated by one or several point light sources and the shading of each sample point in the volume is calculated by using a Phong model. This illumination model provides good perceptual cues but mainly on the orientation of isosurfaces. However, this usually results in images with poor depth cues. Shadows can be used to solve this problem, but shadow generation is costly and may introduce illumination discontinuities on the surface.

We propose two simple and fast approaches for enhancing depth perception of volumetric models. In order to do so, we add ambient occlusion (vicinity shading) and halos in real-time. In both approaches we propose a data structure and an algorithm for fast rendering based on Summed Area Tables. The first method: vicinity occlusion maps (VOM), uses a data structure that is computed on the fly for each frame and allows to generate view dependent ambient occlusion and halos (see Fig. 1) at low cost. The second approach, Density Summed Area Table (DSAT3D) uses a precomputed 3D Summed Area Table of the density values of a model, and allows the computation of view independent ambient occlusion and halos (see Fig. 1) at low cost. The second approach, Density Summed Area Table (DSAT3D) uses a precomputed 3D Summed Area Table of the density values of a model, and allows the computation of view independent ambient occlusion and halos (see Fig. 1) at low cost. The second approach, Density Summed Area Table (DSAT3D) uses a precomputed 3D Summed Area Table of the density values of a model, and allows the computation of view independent ambient occlusion and halos (see Fig. 1) at low cost.
depth map, similar to other recent ambient occlusion methods, while the latter takes into account 3D information thus making this approach view independent. This paper is an extension of [1], where the VOM data structure was presented for halo rendering and ambient occlusion simulation. In this paper, we present a new method that uses DSAT3D data structure, and the CUDA implementations for the construction of both data structures.

Our goal is to enhance depth perception for volumetric models. Our main contributions can be summarized as:

- **Vicinity occlusion map (VOM):** A data structure computed on the fly whose main advantages are: (i) no need for precomputation; (ii) limited (i.e. constant per frame) impact on rendering; and (iii) may be used for computing view-dependent real-time halos (size or color can be changed interactively).
- **Density Summed Area Table (DSAT3D):** A 3D Summed Area Table of the density values of the model that allows for the fast computation of ambient occlusion. Its main features are: (i) view independent ambient occlusion computation; (ii) almost no impact in rendering performance; and (iii) requires precomputing a 3D texture, at the expense of its corresponding memory footprint.

The visual quality of both techniques is similar except for structures where large depth variations appear. In these cases, VOMs might generate oversharpened regions and DSAT3D may be more adequate.

The paper is organized as follows: First, Section 2 overviews previous work. In Section 3 we present some background useful for the remainder of the paper. In Section 4 we describe our first data structure: vicinity occlusion maps, and the algorithms that use it for simulating ambient occlusion and rendering halos. In Section 5 we introduce the Density 3D Summed Area Tables and show how to use it for fast ambient occlusion computation. The results are analyzed in Section 6, where we also discuss the pros and cons of both methods and how they compare to previous approaches. Finally, we conclude on the paper in Section 7 and discuss some directions for future research.

2. Previous work

In this section we analyze previous methods for enhancing the perception of depth in volumetric models. The two main approaches are realistic shading and non-photorealistic techniques.

Realistic shading models: These methods simulate classical rendering effects, such as shadows, by applying different generation processes. There are several proposals for applying shadows in volume rendering [2–4] although the generation process may differ importantly. For instance, Yagel et al. use recursive ray tracing while Behrens and Ratering add shadows to texture-based volume renderings. Kniss et al. [4] also add other effects such as phase functions and forward scattering. A comparison of several shadowing algorithms for volume rendering can be found in a recent work by Ropinski et al. [5] where shadow rays, shadow maps and deep shadow maps are compared, with a special focus on semitransparent objects.

Following the same idea of increasing the realism of the shading model, Stewart developed a technique named vicinity shading that enhances depth perception through the use of obscurances [6]. In essence, he proposed a model that performs shading according to the presence of occluders on the neighborhood of a voxel. Obscurances (also coined as ambient occlusion by Landis [7] and others) is a model of ambient illumination that produces realistic shading at relative low cost. Ambient occlusion-based techniques darken pixels in proportion to the total solid angle occluded by nearby elements. Our technique allows for a fast computation of an approximation to vicinity shading without the need of tracing extra rays into the volume. A similar lighting technique has also been applied by Tarini et al. [8] to enhance molecular visualization, by exploiting the fact that the rendered geometry is built upon atoms, which have a particular spherical shape. Ambient occlusion techniques have become popular thanks to their simplicity and efficiency. Although most papers compute ambient occlusion information in a preprocess [8–11], more recently some estimate the occlusion based on depth maps [12,13], or as an approximation to global illumination [14].

Ruiz et al. [15] implement the obscurances method for volume models. In order to determine the ambient occlusion in a point, several rays are traced sampling the surrounding of that point. This leads to large precomputation times of several minutes even for relatively small models (up to $256 \times 256 \times 415$ slices).

Ropinski et al. [16] have developed an approach for fast ambient occlusion-based rendering of volume models. Their method uses a precomputed set of local histograms (similar to [17]). However, this set may become too big for large datasets. On the other hand, like our approach, they generate ambient occlusion images on the fly, and perform fast updates when transfer function changes. This system also computes color bleeding and volume glows with the same data structure. In a recent work, Schott et al. [18] have presented a fast method for occlusion shading. They render the volume using slicing. They use two buffers: eye buffer and occlusion buffer. Each buffer is
computed incrementally. When a slice is rendered, the occlusion is computed by reading the occlusion buffer, and its value is used to generate the new color for the eye buffer. This color is used to update the occlusion buffer. We further discuss the similarities with our approach in Section 6.

Non-photorealistic techniques: Expressive visualization is a set of techniques that provide visual cues on features of the model in order to achieve a wide range of effects such as artistic renderings, or attention focus. Various strategies to emphasize different parts of the model may be used to enhance depth perception. Contours are generated based on the magnitude of local gradients and the angle between the viewing direction and the gradient by Csebfalvi et al. [19]. Hauser et al. [20] propose two-level volume rendering that is able to use different rendering paradigms (e.g., direct volume rendering or maximum intensity projection) for the different structures within a dataset, and show how to use it as a focus + context technique. Nagy et al. [21] combine line drawings and direct volume rendering techniques. Yuan and Chen [22] present a method for volume rendering images using silhouettes, ridge and valley lines. Bruckner and Gröller [23] use volumetric halos for enhancing and highlighting structures of interest using a GPU-based direct volume rendering. Halos have also been used by Wenger et al. [24], Ebert and Rheingans [25], Svakhine and Ebert [26], or Ritter et al. [27], and Tarini et al. [8]. These methods usually calculate halos on a preprocess and therefore they may not easily be modified on the fly. The halos built by Bruckner and Gröller are similar to ours in that they may flexibly change size and color, although their approach also supports semi-transparent objects (see Section 6 for a deeper discussion). However, their construction process involves several steps while our halos are built in a single step posterior to the rendering, using the vicinity occlusion map. This yields a constant cost per frame (which depends on the window size), often smaller than their method. A recent approach that also uses depth information for halo rendering in line models is due to Everts [28]. This method also requires data preprocess and renders the halos as triangle strips, instead of image post-processing.

In all cases, we render the volume models using GPU-based ray casting [29], which yields real time results. In short, this algorithm renders the bounding box of the volume, and each rasterized fragment generates a ray that traverses the volume and calculates the shading by querying the 3D texture that stores the density values of the model.

3. Preliminaries

In this section we review two concepts used in the remaining of the paper: Summed Area Tables and ambient occlusion.

3.1. Summed Area Tables

Given a table t, its Summed Area Table [30] (SAT) is a structure that stores, for each cell (x,y), the sum of the elements of t from the initial position (0,0) until (x,y) included:

$$\text{SAT}[x,y] = \sum_{i=0}^{x} \sum_{j=0}^{y} t_{ij}$$ (1)

In Computer vision community, this data structure is also called integral image [31]. Once the SAT is built, it provides a way to filter arbitrarily large rectangular regions of an image in a constant amount of time. In order to compute the sum of the values of a rectangular region of t, only four accesses are required (see Fig. 2), independently of the size of the area. This allows us to compute the approximation of occlusion due to neighbor regions in a constant time, as we will see later.

Summed Area Tables may be built incrementally on a CPU by a simple algorithm as depicted in Fig. 3. Therefore, its construction cost is linear with the number of cells in the original table.

3.2. Ambient occlusion in volumetric models

Ambient occlusion [7] for volumetric models was first introduced by Stewart with the name of vicinity shading [32]. This technique was tailored to increase depth perception in volumetric models. The original idea is due to Zhukov et al. [6], who developed a fast estimation of global illumination dubbed obscurances. For a surface point, the algorithm attenuates the illumination coming from all directions in space by measuring the occlusions in the local vicinity of the point. In order to do this, the occlusion is computed by sampling the environment of the voxel to be shaded with more than 1000 sample directions. Although the developed algorithm traces rays in an efficient way, sampling such a high number of directions results in an important impact in rendering time. Recent papers deal with the generation of ambient occlusion based shadows in real-time [16,8]. Unfortunately, most of them require either a set of precomputed structures, or need knowledge of the 3D geometry. Our approach needs no precomputation and does not require knowledge of the geometry, which is often not available in volume rendering. Note that in volume rendering, the shading depends on the transfer function because this function determines the occlusions, and it can be changed interactively.

Given a point p on the surface of a model with surface normal $n_p$, the irradiance reaching p can be defined as

$$E(p) = \int_{\Omega} n_p \cdot \omega L(\omega) \, d\omega$$ (2)

where $L(\omega)$ is a scalar with magnitude equal to the radiance coming from direction $\omega$, and $\Omega$ is the set of directions above the surface, i.e., the direction for which $n_p \cdot \omega > 0$. This can be
approximately evaluated by discretizing the domain \( \Omega \) into \( k \) sectors \( \Omega_i \) with a possible uniform solid angle measure \( \sigma_i \), and, for each sector, evaluating the radiance \( L \) only for a sample direction \( \omega_i \):

\[
E(p) = \sum_{i=1}^{k} n_p \cdot \sigma_i L(\omega_i) / \sigma_i
\]  

(3)

If we consider only an environment where light comes uniformly from each direction, we may even get a simpler formulation, as the radiance reaching a point may be substituted by a binary function that determines the reachability of that point \( (\mathbb{O}(\omega)) \), that is, \( \mathbb{O}(\omega) \) evaluates to 1 if the point light is not occluded in a certain direction. This results in an approximation of the rendering equation:

\[
E(p) = \frac{1}{4\pi} \sum_{i=1}^{k} n_p \cdot \sigma_i \mathbb{O}(\omega_i)
\]  

(4)

4. Ambient occlusion and halo rendering using vicinity occlusion maps

Volumetric models are rendered using ray casting on GPUs [29], which yields real time rendering. Our method consists on generating depictions that increase the insights on the models being visualized. These reduce ambiguity through the addition of depth cues by simulating Stewart’s vicinity shading and rendering halos. For the fast generation of ambient occlusion and halos, we have developed a new data structure dubbed vicinity occlusion map [1].

4.1. VOM data structure

We first focus on simulating ambient occlusion. This results in a new data structure that can also be used to render halos in real time.

The main idea of ambient occlusion is to measure, for a certain point, the amount of ambient light occluded by its neighboring voxels. We reduce the lighting contribution at the point to be shaded by an amount proportional to the occlusions produced by surrounding structures. Given a rendered image of a volumetric object and its associated depth map, we may ambient occlusion this value by analyzing, for a given pixel, the depth values of its closer surrounding pixels, and counting the number of pixels whose depth is smaller (they are closer to the viewer) than the one to shade. Naively counting the number of elements in the depth map would result in a too large number of texture queries, thus reducing the framerate. The solution developed by Crytek [12] consists on subsampling this region using a specific pattern. Such sampling result in visual noise artifacts that are reduced by filtering. We adopt a completely different approach: we compute an average depth using a new data structure: the VOM.

A vicinity occlusion map (VOM) is a data structure that encodes the information contained in a depth map in two tables, one containing the accumulated depth of the neighborhood of a pixel, and another one that contains the number of values that contributed to the sum. We build VOMs from the depth map resulting from ray casting the volumetric model from a viewpoint. The depth map contains, for each fragment, the distance to the observer at the moment the opacity of the ray gets to one. Our data structure is built by processing the depth map and generating its Summed Area Table together with information concerning the number of fragments that effectively contributed to the sum. Therefore, VOMs consist of two tables, dubbed SATdepth (Summed Area Table of the depth map) and SATNdepth (Summed Area Table of a bitmap containing 1 for pixels not belonging to background and 0 otherwise), respectively.

4.2. System overview

In this Section we present the architecture of our framework (see Fig. 4).

Initially, we load the volumetric model on CPU and build a 3D texture from it, that is loaded to GPU texture memory. Then, given a transfer function and a viewpoint, our system performs two
rendering passes. The first consists in a GPU-based ray casting of the model that generates a color map and a depth map. This depth map is passed to the CPU in order to build the vicinity occlusion map (VOM). Then, the VOM is uploaded to GPU and the second rendering pass performs the shading by simply rasterizing a screen-aligned quad, and using the VOM to produce the final result. Hence, only one Ray Casting pass is required for the whole process. Although the VOM could also be computed on the GPU, this requires multiple rendering passes [33]. Being the volume ray casting a fragment shader intensive algorithm, our initial approach was to download the work to the CPU. Despite that, we have also implemented a CUDA version which yields similar results.

4.3. Ambient occlusion using VOMs

We want to measure, for a certain point, the amount of ambient light occluded by its neighboring structures. We approximate occlusion using the VOM to average depth surrounding the point of interest using the following formula:

$$\text{avgdepth} = \frac{\sum_{x \text{, size}_x \text{, y} \text{, size}_y} \text{depth}(i,j) }{\text{frag} \text{size}}$$

(5)

where size\(_x\) and size\(_y\) is the resolution of the 2D region of the depth map we use to compute ambient occlusion. The average depth can be computed using only SAT\(_{\text{depth}}\). However, although this might yield acceptable results, taking into account the number of pixels that contributed to the average depth produces a more precise value. This is achieved using SAT\(_{\text{N}}\). Finally, we use the difference between the average depth and the current depth of the point to shade in order to obtain a better approximation of the percentage of hemisphere covered by the close voxels. From this value, we infer the amount of occlusion produced by opaque voxels around a point in an efficient way. This may be interpreted as approximating Eq. (4) by only taking one layer of occluders (the depth map), lying in the viewing direction.

We consider regions that go from 10 \(\times\) 10 to 25 \(\times\) 25 pixels around the point to be shaded (which means taking into account 100–625 pixels) to create plausible shading and halos, as we will show with examples. From this simple idea we may build a darkening function that, added to the Phong shading function, will generate depth cues that provide a better understanding of the rendered model. The function is implemented in a shader and executes for all fragments whose depth is different from 1.0, that is, for the fragments belonging to the final color of the object. It is modified by subtracting from the color generated by the ray tracing process \((R\text{\_color})\) a certain amount \(\text{dark}\) of the vicinity color, so the fragment color will be

$$\text{F\_color} = R\text{\_color} - \text{dark} \cdot \text{vicinity\_color}$$

(6)

where \(\text{dark}\) is the occlusion factor due to vicinity shading that is computed taking into account the average depth of the surrounding pixels. If we use white as \(\text{vicinity\_color}\), we may darken the fragment, while if we use a color out of the gray range, we may color the selected structure in order to emphasize it. The occlusion factor is computed by taking into account the number of pixels that contributed to the depth calculation, and the resolution of the used region. So for a fragment \((x,y)\) we will use:

$$\text{occ\_vicinity} = (\text{avgdepth} - \text{depth}(x,y)) \cdot \text{vicinity\_color}$$

(7)

where \(\text{vicinity\_Factor}\) is a term that modulates the influence of the vicinity shading in the final image. Once the \(\text{occlusion\_factor}\) is computed, we add it to the final color by filtering its weight in a smooth way:

$$\text{dark\_vicinity} = \text{smoothstep}(0,0.99,\text{occ\_vicinity})$$

(8)

This operation is similar to the depth darkening shown by Luft et al. for surface-based geometry and images with depth [34]. In our case, we compute the difference with the average depth, instead of a gaussian filter of the depth map. Moreover, we only take into account non-background pixels, therefore, halos using this value will not decay as fast as the depth darkening for sharp angles.

In Fig. 5 we show a comparison of renderings without (left) and with (right) vicinity shading. Increasing the \(\text{vicinity\_Factor}\) reduces overall lighting and increases depth cues. If we want to increase the contrast in areas of the image enhanced by vicinity

---

**Fig. 4.** Application architecture: The CPU reads a volumetric model from a DICOM file, builds a 3D texture from it, and uploads it to the GPU. Then, we perform a GPU ray casting on the model which generates a color map and a depth map. The depth map is passed to the CPU, which is used to generate the vicinity occlusion map (VOM). With the VOM and the color map, the GPU builds the final image in a composting process.
factor, we replace this factor with a power of the difference in depths.

Summed Area Tables may become very large in terms of bit depth. The number of bits required to store each component in a SAT of resolution \( w \times h \) is

\[
N = \log_2 w + \log_2 h + P_i
\]

where \( P_i \) is the precision required by the input. This makes the management of SAT quite difficult. As noted by Hensley [33], this may overflow the texture resolution if not handled accurately. In our case, we perform the SAT computation in long integer format and then transfer the result into a pair of 32 bits textures to the GPU (one for the depth sum and another for the accumulated number of elements different from 0).

The \( \text{avg}_\text{depth} \) computation requires four accesses for the depth and four more for the number of pixels that contributed to the depth map, as shown in Fig. 2. We might code both tables into a single 64 bit texture and therefore save four texture accesses. However, most of these optimizations have little impact, because the major cost incurred by our algorithm is data transfer, as it will be analyzed later.

4.4. Halo rendering

In order to render halos on selected parts of the model, we perform a similar algorithm to the one used to simulate vicinity shading. In this case, we have to take into account that the halo has to be rendered around the object of interest and has to decay as we go away from the object.

The user may decide the halo extension (in pixels) by manipulating a slider. Moreover, the user can also change the halo color interactively. In order to render a halo, for each pixel, we query the VOM to calculate the average depth of the neighborhood, and, if the current fragment is outside the object we query the VOM to calculate the average depth of the halo color interactively. In order to render a halo, for each pixel, we manipulate a slider. Moreover, the user can also change the halo color interactively. In order to render a halo, for each pixel, we query the VOM to calculate the average depth of the neighborhood, and, if the current fragment is outside the object we query the VOM to calculate the average depth of the halo color interactively.

The \( \text{avg}_\text{depth} \) computation requires four accesses for the depth and four more for the number of pixels that contributed to the depth map, as shown in Fig. 2. We might code both tables into a single 64 bit texture and therefore save four texture accesses. However, most of these optimizations have little impact, because the major cost incurred by our algorithm is data transfer, as it will be analyzed later.

4.4. Halo rendering

In order to render halos on selected parts of the model, we perform a similar algorithm to the one used to simulate vicinity shading. In this case, we have to take into account that the halo has to be rendered around the object of interest and has to decay as we go away from the object.

The user may decide the halo extension (in pixels) by manipulating a slider. Moreover, the user can also change the halo color interactively. In order to render a halo, for each pixel, we query the VOM to calculate the average depth of the neighborhood, and, if the current fragment is outside the object we query the VOM to calculate the average depth of the halo color interactively. In order to render a halo, for each pixel, we query the VOM to calculate the average depth of the neighborhood, and, if the current fragment is outside the object we query the VOM to calculate the average depth of the halo color interactively.

The \( \text{avg}_\text{depth} \) computation requires four accesses for the depth and four more for the number of pixels that contributed to the depth map, as shown in Fig. 2. We might code both tables into a single 64 bit texture and therefore save four texture accesses. However, most of these optimizations have little impact, because the major cost incurred by our algorithm is data transfer, as it will be analyzed later.

\[
F_{\text{color}} = R_{\text{color}} + d_{\text{dark}} \cdot \text{halo}_{\text{color}}
\]

where \( R_{\text{color}} \) is the color generated by the Ray Casting algorithm, and the amount of darkening \( d_{\text{dark}} \) due to the halo is determined by

\[
d_{\text{dark}} = \frac{\text{avg}_\text{depth} - \text{depth}(x,y) \cdot \text{halo}_{\text{weight}} \cdot \text{Num}_\text{elems}}{\text{resolution}}
\]

where \( \text{halo}_{\text{weight}} \) is a weight that can be changed by the user, \( \text{Num}_\text{elems} \) is the number of elements that contributed to the average depth calculation, and \( \text{resolution} \) is the size of the region of interest. Dividing the number of fragments whose depth is not zero by the resolution of the search region, we create a function that decays with the distance to the object.

The result is a halo painted with the color selected by the user and whose intensity and area of influence may be changed interactively. All these changes are possible because the number of texture accesses required to perform the effect is constant. A couple of examples of halos generated using this technique are shown Fig. 6.

4.5. Features

Our algorithm performs a real-time computation of both vicinity shading and halos. Since all the calculations are carried out from scratch each frame, the parameters involved can be changed interactively without cost. We thus provide the user control of several parameters in order to improve the depth perception or structure accentuation:

- **Vicinity weight and size**: Depth complexity among models or regions of a model may vary, and therefore, applying a fixed function may enlighten a certain part and hide another. In our application, the user may manipulate a slider to change the default value given to the darkening function. We may see the effect of such changes in Fig. 7. The number of pixels taken into account can also be interactively changed in order to fine tune the results if necessary. Without extra cost, we let the user change the emphasizing color, allowing to change from darkening to enlightening if the depth variations need to be accentuated in a different way.

- **Halo size and color**: Thanks to the use of Summed Area Tables, different sizes of halos require the same number of texture accesses. Therefore, we let the user change the halo size, again by using a slider. Nonetheless, the color can also be defined interactively because different structures or different contexts might require a different color to emphasize the object of interest. In Fig. 8 we compare two different colors: black and red, for halo-based shape emphasis.

- **Structure selection**: Our application allows interactive selection of structures from a set of previously segmented structures,
and the shading effects are restricted to them. This allows us to create halos or colored shading for specified parts of the model, in order to provide better visual insights on the structures of interest. Fig. 9 illustrates this effect, separating a vascular tree in the liver from the surrounding structures.

From the effects presented above, the only one requiring extra information is structure selection, where we need some extra information that identifies the structures. This can be implemented using texture of identifiers, built in a preprocess. The rendering requires some changes: When the color image and depth map are generated, the ray caster identifies, for each rendered fragment whose opacity is 1.0, the structure it belongs to with an extra fetch to the 3D texture of identifiers. This then produces as a result a modified depth map, that only stores depth values for the structure of interest. However, we store in the alpha channel of the color map, the depth reached by the ray (even if it did not intersect the structure of interest). This way, we may compare the depth map of the selected structure and the depth of the color buffer. Therefore, when evaluating if a pixel has to be shaded with a halo color, we can test if there is an occluding object in front of it and avoid painting halos there. We show this effect in Fig. 9. Note that some ribbons in front of the vessels we want to highlight do correctly occlude the halo. With structure selection, we may also render a halo around a semi-transparent structure, or around a selected structure that lies within a semi-transparent volume.

We may also simulate a 3D halo by analyzing the generated depth map. We add halos, not only around the object, but also at the pixels of the image where the depth map generated by the selected object has a discontinuity. The result can be seen in Fig. 10. Adding simulated 3D halos around a pollen grain. Left image shows the 2D halos method, while right image adds (in yellow) the simulated 3D halos generated by analyzing depth discontinuities. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.6. VOM construction using CUDA

We have also implemented a CUDA-based version of the SAT construction. The algorithm builds the Summed Area Tables in a separate way: first, columns are summed, and then, the Summed Area Table is computed by analyzing the previously generated array in rows. Although the construction times are fast, unfortunately, the connection between the regular graphics pipeline and the CUDA pipeline is still not properly accelerated. Thus, we still require downloading the depth map to CPU before it may be uploaded to CUDA memory. We compare the data management of the two versions of our algorithm in Fig. 11. Due to the download, that is the bottleneck of the algorithm, the framerates obtained are only similar to the framerates obtained with the CPU version. Future versions of CUDA are expected to allow sharing texture memory between CUDA and OpenGL. This would save bandwidth and time. Therefore, we believe the CUDA version of our
algorithm may improve framerates over the CPU version. A more GPU-friendly approach, though requiring multiple rendering passes was developed by Hensley [33]. However, we believe that a CUDA-based approach will be more flexible, for instance if dealing with non-power-of-two images.

As shown, the pipeline necessary to interleave OpenGL and CUDA computations is not currently the most efficient. However, we may measure the amount of time required to compute a Summed Area Table using CUDA and using the CPU. This demonstrates that the CUDA approach is certainly more efficient, and with future versions of the driver, it may improve the framerates. The time needed to build the vicinity occlusion map only depends on the resolution of the image, and it is shown in Table 1.

### 4.7. Results

We have implemented the proposed method and have tested the results in a PC equipped with an Intel Core 2 Duo CPU running at 3.0 GHz. It also has 4 GB of RAM memory and a nVidia GeForce GTX 280 GPU with 1 GB of RAM. We have analyzed models with different sizes. The dimensions of the models are shown in the following list:

- Brain: $128 \times 128 \times 58$ slices.
- Ear (Baby): $256 \times 256 \times 98$ slices.

With this machine, we may obtain real-time framerates for fairly complex models (of up to $512 \times 512 \times 600$ slices). The penalty suffered by our shading approach is low compared to the cost of ray tracing the scene. All the examples of the paper were rendered in a Viewport of $512 \times 512$ resolution and the ray casting samples three times per voxel, as lower density sampling usually results in poor renderings. The quality of the images is really high and the influence of vicinity shading and halo sizes may be changed quite easily by the user. The ray casting process dominates the overall cost, and this mainly depends on the transfer function used. That is the reason why we obtain similar framerates for models of different number of slices. As we can see in Figs. 5 and 6, we have used models that go from high opacity such as the bonsai, or with large semi-transparent regions, such as the body.

Concerning efficiency, the main advantage of our approach is that its cost is constant, because the number of texture accesses is constant independent of the size of the model. However, it is linear with the number of pixels of the viewport.

Table 2 shows the results obtained with our method. We may see that the exploration can be done in real time. If a better framerate is required, the size of VOM may be reduced (in order to reduce the transmission cost) either by quantizing the results or by building a mip map, however, this last requires fine tuning in the shader in order to achieve comparable results.

As already commented, the CUDA version obtains similar results due to the extra data transfer required. The results are shown in Table 3.

Although VOMs are suitable for increasing depth information of isosurfaces detected by volume ray casting, they may also be used for semi-transparent objects in a similar way. The only key point is the depth map creation, that must be generated accordingly to the information of the structure we want to analyze, so we will require the proper identification of the structures of interest.

On the other hand, objects with a lot of disconnected structures, or structures that are partially isolated in 3D space may result in over darkened regions with this algorithm. This is due to the fact that the occlusion function is built on a 2D basis, and not taking into account 3D geometry. In most cases, this will probably not be a visual problem (such as in Fig. 1 where frontmost ribs darken the ones in the bottom). However, in some cases this might be noticeable and even yield unexpected result.

### Table 2

<table>
<thead>
<tr>
<th>Model</th>
<th>Viewport</th>
<th>Ray casting (fps)</th>
<th>VOM (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brain</td>
<td>256 x 256</td>
<td>100.19</td>
<td>100.11</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>100.18</td>
<td>58.04</td>
</tr>
<tr>
<td>Ear (Baby)</td>
<td>256 x 256</td>
<td>100.22</td>
<td>83.30</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>87.08</td>
<td>43.19</td>
</tr>
<tr>
<td>Engine</td>
<td>256 x 256</td>
<td>99.00</td>
<td>63.98</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>59.16</td>
<td>35.41</td>
</tr>
<tr>
<td>Intestines</td>
<td>256 x 256</td>
<td>36.12</td>
<td>30.89</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>22.97</td>
<td>18.46</td>
</tr>
<tr>
<td>Bonsai</td>
<td>256 x 256</td>
<td>39.70</td>
<td>33.35</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>24.18</td>
<td>19.17</td>
</tr>
<tr>
<td>Body</td>
<td>256 x 256</td>
<td>10.70</td>
<td>10.15</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>6.46</td>
<td>5.87</td>
</tr>
</tbody>
</table>

Note how the impact of adding vicinity shading and halos is relatively low compared to the complete rendering process.
as in the intestines picture in Fig. 12. In this image, the vicinity shows that some regions (far from the user) are slightly over darkened.

![Image](image-url)

**Table 3**

Comparison of framerates between our original algorithm (VOM) and the CUDA-accelerated version of the Summed Area Table computation (VOM\(_{\text{cuda}}\)).

<table>
<thead>
<tr>
<th>Model</th>
<th>Viewport</th>
<th>VOM (fps)</th>
<th>VOM(_{\text{cuda}}) (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brain</td>
<td>256 x 256</td>
<td>100.14</td>
<td>100.11</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>60.11</td>
<td>58.04</td>
</tr>
<tr>
<td>Ear (Baby)</td>
<td>256 x 256</td>
<td>98.81</td>
<td>83.30</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>46.45</td>
<td>43.19</td>
</tr>
<tr>
<td>Engine</td>
<td>256 x 256</td>
<td>85.72</td>
<td>63.98</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>37.48</td>
<td>35.41</td>
</tr>
<tr>
<td>Intestines</td>
<td>256 x 256</td>
<td>34.90</td>
<td>35.82</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>18.80</td>
<td>18.46</td>
</tr>
<tr>
<td>Bonsai</td>
<td>256 x 256</td>
<td>38.01</td>
<td>33.35</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>19.68</td>
<td>19.17</td>
</tr>
<tr>
<td>Body</td>
<td>256 x 256</td>
<td>21.19</td>
<td>19.88</td>
</tr>
<tr>
<td></td>
<td>512 x 512</td>
<td>11.00</td>
<td>10.79</td>
</tr>
</tbody>
</table>

Fig. 12. Close up of a vicinity occlusion rendering. Increasing the vicinity factor shows that some regions (far from the user) are slightly over darkened.

as in the intestines picture in Fig. 12. In this image, the vicinity factor is raised a little bit in order to make the darkened regions more obvious. Note that some regions are too dark because the geometry acting as occluder is close to the viewer, and hence there is a large depth jump in the depth map, while the actual occlusion in 3D might be lower than the one estimated, as the occluding geometry does not span for a large 3D region in space. This is not a complete disadvantage in the sense that it still helps judging the 3D arrangement of structures, but a better approximation might yield a more qualitative result.

In order to improve the shading quality, the information of the 3D occlusion should be taken into account. This is the objective of the DSAT3D data structure and rendering algorithm, presented in the following section.

### 5. Ambient occlusion using Density Summed Area Tables

So far, we have presented a data structure tailored to simulate ambient occlusion effects in real-time for volumetric models. As stated, VOMs take into account 2D information captured from the depth map. This information is view-dependent. Thus, it does not take into account the volume around the point to shade, but only the 2D projection of the dataset on the view direction. The practical consequence is that non-opaque regions do not contribute to the shading because the depth map is built at the distance the intersection ray reaches total opacity. A more realistic approach should take into account this information. Ruiz et al. [15] explore the surrounding of a point by tracing rays in all directions. Nevertheless, we want to avoid tracing so many rays, because it results in an important performance penalty.

#### 5.1. DSAT3D data structure

As stated, we will build a 3D version of a Summed Area Table. 3D Summed Area Tables have been used previously for importance information storage [35]. In our case, we encode the density values of the dataset. The construction of a 3D Summed Area Table is similar to the 2D version. It can also be done incrementally, as shown in Fig. 13.

In order to compute the average density of a region with the DSAT3D we need eight texture accesses (as compared with the four queries required by the 2D SAT), as shown in Fig. 14. The average density on a cubic region is therefore computed as

\[ D = (V_1 - V_2 - V_5 + V_7 + V_6 + V_8)/N_{\text{vox}}, \]

where \(N_{\text{vox}}\) is the number of voxels in the analyzed region.

#### 5.2. System overview

Our framework builds a 3D Summed Area Table of the density values of the volume, once the model is loaded. Then, this structure and the 3D volume are passed to the GPU. As a result, the rendering algorithm will be able to approximate for each point, the occlusion caused by close structures, by analyzing the opacity of the neighboring voxels in a single ray casting pass. The pipeline of this algorithm is depicted in Fig. 15.

In contrast with the previous system, the DSAT3D may be built as a preprocess and used during the rendering. As a consequence, we will obtain higher frame rates because there is no need to further transfer information from the GPU to the CPU and backwards. Moreover, our approach is valid even for transfer function changes, because the occlusion is computed by evaluating the transfer function on the density values, that remain constant.

#### 5.3. Ambient occlusion using DSAT3D

In order to simulate ambient occlusion at a point, we evaluate the opacity of the voxels that surround the point. We could sample the density of each neighboring voxel, determine its opacity by applying the transfer function, and reduce illumination according to it. However, like in the 2D version, this would yield too many texture reads. Instead, what we do is evaluating the average density values of the region by using the DSAT3D. The naive approach would consist in evaluating the average density, \(D\), of a cubic region around the point of interest. However, taking \(D\) as the average density and applying the transfer function in order to estimate occlusion will introduce an error that might become excessive if large variations of the density values happen within. This is especially true if the average takes into account a large number of voxels. With little more effort, we may evaluate occlusion as a sum of the occlusion caused by the eight cubic regions around the point of interest, as depicted in Fig. 16. This yields a much better approximation for close structures. Note that this works not only as a better approximation for the average density, but the occlusion is computed with respect to the point of interest eight times. The occlusion value may be very different from the occlusion if we had averaged the density.

Thus, we approximate Eq. (4) by evaluating the occlusion the following way:

\[ \text{Occlusion} = \sqrt[8]{\sum_{i=1}^{8} \text{Opacity}(D_i)/8} \]

where \(D_i\) are the average densities depicted in Fig. 14, and \(\text{Opacity}\) function evaluates the opacity by applying the Transfer Function to the given density value. We introduced a square
In case we used a large query area, the use of octants instead of the overall average would not be precise enough. However, for the ranges we use, that go from 10 to 20 texels far from the point to be shaded, the effects are a good approximation. If more precision was required, we could use a multiresolution approach, a set of close octants within a closer range to the point of interest and a higher weight, and another one with larger area and lower weight.

The darkening value is computed applying the following function:

\[
dark_v = \text{smooth step} (\text{minimum Dark}, 0.99, \text{Occlusion})
\]  

This helps the user to fine tune the amount of darkening desired. Finally, the resulting color is computed by evaluating the same function as with the previous approach:

\[
F_{\text{color}} = R_{\text{color}} \cdot \text{dark}_v \cdot \text{vicinity}_{\text{color}}
\]  

### 5.4. DSAT3D construction using CUDA

We have also implemented a CUDA-based algorithm in order to improve construction timings. As shown in Table 7, the timings are better for the CUDA version as the model sizes increase. CUDA timings are very good for large models, as they are substantially smaller than the time required to load big volume models. Like in the previous case, the bottleneck is the data communication bus,
because the resulting information cannot remain as texture memory and has to be downloaded again to CPU, as depicted in Fig. 17. This is something expected to be solved as soon as the Frame Buffer Objects are incorporated in CUDA.

For the GPU implementation, the SAT3D computation algorithm had to be changed. Indeed, porting naively the CPU algorithm to GPU leads to a memory pattern that did not fit the GPU. Even though the input was read a few times, and locally, it did not meet the memory pattern requirements to use maximal performance of the GPU. We thus implemented a new algorithm based on the following observation: the SAT3D computation can be separated.

\[
SX(i,j,k) = \sum_{x \leq i} in(x,j,k) \\
SY(i,j,k) = \sum_{y \leq j} SX(i,y,k) \\
SAT(i,j,k) = SZ(i,j,k) \\
= \sum_{z \leq k} SY(i,j,z) \\
= \sum_{z \leq k} \sum_{y \leq j} SX(i,y,z) \\
= \sum_{z \leq k} \sum_{y \leq j} \sum_{x \leq i} in(x,j,z)
\]

The algorithm is now changed to the following:

1. Compute SX in a first pass, Z-slice after Z-slice, by loading a line in shared memory, perform a parallel computation (see [36]) for the line, and write back the line.

2. Compute SY in a single pass reading memory with a nice GPU pattern—coalesced reads.

3. Compute SZ in a single pass reading memory with a nice GPU pattern—coalesced reads.

This new algorithm leads to a significant performance improvement:

1. CPU algorithm on GPU (naive port)—3DSAT computation w/o memory transfers—512 × 512 × 512: 3.997 s on a Tesla M1060.
2. GPU-friendly algorithm—3DSAT computation w/o memory transfers—$512 \times 512 \times 512$: 0.191 s on a Tesla M1060.

This performance gap is explained by the memory pattern of the algorithm. The first naive algorithm uses a scattered memory pattern in reads (reading memory that has just been written), when the second algorithm reads data in a coalesced manner which is the GPU friendly way thus yielding much higher memory bandwidth.

5.5. Results

The presented approach can be used to compute ambient occlusion in a fast way, at the cost of a large data structure. The results are similar to the ones obtained with the 2D version for models that do not present isolated disconnected features. We may see an example in Fig. 18, where the ear of the Baby model is shaded using VOM (left) and DSAT3D (right).

We can see a couple more examples in Fig. 19. Left column shows the original ray casting result and the right one shows the ambient occlusion enhanced rendering.

Like the previous results, the framerates correspond to a PC equipped with an Intel Core 2 Duo CPU running at 3.0 GHz. It also has 4 GB of RAM memory and a NVidia GeForce GTX 280 GPU with 1 GB of RAM. The raycasting analyzes three samples per voxel in order to guarantee a high quality result. All images were rendered using a $512 \times 512$ viewport. Results are presented in Table 4.

We may see that the time impact is lower for the DSAT3D than for the vicinity occlusion maps. The main reason is that texture queries to determine occlusion are only carried out when reaching the isosurface to shade, not along the whole ray traversal, which would result in an important rendering penalty. This, together with the fact that the 3D texture has been precomputed, makes this approach faster. An analysis of the impact is presented in Table 5 where we show the values for the $512 \times 512$ viewport. Note how we may deduce that the cost of the 2D version is roughly constant per frame if we compare models that render at similar speeds and have similar resolutions, such as the Intestines and the Bonsai models. The cost of implementing a DSAT3D occlusion is lower, although, as already stated, it requires more memory.

The new data structure, DSAT3D, stores a sum of the density values. This means that the storage required to hold all the values is higher than the original 3D model. This is a typical tradeoff of space versus speed, but might become a problem if the amount of memory is limited. For the models we tested, the amount of extra storage required for having a DSAT3D structure is high, although it may compete with other approaches that store more elaborated information such as the method by Ropinski et al. [16]. In their case, the time and storage requirements for large models (such as the ones presented here) may be prohibitive. In our case, we usually need a 32 bit texture with the same dimensions as the input dataset (see Table 6). This more than doubles memory requirements, but we did not need to compress this structure.

As shown in Table 7, the timings are specially good for large models when using CUDA.

<table>
<thead>
<tr>
<th>Model</th>
<th>Viewport</th>
<th>Ray casting</th>
<th>DSAT3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brain</td>
<td>256 × 256</td>
<td>100.19</td>
<td>100.18</td>
</tr>
<tr>
<td></td>
<td>512 × 512</td>
<td>100.18</td>
<td>100.15</td>
</tr>
<tr>
<td>Ear (Baby)</td>
<td>256 × 256</td>
<td>100.22</td>
<td>100.20</td>
</tr>
<tr>
<td></td>
<td>512 × 512</td>
<td>87.08</td>
<td>80.09</td>
</tr>
<tr>
<td>Engine</td>
<td>256 × 256</td>
<td>99.00</td>
<td>94.63</td>
</tr>
<tr>
<td></td>
<td>512 × 512</td>
<td>59.16</td>
<td>58.17</td>
</tr>
<tr>
<td>Intestines</td>
<td>256 × 256</td>
<td>36.12</td>
<td>35.82</td>
</tr>
<tr>
<td></td>
<td>512 × 512</td>
<td>22.97</td>
<td>22.04</td>
</tr>
<tr>
<td>Bonsai</td>
<td>256 × 256</td>
<td>39.70</td>
<td>38.82</td>
</tr>
<tr>
<td></td>
<td>512 × 512</td>
<td>24.18</td>
<td>22.84</td>
</tr>
<tr>
<td>Body</td>
<td>256 × 256</td>
<td>10.70</td>
<td>10.40</td>
</tr>
<tr>
<td></td>
<td>512 × 512</td>
<td>6.46</td>
<td>6.25</td>
</tr>
</tbody>
</table>

Comparison of the impact incurred by using the both methods presented in this paper for the $512 \times 512$ viewport.

<table>
<thead>
<tr>
<th>Model</th>
<th>Ray casting</th>
<th>VOM (impact)</th>
<th>DSAT3D (impact)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brain</td>
<td>100.2</td>
<td>58.04 (−42.06%)</td>
<td>100.2 (−0.03%)</td>
</tr>
<tr>
<td>Ear (Baby)</td>
<td>87.08</td>
<td>43.19 (−50.39%)</td>
<td>80.09 (−8.02%)</td>
</tr>
<tr>
<td>Engine</td>
<td>59.16</td>
<td>35.41 (−40.14%)</td>
<td>58.17 (−1.67%)</td>
</tr>
<tr>
<td>Intestines</td>
<td>22.97</td>
<td>18.46 (−19.65%)</td>
<td>22.04 (−4.05%)</td>
</tr>
<tr>
<td>Bonsai</td>
<td>24.18</td>
<td>19.17 (−20.71%)</td>
<td>22.84 (−5.55%)</td>
</tr>
<tr>
<td>Body</td>
<td>11.98</td>
<td>10.79 (−9.88%)</td>
<td>11.82 (−1.35%)</td>
</tr>
</tbody>
</table>

Note how the impact of the DSAT3D system is really low, up to an 8% at most.
approaches for ambient occlusion and halo rendering.

6. Discussion

We begin our discussion by comparing VOMs with previous approaches for ambient occlusion and halo rendering.

As in Stewart’s approach [32], our shading algorithm reduces ambiguity by adding depth cues on the surface of rendered models. However, instead of tracing rays through the neighbor voxels, we analyze the depth information by querying the vicinity occlusion map. The efficiency of this approach makes changing lighting parameters interactive. Thus, the user may change how vicinity shading affects the overall rendering and, even the color of the shading. Changing shading color helps accentuating structures in a stylized way.

Ropinski et al. [16] also compute ambient occlusion in real time, but require preprocessing times of several hours and a huge amount of information storage which do not allow them to deal with large models. Compared to this approach, ours does not require preprocessing. Moreover, the amount of generated information is not so high, and therefore the models we can deal with are larger.

Our approach achieves similar results than the one of Schott et al. [18]. Their approach works for a slice-based renderer and ours for a ray casting, although it could be easily adapted to the slice-based rendering. In contrast to theirs, our approach does take into account only the neighborhood in image space, while their approach computes occlusion taking into account all the information in the viewing direction. On the other hand, our method is insensitive to the change of the kernel size, that can be interactively adjusted if necessary with no penalty.

Compared to Ruiz et al. [15], our approach achieves real-time frame-rates, because it takes advantage of the use of Summed Area Tables, that limit the number of texture access to a constant count, even for arbitrary sizes. Moreover, its cost is limited, as the work to be done per frame is constant. This has the advantage of guaranteeing interactive frame rates but, for small models (with high frame rates), the impact seems to be more important.

Compared to other surface depth-based ambient occlusion methods [12,13], we do not need to sample so exhaustively the depth map, and the numbers obtained from the Summed Area Tables already yield a filtered value of the depth map, thus not requiring further filtering for avoiding sampling artifacts when camera moves.

Our halo rendering method also takes advantage of VOMs in order to reduce the computation cost. As in Bruckner and Gröller [23], our halos are flexible because they are generated on the fly, and we may change their size and color interactively. They compute halos on a volumetric way, so 3D information is taken into account instead of 2D, which improves the quality of the halos. We may simulate some sort of 3D halos by analyzing depth discontinuities in the object of interest, as shown in Fig. 10. However, our halos are not truly 3D. On the other hand, in contrast to their approach, we generate the halos in a ray casting method by analyzing depth maps in a single (extra) pass. Thus, our approach has a low impact on rendering. It is only bounded by the size of the viewport, its cost is linear with the number of pixels, as it performs the computations on 2D views. Their approach also fully supports transparency. In our case, halos can be generated around transparent objects, but semitransparent objects do not occlude, not even partially, our generated halos.

Concerning the DSAT3D, it is an example of trading of space for speed. The required space is relatively high, but has the advantage of the impact in rendering, which is almost null. Compared to Ropinski et al. [16], we improve the preprocessing times, as ours is only a matter of seconds (in the case of a CPU processing) while theirs may take up to several hours. Furthermore, the CUDA implementation takes less than two seconds, which is lower than the time required to load a large volumetric model. In any case, we may deal with larger models. Notwithstanding, the memory footprint may be a limitation for large models on GPUs with small amount of memory.

Finally, the two algorithms presented here yield quite similar results for a large number of models. In most cases, deciding which to apply will consist in a tradeoff between memory and speed. For disconnected structures, the results may be slightly different, although the result will probably be acceptable in both

<table>
<thead>
<tr>
<th>Model</th>
<th>Sat3D generation (s)</th>
<th>Maximum value</th>
<th># bits required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brain</td>
<td>0.04</td>
<td>31,238,068</td>
<td>25</td>
</tr>
<tr>
<td>Ear (Baby)</td>
<td>0.74</td>
<td>198,720,120</td>
<td>28</td>
</tr>
<tr>
<td>Engine</td>
<td>2.07</td>
<td>187,475,047</td>
<td>28</td>
</tr>
<tr>
<td>Intestines</td>
<td>5.75</td>
<td>4,294,967,192</td>
<td>32</td>
</tr>
<tr>
<td>Bonsai</td>
<td>6.08</td>
<td>136,596,356</td>
<td>28</td>
</tr>
<tr>
<td>Body</td>
<td>17.1</td>
<td>4,294,967,162</td>
<td>32</td>
</tr>
</tbody>
</table>

Note that the DSAT3D construction using CUDA takes lower than the time required to load the model, which is an acceptable result. The CPU algorithm for large models is more time consuming.

Fig. 20. 2D vicinity occlusion shading versus 3D ambient occlusion using Summed Area Tables. Top row compares the 2D ambient occlusion and the 3D ambient occlusion algorithms implemented using Summed Area Tables. Middle row shows the depth map used to build the VOM data structure and the opacity map computed from the DSAT3D. Finally, we may see the original ray casting without depth enhancement and the use of both VOM and DSAT3D to improve ambient occlusion computation: (a) VOM; (b) DSAT3D; (c) depth VOM; (e) opacity DSAT3D; (f) original RC; and (g) VOM-DSAT3D.
cases. We show a comparison in Fig. 20 between the initial ray casting (f) and the results applying VOM and DSAT (a and b, respectively). DSAT3D does not shade if fine structures are isolated as in the hole in the middle of the model. However, it does take into account close geometry (b bottom left) even if there is no large depth change (compare to a) that may generate VOM shadows. In order to interpret where the shadows come from, we also provide the depth map generated for the VOM in c and the opacity map created through the DSAT3D ambient occlusion execution in d. In such cases, we also have tested how a combination of both effects look like, shown in image g.

Although combining VOM and DSAT3D requires an extra rendering step, the bottleneck is the VOM computation, and therefore the frame rates we obtain are the same ones as the VOM. This combination may be useful for models that combine sparse and dense regions.

7. Conclusions and future work

In this paper we have presented a fast method for ambiguity reduction in volume rendering. We achieve this through the use of two different techniques: vicinity shading and halo rendering. For ambient occlusion simulation, we have presented two different data structures: VOM and DSAT3D. Both of them take advantage of the use of Summed Area Tables to accelerate rendering.

VOMs are generated at each frame, and provide a fast method for screen-space ambient occlusion computation. The impact on rendering is constant per frame, and depends on the size of the viewport. With this data structure, we are also able to generate halos on the fly. Our approach is fast and highly configurable, and therefore, the user gains flexibility in the exploration process through an intuitive interface.

DSAT3D consists in a 3D Summed Area Table of the density values of the original model. With this structure, we can estimate ambient occlusion by analyzing 3D geometry around the point of interest. This makes the simulation to be view independent. This method is slightly superior to VOMs for objects consisting in discontinuous or separated structures. For other objects, the results are similar to VOMs but with a lower impact on rendering. The cost is an increase in memory consumption, which may be a problem in environments with limited GPU memory. As a consequence, one of the issues we want to focus in the future is the reduction of DSAT3D size.

Acknowledgments

This work has been supported by Project TIN2007-67982-C02-01 of the Spanish Government. The authors want to thank the anonymous reviewers for their valuable comments.

References