Analytic Ambient Occlusion using Exact from-Polygon Visibility

Abstract

This paper presents a new method to compute exact from-polygon visibility, as well as a possible application to the calculation of high quality ambient occlusion. The starting point of this work is a theoretical framework which allows to group lines together according to the geometry they intersect. By applying this study in the context of from-polygon visibility, we derive an analytical definition which permits us to group the view rays according to the first geometry they encounter. Our new algorithm encodes lazily the visibility from a polygon. Contrary to previous works on from-polygon visibility, our approach is very robust. For each point in the scene, the algorithm efficiently calculates the exact fragments of visible geometry. By combining this information with a analytical definition of ambient occlusion, we achieve results that are not sensitive to noise or other visual imperfections, contrary to ambient occlusion methods which are based either on sampling or visibility approximations.

Keywords:
Visibility, Ambient Occlusion, Plücker Coordinates

1. Introduction

Ambient occlusion is an approximation of the darkening which occurs when the ambient light is blocked by nearby objects. The technique is usually used to improve the realism of rendered images at lower cost compared to global illumination algorithms. The ambient occlusion of a point is defined as the cosine-weighted fraction of the upper hemisphere where incoming light cannot reach the point [1]. This corresponds to a surface integral, which is usually evaluated by sampling the point's upper hemisphere and casting rays to test for occlusions. In this case, the results are sensitive to noise according to the sampling strategy and the density. To avoid this problem, some methods use an analytical evaluation of the integral. However, they still approximate the visibility computations geometry, which leads to visual artifacts.

In this paper, we present a high quality method based on exact visibility, which calculates analytical ambient occlusion. Therefore, our approach does not suffer from noise or any visual artifact. Although calculating the exact visibility from a point is possible, this would be too expensive if applied to all the points in a scene. We propose a different strategy based on from-polygon visibility to take advantage of the visibility coherence between neighbor points on a same surface.

Two main contributions are presented:

- First of all, we describe a new algorithm which calculates exact and analytical visibility from a surface. This solution brings from-polygon visibility at a higher level of robustness and efficiency than any of the previous methods.
- Secondly, we apply our approach in the context of ambient occlusion to produce high quality and noise free results without visual artifact.

The starting point of our method are the equivalence classes proposed by Pellegrini [2, 3], which allow grouping lines according to the geometry they intersect. We build on this theoretical framework and provide a new definition which groups the rays issued from a surface according to the first triangle they intersect. This information is encoded into a visibility data structure, which is built on demand. Using the data structure we can efficiently solve the exact visibility of a group of triangles for each point, taking advantage of visibility coherence between neighbor points. This information is used with a closed form solution of the ambient occlusion integral to achieve high quality noise free results.

The paper is organized as follows: The first section summarizes the works related to this paper. The second section details some useful mathematical

notions, followed by the geometrical concepts which motivated our research. In the fourth section we discuss our work from a theoretical point of view. We present the design of a structure capable to retain the visibility from a surface and we analyze the means by which this information can be extracted. The practical aspects of the algorithm will be addressed in Section 5. Finally, in Section 6, we test our algorithm's behavior on different configurations and analyze the results. We also address some of the limitations of our method, in correlation with possible perspectives.

2. Related Work

2.1. Ambient Occlusion Theory

Ambient occlusion (Equation 1) is an empirical illumination model used to simulate global illumination effects, at a less expensive cost. It was first introduced by Zhukov et al. [4], under the name of obscurances, and later adapted to the movie industry [5, 6]. Roughly speaking, ambient occlusion is a geometrical property of a point which approximates the percentage of ambient light blocked by the directly visible geometry close to the point.

$$AO(x) = \frac{1}{\pi} \int_{\omega \in \Omega} vis(x, \omega)(N \cdot \omega) d\omega \qquad (1)$$

Here x is the surface point receiving the occlusion and N its normal. Ω denotes the upper hemisphere with respect to N and $vis(x, \omega)$ is a visibility function that returns a zero value when no geometry is visible in direction ω, and one otherwise.

Obscurances and ambient occlusion are set apart by a major difference: The first one takes into account an attenuation function, dependent on the distance to the blocking surfaces, while the second one tests only the visibility along a view direction. Therefore, from a geometrical point of view, projectively equivalent objects produce the same ambient occlusion.

Several computation methods exist. According to [4], the ambient occlusion for a fully visible polygon resembles the form factor corresponding to the diffuse radiative transfer between the polygon and the considered point. This involves a visibility calculation and an evaluation of the integral in Equation 1. Usually, approximations are made when performing both these steps. Our goal is to present an exact method which yields accurate and noise-free results. Therefore, in this section we focus only on those methods that either propose analytical solutions or achieve results matching those produced using a ray tracer. A more comprehensive survey of the different obscurance and ambient occlusion techniques has been provided by Méndez and Sbert [7].

2.2. Ambient Occlusion Computation

Ray traced solutions. The reference method is the ray traced ambient occlusion [8]. This technique evaluates the integral by sampling the point's upper hemisphere and casting rays into the environment to test for occlusions. The quality of the final images is therefore dependent on the number of rays traced and the results usually contain a large amount of noise. The sampling technique also has an impact on the computations, as shown in [9]. While this technique remains the gold standard for off-line renderers, most methods trade accuracy for speed, by approximating the scene geometry and/or the visibility computations.

Analytic solutions. Bunnell [10] represents the scene by a hierarchical structure of disks. The occlusion between these disks is then analytically approximated using disk-to-disk form factor evaluation. Hoberock and Jia [11] construct on this approach in order to improve its robustness and the results quality. Both algorithms approximate occlusion by summing shadow contributions from neighbor disks. Although the use of analytic expressions yields noise free images, their results suffer from various visual artifacts which are mainly due to the approximated visibility.

McGuire [12] proposes an analytical method which yields quality noise-free results. He builds an ambient occlusion volume for each polygon and locates the visible points in the volumes using rasterization. Ambient occlusion values are calculated using an analytic formula derived from the global illumination theory [13]. Although achieving noiseless high quality results, the method suffers from over-occlusion, which results in over-darkened areas and a loss of details. This limitation is solved by Laine and Karras [1], which redefine the bounding volumes and propose a technique to avoid over counting occlusions. However, they replace the analytical evaluation of the ambient occlusion integral with an improved stochastic sampling. A second ray tracing method using BVH traversal is presented. Although approximations are made in both the evaluation of the integral and the visibility computations, their results have a comparable quality to those produced by a ray-tracer.

Szirmay-Kalos et al. [14] introduce *volumetric ambient occlusion*, a new definition of a point's openness, which is more adapted to GPU evaluation. The directional integral in Equation 1 is transformed into a volumetric one and ray tracing is replaced with containment tests. Although the new integral can be evaluated more

accurately while keeping a low number of samples, the results are sensitive to noise and require a low pass filter to reduce it.

Context based solutions. There are other studies which target specific applications, such as molecular visualization [15], tree [16] or character [17, 18] rendering. These methods are based on approximations or assumptions valid only in a particular context and are therefore not related to this work.

Screen space ambient occlusion (SSAO) methods [19], represent a recent class of techniques, which target the gaming industry and are primarily oriented towards speed and performance, to the detriment of visual quality. Roughly speaking, the ambient occlusion between nearby geometry is crudely approximated in screen space. This approximation is the source of many limitations, and has a negative impact on image quality. Therefore, their study is outside the scope of this paper.

Although having an empirical foundation, ambient occlusion represents an illumination model which translates a geometric property of a point. The computations based on sampling techniques [8] are subject to noise, according to the method and the number of samples. On the other hand, using an analytic evaluation of the integral ensures noise free results [10, 11, 12]. However, this cannot compensate for the approximated visibility, which becomes the source of various artifacts, such as disk-shaped artifacts [10] and over-occlusion [12]. Therefore, in order to achieve noise-free and high quality results, we combine an analytic expression with an exact visibility computation.

2.3. Exact From Polygon Visibility

Exact solutions to visibility problems are important because they help to improve our understanding about visibility. However, exact from-polygon visibility is very complex, because it is a four-dimensional problem. As an example, the 3D visibility complex [20] is a data structure which aims at encoding exactly all the visibility information in a 3D environment. However, its construction is complicated and suffers from degeneracies. The visibility skeleton [21] is a partial construction of the 3D visibility complex. Thus it does not encode all the visibility. Several algorithms have been proposed [22] but its construction remains a challenging task [23].

The most practical solutions in the literature rely on the same theoretical framework described by Pellegrini [3] in Plücker space, a 5D space of lines. His analysis provides a worst case complexity of $O(n^{4+\epsilon})$ in space and time, where n is the number of triangles. This underlines the complexity inherent to the visibility. A very special care must be given to the algorithm design to remain practicable. As an example, a first approach is proposed by Mount and Pu [24]. But their experiments do not exceed 14 random triangles, because, as stated by the authors, they reach the theoretical complexity upper bound.

The two first practicable algorithms are provided by Nirenstein *et al.* [25] and Bittner [26] (see [27] for details about the differences between the two solutions), and further improved by Haumont *e*t al. [28] and Mora *et al.* [29] respectively.

It is worth underlying the major differences between the two approaches initiated by Nirenstein and Bittner. In particular, they provide different outputs and compute different informations.

Algorithm output. Nirenstein and Haumont design their algorithms to answer if two polygons are mutually visible. Thus, the algorithms output a boolean value. In contrast, the methods proposed by Mount, Bittner and Mora aim to compute and store all the visibility information. In this case, the output is a partition of Plücker space encoded in a Binary Space Partitioning tree.

Occlusion versus visibility. All the previous methods, with the exception of Bittner's algorithm, calculate occlusion between two polygons. More precisely, they distinguish two kinds of lines: occluded and unoccluded. This is a lot less complex than computing a full visibility information, which implies to differentiate the lines originating from a polygon according to the first geometry they encounter.

Bittner [26] is the only one to compute full visibility in his PhD thesis. But his experiments are restricted to random triangles or 2.5D environments with moderate complexity in order to compute Potentially Visible Sets.

Despite these differences, all the previously listed algorithms share a common drawback: They all rely on Constructive Solid Geometry (CSG) operations in Plücker space, which consists in computing subtractions of 5D polyhedrons. This has several negative impacts.

5D CSG problems. First of all, CSG operations are not reliable and computationally expensive because of the space dimension. All the above methods can only be used as a pre-process step. Secondly, these calculations are very complex to implement since they requires non-trivial geometric algorithms defined in nD (vertex enumeration [30], face lattice computation [31, 32], intersections of nD polyhedrons [33]). All this results in algorithms that are prone to numerical instabilities and which lack robustness.

In this paper, we start from Pellegrini's framework

to derive a totally different approach from the previous work. Our algorithm computes from-polygon visibility, and not occlusion. The output is a data structure encoding the visibility information. Contrary to all the previous works:

- Our algorithm works at run time, computing the visibility information lazily, where and when it is required.
- Our method avoids all 5D CSG operations, which brings from-polygon visibility at a high level of robustness and efficiency. This has an important impact on the implementation, making it simpler.
- Our method encodes from-polygon visibility in order to take advantage of the visibility coherence from a surface and speed-up point to polygon visibility queries.

Our algorithm capacities are demonstrated through its application in computing high quality ambient occlusion using 3D scenes with different complexities.

3. Theoretical Basis

3.1. An Analytical Solution to the Ambient Occlusion Integral

The following assumption is made throughout the rest of this paper: Any surface (or polygon) refers to a plane convex polygon having a front and a back face.

Let x be a point on a surface S with normal N, and let T be a triangle (convex polygon) which is entirely visible from x. Computing the ambient occlusion due to T is equivalent to calculating the form factor between x and T [4]. In the simplified case of a point and a directly visible polygon, the form factor can be evaluated in closed form using Lambert's formula [34]. For T, the ambient occlusion integral (Equation 1) becomes the sum in Equation 2.

$$AO(x,T) = \frac{1}{2\pi}\sum_{i=1}^{n}(\cos^{-1}\frac{v_i \cdot v_{i+1}}{\|v_i\| \cdot \|v_{i+1}\|})(\frac{v_i \times v_{i+1}}{\|v_i \times v_{i+1}\|} \cdot N) \quad (2)$$

Where n is the number of vertices of T and $v_i = t_i - x$ (the vectors defined by x and each vertex of T).

If there is more than one visible polygon, the total ambient occlusion is the sum of the integrals over each surface.

Equation 2 can only be used under the assumption that for a point x all the exact fragments of visible geometry are known. However, calculating these fragments for every point on a surface can be a very expensive task. On the other hand, neighbor points have similar views of the surrounding environment. Therefore, in this paper, instead of computing the visibility from each point, we capture the visibility from the whole polygon in order to use the coherence of view directions.

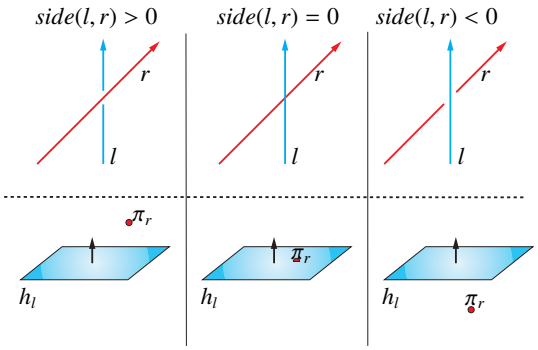

Figure 1: Above: The relative orientation of two lines is given by the sign of the side operator. Under: Since $side(l,r) = h_l(\pi_r)$, the side operator is equivalent to testing the position of the Plücker point of one line against the dual hyperplane of the other line

The next subsection briefly reviews the Plücker coordinates, which are useful to calculate visibility from surfaces.

3.2. Plücker Coordinates

Plücker coordinates [35] provide an efficient representation of oriented lines. Using the Plücker coordinates, any real line can be mapped to a point in a five dimensional projective space, known as the Plücker space ($\mathbb{P}^5$). Let $p = (p_x, p_y, p_z)$ and $q = (q_x, q_y, q_z)$ be two distinct 3D points which define an oriented line l. This line corresponds to a set of six coefficients called the Plücker coordinates, π_l, of the line l.

$$\pi_l = (l_0, l_1, l_2, l_3, l_4, l_5)$$

where

$$\begin{aligned} l_0 &= q_x - p_x & l_3 &= q_z p_y - q_y p_z \\ l_1 &= q_y - p_y & l_4 &= q_x p_z - q_z p_x \\ l_2 &= q_z - p_z & l_5 &= q_y p_x - q_x p_y \end{aligned}$$

Any Plücker point is in bijection with its dual hyperplane:

$$h_l(x) = l_3x_0 + l_4x_1 + l_5x_2 + l_0x_3 + l_1x_4 + l_2x_5 = 0$$
$$with\ x(x_0, x_1, x_2, x_3, x_4, x_5) \in \mathbb{P}^5.$$

Therefore, there are two geometrically dual ways to describe an oriented real line in Plücker space: either as

a point or as its dual hyperplane.

An important property of the Plücker space is that we can characterize the relative orientation of two lines. This can be done using the so-called *side operator*:

$$side(l, r) = l_3 r_0 + l_4 r_1 + l_5 r_2 + l_0 r_3 + l_1 r_4 + l_2 r_5$$

where l and r are two lines and $\pi_l\,(l_0, l_1, l_2, l_3, l_4, l_5)$ and $\pi_r\,(r_0, r_1, r_2, r_3, r_4, r_5)$ their Plücker coordinates. In particular, two lines are incident (or parallel) if their side operator equals zero. We can notice that $side(l, r) = h_l(\pi_r)$. From a geometric point of view, this translates to the fact that computing the relative position of two lines comes down to testing the position of the Plücker point of one line against the dual hyperplane of the second line (see Figure 1).

We recall here an important theorem which will be used in the context this paper.

Theorem 1. Let A and B be two convex polygons. All the lines intersecting both A and B are contained in a minimal convex polyhedron defined in Plücker space (we note this $poly(A \rightarrow B)$). The vertices of this polyhedron are the Plücker points corresponding to the lines defined by one vertex of A and one vertex of B. The bounding hyperplanes are the Plücker hyperplanes corresponding to the support lines of the edges of A and B. This theorem is demonstrated in [36].

Corollary. Testing the orientation of $poly(A \rightarrow B)$ with respect to a hyperplane is equivalent to calculating the position of the lines stabbing A and B with respect to the hyperplane. If all the vertices of $poly(A \rightarrow B)$ have the same sign with respect to the hyperplane, the polyhedron lies completely in the positive or negative half-space defined by the hyperplane. Thus, all the stabbing lines have the same orientation. Otherwise, $poly(A \rightarrow B)$ is intersected by the hyperplane and the orientation of the lines stabbing A and B is no longer coherent.

3.3. Equivalence Classes of Oriented Lines

A well-known application of the side operator is an easy and robust solution to the line-triangle intersection problem. An oriented line intersects a triangle if its relative orientation is consistent with respect to the lines spanning the triangle's edges. This is illustrated in Figure 2. The lines spanning the triangle's edges correspond to three hyperplanes in $\mathbb{P}^5$. These hyperplanes split the Plücker space into cells. Any line stabbing the triangle will map to a Plücker point located in the same cell, which corresponds to the intersection of the three

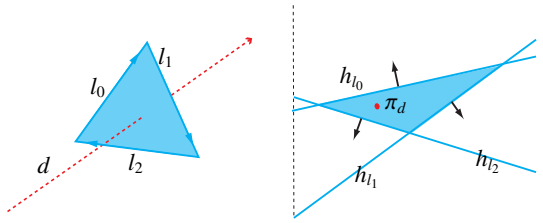

Figure 2: Left: a triangle in 3D space. l_0, l_1, l_2 are the lines spanning the triangle's edges. d is a line stabbing the triangle. Right: h_{l_0}, h_{l_1} and h_{l_2} are the Plücker hyperplanes mapped from l_0, l_1 and l_2 respectively. π_d is the Plücker point mapped from the stabbing line d. π_d has a consistent orientation in respect to l_0, l_1, l_2. From a geometrical point of view, π_d lies at the intersection of the half spaces induced by h_{l_0}, h_{l_1} and h_{l_2}. Thus, these 3 hyperplanes provide an analytical representation of all the lines stabbing the triangle.

half spaces induced by the hyperplanes. Therefore, this cell contains all the lines (ie. their Plücker points) stabbing the triangle, whereas the other cells contain all the lines missing the triangle. This provides an analytical representation of all the lines intersecting a triangle.

This property has been extended to any set of convex polygons by Pellegrini [2, 3]. Let T_{set} be a set of oriented triangles (or convex polygons). The Plücker hyperplanes corresponding to the lines spanning the triangles edged divide the Plücker space into cells. This builds an arrangement, having the following property: All the points belonging to the same cell have the same sign with respect to its bounding hyperplanes. Therefore, in Plücker space, all the lines (i.e. their Plucker points) belonging to the same cell intersect the same subset of triangles in T_{set}. The decomposition of Plücker space into cells allows to group lines together according to the subset of triangles they intersect. This defines an equivalence relation on lines. As a consequence, each cell corresponds to an equivalence class. To sum up, the Pellegrini approach allows an exact and analytical representation of all the sets of lines generated by a set of triangles.

It is important to note that these equivalence classes correspond to an information of occlusions, rather than one of visibility. Pellegrini's definition provides the information of which lines intersect which subset of triangles, but not the order in which these triangles are intersected. Speaking in terms of visibility, using this definition, we cannot know which triangle represents the first visible geometry along a view direction. Therefore, in order to represent the visibility from a polygon, Pellegrini's definition is no longer sufficient. The lines need to be grouped according to the first triangle they intersect.

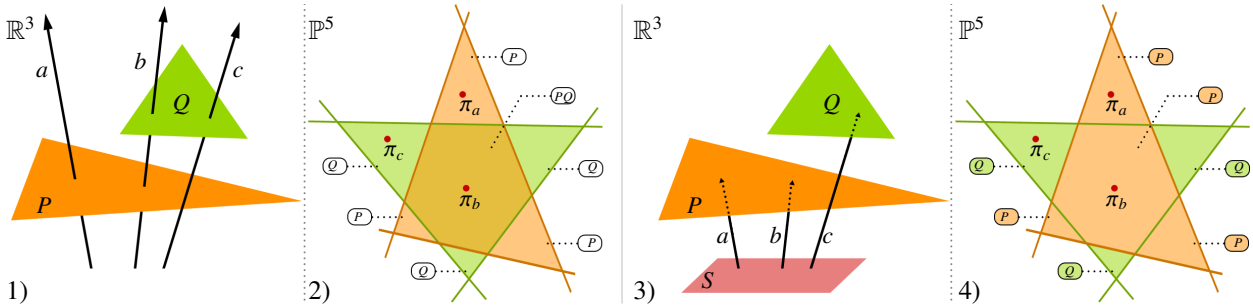

Figure 3: Left (1,2): Equivalence classes according to Pellegrini. 1) Two triangles (P, Q) and three lines (a, b, c) in various configurations. 2) The arrangement of hyperplanes (illustrated by 6 2D lines) mapped from the triangles' edges. Filled cells are set of lines intersecting at least one triangle. The intersected triangles are associated with the corresponding cells. π_a, π_b and π_c are the Plücker points mapped from a, b and c, respectively. They are located in the cells according to the triangles they stab. For example, π_b has a consistent orientation with respect to the 6 hyperplanes, since b intersects the two triangles. However, there is no indication on which is the first intersected triangle. Right (3,4): Our new classes in the context of visibility from a surface S. 3) The three lines (a, b, c) can be associated to visibility rays originating from S. 4) The triangles formerly associated with the central cell have been depth sorted to find the first intersection with respect to S. Thus, only P is associated with the cell and no ambiguity remains on which is the first triangle intersected by b.

The next section contains a theoretical description of our method. We modify Pellegrini's definition in order to describe a visibility information and we demonstrate how this information can be stored and exploited in the context of ambient occlusion computation.

4. Our method

4.1. From Occlusion to Visibility: Classes of Rays

We aim at describing the visibility from a surface S, over a set of triangles. Without loss of generality, we assume that the triangles are not intersecting each other. Otherwise, the existing intersections can be handled as a preprocess step.

Pellegrini's definition can be restrained to the view rays originating from a surface S. Each cell in the arrangement will thus contain all the rays stabbing both S and the triangles associated with the cell. Since these triangles do not intersect each other, one of them is stabbed before all the other by all the rays in the equivalence class. Moreover, all the other triangles in the cell are behind the support plane of this triangle, with respect to S. Thus, they can be sorted in order to find the first intersected one.

In conclusion, the rays originating from S can be grouped together according to the first triangle they intersect. The result is continuous sets of rays. This corresponds to an analytical representation of the triangles directly visible from S. Therefore, we have a new visibility equivalence relation, with respect to a surface. Each cell of the new partition corresponds either to a single visible triangle, or to an empty zone, where no geometry is visible from S. The triangle associated with a non-empty cell is the first triangle intersected by a coherent group of rays originating from S. Figure 3 gives a comparison between Pellegrini's definition and our new visibility classes.

It is important to note that each visibility class contains both a directional and a depth information (since the rays are grouped with respect to the geometry in the scene and according to the first triangle they stab).

From an algorithmic point of view, the visibility classes can be stored using a binary space partitioning (BSP) tree, since the visibility classes are defined by an arrangement of hyperplanes. The inner nodes contain the hyperplanes corresponding to the lines spanning the triangles' edges, whereas each leaf represents a cell of the arrangement, and thus a visibility class. This can either correspond to a set of rays missing all the triangles, or to a set of rays intersecting the same first triangle. In the latter case, the triangle is associated with the leaf. Therefore, our data structure is actually a ray partition structure, $RP(S)$.

Up to this point, an analytical representation of the visibility from a surface has been described, as well as the structure suitable to encode it. The next subsection focuses on how $RP(S)$ can be used to extract the exact visibility from any point on S.

4.2. Extracting the Exact Visibility Information

Let *RP(S)* be a structure which stores the visibility from the surface S over a set of triangles, as previously described. Given a point x on S, we want to compute

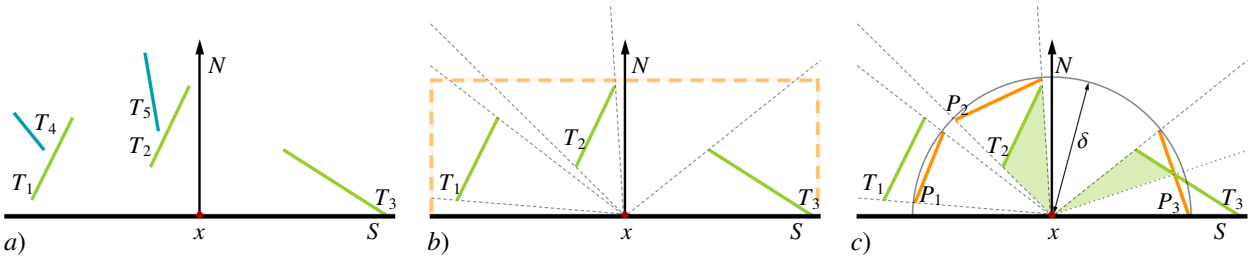

Figure 4: Extracting and using the visibility information for point x. **a)** The triangles T_1, T_2, T_3, T_4 and T_5 are visible from the surface S. This information is contained in the data structure. However, both T_4 and T_5 are not visible from x. **b)** The point x and the bounding box faces form the initial view beams, which are used to guide the visibility extraction process for x. **c)** Three triangles (T_1, T_2 and T_3) are entirely visible from x. They subtend the same solid angle as the patches P_1, P_2 and P_3, respectively. Thus they produce the same ambient occlusion. δ is the maximum occlusion distance, beyond which a visible triangle no longer influences the AO computation. This extra condition allows to completely reject triangle T_1 and to determine the part of T_3 which can be taken into account.

its ambient occlusion value, using Equation 2. Therefore, we need to find the exact visible geometry from x. The data structure already contains all the visibility information from S. Thus, the visibility from x does not need to be calculated, but extracted. This is equivalent to finding the classes corresponding to the rays originating from x only. The need for such an operation is explained in Figure 4a.

Considering a ray originating from x, we want to determine its visibility class. This is equivalent to locating the ray's corresponding Plücker point into the data structure. The position of the point is tested against the hyperplanes contained in the inner nodes, until a leaf is reached. If a triangle is associated with the leaf, this triangle is visible from x in the direction of the ray. If not, the ray does not intersect any geometry.

The above operation is generalized to the infinite set of rays contained in a view beam. In order to test the continuous set of rays originating from x we can construct the initial beams using the faces of the bounding box containing all the geometry visible from S (see Figure 4b). Let P be such a face. The set of rays contained in the beam defined by x and P needs to be tested against the hyperplanes contained in the inner nodes. This process is illustrated in Figure 5 and consists in the following steps: x and the line corresponding to a given hyperplane (noted hp) form a plane. Testing the orientation of the set of rays stabbing P with respect to hp is equivalent to testing the position of P with respect to this plane. If P lies in the positive (or negative) halfspace induced by the plane, the set of rays has a positive (or negative, respectively) orientation with respect to the hyperplane hp. Otherwise, P can be subdivided by the plane into two convex fragments, leading to two sets of rays, one with a positive orientation, the other with a negative orientation.

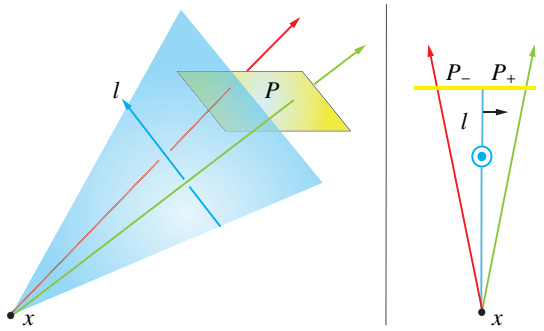

Figure 5: Illustration of how a beam (set of rays), defined by x and P can be divided into two coherent sets of rays. The point x and the line l define a plane that sets apart the lines having a positive or negative orientation in respect to l. As a consequence, if the polygon P is split by the plane, we can compute the two relevant polygons P^+ and P^- so that they represent respectively the positive and negative subset of lines with respect to l.

By repeating the above procedure for all the initial faces, all the view rays are located into the leaves of $RP(S)$. The result is a list of patches (fragments of initial faces), each one corresponding to a coherent set of rays belonging to a visibility class. Thus each patch represents a continuous set of rays intersecting the same first triangle. Note that the patches only represent view directions, serving as guide for the rays issued from x, without altering the data structure.

The next subsection gives further details on how this exact visibility information is used to compute the ambient occlusion.

4.3. Calculating the Ambient Occlusion

Up to this point, the visibility information has been extracted for point x using the data structure $RP(S)$. However, since ambient occlusion is a local property,

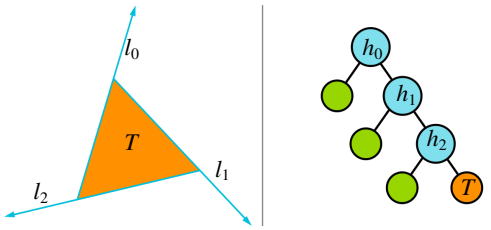

Figure 6: Left: l_0, l_1, l_2 are the lines defined by the edges of a triangle T. h_0, h_1, h_2 are their dual hyperplanes in Plücker space. Right: The corresponding space partition. The four leaves represent the four visibility classes generated by the triangle: Three *free* classes (left leaves) corresponding to the rays missing the triangle, and one *visible triangle* class (right leaf) corresponding to the rays intersecting T (T is associated with this leaf).

it is only calculated for the geometry situated within a given radius from the point. Therefore, the visibility information previously obtained needs to be limited to this local environment.

Let δ be the maximum occlusion distance for the points on surface S (and thus for x). Let $hemisphere(x, \delta)$ be the upper hemisphere centered at x and having a δ radius. Thus, if T is a triangle visible from x, we need to calculate its position with respect to $hemisphere(x, \delta)$. Without loss of generality we can extract the initial visibility for x (Section 4.2) using a patch representation of $hemisphere(x, \delta)$ instead of a bounding box. The process is the same as described in Section 4.2, except for one detail: Before applying Equation 2 to a fragment located in a leaf, we need to test its position with respect to the triangle associated with the leaf. Let P be a patch which has been located into a leaf having T as associated triangle. Three cases can occur (see Figure 4c for an illustration):

- T is in front of P, with respect to x. Thus, the visible fragment of T lies completely within the $hemisphere(x, \delta)$ and contributes to the ambient occlusion for x. Equation 2 can be applied to P.
- T is behind P, with respect to x. Thus, the visible fragment of T lies completely outside the $hemisphere(x, \delta)$ and has no contribution to the ambient occlusion for x.
- T intersects P. Thus, the visible fragment of T intersects the $hemisphere(x, \delta)$ and it partially contributes to the ambient occlusion for x. In this case, P is clipped against T and Equation 2 is applied to the resulting fragment.

The above discussion was made under the assumption that locating T with respect to $hemisphere(x, \delta)$ is equivalent to calculating the position of T with respect to P. This point is further discussed in Section 5.2.

In conclusion, we have defined a method which encodes the visibility from a surface, extracts the visibility for a point on the surface, and uses this information to calculate the ambient occlusion value for the point. The next section will focus on the practical aspects of our method.

5. Algorithm

5.1. Overview

The previous section has detailed the two theoretical steps of our method: building a ray partition for a surface and extracting from this structure the visibility information for a point on the surface. However, in practice, the two steps are not independent. The data structure is actually built on-demand depending on when and where the visibility information is needed. This lazy construction is directed by the visibility queries so that only the required classes are computed. We distinguish three types of classes:

- A *visible triangle class*: Any class representing a set of rays stabbing the same first triangle.
- A *free class*: Any class representing a set of rays which do not stab any geometry.
- An *undefined class*: Any class that has not yet been needed by a visibility query. Thus, it is neither marked as *free* nor *visible triangle*.

We note $RP(S, T)$ the BSP representation of the visibility classes generated by T (see Figure 6). The structure of a node is defined in Algorithm 1 (lines 1-4). Each inner node contains a hyperplane, whereas a leaf may contain two types of information:

- An associated triangle. All the rays located in the leaf intersect this triangle.
- A list of potentially visible geometry. Since the data structure is developed on-demand, some leaves may temporarily correspond to *undefined* classes, thus containing rays which intersect different triangles in an unknown order. All these triangles are associated with the leaf as a list of potentially visible geometry.

A visibility query can be summarized as follows: Given a point x on a surface S and a patch P from the polygonal hemisphere representation, we want to find

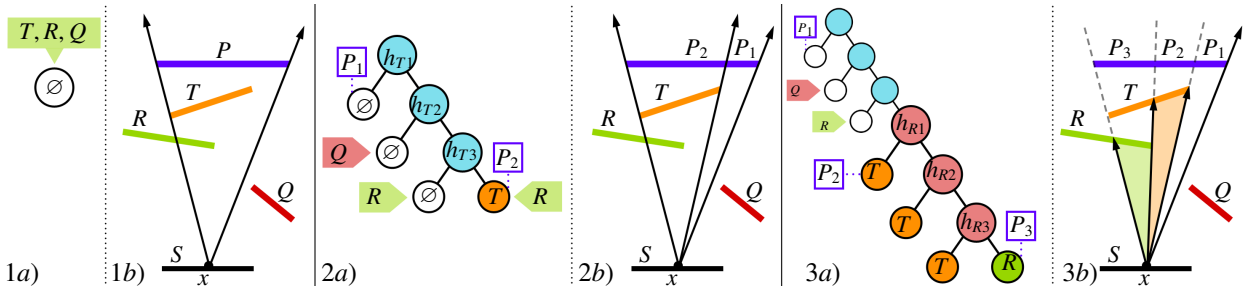

Figure 7: Illustration of the algorithm (see Algorithm 1 for pseudocode) for point $x \in S$, a patch P and three triangles (T, R, Q). **1a)** The algorithm starts with an empty leaf, associated with the potentially visible geometry (T, R, Q). **1b)** The initial beam contains all the rays originating from x and stabbing P. **2a)** One triangle is chosen randomly (T in this case) and the root leaf is replaced by $RP(S, T)$. The inner nodes contain the hyperplanes corresponding to T's edges (h_{T1}, h_{T2}, h_{T3}). The left leaves represent sets of rays missing the triangle, while the right leaf represents all the rays stabbing T. The remaining geometry (R, Q) is then located into the tree, using the technique described by the procedure *locateTriangle* (Algorithm 1). Since $poly(S \rightarrow R)$ is found to intersect h_{T3}, the triangle is sent in both left and right leaves. **2b)** Next, the continous set of rays stabbing P are located into the tree. An intersection occurs in the first inner node ($P \rightarrow P_1, P_2$), identifying thus the rays missing the triangle (P_1). The patch P_1 is discarded, while P_2 lands in the right leaf. This leaf has an associated triangle (T) and some potentially visible geometry (R). Therefore, the rays stabbing P_2 intersect T and may also intersect Q. The position of R is tested against the support plane of T. Since R is located in front of T (with respect to S), it will be used to further develop the tree. **3a)** $RP(S, R)$ replaces the right leaf. Note that the intersection information is inherited in the newly added tree. The new left leaves correspond to classes of rays intersecting T, but missing R. The new right leaf represents the class of rays intersecting both R and T, in this particular order. **3b)** P_2 is analytically split into P_2 and P_3, which are located according to the triangles they intersect.

the visible triangles from x in direction of P. We also want the exact fragments of P corresponding to each coherent set of view directions. Therefore, the algorithm subdivides P into several convex fragments, so that each one of them corresponds to a set of rays belonging to a single class. Algorithm 1 details such a query and Figure 7 applies it to a given configuration.

In the beginning, all the potentially visible geometry is associated to a single leaf. Triangles are added to the tree by replacing an existing leaf with $RP(S, triangle)$.

For each inner node, the algorithm tests the orientation of the rays stabbing P with respect to the hyperplane stored in the node (lines 27-34). This corresponds to the procedure illustrated in Figure 5 and described in Section 4.2. The *makePlane* procedure calculates the plane defined by x and the hyperplane (line 28). If P lies in the positive (resp. negative) half-space, then all the lines stabbing it have a positive (resp. negative) orientation, and the algorithm continues in the right (resp. left) subtree (lines 30-31). Otherwise, P is split against the plane and the algorithm continues recursively in both subtrees with the relevant parts of P (line 32). When a patch reaches a leaf, several alternatives can occur:

- The leaf corresponds to a *free* class. Thus, the patch represents a set of coherent rays missing all the triangles and is discarded.

- The leaf corresponds to a *visible triangle* class. Thus, the patch represents a coherent set of rays viewing the triangle associated with the leaf. Before returning or discarding the patch, its position with respect to the triangle needs to be determined according to the procedure explained in Section 4.3 and represented by the *visibleFragment* procedure (lines 39-41). It is important to note that this query is answered without developing the tree and thus taking advantage of previous queries.

- The leaf corresponds to an *undefined* class. Thus, the patch represents a set of rays which may intersect the potentially visible geometry associated with the leaf. If the leaf also contains an associated triangle (line 36), the set of rays intersects this triangle. Therefore, the associated geometry can be sorted with respect to the support plane of this triangle (line 37, *depthCulling* procedure). This eliminates all geometry located behind the intersected triangle (with respect to S and independently of x). If some geometry remains, we cannot answer the query without further developing the tree.

In order to develop the tree, the algorithm chooses a random triangle (RT) among the geometry associated with the current leaf (line 44). This triangle is merged into the tree by replacing the leaf by $RP(S, RT)$ (line 45). Next, the remaining geometry needs to be inserted into $RP(S, RT)$. We use a conservative insertion algorithm to locate each concerned triangle with respect to the hyperplanes, until it reaches a leaf (lines 46-48).

Algorithm 1 Given a point x on a surface S and a patch P, the visibility query finds the exact fragments of P corresponding to coherent set of rays intersecting the same first triangle.

```
 1: Structure Node
 2: hyperplane : [inner node] Plücker
 3: geometry : [leaf] List of potentially visible triangles
 4: class : [leaf] The visibility class: ∅ (if free class) or T (if
    visible triangle class)
 5: ____________________________________________
 6: global Polygon S
 7: ____________________________________________
 8: locateTriangle (Node n, Triangle T)
 9: if n is a leaf then
10:    n.geometry ← n.geometry ∪ T
11: else
12:    pos ← position(poly(S → T), n.hyperplane)
13:    if pos > 0 then locateTriangle (n.right, T)
14:    else if pos < 0 then locateTriangle (n.left, T)
15:       else
16:          locateTriangle (n.right, T)
17:          locateTriangle (n.left, T)
18:       end if
19:    end if
20: end if
21: ____________________________________________
22: visibilityQuery(Node n, Patch P, Point x)
23: return List of Patches
24: loop
25:    ____________________________________
26:    // Querying the data structure
27:    while n is not a leaf do
28:       Plane pl ← makePlane (x, n.hyperplane)
29:       pos ← position (pl, P)
30:       if pos > 0 then n ← n.right
31:       else if pos < 0 then n ← n.left
32:       else return visibilityQuery(n.right, P ∩ pl+, x)
                    ∪ visibilityQuery(n.left, P ∩ pl−, x)
33:       end if
34:    end while
35:    // A leaf has been reached
36:    if n.geometry ≠ ∅ and n.class ≠ ∅ then
37:       depthCulling(n.geometry, n.class)
38:    end if
39:    if n.geometry = ∅ then
40:       return visibleFragment(n.class, P)
41:    end if
42:    ____________________________________
43:    // Building the data structure
44:    RT ← random triangle in n.geometry
45:    n ← root of RP(S, RT)
46:    for T in n.geometry, T ≠ RT do
47:       locateTriangle (n, T)
48:    end for
49: end loop
```

This is achieved by the *locateTriangle* procedure, whose details are given in Section 5.2. After this step, the algorithm continues from the root of $RP(S, RT)$ until it finds the visibility class for the current fragment.

5.2. Important Points

Conservative insertion. When a triangle RT is merged into the tree, the geometry associated with the replaced leaf is inserted into $RP(S, RT)$. The *locateTriangle* procedure illustrates the insertion algorithm. Let T be one of these triangles. The visibility of T is described by all the rays stabbing both S and T. The main operation is testing the relative position of these rays with respect to the hyperplanes in the inner nodes. As stated by Theorem 1 (Section 3.2), these rays are contained in $poly(S \rightarrow T)$. Using the Theorem's corollary, we can determine the position of $poly(S \rightarrow T)$ with respect to the hyperplanes (line 12). A particular case occurs when a hyperplane intersects the polyhedron and thus the orientation of the rays is no longer coherent (line 15). As explained in Section 2.3 the two convex polyhedrons resulting from such an intersection can be calculated, at great expense and with loss of robustness. Instead of splitting $poly(S \rightarrow T)$, we consider it as belonging to both half-spaces (lines 16-17). This conservative insertion avoids expensive computations and numerical errors.

Hemisphere representation. The algorithm extracts the exact visibility from a point. However, the occlusions are only considered in a limited environment. Thus, if a visible triangle intersects the upper hemisphere, only a fragment must be taken into account. This fragment is obtained by clipping the corresponding patch against the triangle. This is the only operation dependent on the chosen tessellation. Clipping a polygon against a sphere yields a non polygonal object. Therefore, an approximation needs to be done, in order to use Equation 2. In order to minimize its error one might choose a fine tessellation. However, in practice, various choices produce the same visual effect. The explanation comes from the fact that each hemisphere patch is subdivided several times. This becomes equivalent to having a sufficiently fine initial subdivision.

Random selection of geometry. The algorithms efficiency is related to the balance of the tree. To develop the data structure, the algorithm randomly chooses a potentially visible triangle. Obviously, some choices may lead to a more balanced tree than others. However this is not predictable. Moreover, an optimal choice may not even exist. As noted in Section 4.1, each visibility class contains both a directional and a depth information. Inserting a triangle into the BSP tree is equivalent

to determine its position with respect to the hyperplanes in the node, which represent the support lines of the previously inserted triangles. Moreover, the same triangles are also used to depth sort the potentially visible geometry. Choosing the best triangle for the depth sort does not ensure an optimal choice for the directional sort and vice-versa. A random choice gives better results, and more important, it has a consistent behavior whatever the scene is.

5.3. Falloff Function

When calculating obscurances instead of ambient occlusion, the visibility function in Equation 1 is replaced with a falloff function. The latter attenuates the contribution of a triangle depending on its distance with respect to the point for which the computations are being made. However, this replacement results in a new integral for which a closed form solution may not even exist. Therefore, in our implementation, we chose a falloff function which weights the ambient occlusion value for a visible polygon using an average distance between the point and the triangle (see Equation 3). This distance is calculated using the barycenter of the visible geometry (see Equation 4). If the area of this triangle is too important, it can be divided, in order to have a better approximation.

$$AO'(x,T) = F(x,T)AO(x,T) \tag{3}$$

where

$$F(x,T) = \sqrt{\frac{dist(x, barycenter(T))}{\delta}} \tag{4}$$

Our tests showed that this solution provides high quality results at a negligible cost.

5.4. Framework

In order to illustrate the efficiency and the reliability of our exact visibility algorithm, we plug it into a ray-tracing rendering software for computing high quality ambient occlusion. For the completeness of this work, we describe the functioning of our framework.

- Using the primary rays, all the image points are grouped together with respect to the triangle they belong to. This builds a list of visible triangles.
- For each visible triangle, an empty tree is created and associated with the potentially visible geometry.
- Next, for each image point, we build a patch representation of its upper hemisphere. For each patch we use Algorithm 1 to compute the visible parts of the nearby geometry. In the end, all the visible fragments of geometry are used to analytically compute the ambient occlusion.

Black box. This framework provides an example on how our visibility algorithm (Algorithm 1) can be used. It is important to note that our method is designed as a black box, which can be integrated with a rendering application. The input data consists of the points to be shaded, grouped by their source surfaces, as well as the geometry of the scene. The algorithm returns, for each point, its final ambient occlusion value.

Local complexity. For each triangle visible from the camera, we are only taking into account its local environment. The data structures are built successively and independently per triangle. This treatment ensures that the global size of the scene has little impact on the computations. The only important factor is the local complexity specific to each visible triangle. In addition, computing the visibility per triangle limits the memory consumption since each data structure is deleted as soon as all the related image points have been shaded. Moreover, the implementation can be easily multi-threaded: A thread gets a visible triangle from the list, computes the ambient occlusion for its image points and starts over until the list becomes empty.

Geometry selection. The selection of the potentially visible geometry for a surface S is done using a hierarchical sphere culling process [37]. This is done once for each surface. Therefore in order to select all the necessary geometry for each point on S, we compute the bounding sphere of S and increase its radius by δ. Each triangle is tested against this sphere, using the approach described in [38]. Any triangle intersecting the sphere is considered as potentially visible. A second test eliminates the triangles located below S (with respect to the camera) and those above a plane parallel to S and situated at a distance of δ.

6. Results

All tests were run on 4 cores of a 2.67 GHz Intel Core i7 920 processor with 6GB of memory. The images were rendered at 1280×960 pixels. We compared the performance of our method against the Mental Ray® rendering software (abbreviated MR through the rest of this paper). Our choice was motivated by the fact that MR is a well known, high quality ray tracing production application. Comparing our approach to previous works

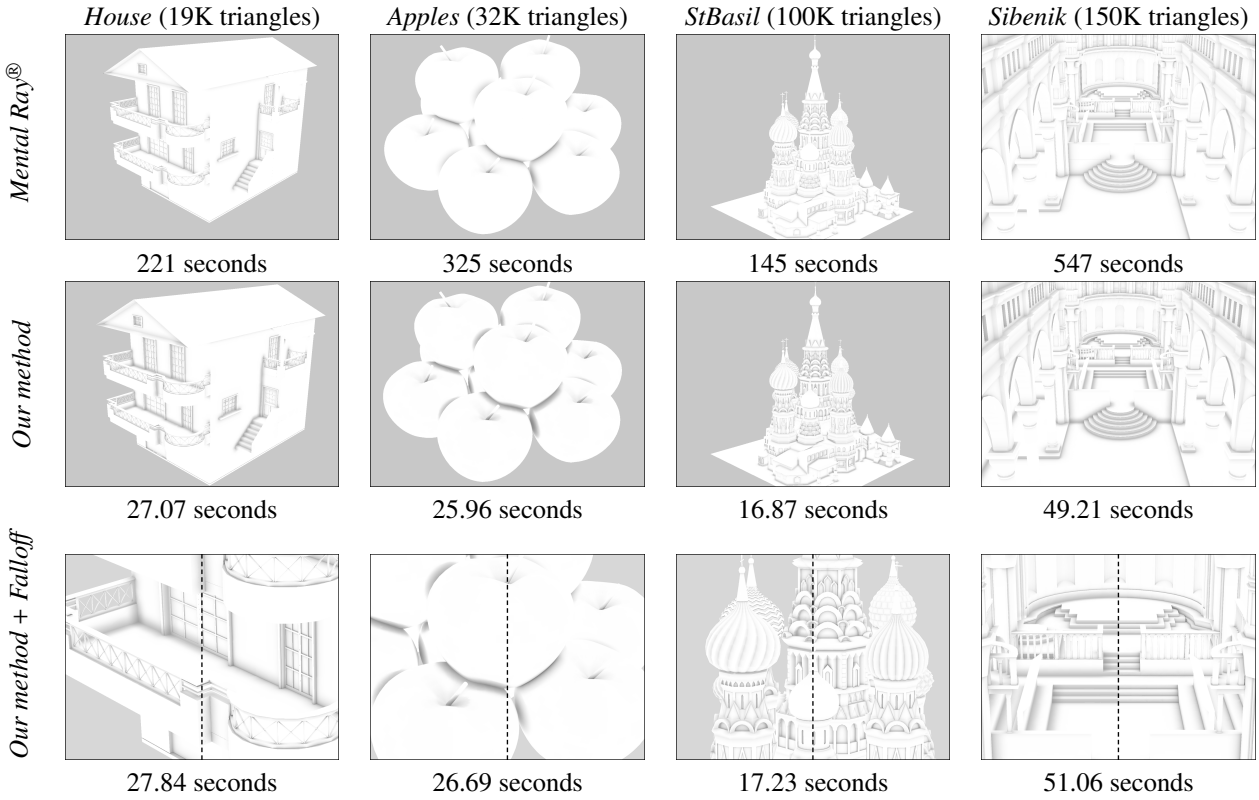

Figure 8: The first row presents the results obtained using MR with 1024 occlusion rays per pixel. The second row contains the images achieved using our method. At comparable quality, our method yields better rendering times than MR. The third row offers a comparison between our base version (ambient occlusion, left part of the image), and our enhanced one (with falloff, right part of the image). The extra computational time required to apply the falloff function counts for less than a second for each scene, thus representing a negligible cost.

	Our Method					**Mental Ray®**	
	Memory	**Time**				**Time**	**Comparison**
	Max Size 1th (4th) [MB]	Time / Point [ms]	Init [%]	Queries [%]	Total [s]	MR Total [s]	Acceleration Factor
House	9.02 (36)	0.0488	3.95	96.05	27.07	221	× 8.16
Apples	3.33 (13)	0.0586	3.54	96.46	25.96	325	× 12.51
StBasil	4.43 (18)	0.0606	6.78	93.22	16.87	145	× 8.59
Sibenik	8.87 (36)	0.0476	3.03	96.97	49.21	547	× 11.11

Figure 9: Memory and time consumptions for our algorithm, as well as the time obtained using MR. The first column represents the maximum memory load reached during the process. The first value corresponds to the memory load for a single thread. The second value is the maximum obtained by summing the maximum values reported for the 4 threads during an execution. The *Time/Point* column shows the average time spent on calculating the ambient occlusion of an image point, using our data structure. The *Init* column gives the time percentage spent on initializing the trees, while the *Queries* column gives the percentage spent querying the trees. The *Total* column gives the time for the whole process. The *MR Total* column indicates the time achieved by MR, while the last column (*Acceleration Factor*) provides the acceleration ratio between our method and MR.

based on Pellegrini's approach is not conceivable. They compute occlusion instead of visibility, except in [26]. For all the previous methods, robustness issues restrict their application to environments of moderate size and complexity. In contrast, comparing to MR, we want to show that our approach is robust, scalable and competitive with respect to a standard production solution.

Since we seek for high quality ambient occlusion through analytical computations, we also include a visual comparison with the Ambient Occlusion Volumes [12] technique (abbreviated AOV through the rest of this paper). To our knowledge, AOV is currently the most

	Our Method				Mental Ray®	
	Memory		Time		Time	Comparison
δ	Max Occ / Surface	Max Size 1th (4th) [MB]	Time / Point [ms]	Total [s]	MR Total [s]	Acceleration Factor
2	455	2.43 (10)	0.014	13	459	× 35.31
5	777	5.71 (23)	0.037	38	531	× 13.97
9	1270	21.7 (87)	0.092	101	592	× 5.86
13	2136	36.85 (147)	0.146	169	643	× 3.80
17	2738	51.66 (206)	0.288	342	683	× 2.00
21	3246	64.42 (258)	0.438	527	727	× 1.38
25	3836	80.87 (323)	0.639	776	744	× 0.96

Figure 10: The variation of the parameters in Figure 9 with respect to the δ value. The *Max Occ / Surface* represents the maximum number of potentially visible polygons for a surface. All the other columns retain their previous definitions provided in Figure 9. Although both memory and time consumption increase, our method remains competitive for all the considered δ values.

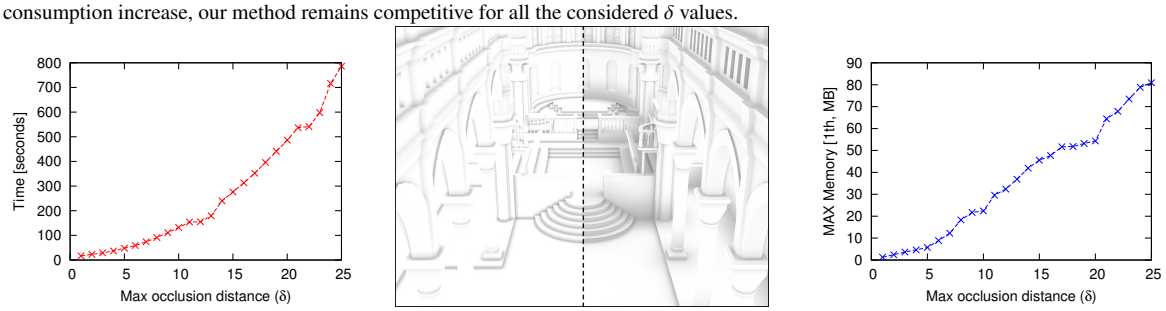

Figure 11: Increasing the maximum occlusion distance (δ). The two graphs illustrate the time consumption (left) and the maximum memory load (right) with respect to the variation of δ. The data corresponds to the values from Figure 10, *Time Total* and *Max Size*. Both graphs show that our method remains practicable, despite the increase with respect to the δ radius. The central image offers a visual comparison between two images corresponding to different δ values (left: $\delta = 2$ and right: $\delta = 25$).

accomplished analytic technique. Our intention is to show that our method does not exhibit any visual artifact contrary to AOV. The AOV algorithm was executed on a GeForce GTX 285 GPU.

Four scenes were chosen to illustrate the behavior of our method. The first one, *House*, is a moderate architectural model combining large and simple areas with some detailed features. *Apples* and *StBasil* are two models with regular meshes, but completely different visual complexities. And finally, *Sibenik* is a complex model, which is often used to illustrate different ambient occlusion techniques.

Ambient occlusion is a local property, usually applied in the final composition as a complement to a direct illumination model. Thus, we chose a δ value sufficient for obtaining visually pleasant results on which all the details are visible. We also include an analysis of the impact of δ's variation, illustrated using our most complex scene (*Sibenik*).

6.1. MR Comparison on Quality and Time

Quality. MR has been configured to produce images which are visually comparable to the results obtained using our method. We sampled the entire hemisphere, using 1024 samples. This value represents the minimum required to remove all the noise in the final images. Figure 8 shows the results of both methods.

Time. Since the images obtained using MR are visually comparable to the results achieved by our method, it is pertinent to compare the computation times required for both methods. The last three columns of Figure 9 illustrate this comparison. The considered MR execution times correspond to the ambient occlusion calculations only. It can be noticed that our algorithm is faster on all models. Moreover, as shown in Figure 10, our method remains competitive for larger δ values.

6.2. Time Analysis

The total time required by our method can be divided into two steps. The first one concerns the preliminary set-up for each surface: selecting geometry and calculating the necessary hyperplanes, computing a polygonal representation of the upper-hemisphere, and initializing the data structure. The second step involves the visibility queries which compute the needed ambient

occlusion values while developing the tree. These values can be found in Figure 9 (*Init* and *Queries* columns). We notice that the computation time is dominated by the visibility queries, which represent the core of our method.

Our method calculates ambient occlusion for each image point belonging to a visible triangle. Thus, in order to better understand the behavior of our method, we indicate the total time spent on each image (Figure 9, *Total* column), as well as the average time spent on each image point for which a data structure has been queried (Figure 9, *Time (ms) / Point* column). This column shows that although the chosen scenes have different complexities, the variation of the average time spent per image point is moderate. As previously explained, the algorithm handles the surfaces one after another. Therefore, the computations are local to each visible source triangle and limited to its close neighborhood, restricted within a δ radius. As a result, our approach escapes the global complexity of the scene.

The last row from Figure 8 show a visual comparison between the results obtained using the base (ambient occlusion) version of our method and the enhanced one (including falloff). As indicated by the rendering times, the extra computational cost is negligible. The reason for this is that the majority of the required information is already present in the data structure, and the additional calculations consist of basic operations.

6.3. Memory Analysis

Since the algorithm builds one data structure in Plücker space for each visible triangle from the camera, the memory consumption varies during the rendering process. Therefore, we report the maximum memory load that was reached for each scene (see Figure 9, column *Max Size*), for a single thread. However, since we ran our tests on 4 threads, we are also indicating the maximum memory footprint reported for four simultaneously built data structures. Figure 13 illustrates the memory consumption reported for a single thread. Although several peaks are present (corresponding to large data structures which encode a complex environment), most of the memory footprint remains low.

As shown in Figure 9 (*Max Size* column), the memory consumption is significantly different between the considered scenes. Both *House* and *Sibenik* have more important memory consumptions than *Apples* and *StBasil*. These last two models have regular meshes, while the first two are irregular. For each visible triangle, our algorithm attempts to take advantage of the visibility coherence between the points on the triangle. Thus, if a large area surface "views" an increased amount of geometry, a loss of the visibility coherence may occur. This is the case with *House* and *Sibenik*.

6.4. The δ parameter

Time. The computation time according to the δ parameter is illustrated by the first graph of Figure 11. When δ increases, our algorithm needs to consider more potentially visible geometry. This leads to an over-cost required for computing a larger data structure. As a result, the algorithm performance decreases compared to MR. This is illustrated by the last column of Figure 10. The last row reports the crossing point between our method and MR (the resulted image is presented in Figure 11, right side of center image). Although our method can handle larger δ values, the interest of performing exact ambient occlusion calculations is diminished when the distance becomes important. As explained by Laine and Karras [1], distant geometry has a very low contribution to the ambient occlusion and can be calculated using simplified geometry, without introducing artifacts or perturb the quality of the result. Therefore, we believe that an exact and analytical method should not be applied on the raw data in the far-field. In Section 6.6 we analyze how the far field occlusion can be taken into account as a possible development of our algorithm.

Memory. As previously stated, an increase in δ is equivalent to an increase in the potentially visible geometry. The third graph in Figure 11 illustrates the memory footprint with respect to δ variation. Although the memory consumption increases with the distance, it is far from being a limitation even for the largest radius.

According to Pellegrini [2, 3], the complexity of an arrangement of hyperplanes in Plücker space is $O(n^{4+\epsilon})$ in memory, where n is the number of triangles. Such a complexity may let one think that no practicable application can be build on this theoretical framework. However, our results prove the contrary. A least square fitting analysis using our experiments indicates a practical memory complexity of our application of $O(n^{1.82})$.

Local complexity of the arrangement. The practical complexity achieved by our method can be explained by the local character of the computations involved. While the arrangement from the theoretical framework concerns all the lines in Plücker space, we are limiting our computations to the arrangement of rays originating from a surface. Thus, we are only constructing a local partition of rays, with respect to the considered surface. Moreover, the lazy evaluation of the visibility avoids computing useless parts of this arrangement.

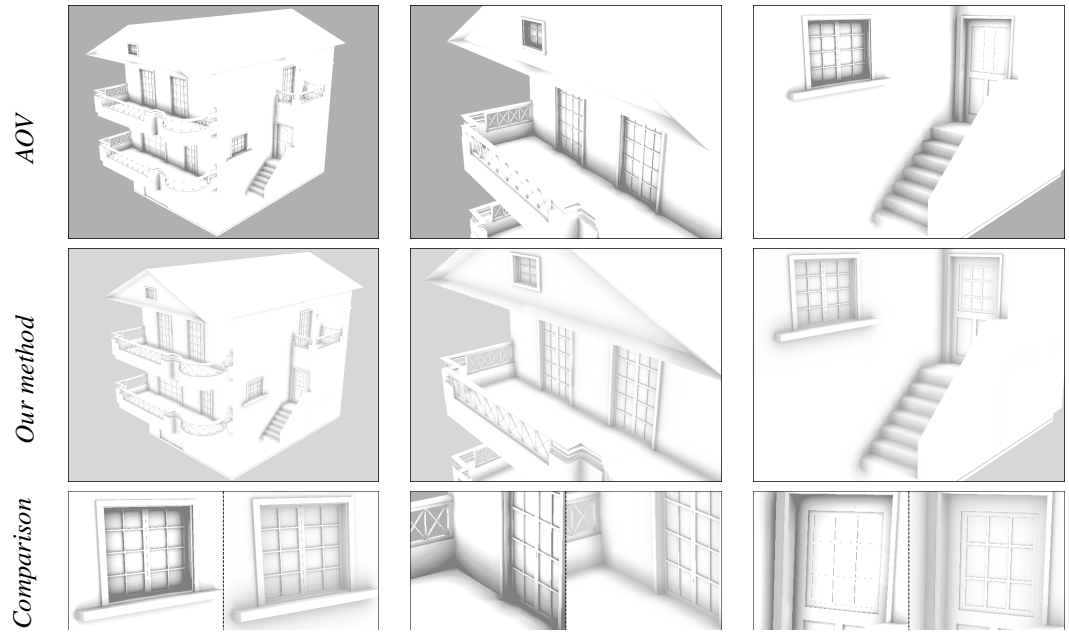

Figure 12: The first row presents the results obtained using the Ambient Occlusion Volumes[12] technique. The main artifact is over-occlusion which results in a loss of details, especially in the corners of windows or doors. The second row contains the images achieved using our method. The color variations from gray to white are smooth and all the details are present. The third row presents a comparison, which further underlines the visual imperfections of AOV and how they are avoided in our results.

6.5. AOV Comparison

Figure 12 provides a visual comparison between AOV and our method. Although both methods produce noise free results, the images obtained using AOV suffer from artifacts resulting from the approximated visibility. The main problem is over-occlusion, which has a negative impact especially in the areas where details are present, such as the windows or the doors. Since our method computes the exact visible fragments of the surrounding geometry, all the details are rendered correctly. As an indication, the main image took 221 seconds using MR, 27 seconds using our method and 0.074 seconds using AOV.

AOV is an ambient occlusion technique which makes some sacrifices on the visibility accuracy in order to achieve fast rendering times. The cost of these approximations are the visual artifacts, as illustrated in Figure 12. In contrast, our visibility algorithm provides exact informations and thus artifact free results. This accuracy comes with a cost in time. However, as shown by the previous tests, our algorithm remains competitive with respect to a standard industry solution such as MR.

6.6. Discussions and Future Work

This section outlines some of the limitations of our method, as well as some possible solutions which are to be addressed in the future.

If the considered local environment goes beyond a very important δ radius, the algorithm reports an increased memory load. More precisely, a visible polygon for which a large set of geometry needs to be analyzed will result in an expensive data structure, both in terms of memory consumption and execution time. An interesting solution would be to deal with sequential layers of geometry, instead of considering at once all the potential occluders for a surface. For a chosen δ value, we can start with the geometry contained into a δ/k radius. If some queries require further development, the geometry contained between δ/k and $2 \times \delta/k$ is added and so forth (k can be a constant or not). The advantage of such an approach is that if we decide to calculate the ambient occlusion for a radius of $\Delta > \delta$, we can reuse the previous computations.

This leads us to another possible development of our ambient occlusion algorithm. The current algorithm treats all the potentially visible geometry within a δ radius equally. No distinction is being made between the near and the far field. Therefore, it may construct a complex tree in order to add a very small occlusion effect from a far away object. Moreover, as outlined by Laine and Karras [1], such distant geometry may be used in a simplified form without altering the visual re-

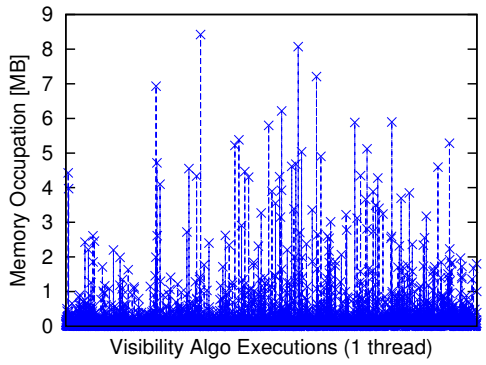

Figure 13: Memory consumption for our algorithm, during an execution, measured for a single thread. The scene is *Sibenik*. Each vertical bar corresponds to a data structure build for a visible polygon from the camera. The peaks represent large data structures, corresponding to polygons having a complex environment. However, the majority of the data structures have a low memory footprint.

sults. Since only the near geometry needs to be exact and thus processed accurately, we could adapt our current exact method to handle far field occlusion using an approximation of the actual geometry. Therefore, our algorithm would be building complete data structures only for close and relevant geometry, and approximate the occlusion from far away objects. Thus, we could improve the memory footprint and the time consumption without modifying the visual quality of the result.

7. Conclusions

In this paper we have proposed a new method which computes exact from-polygon visibility. Our approach is based on a new concept of visibility equivalence classes of lines. Previous methods present in the literature have either proposed solutions which compute the occlusion between two polygons, or limited calculations of from-polygon visibility. Moreover, since these algorithms rely on 5D CSG operations, they suffer from numerical instability and their application is restricted. In contrast, our method uses basic geometric operations to lazily encode an analytic definition of the visibility from a polygon over a set of triangles. The proposed data structure is conservatively is built during render time, while the view rays are classified with respect to the geometry they encounter. Thus, the algorithm benefits from the coherence between these rays. Our method is robust, efficient, and can be easily applied to various geometric configurations, as demonstrated by our tests. Using our from-polygon visibility approach, we have proposed an analytical solution to the ambient occlusion calculation. Our results are of high quality and noise free. Moreover, our computation times are competitive with respect to a production ray tracer.

Acknowledgments: *Sibenik* by M. Dabrovic. *StBasil* by ArchiDOM. *House* by "elias_ts" on `turbosquid.com`. *Apples* by `andeciuala.blogspot.com`.

References

[1] S. Laine, T. Karras, *Two Methods for Fast Ray-Cast Ambient Occlusion*, Computer Graphics Forum 29 (2010) 1325–1333.

[2] M. Pellegrini, *Ray-shooting and isotopy classes of lines in 3-dimensional space*, in: F. Dehne, J.-R. Sack, N. Santoro (Eds.), Algorithms and Data Structures, volume 519 of *Lecture Notes in Computer Science*, Springer Berlin, 1991, pp. 20–31.

[3] M. Pellegrini, Handbook of Discrete and Computational Geometry - 2nd edition, pp. 839–856. *Ray shooting and lines in space*.

[4] S. Zhukov, A. Iones, G. Kronin, *An Ambient Light Illumination Model*, in: Eurographics Symposium on Rendering, pp. 45–56.

[5] H. Landis, *Production-Ready Global Illumination*, in: Siggraph Course Notes, volume 16.

[6] R. Bredow, *Renderman on Film*, SIGGRAPH 2002 Course Notes Course 16 (2002).

[7] A. Méndez-Feliu, M. Sbert, *From obscurances to ambient occlusion: A survey*, The Visual Computer 25 (2009) 181–196.

[8] A. Méndez, M. Sbert, L. Neumann, *Obscurances for ray-tracing*, Eurographics 2003 Poster Presentation, 2003.

[9] A. Méndez, M. Sbert, *Comparing hemisphere sampling techniques for obscurances computation*, in: Proceedings of the International Conference on Computer Graphics and Artificial Intelligence (3IA 2004).

[10] M. Bunnell, *Dynamic Ambient Occlusion and Indirect Lighting*, in: M. Pharr, R. Fernando (Eds.), GPU Gems 2, Addison-Wesley Professional, 2005, pp. 223–233.

[11] J. Hoberock, Y. Jia, *High-Quality Ambient Occlusion*, in: H. Nguyen (Ed.), GPU Gems 3, Addison-Wesley Professional, 2007, pp. 257–274.

[12] M. McGuire, *Ambient Occlusion Volumes*, in: Proceedings of High Performance Graphics 2010.

[13] D. R. Baum, H. E. Rushmeier, J. M. Winget, *Improving radiosity solutions through the use of analytically determined form-factors*, in: Proceedings of the 16th annual conference on Computer Graphics and Interactive Techniques, SIGGRAPH '89, ACM, New York, USA, 1989, pp. 325–334.

[14] L. Szirmay-Kalos, T. Umenhoffer, B. Tóth, L. Szécsi, M. Sbert, *Volumetric Ambient Occlusion for Real-Time Rendering and Games*, IEEE Comput. Graph. Appl. 30 (2010) 70–79.

[15] M. Tarini, P. Cignoni, C. Montani, *Ambient occlusion and edge cueing to enhance real time molecular visualization*, IEEE Transaction on Visualization and Computer Graphics 12 (2006).

[16] K. Hegeman, S. Premoze, M. Ashikhmin, G. Drettakis, *Approximate Ambient Occlusion For Trees*, in: C. Sequin, M. Olano (Eds.), Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games.

[17] J. Kontkanen, T. Aila, *Ambient Occlusion for Animated Characters*, in: T. A.-M. Wolfgang Heidrich (Ed.), Rendering Techniques 2006 (Eurographics Symposium on Rendering), Eurographics.

[18] A. G. Kirk, O. Arikan, *Real-time ambient occlusion for dynamic character skins*, in: Proceedings of the 2007 symposium on Interactive 3D graphics and games, I3D '07, ACM, New York, USA, 2007, pp. 47–52.

[19] M. Mittring, *Finding next gen: CryEngine 2*, in: ACM SIGGRAPH 2007 courses, ACM, New York, USA, 2007, pp. 97–121.

[20] F. Durand, G. Drettakis, C. Puech, *The 3D visibility complex*, ACM Trans. Graph. 21 (2002) 176–206.

[21] F. Durand, G. Drettakis, C. Puech, *The visibility skeleton: a powerful and efficient multi-purpose global visibility tool*, in: Proceedings of the 24th annual conference on Computer graphics and interactive techniques, SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., New York, USA, 1997, pp. 89–100.

[22] L. Zhang, H. Everett, S. Lazard, S. Whitesides, *Towards an implementation of the 3D visibility skeleton*, in: Proceedings of the twenty-third annual symposium on Computational geometry, SCG '07, ACM, New York, USA, 2007, pp. 131–132.

[23] S. Lazard, C. Weibel, S. Whitesides, L. Zhang, *On the computation of 3D visibility skeletons*, in: Proceedings of the 16th annual international conference on Computing and combinatorics, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 469–478.

[24] D. M. Mount, F.-T. Pu, *Binary Space Partitions in Plücker Space*, in: ALENEX '99: Selected papers from the International Workshop on Algorithm Engineering and Experimentation, Springer-Verlag, London, UK, 1999, pp. 94–113.

[25] S. Nirenstein, E. Blake, J. Gain, *Exact from-region visibility culling*, in: Eurographics workshop on Rendering, Eurographics Association, 2002, pp. 191–202.

[26] J. Bittner, *Hierarchical Techniques for Visibility Computations*, Ph.D. thesis, Czech Technical University in Prague, 2002.

[27] F. Mora, L. Aveneau, M. Mériaux, *Coherent and Exact Polygon-to-Polygon Visibility*, in: Proceedings of WSCG'05.

[28] D. Haumont, O. Makinen, S. Nirenstein, *A Low Dimensional Framework for Exact Polygon-to-Polygon Occlusion Queries*, in: Eurographics Workshop on Rendering, pp. 211–222.

[29] F. Mora, L. Aveneau, *Fast and Exact Direct Illumination*, Proceedings of CGI'2005, New York, Stony Brooks.

[30] K. Fukuda, A. Prodon, *Double Description Method Revisited*, in: Combinatorics and Computer Science, pp. 91–111.

[31] D. Avis, K. Fukuda, *Reverse search for enumeration*, Discrete Appl. Math. 65 (1996) 21–46.

[32] V. Kaibel, M. E. Pfetsch, *Computing the face lattice of a polytope from its vertex-facet incidences*, Computational Geometry: Theory and Applications 23 (2002) 281–290.

[33] C. L. Bajaj, V. Pascucci, *Splitting a Complex of Convex Polytopes in any Dimension*, in: In Proceedings of the 12th Annual Symposium on Computational Geometry, ACM, 1996, pp. 88–97.

[34] J. H. Lambert, *Photometria, sive, De mensura et gradibus luminis, colorum et umbrae* , V.E. Klett, Augustae Vindelicorum :, 1760.

[35] K. Shoemake, *Plücker coordinates tutorial*, Ray Tracing News, 1998.

[36] L. Aveneau, S. Charneau, L. Fuchs, F. Mora, *A Framework for n-Dimensional Visibility Computations*, in: L. Dorst, J. Lasenby (Eds.), Guide to Geometric Algebra in Practice, Springer, 2011, pp. 273–296.

[37] P. Hubbard, *Interactive collision detection*, in: Virtual Reality, 1993. Proceedings., IEEE 1993 Symposium on Research Frontiers in, pp. 24 –31.

[38] J. Arvo, Graphics gems, Academic Press Professional, Inc., 1990, pp. 335–339.