Ambient occlusion using cone tracing with scene voxelization

Eduardo Ceretta Dalla Favera
Tecgraf/PUC-Rio - Computer Science Department
Pontifical Catholic University of Rio de Janeiro, Brazil
Email: eduardo.ceretta@gmail.com

Waldemar Celes
Tecgraf/PUC-Rio - Computer Science Department
Pontifical Catholic University of Rio de Janeiro, Brazil
Email: celes@tecgraf.puc-rio.br

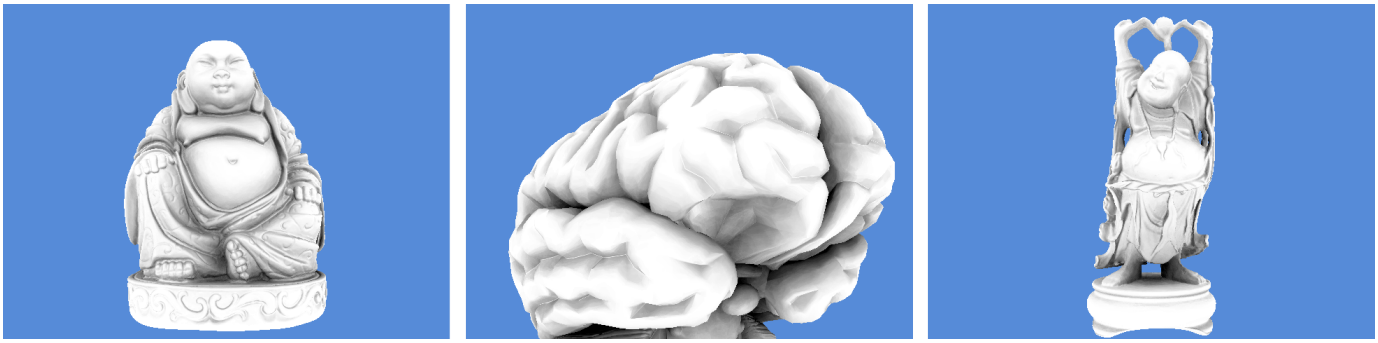


Fig. 1. Ambient occlusion rendering using our proposal: Buddha – 75.36 fps (left), Brain – 68.82 fps (middle), Happy – 106.04 fps (right).

***Abstract*—Ambient occlusion is a low-cost technique to simulate indirect ambient illumination in a realistic way. The goal is to estimate the amount of incident ambient light at each visible point. In this paper, we propose a novel ambient occlusion method that produces good quality results in real time. Using an efficient voxelization algorithm, we create a volumetric description of the scene geometry in a regular grid. During scene rendering, the hemisphere around each visible point is sampled by a set of cones, each one representing a package of rays. The volume of each cone is sampled by a series of spheres. The obstructed volumes of the spheres are used to estimate the amount of rays that are blocked by the scene geometry. The final ambient occlusion at each visible point is computed by considering all cones in the hemisphere. This approach has shown to be quite adequate: the intersection of each sphere with the voxelized scene is performed in a very efficient manner, and good quality results are achieved with a small number of cones. Computational experiments demonstrate the efficiency and effectiveness of our proposal.**

***Keywords*-Ambient occlusion; cone tracing; global ilumination**

I. Introduction

The pursuit of realism of virtual scenes is one of the fields of computer graphics in constant development. A good way to achieve this realism is through the correct modeling of lighting in a 3D environment. Global illumination comprises a set of techniques that intend to approximate, as closely as possible, the way light interacts with the objects in the scene. However, in practice, few applications can incorporate these techniques, because global illumination model is very complex and hardly there exist available computational resources for it to be done efficiently.

In order to achieve a global illumination aspect in real time, some techniques can be dismembered and applied individually. One example is the ambient occlusion technique, which produces a global illumination effect with relatively low computational cost. Differently from other global techniques, such as the computation of reflections and refractions, ambient occlusion requires only the launch of primary rays to estimate ambient illumination at each visible point, and only considers the scene geometry in the neighborhood of the point. Hence, the computation can be performed in real time. The resulting effect does enhance the tridimensional characteristics of objects in the scene, making the final rendered image more realistic.

When light directly hits a surface, this light is reflected in several directions, which contributes to the illumination of other objects; this creates the effect of indirect lighting: light reflected from all surround objects contributes to the illumination of a given point. The ambient illumination aims to represent the incidence of such reflected light. If a point is occluded by objects in its local vicinity, it will receive little reflected light. The ambient occlusion technique estimates the amount of light that is blocked by the surround objects.

To physically-correctly compute the amount of ambient light that reaches a point, one can trace a set of rays to sample the normal hemisphere of the corresponding surface point. The ambient occlusion factor, the amount of light blocked by the surround objects, is given by counting the number of rays that hit objects in the scene. Figure 2 illustrates this idea.

The ambient occlusion at a point P of normal $\hat{n}$ is calculated

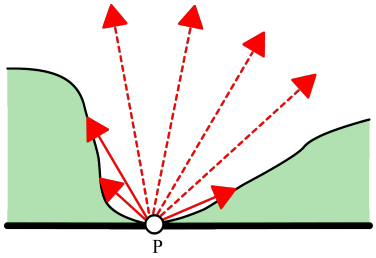


Fig. 2. Rays traced from P to diverse directions. The dashed rays did not hit the surround geometry.

using the following equation [1]:

$$A(P,\hat{n}) = \frac{1}{\pi}\int_{\Omega} V(P,\hat{\omega})(\hat{\omega}\cdot\hat{n})d\omega \qquad (1)$$

where $\hat{\omega}$ represents the directions of the hemisphere Ω, and $V(P,\hat{\omega})$ is the visibility function, which must be 1 if the direction $\hat{\omega}$ is obstructed by the geometry and 0 otherwise.

Several solutions have been proposed to incorporate the ambient occlusion effect in 3D scene illumination. Ray tracing results in very good image quality but presents a high computational cost. Approaches that estimate the occlusion in a pre-processing step generate good results efficiently, although do not support dynamic scenes [2]. Real-time methods with support for dynamic scenes, generally, utilize the information in screen space to sample the neighborhood of a pixel to approximate its occlusion [1], [3], [4]. However, due to limited available information and view dependency, the estimated results are not accurate and vary with camera position. These limitations are overcome by algorithms that compute the occlusion in object space. This however requires the use of an appropriate data structure to accelerate the access to geometry information. Despite acceleration techniques, object-space methods tends to perform worse than screen-space methods [5], [6].

This work proposes a novel object-space method that efficiently generates high-quality results. The proposed method uses a regular grid to efficiently access the geometry information of the scene, allowing a fast evaluation of ambient occlusion at each pixel. We employ the real-time algorithm proposed by Eisemann and Dcoret [7] to voxelize the scene in a binary regular grid. The ambient occlusion at each visible point is determined with the tracing of a set of cones distributed in the corresponding hemisphere. Each cone represents a package of rays. The volume of each cone is sampled by a sequence of spheres, which are used to efficiently estimate the percentage of obstructed volume and thus the amount of blocked rays. The effectiveness of our method is demonstrated by applying it to a set of different models. Figure 1 illustrates achieved results. Our main contribution relies on the appropriate and efficient use of spheres to compute obstructed volumes, which are translated to a good estimation of the amount of ambient light that is blocked.

The rest of this paper is organized as follows: Section II briefly exposes some related and inspiring works. The proposed method is explained in Section III. Achieved results are exposed in Section IV, and Section V concludes the work and presents final considerations.

II. Related Work

Different methods have been proposed aiming to resolve ambient occlusion. Among them, the ray tracing technique is the direct translation of the ambient occlusion integral using the Monte Carlo method. Although accurate, the computational cost of this method is too high.

Screen-space methods have been used to approximate the occlusion integral in real time [8], [1], [9], [3], [4]. In general, these methods use deferred shading [10] to save buffers with geometry information in screen space. Only the information available in the generated buffers are used to compute ambient occlusion. The main advantages of such methods are good performance and natural support for dynamic scenes. However, these techniques are view-dependent and can result in inaccurate ambient occlusion due to the limited available information [5].

Szirmay-Kalos et al. [4] proposed the conversion of the directional integral of the ambient occlusion in a volumetric integral easily resolved on the GPU. Instead of tracing rays to sample the visibility of each point, they proposed to estimate ambient occlusion by computing the occluded volume of a sphere tangent to the point. The occluded volume of the tangent sphere is computed by sampling the sphere with a set of cylinders (pipes) in eye space. The cylinders have the same sectional area and are parallel to the z axis. The occluded volume of each cylinder is efficiently computed by fetching the depth buffer. The final occlusion is defined by the ratio between the occluded volume and the total volume.

Object-space algorithms tends to be more accurate, but have to deal with a large amount of information. In order to gain performance, different acceleration data structures have been employed. Papaioannou et al. [5] proposed an accelerated ray-marching technique in order to verify point visibility. Their technique consists in casting rays in several directions of the hemisphere, but the rays are iterated with constant steps in a regular grid that voxelizes the scene. At each iteration, it is checked if the ray hits a full voxel. For real-time scene voxelization, the authors indicated the method presented by Eisemann and Décoret [7].

Scene voxelization consists in representing the scene volume by a regular grid. Eisemann and Décoret [7] proposed a single-pass algorithm to voxelize the interior of *watertight* 3D models. According to Nooruddin and Turk [11], a model is watertight if for any connected component in space (separated by the geometry), all its points share the same classification: being in the interior or exterior. A point in space is considered interior/exterior if the number of intersections with the model of any ray originating at this point is odd/even.

The efficient voxelization technique proposed by Eisemann and Décoret [7] results in a binary regular grid represented by a 2D texture. The size of the texture defines the x and y dimensions of the grid, and the number of bits in the RGBA

representation of a texel forms the z dimension of the grid. The z dimension of a grid represented by a 32-bit RGBA texture is therefore 128.

Crassin et al. [6] introduced an approach using an octree of voxels to represent the geometry of the scene. This representation is constructed on the GPU and is used to make visibility tests using cone tracing. The structure supports dynamic scenes due to a real-time algorithm to update the octree: the octree is built once for static objects and is updated when there is movement or geometry modification. The cone tracing technique attempts to approximate the effect of a package of rays emitted from a point. The axis of the cone is traversed and the octree structure is accessed at different levels according to the corresponding cone radius. The ambient occlusion is calculated through the casting of cones in the hemisphere of a point, and the occlusion of each cone is summed and normalized by the total number of cones.

A. Discussion and Proposal

Screen-space methods obtain results with considerable efficiency, but suffer from the absence of enough information to accurately compute ambient occlusion; the view dependency aggravates inaccuracies when the main occluder is not visible. The proposal by Papaioannou et al. [5] uses ray-marching in a voxelized scene, reducing the view dependency; however, the ray-marching process still requires the use of a large number of rays to produce a high-quality result, what degrades performance.

The technique presented by Crassin et al. [6] builds an octree structure, taking advantage of graphics hardware, and uses cones to guide the way the levels of the octree is accessed. The performance is good and the results have good quality. Nevertheless, the creation and maintenance of this structure is complex, and an octree can be inappropriate to sample neighbor regions across different octants.

Similar to Papaioannou et al. [5], we have decided to employ the technique proposed by Eisemann and Décoret [7] and make use of a binary regular grid to voxelize the scene. In order to avoid the large number of rays being traced, similar to Crassin et al. [6], we have opted for using cone tracing. However, inspired by the work of Szirmay-Kalos et al. [4], we sample each cone by a set of spheres and compute the occluded volume of each sphere by efficiently fetching the binary voxelization grid. The obstructed volumes of the spheres are combined to express the amount of light blocked in each cone. As a result, we end up with an efficient and reasonable accurate object-space algorithm for ambient occlusion computation of dynamic scenes.

III. Proposed Method

The proposed method relies on estimating the amount of blocked light by a series of cones used to sample the hemisphere at each visible point. Each cone is, in turn, sampled by a sequence of spheres. The obstructed volume of each sphere is efficiently computed by fetching the voxelization data structure, and the amount of light blocked in each cone is evaluated by combining the sphere obstructed volumes.

Our method consists in a three rendering-pass algorithm:

- Geometry buffer creation: an object-space procedure based on the deferred shading technique.
- Scene voxelization: a second object-space procedure to create a binary regular grid.
- Ambient occlusion computation: a screen-space procedure to compute the occlusion of each visible point.

The next subsections describe in detail these procedures, especially the third one where resides our main contribution.

A. Geometry Buffer Creation

Ambient occlusion computation is required only for the visible points. We thus employ a first pass to create the buffers that hold the geometry data needed for this computation. Using the conventional deferred shading technique, the scene objects are sent through the graphics pipeline, using a fragment shader to direct the information to the geometry buffers (G-buffers). In our case, we create one buffer to hold the position and another to hold the normal at each pixel, both in eye space. This geometry information is used to determine the hemisphere associated to each pixel, thus defining cone orientations.

B. Scene Voxelization

Our method uses scene voxelization as the structure to allow easy access to the geometry information during the ambient occlusion computation. We use the efficient voxelization technique proposed by Eisemann and Décoret [7], building the grid at each frame. In this way, we provide support for dynamic scenes. The advantage of using a regular grid relies on its simplicity of computation and access. The voxelization is represented by a binary regular grid encoded in a 2D texture as shown in the Figure 3.

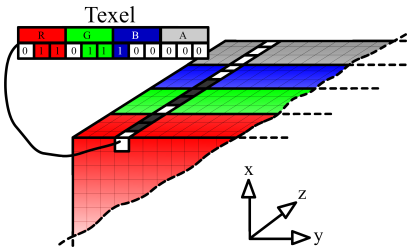


Fig. 3. Example of a regular grid. The x and y dimensions of the grid are delimited by the texture dimension. The z dimension is delimited by the number of bits in a texel (a value of 128 if we consider 32 bits per component).

The technique proposed by Eisemann and Décoret [7] works for *watertight* models. It produces a solid voxelization of the scene, assigning 1 to a voxel (represented by a bit) that is interior to any object and 0 to a voxel that is exterior to all objects. This is efficiently built by sending both the front and back faces of each object through the graphics pipeline and by employing XOR operations to correctly combine the rasterized fragments. We build the voxelization of the scene also in eye space.

The main limitation of this voxelization representation is its small resolution along the z axis: 128 considering 32 bits per component. To alleviate this limitation, we propose to use the information acquired in the previous step to slightly improve the voxelization along the z axis. As mentioned, the previous step creates a buffer that holds the position in eye space of each visible point. We then use the $z_{nearest}$ coordinate stored at each pixel to reduce the voxelized space along each pixel in the z direction, instead of using the near plane distance for all the pixels. This improvement is based on the concept of local slice map, presented by Eisemann et al. [12]. Figure 4 exhibits the difference between a grid using the near plane and the proposed one using the nearest fragment.

The voxel index, along the z axis, associated to a given z coordinate, is given by the following equation:

$$z_{GridIndex} = \frac{z - z_{nearest}}{z_{far} - z_{nearest}} \tag{2}$$

where z_{far} represents the distance to the far plane.

One can note that a similar approach could be used to replace the far plane distance. However, obtaining the information of the farthest fragment would involve another geometry rendering pass, what would be too costly. The nearest fragment is gotten for free due to the previous G-buffer computation.

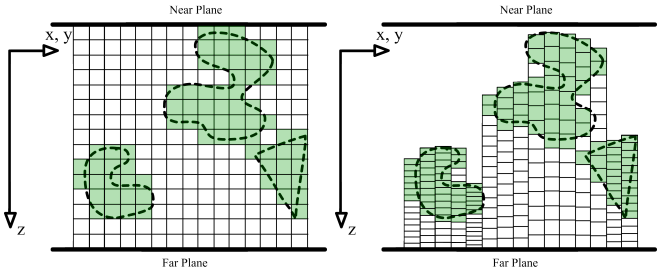


Fig. 4. Conventional grid using the near plane (a); and proposed grid using the nearest fragment (b).

C. Ambient Occlusion Computation

Our approach to resolve the ambient occlusion integral involves the replacement of ray tracing by cone tracing. Thus, the hemisphere associated to each point is sampled by a set of cones. Figure 5a illustrates the hemisphere sampling process.

Each cone apex is positioned at the location stored in the G-buffer (eye space). The hemisphere is oriented according to the stored normal and is limited by choosing a distance of influence, defining a spherical cap with the chosen distance as its radius ($Rmax$). To orient the cones, the spherical cap is regularly subdivided using spherical coordinates. The cone aperture is set to 30 degrees; this value has delivered good results: a smaller value covers less hemisphere volume than needed; a higher value has turned difficult to accommodate the sequence of spheres inside the cone.

In fact, the volume of each cone is sampled by disposing a series of spheres along the cone's axis. Consecutive spheres are tangent to each other, and the radii are chosen according to the cone aperture. Figure 5b shows this arrangement.

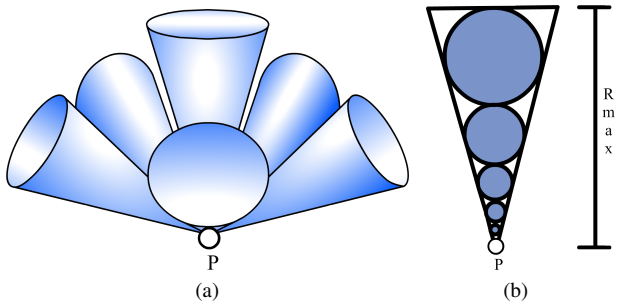


Fig. 5. Sampling the hemisphere by a set of cones (a); sampling each cone by a sequence of spheres (b).

The ambient occlusion is estimated by first evaluating the obstructed volume of each sphere. This is resolved inspired by the method proposed by Szirmay-Kalos et al. [4]. However, instead of using the depth buffer, our technique computes the number of full voxels inside the sphere to estimate its occluded volume. This subject is detailed in Subsection III-C1. The occluded sphere volumes are then combined to get the amount of light blocked in each cone, and the final ambient occlusion considers all the cones, as described in Subsection III-C2.

1) Sphere Occlusion: The obstructed volume of each sphere is computed by using the Monte Carlo integration scheme. We estimate the obstructed volume by a small set of prisms (columns), within the sphere, oriented along the z-axis. The obstructed sphere volume is set proportional to the obstructed volumes of all prisms. The great disk of each sphere (represented by the intersection between the sphere and a plane containing the center of the sphere), perpendicular to the z-axis, is sampled using Poisson disk distribution. Each sample defines a prism delimited by the sphere, and each prism is aligned to one column, along z, of the voxelization grid. Given the x and y coordinates of each sample, it is easy to compute the limits, along the z-axis, of the corresponding prism: z_{in} and z_{out}. Figure 6 illustrates the sphere sampling process.

The obstructed volume of each prism is computed in a very efficient way. The x and y coordinates of each sample is also used to fetch the 2D texture representing the voxelization grid. The corresponding texel encodes the obstructed volume of the entire column along the scene. Knowing the prism limits, z_{in} and z_{out}, our goal is to count the number of bits set to 1 within this range.

To do that, we employ a two-step procedure. First, we eliminate the bits outside the limits; then, we count the number of remaining bits. This procedure is performed with the help of built-in functions supported by modern graphics card, and is illustrated in Figure 7. For each color channel, the bits outside the limits are eliminated using the function *bitfieldExtract* and the remaining bits are counted using the function *bitCount*. Both functions perform very efficiently.

2) Amount of Blocked Light: Once we have the obstructed volume of each sphere, we compute the amount of blocked

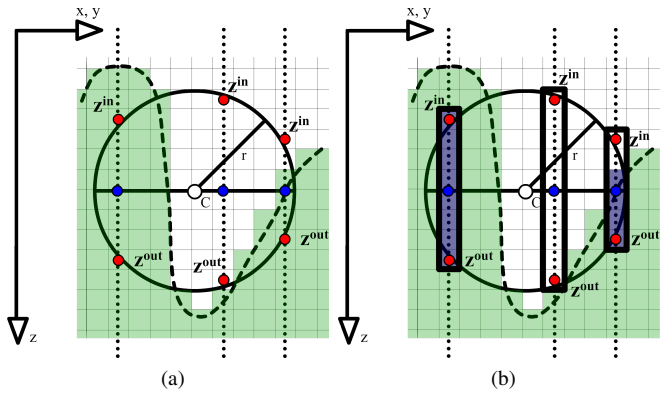


Fig. 6. A sphere in the voxelized space with the great disk and samples (blue dots). The points z_{in} and z_{out} indicate the limits of the prisms within the sphere (a); the resulting range of full voxels inside the sphere is displayed in blue (b).

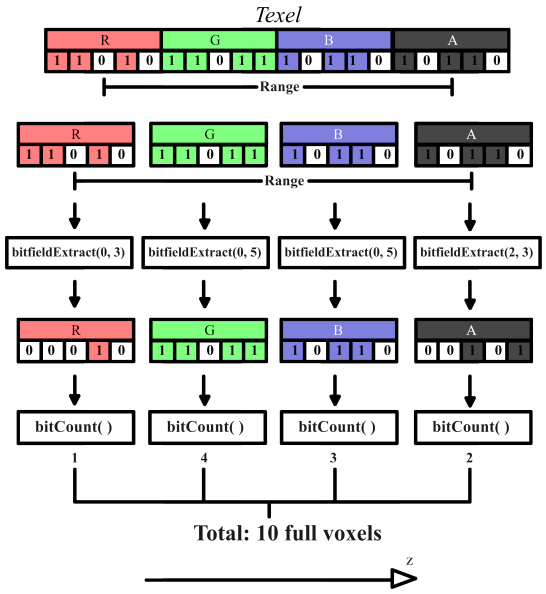


Fig. 7. Procedure to count the number of bits set to 1 along a prism inside a sphere.

light in the cones. Each cone represents a package of rays. To compute the amount of blocked light is to compute the amount of rays that hit the geometry of the scene before the distance of influence is reached.

The amount of blocked rays in a cone results from accumulating the amount of obstructed volume by the sequence of spheres in a front-to-back fashion [13]. Figure 8 illustrates the adopted procedure. We first consider the smallest sphere, the one closest to the cone apex. Let us consider that x rays are emitted from P. Without losing the generality, let us say that the obstructed volume of this sphere is 30%. We then assume that this sphere absorbs 30% of the rays; as a consequence, the amount of rays that enters the second sphere is $x(1-0.30)$. If the second sphere obstructs 25% of the rays, the third sphere will be reached by only $x(1-0.30)(1-0.25)$ rays, and so on. After accumulating the result of the last sphere, we have the amount of blocked rays in the cone, which is directly translated to represent the amount of blocked light.

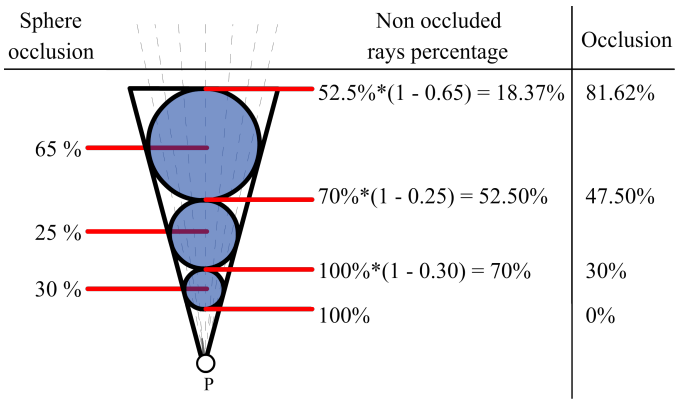


Fig. 8. Cone with occlusions values of each sphere. The column in the middle indicates the portion of the rays not occluded before and after each sphere. The occlusion along the axis of the cone is displayed in the right column.

Finally, we average the results of all cones to estimate the amount of blocked light at each visible point. Note that all the procedure can be efficiently implemented on graphics hardware. A few parameters control the quality of the resulting images: the number of cones to sample the hemisphere, the number of spheres to sample each cone, and the number of prisms to sample each sphere.

IV. Results

Our method was implemented in C++ using OpenGL and the GLSL shading language. Several computational tests were accomplished aiming to measure the performance and to analyze the quality of the obtained results. The tests were run on a machine with a 2.8 GHz Intel i7 processor equipped with a Nvidia GeForce GTX 480 graphics card. All the frame rates presented here represent average performance of the algorithms.

We first present an analysis of the achieved results by varying the parameters that control the proposed method, testing with three different models. We compare both image quality and performance. Then, we analyze the performance of each rendering pass of our algorithm varying screen resolution and geometry complexity. Finally, we compare our achieved results with the results produced by a screen-space algorithm.

A. Quality Test

In order to verify the quality of the achieved results of our proposal, we vary the parameters that control the method. Three settings were defined: *low*, *medium*, and *high* quality. The best choice may depend on the available graphics hardware. Our goal is to analyze the quality of the results for different settings. Table I exhibits the parameters used in each quality setting. Note that we have opted for fixing the number of cones to 6. In fact, this has shown to be an appropriate, and relatively cheap, choice.

TABLE I
PARAMETERS USED FOR EACH QUALITY SETTING.

Quality	Number of cones	Number of spheres per cone	Number of prisms per sphere
Low	6	3	3
Medium	6	5	6
High	6	7	12

As a quality reference, we used the ray-tracing ambient occlusion application Optix [14]. It was developed by Nvidia using CUDA and thus runs on the GPU. It produces high quality results but does not run in real time.

We ran the quality test considering three distinct models: *karburator*, *jsyrlin*, and *buddha*. Table II shows the observed time of execution of our algorithm considering the three different parameter configurations. Figures 9, 10, and 11 display the achieved results for a visual comparison. These results show that our algorithm tends to get closer to Optix results as we tune the quality parameters. One can also note that the proposed method generates good results even with the defined low-quality parameters.

TABLE II
COMPARISON AMONG DIFFERENT QUALITY SETTINGS FOR THREE DISTINCT MODELS.

Model	# Verts	# Triangs	Quality	Time (ms)	FPS
karburator	251k	500k	Low	5.03	198.81
			Medium	12.12	82.51
			High	29.92	33.42
jsyrlin	253k	505k	Low	5.08	196.85
			Medium	12.55	79.68
			High	31.30	31.95
buddha	757k	1,514k	Low	6.81	146.84
			Medium	13.27	75.36
			High	29.59	33.80

A limitation of our method is the low voxelization resolution along the z axis. In its current implementation, it is limited to 128 voxels due to the maximum number of bits in a single texel. This restriction makes little details, mainly small cavities, to be incorrectly illuminated. Figure 9 exhibits this limitation at the cavity of a pipe: it appears dark in the reference image but only slightly darkened in the images produced by our algorithm. The grid resolution can be extended by using more textures. However, this affects the overall performance due to the need of more time to build the grid and more accesses to the grid texture during the occlusion calculation; such an extension will be considered in a future research.

Table II shows that the performance decreases as we improve image quality. As graphics hardware evolves, higher quality configurations may be employed in real-time applications. Even though, we evaluate that the defined medium-quality configuration has presented a good tradeoff between quality and performance. Based on these results, we have chosen the medium-quality setting as the one used in the subsequent tests.

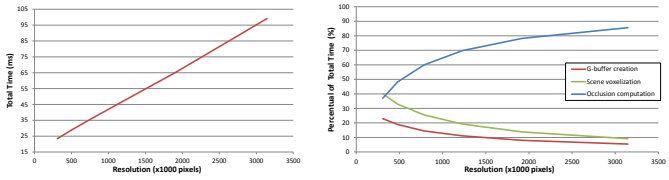
B. Performance Analysis

The proposed algorithm consists in three steps: geometry buffer creation, scene voxelization, and ambient occlusion computation. To better analyze the performance of our method, we measured the rendering time spent at each step in diverse situations, running the algorithm with the increase of image resolution and of geometry complexity.

Table III shows the rendering time of a model composed of 3 million vertices and 6 million triangles in different resolutions. The plot in Figure 12a depicts that the algorithm performance varies linearly with the number of pixels, as expected. The plot in Figure 12b presents the percentage of time spent at each step of the algorithm as the screen resolution increases.

TABLE III
EXECUTION TIME FOR DIFFERENT SCREEN RESOLUTIONS.

Resolution	Time (ms)	FPS
640x480	23.37	42.78
800x600	28.34	35.28
1024x768	36.45	27.43
1280x960	47.87	20.88
1600x1200	65.52	15.26
2048x1536	99.18	10.08



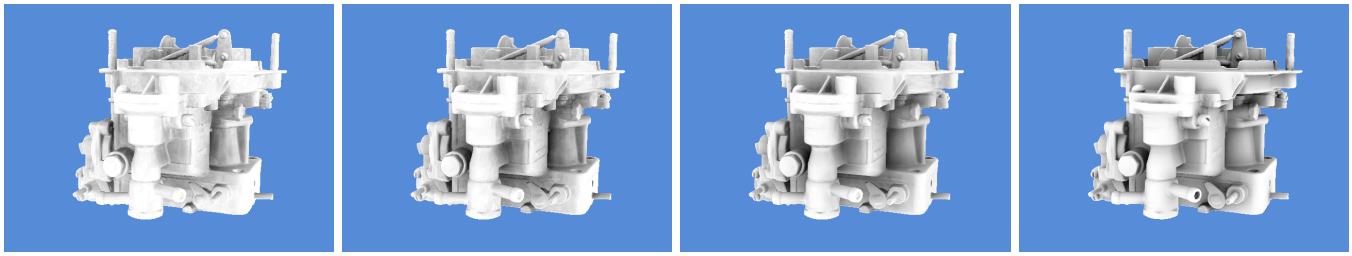
(a) (b)

Fig. 12. Performance variation with respect to image resolution for rendering the image (a), and the percentage of time spent at each step (b).

Table IV arranges the rendering time considering scenes with different geometry complexities. In order to eliminate screen-space interferences, all the scenes were rendered maintaining, as closely as possible, the same amount of visible pixels (an auxiliary procedure using occlusion query was applied to count the pixels). The plot in Figure 13a exhibits a global tendency of linear variation, again as expected. The plot in Figure 13b presents the percentage of time spent at each step of the algorithm as the complexity of the geometry increases.

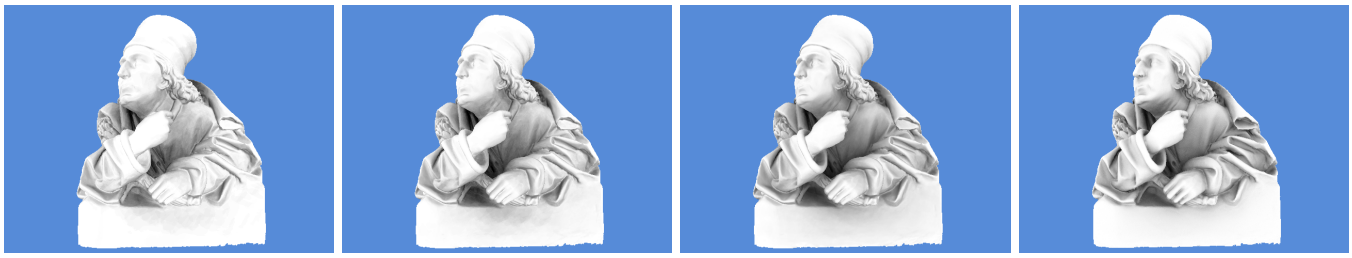
TABLE IV
EXECUTION TIME FOR DIFFERENT GEOMETRY COMPLEXITIES.

Model	# Verts	# Triangs	Time (ms)	FPS
bunzipper	36k	69k	6.11	163.67
jsyrlin	253k	505k	8.13	123.00
hand	327k	655k	8.46	118.20
bareliefply	507k	1,000k	10.49	95.33
happyvrip	544k	1,088k	10.45	95.69
buddha	757k	1,514k	10.65	93.90



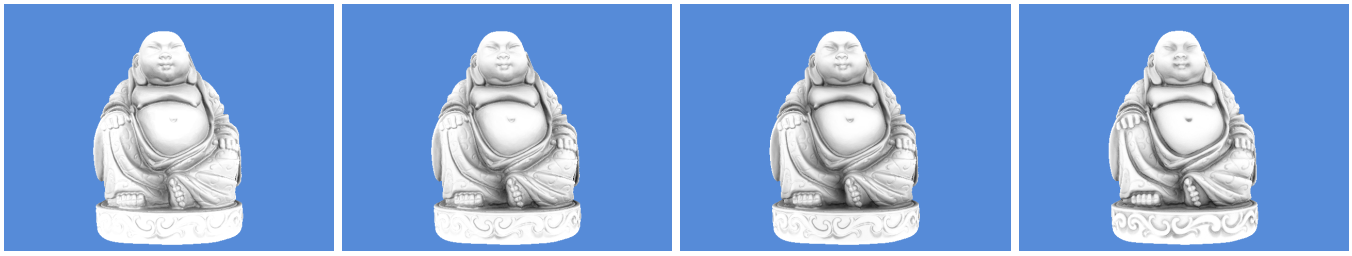
(a) Low (b) Medium (c) High (d) Optix

Fig. 9. Comparison among different quality settings for the *karburator* model.



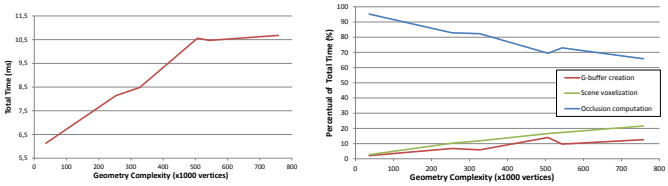
(a) Low (b) Medium (c) High (d) Optix

Fig. 10. Comparison among different quality settings for the *jsyrlin* model.



(a) Low (b) Medium (c) High (d) Optix

Fig. 11. Comparison among different quality settings for the *buddha* model.



(a) (b)

Fig. 13. Performance variation with respect to geometry complexity for rendering the image (a), and the percentage of time spent at each step (b).

C. Comparison with a Screen-Space Algorithm

Our last test compares our method with a screen-space algorithm. The screen-space algorithm, denoted by SSAO, was developed by NVidia and is part of the Direct3D SDK examples [15].

Table V compares the performance achieved by each algorithm to render the *dragon* model using two different poses. Figure 14 displays the achieved images, comparing both with the quality reference *Optix*. The results are in accordance with the expected behavior: a better performance for the screen-space algorithm, and a better quality for our object-space algorithm. However, it is worth mentioning that our proposal presents quite competitive performance and much better quality.

TABLE V
PERFORMANCE COMPARISON BETWEEN SSAO ALGORITHM AND OUR METHOD.

Algorithm	Pose 1 (FPS)	Pose 2 (FPS)
SSAO	87.8	88.2
Our method	80.1	78.3

Figure 15 highlights the differences between the achieved results and points out critical parts for the screen-space algorithm.

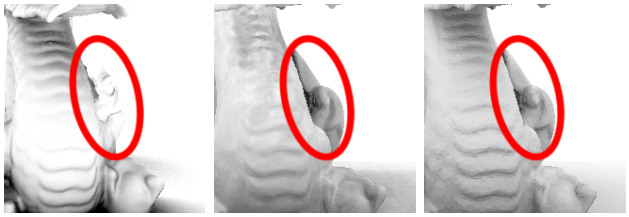
V. CONCLUSION

In this paper, we presented a novel method to compute ambient occlusion in real time. The proposed method per-

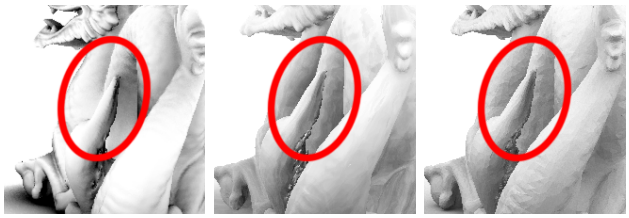
(a) SSAO (b) Our method (c) Optix

(d) SSAO (e) Our method (f) Optix

Fig. 14. Image comparison between SSAO algorithm and our method.



(a) SSAO (b) Our method (c) Optix



(d) SSAO (e) Our method (f) Optix

Fig. 15. Details not correctly captured by the screen-space algorithm.

forms the occlusion computation in object space and uses the efficient voxelization algorithm presented by Eisemann and Décoret [7]. This allows fast computation of obstructed volume of spheres. The ambient occlusion of each visible point is defined by sampling the corresponding hemisphere with a set of cones, each one representing a package of rays. The amount of rays emanating in each cone that is obstructed by the geometry of the scene is computed by sampling each cone by a sequence of spheres. A set of computational experiments were used to analysis both performance and quality of the proposed method. The achieved results demonstrate that our method produces images in real time with quality similar to a ray casting algorithm. Still, the performance of our proposal is competitive when compared to a screen-space algorithm.

In the future, we plan to investigate the use of a grid with higher resolution in the z direction to voxelize the scene. We also intend to investigate the use of ambient occlusion in scientific visualization of unstructured meshes. The generation of a regular grid for static scenes in a pre-processing phase can also be explored in order to improve the overall performance for static scenes.

ACKNOWLEDGMENT

We thank CAPES (Brazilian National Research and Development Council) and CNPq (Brazilian National Council for Scientific and Technological Development) for the financial support to conduct this research. We also thank the anonymous reviewers for the valuable feedback.

REFERENCES

[1] P. Shanmugam and O. Arikan, "Hardware accelerated ambient occlusion techniques on gpus," in *In I3D 07: Proceedings of the 2007 symposium on Interactive 3D graphics and games, ACM*. Press.

[2] M. Sattler, R. Sarlette, G. Zachmann, and R. Klein, "Hardware-accelerated ambient occlusion computation," in *Vision, Modeling, and Visualization 2004*, B. Girod, M. Magnor, and H.-P. Seidel, Eds. Akademische Verlagsgesellschaft Aka GmbH, Berlin, Nov. 2004, pp. 331–338.

[3] L. Bavoil, M. Sainz, and R. Dimitrov, "Image-space horizon-based ambient occlusion," in *SIGGRAPH '08: ACM SIGGRAPH 2008 talks*. New York, NY, USA: ACM, 2008, pp. 1–1.

[4] L. Szirmay-Kalos, T. Umenhoffer, B. Tth, L. Szcsi, and M. Sbert, "Volumetric ambient occlusion for real-time rendering and games." *IEEE Computer Graphics and Applications*, vol. 30, no. 1, pp. 70–79, 2010. [Online]. Available: http://dblp.uni-trier.de/db/journals/cga/cga30.html#Szirmay-KalosUTSS10

[5] G. Papaioannou, M. L. Menexi, and C. Papadopoulos, "Real-time volume-based ambient occlusion," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, pp. 752–762, 2010.

[6] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using voxel cone tracing: An insight," Technical Talk at SIGGRAPH, aug 2011.

[7] E. Eisemann and X. Décoret, "Single-pass GPU solid voxelization for real-time applications," in *Graphics Interface 2008, GI '08, May, 2008*. Windsor, Canada: Canadian Information Processing Society, May 2008, pp. 73–80.

[8] M. Mittring, "Finding next gen: Cryengine 2," in *SIGGRAPH '07: ACM SIGGRAPH 2007 courses*. New York, NY, USA: ACM, 2007, pp. 97–121.

[9] R. Dimitrov, L. Bavoil, and M. Sainz, "Horizon-split ambient occlusion," in *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, ser. I3D '08. New York, NY, USA: ACM, 2008, pp. 5:1–5:1. [Online]. Available: http://doi.acm.org/10.1145/1342250.1357017

[10] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The triangle processor and normal vector shader: a vlsi system for high performance graphics," in *SIGGRAPH*, 1988, pp. 21–30.

[11] F. S. Nooruddin and G. Turk, "Simplification and repair of polygonal models using volumetric techniques," *IEEE Transactions on Visualization and Computer Graphics*, vol. 9, pp. 191–205, 2003.

[12] E. Eisemann and X. Dcoret, "Fast scene voxelization and applications," *ACM SIGGRAPH 2006 Sketches on SIGGRAPH 06*, p. 8, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1179849.1179859

[13] N. Max, "Optical models for direct volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, pp. 99–108, June 1995. [Online]. Available: http://dl.acm.org/citation.cfm?id=614258.614298

[14] Nvidia, "Nvidia optix 2 ray tracing engine examples," Jun. 2011, `http://developer.nvidia.com/optix-interactive-examples`. [Online]. Available: http://developer.nvidia.com/optix-interactive-examples

[15] ——, "Nvidia direct3d sdk 10 code samples," Jun. 2011, `http://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html`. [Online]. Available: http://developer.download.nvidia.com/SDK/10.5/direct3d/samples.html