Lazy visibility evaluation for exact soft shadows

Frédéric Mora, Lilian Aveneau, Apostu Oana, Djamchid Ghazanfarpour

Volume xx (200y), Number z, pp. 1–13

Lazy visibility evaluation for exact soft shadows

F. Mora[1], L. Aveneau[2], O. Apostu[1], and D. Ghazanfarpour[1]

[1]University of Limoges - XLIM-CNRS, France
[2]University of Poitiers - XLIM-CNRS, France
{frederic.mora, lilian.aveneau, oana.apostu, djamchid.ghazanfarpour}@xlim.fr

Abstract

This paper presents a novel approach to compute high quality and noise-free soft shadows using exact visibility computations. This work relies on a theoretical framework allowing to group lines according to the geometry they intersect. From this study, we derive a new algorithm encoding lazily the visibility from a polygon. Contrary to previous works on from-polygon visibility, our approach is very robust and straightforward to implement. We apply this algorithm to solve exactly and efficiently the visibility of an area light source from any point in a scene. As a consequence, results are not sensitive to noise, contrary to soft shadows methods based on area light source sampling. We demonstrate the reliability of our approach on different scenes and configurations.

Keywords: Analytical visibility, exact soft shadows

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Visible surface algorithms, Shadowing

1. Introduction

Shadows and especially soft shadows play an important role in visual perception as they unveil spatial relationships between objects. This makes soft shadows essential to realistic rendering. However, they are very expensive to compute since they require solving the visibility of an area light source from any visible point in a scene. The accuracy of any solution to this problem determines the soft shadows quality and, as a consequence, the realism of the scene.

Dedicated algorithms were developed to reduce the computational complexity inherent to soft shadows. Interactive solutions exist, but they make crude assumptions leading to approximate or plausible soft shadows. They are forced to sacrifice the correctness of the shadows in order to reach interactive frame rates. Geometry-based approaches can lead to fine results, using silhouette edges [LLA06, FBP08] for example. While the accuracy of the visibility computation is crucial for soft shadows, the wide majority of algorithms are based on sampled light sources. As a consequence, whatever their accuracy, they are still sensitive to noise according to the sampling strategy and density.

In this paper, we present a new high quality soft shadow algorithm based on exact visibility computed from the area light source. Our method relies on an equivalence relation allowing to group lines according to the geometry they intersect [Pel04]. We build on this theoretical framework to provide a novel and practical approach to lazily compute the exact visibility from a polygon. This visibility data allows to solve efficiently and exactly the visibility of an area light source for each point to be shaded, taking advantage of the visibility coherence between neighbour points. Since the direct illumination is analytically evaluated using exact visibility, the shadows are optimal and noise-free. While exact approaches suffer from implementation and robustness issues, our method is easy to implement, scalable and numerically robust. In addition, our results show that it outperforms an optimized ray tracer.

The paper is organized as follows: The first section outlines the works related to this paper. The second section presents the geometrical and mathematical framework we use. Moreover, we demonstrate an essential theorem to our algorithm. The third section details our exact from-polygon visibility algorithm which is the basis to solve the visibility of an area light source. At last, in section 4, we test our algorithm's behaviour in different configurations. Before con-

cluding, we discuss these results, we analyse some limitations and we consider several perspectives.

2. Previous work

Many contributions on soft shadows can be found in the literature. A survey has been presented by Hasenfratz *et al.* [HLHS03]. Recently, Eisemann *et al.* [EASW09] provide another interesting lecture. In this section, we only focus on widely used or recent physically-based algorithms able to compute very high quality or exact soft shadows. In addition, since the core of our method relies on an exact from-polygon visibility algorithm, we also focus on this topic.

Soft shadows in ray and beam tracing

Ray tracing easily supports soft shadows using shadow rays and stochastic sampling of the light sources. As a counterpart, the results are sensitive to noise. This requires increasing the number of samples in order to avoid visual artifacts. But this drastically slows down the rendering. Packet [WBWS01] and frustrum [RSH05, WIK*06] traversal techniques improve ray-tracing efficiency. They take advantage of the spatial coherence and have efficient implementations, using SIMD instructions. Moreover, they can be designed for a specific graphic architecture [BW09]. Despite these improvements, soft shadows remain a challenging task. If the sample distribution can be improved using importance or interleaved sampling, the results remain sensitive to noise.

Beam tracing [HH84] methods are suited for soft shadows computation. A single beam is cast to the light to solve exactly its visibility. As a consequence, results are not sensitive to noise. Beam tracing is usually considered as lacking robustness, which can be improved by adaptive methods [GH98]. Overbeck *et al.* [ORM07] propose a robust beam-triangle intersection to handle more complex models with soft shadows.

Packets, frustrum or beam techniques take advantage of the spatial coherence from each point to be shaded. Our algorithm uses the spatial coherence from the light source. This is an important difference since the shading of the points is not an independent operation anymore.

Silhouette based soft shadows

Akenine-Möller and Assarsson [AMA02, AAM03] propose the soft shadow volume algorithm on graphics hardware. It extends the shadow volume algorithm [Cro77] using penumbra wedges. A wedge is the place where the light source view can change because of a silhouette edge. Given a point inside a wedge, the relevant silhouette edge is projected onto the light to determine the light samples visibility. However, overlapping silhouette edges lead to overestimating the shadow and silhouette edges are underestimated since they are computed only from the center of the light source. As a consequence, soft shadows are approximated.

The soft shadow volume algorithm is extended to offline ray-tracing by Laine *et al.* [LAA*05]. All the silhouette edges are taken into account and the algorithm deals properly with the overlapping ones. Special rules determine the light samples affected by each silhouette edges. Then a single ray is shot to recover the samples visibility. Lehtinen *et al.* [LLA06] remark that this algorithm can be overconservative and less efficient than a classical ray-tracer. They change the data structure used to store the silhouette edges, improving both the algorithm efficiency and its memory consumption.

Forest *et al.* [FBP08] build on the works of [AAM03] and [LLA06]. They remove the shooting ray step and provide a dedicated implementation on graphics hardware allowing several frames per second on moderate scenes using an average of 16 light samples per light sources.

Research on silhouette based algorithms has lead to very efficient solutions. However, they require triangles connectivity with consistent orientation. Since these methods are object-based, visibility computations are accurate. But they are still subject to noise because of the light source sampling. The sampling strategy has a significant impact as shown in [FBP08].

Other methods

Laine *et al.* [LA05] change the pre-processing order of ray tracing. Instead of finding a triangle hit by a shadow ray, they search all the shadow rays intersecting a given triangle. Notice that the same idea was used to provide interactive GPU algorithms based on triangles [ED07, SEA08] or extended to silhouette edges [JHH*09]. However, approximations are made to reach such frame rates. In this work, we also focus on the sets of lines intersecting triangles. However, we use an analytical representation of those sets.

Exact from-polygon visibility

Exact solutions to visibility problems are important to better understand its underlying nature, and design new algorithms. However, exact from-polygon (and thus from-region) visibility is very complex, because it is a four-dimensional problem [DDP02]. There are few solutions in the literature. They all rely on the same theoretical framework described by Pellegrini [Pel04] in the Plücker space, a 5D space of lines. His analysis provides a worst case complexity of $O(n^4 log\, n)$ in space and time, where n is the number of triangles. This underlines the complexity inherent to the visibility in space. To remain practicable, a very special care must be given to the algorithm design. As an example, a first approach is proposed by Mount and Pu [MP99]. 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.* [NBG02] and Bittner [Bit02] (see [MAM05] for details about the differences between the two solutions), and further improved by Haumont *et* al. [HMN05] and Mora *et al.* [MA05] respectively. It is worth underlying a major difference between the two approaches initiated by Nirenstein and Bittner respectively :

- Nirenstein and Haumont design their algorithms to solve

whether two polygons are mutually visible. Thus the algorithm output is a boolean value. Whenever possible, the computation of the whole visibility set is avoided since the process can be terminated as soon as a visibility is found.

- Mount, Bittner and Mora algorithms aim to compute and store all the visibility information. In this case, the output is a partition of the Plücker space encoded in a Binary Space Partitioning tree.

Despite this difference, they all rely on Computing Solid Geometry in the Plücker space which is computationally very expensive. As a consequence, they are only used as a pre-process step. In addition 5D CSG operations are very complex to implement and prone to numerical instability. Thus, the reliability and the scalability of these solutions is restricted.

The soft shadow algorithm presented in this paper relies on a novel approach to capture the visibility from a polygon. It also leads to the construction of a BSP tree in the Plücker space. But in contrast to the previous works, the tree is built lazily and at run time, not as a pre-process step. Visibility is computed on-demand when and where it is required, according to the image resolution. It does not rely on any expensive and complicated 5D CSG operation. This makes the algorithm very easy to implement, efficient and computationally robust, whereas previous methods are complex and suffer from numerical instability.

3. Geometrical basis

In this section we introduce the geometrical knowledge underlying the work presented in this paper. In particular, we demonstrate a result on the orientation of the lines stabbing two polygons. This result is important for our algorithm.

3.1. Plücker's coordinates

The Plücker space is a five dimensional projective space, denoted $\mathbb{P}^5$, well known in computer graphics as an efficient solution for dealing with real 3D lines [Sho98]. We only recall here the properties used in this paper. Let's consider an oriented 3D line l going through two distinct points p and q of coordinates (p_x, p_y, p_z) and (q_x, q_y, q_z) respectively. The line l maps to the Plücker point denoted $\pi_l \in \mathbb{P}^5$. The six coordinates of π_l, denoted $(l_0, l_1, l_2, l_3, l_4, l_5)$ are defined by:

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

Any Plücker point is in bijection with its dual hyperplane:

$$h_l(x) = l_3 x_0 + l_4 x_1 + l_5 x_2 + l_0 x_3 + l_1 x_4 + l_2 x_5 = 0$$

with $x \in \mathbb{P}^5$ with coordinates $(x_0, x_1, x_2, x_3, x_4, x_5)$. Thus, any line l in the 3D space can be mapped to the Plücker space as a point π_l or its dual hyperplane h_l. We now recall the definition of the so-called *side operator*:

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

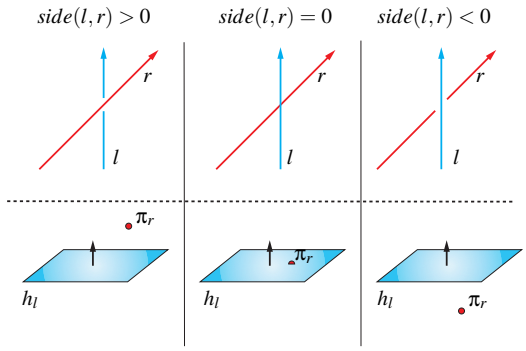

Figure 1: *Above: The relative orientation of two lines is given by the sign of the side operator. Under: Since $side(l,r) = h_l(\pi_r)$, applying the side operator comes down to testing the position of the Plücker point of one line against the dual hyperplane of the other line*

where l and r are two 3D lines, $(l_0, l_1, l_2, l_3, l_4, l_5)$ and $(r_0, r_1, r_2, r_3, r_4, r_5)$ are respectively the coordinates of their Plücker point, π_l and π_r. The sign of the side operator dertermines the relative orientation of the two lines. In particular, two lines are incident (or parallel) if their side operator equals zero. We can notice that: $side(l,r) = h_l(\pi_r)$. This leads to the geometrical interpretation of the side operator as depicted in Figure 1.

3.2. On the lines intersecting the same triangles

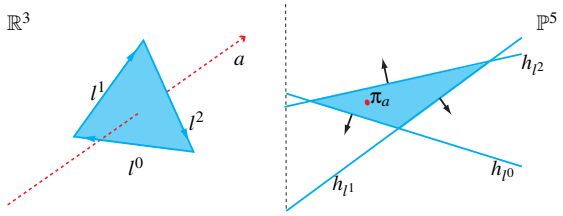

Figure 2: *Left: A triangle in 3D space. l^0, l^1, l^2 are the lines spanning the triangle's edges. a is a line stabbing the triangle. Right: h_{l^0}, h_{l^1} and h_{l_2} are the Plücker hyperplanes mapped from l^0, l^1 and l^2. π_a is the Plücker point mapped from the stabbing line a. π_a has a consistent orientation with respect to l^0, l^1 and l^2. From a geometrical point of view, π_a lies at the intersection of the halfspaces 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.*

A direct application of the side operator is an easy and robust line-triangle intersection test. A line intersects a triangle if its orientation is consistent with respect to the lines spanning the triangle edges. This is depicted in Figure 2. Beyond this intersection test, we can notice that the 3 hyperplanes related to the triangle edges divide the Plücker space into

cells and one of them contains all the lines (*i.e.* their Plücker points) stabbing the triangle, while the other cells contain the lines missing the triangle.

Pellegrini [Pel04] develops a more general approach. He

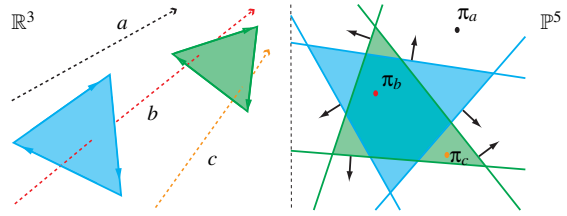

Figure 3: *Left: 2 triangles and 3 lines a, b, c in various configurations. Right: The arrangement of hyperplanes (illustrated by 6 2D lines) mapped from the 6 triangles edges. They divide the Plücker space into cells. Filled cells are set of lines intersecting at least one triangle.* π_a, π_b *and* π_c *are the Plücker points mapped from a, b and c. They are located in the cells they belong to, according to the triangle(s) they stab. For example,* π_b *has a consistent orientation with respect to the 6 hyperplanes since b intersects the two triangles. Its relevant cell holds all the lines intersecting the two triangles.*

uses the Plücker space as a framework to provide theoretical bounds on various problems involving lines. Let S be a set of several triangles (or convex polygons) and L_S be the lines spanning the triangles edges in S. Each line in L_S can be mapped to Plücker space as a hyperplane. This builds an arrangement in Plücker space: A decomposition of the space into cells by a set of hyperplanes. All the points in a same cell satisfy the following property: They all have the same sign with respect to its bounding hyperplanes. This is illustrated in Figure 3 with 2 triangles. Thus, in Plücker space, all the lines (*i.e.* their Plücker points) belonging to the same cell intersect the same subset of triangles in S. Notice that this subset can be empty if the lines miss all the triangles in S.

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. Each cell corresponds to an equivalence class (sometimes called isotopy class [Pel91] or orientation class [CEG*96]). To sum up, the Pellegrini approach allows an exact and analytical representation of all the sets of lines generated by a set of triangles S, using the set of lines L_S.

3.3. Orientation of lines intersecting two polygons

We focus on the set of lines intersecting two convex polygons. We prove the following theorem:

Theorem 1 *Let A and B be two convex polygons with n and m vertices respectively. We define* $vvset = \{v_{ij}, i \in [1,n], j \in [1,m]\}$ *the set of the lines* v_{ij} *defined by one vertex of A and one vertex of B. Let l be any line and q a line intersecting A and B:*

$$side(l,x) \geq 0,\ \forall x \in vvset \Rightarrow side(l,q) \geq 0,$$
$$side(l,x) \leq 0,\ \forall x \in vvset \Rightarrow side(l,q) \leq 0.$$

In other words, if all the vertex-to-vertex lines of two polygons have a positive (resp. negative) orientation with respect to any line l, all the stabbing lines of the two polygons will have a positive (resp. negative) orientation with respect to l. This is also illustrated by Figure 5. We refer the reader to the Appendix at the end of this paper for a detailed proof of the theorem 1.

Using this theorem, we can determine if the set of lines stabbing two polygons has a consistent orientation with respect to a given line. The visibilty algorithm presented in the next section uses this essential result.

It can be proved that the Plücker points of the lines in *vvset* are the vertices of the smallest convex polyhedron in the Plücker space containing the stabbing lines of A and B. A demonstration can be found in [CAF06, ACFM11], but it requires advanced knowledge in geometric algebra. For the understanding and the correctness of this work, the theorem demonstrated in this paper is sufficient.

4. Algorithm

Since equivalence classes are continuous sets of lines that hit/miss the same subset of triangles, they represent coherent paths through the scene independently of any viewpoint. Therefore, two lines belonging to the same equivalence class are spatially coherent. We use this property to build a Plücker space partition representing the visibility of an area light source.

4.1. Overview

We consider a convex area light source L, a triangle T, and define their occluders O as the triangles intersecting their convex hull. To represent the light source visibility, we focus on the sets of lines intersecting L and which either miss all the occluders in O, or hit at least one occluder in O. Thus, we define:

- A *visible class*: Any equivalence class representing a set of lines that do not intersect any occluders.
- An *invisible class*: Any equivalence class representing a set of lines that intersect at least one occluder.
- An *undefined class*: An equivalence class that is not yet found as *visible* or *invisible*.

Our algorithm builds a BSP tree in Plücker space, providing a hierarchical representation of the equivalence classes generated by the occluders. Each leaf represents one of these three classes. The algorithm is lazy: The BSP tree is grown on-demand depending on when and where visibility information is needed. The construction only relies on two operations: Inserting an occluder into the tree and growing the

tree. The former allows to find the leaves affected by an occluder in the tree, while the latter grows the tree by replacing a leaf with an occluder's equivalence classes.

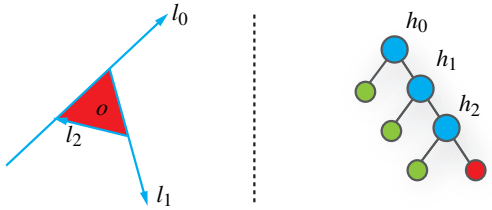

Figure 4: *Let l_0, l_1, l_2 be the lines defined by the edges of an occluder o, and h_0, h_1, h_2 their dual hyperplanes in Plücker space. We can build a BSP tree whose leaves are the four equivalence classes generated by the triangle: Three visible classes (left leaves) and one invisible class (right leaf). We note $bsp(o)$ such a representation.*

Growing a BSP tree:
Let o be an occluder, we note $bsp(o)$ the BSP representation of the equivalence classes generated by o (see Figure 4). The tree grows each time an occluder is merged: The visible/invisible classes generated by the occluder are added to the tree. Merging o into the tree consists in replacing a leaf by the root of $bsp(o)$. As a consequence, each inner node contains a hyperplane corresponding to an occluder's edge.

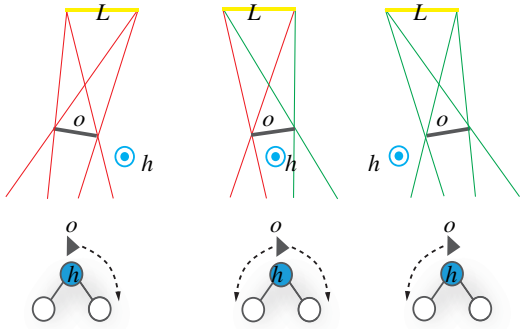

Figure 5: *2D illustration of an insertion using Theorem 1. An occluder o intersects a subset of all the lines originating from the light L. This subset may have a negative (left example), a positive (right example) or a mixed (center example) orientation with respect to any hyperplane h from an inner node. Theorem 1 allows to define this orientation using the vvset lines, i.e. the lines defined by one vertex of o and one vertex of L. As a consequence the occluder o is inserted into the right or left or both subtrees of the node."*

Inserting an occluder:
Inserting an occluder relies on the Theorem 1. The procedure is illustrated by Figure 5 and detailed in Algorithm 1 by the `insertOccluder` function. Occluders are inserted into the tree and located into the leaves (and thus the classes) they may affect. Inserting an occluder o comes down to testing the relative orientation of a hyperplane (from an inner node) and the lines occluded by o. Since the occlusion created by an occluder o is the set of lines intersecting both L and o, we use Theorem 1 to determine the orientation of the occluded lines with respect to the hyperplane. In Algorithm 1, the function `insertOcluder` (line 9) describes the process. If the current node n is a leaf which is not an invisible class, it is affected by the occluder o and thus it is stored in the leaf (line 12). Otherwise, if n is an inner node, we have to test the relative orientation of the lines blocked by o with respect to $n.hyperplane$, the hyperplane contained in the node n. This relies on Theorem 1 using $o.vvset$, the lines defined by one vertex of o and one vertex of the light (line 14). If the orientation is positive (resp. negative) o is inserted in the left (line 15) (resp. right (line 17)) child of n. Otherwise, o is inserted in the both children (line $19-20$). Those are the 3 cases illustrated by Figure 5.

The BSP tree construction is driven by the visibility queries in order to compute only the required equivalence classes. The following section presents this process and gives the details of our visibility algorithm. Next, we describe how it is used in our soft shadow framework.

4.2. Visibility algorithm

We consider the following visibility query: Given a point xyz on the triangle T, we want to find out the visible parts of the light L from xyz through the occluders O. This involves all the lines originating from xyz and intersecting L. As a consequence, we have to find the subsets of those lines belonging to a visible class. At first, we explain how those sets can be represented using convex fragments of L. Let us consider a

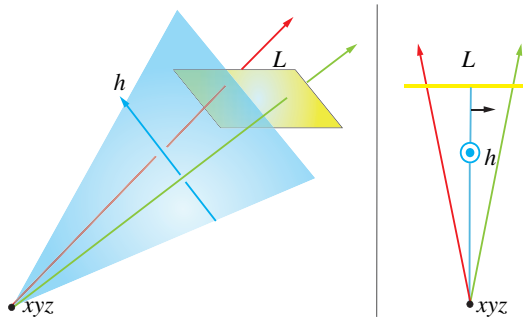

Figure 6: *The point xyz and the line h define a plane that sets apart the lines stabbing L and having a positive or negative orientation with respect to h*

line h spanning an occluder edge in O. We observe that h and xyz define a plane p that sets apart the lines intersecting xyz and L in two sets: The first one with a positive orientation with respect to h, the second one with a negative orientation. Figure 6 gives an illustration. Notice that the orientation of p is coherent with the orientation of h. As a consequence, if L is split by p, we can compute the two relevant polygons $L \cap p^+$ and $L \cap p^-$ so that they represent respectively the

positive and negative subset of lines with respect to h.
The visibility algorithm (Algorithm 1) finds the equivalence

Algorithm 1 Visibility algorithm: Given a point xyz on T, the algorithm answers the query "which parts of L are visible from xyz ? L is used to drive the BSP tree construction and may be split into several fragments representing homogeneous sets of lines from xyz. The fragments reaching visible classes are the visible parts of L from xyz

```
 1: Node {
 2:    // if it is an inner node
 3:    Plucker hyperplane //a line from an occluder's edge
 4:    Node left, right // left and right children
 5:    // if it is a leaf
 6:    Enum class // visible, invisible or undefined
 7:    Occluder[] occluders //occluders in the leaf if it is undefined
 8: }
 9: insertOccluder(Node n, Occluder o)
10: if n is a leaf then
11:    if n is not an invisible class then
12:       n.occluders ← n.occluders ∪ o
13:    end if
14: else if orientation( o.vvset, n.hyperplane) > 0 then
15:    insertOccluder( n.left, o )
16: else if orientation( o.vvset, n.hyperplane ) < 0 then
17:    insertOccluder( n.right, o )
18: else
19:    insertOccluder( n.left, o )
20:    insertOccluder( n.right, o )
21: end if
22:
23: BSP_query(Node n, Polygon L, Point xyz ) return Polygons
24: loop
25:    while n is not a leaf do
26:       Plane p ← makePlane( xyz, n.hyperplane )
27:       if position( p, L ) > 0 then
28:          n ← n.left
29:       else if position( p, L ) < 0 then
30:          n ← n.right
31:       else
32:          // split the light and work recursively
33:          return BSP_query(n.left, L∩p+, xyz)
                    ∪ BSP_query(n.right, L∩p−, xyz)
34:       end if
35:    end while
36:    if n.occluders is empty then
37:       return (n.class is visible) ? L : ∅
38:    else
39:       ro ← random occluder in n.occluders
40:       n ← root of bsp(ro)
41:       for o in n.occluders, o ≠ ro do
42:          insertOccluder( n, o )
43:       end for
44:    end if
45: end loop
```

classes related to the visibility of L from xyz. It subdivides L into several convex fragments so that each of them belongs to a single equivalence class. Because it is lazy, it combines the query (lines 25-35) and the growing (lines 36-44) of the data structure in a single step.
Algorithm 1 starts with all the occluders associated to a single undefined leaf. For each inner node, we compute the plane defined by xyz and the line stored in the node (line 26), as illustrated by Figure 6. Then, L is tested against this plane to determine the orientation of the lines stabbing L and xyz (line 27). If L lies in the positive (resp. negative) halfspace of the plane, then all the lines have a positive (resp. negative) orientation, and the algorithm continues in the left (resp. right) subtree (line 28 or 30). Otherwise, L is split against the plane and the algorithm continues recursively in both subtrees with the relevant parts of L (line 33). When a fragment reaches a leaf, two alternatives occur:

- The leaf has no occluders, therefore it is either a visible or an invisible class. In the former case, the fragment is a convex part of L which is visible from xyz, thus it is returned (line 37). Otherwise, the fragment is invisible from xyz and it is discarded.
- The leaf has some occluders, the class is undefined and we cannot answer the query without further developing the tree.

In the latter case, the algorithm chooses a random occluder (line 39) ro among the occluders associated with the current leaf. This occluder is used to grow the tree by replacing the leaf by $bsp(ro)$, the BSP representation of the four classes generated by ro (line 40). Next, the remaining occluders are inserted into $bsp(ro)$ which is now a new part of the tree. The procedure at line 9 achieve an occluder insertion as described in Figure 5. When all the occluders have been inserted in $bsp(o)$, the algorithm continues from the root of $bsp(o)$ until it finds the equivalence class for the current light fragment.

4.3. Key points

This section underlines significant points of the algorithm.
Random selection of the occluders:
The algorithm's efficiency is related to the balance of the tree. To develop the BSP tree, the algorithm chooses an occluder randomly. Obviously, some choices may lead to a more balanced tree than others. However this is not predictable. In fact, we have tested different heuristics, trying to make a "good choice". Unfortunately, all of them achieved poor improvements compared to the extra computational cost. Moreover, their behaviour can be very different according to the nature of the scene. We have reached the conclusion that a random choice gives better results, and more important, it has a consistent behaviour independent of the rendered scene. This is similar to the choice of the pivot in the well known quicksort algorithm: Although a random pivot is not the optimal choice, it leads to the best performance in practice.

Conservative insertion:
If all the lines stabbing an occluder o do not have the same

orientation with respect to a hyperplane, o is inserted in both subtrees of the relevant node. As a consequence the insertion is conservative. This step could be computed exactly as in [HMN05,MA05]. But, as explained in section 2, this is expensive and prone to numerical errors. Our algorithm avoids these problems to remain simple and robust.

Tree growth:
Notice that all the queries will not develop the BSP tree. We can expect some queries to develop new equivalence classes, at least at the begining since the BSP tree is empty. We can expect queries to take advantage of the previous computations thanks to the visibility coherence. This is a key point of the algorithm's efficiency.

4.4. Soft shadows framework

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 soft shadows. Algorithm 2 describes the process.

Algorithm 2 The following pseudocode illustrates how our visibility algorithm is plugged in a ray tracer software to analytically compute soft shadows

```
1:  build visible_triangle, the triangle list visible from the camera
2:  foreach light L in the scene do
3:      triangle_list ← visible_triangle
4:      // the following loop parallelization is straightforward
5:      while triangle_list is not empty do
6:          remove a triangle T from the triangle_list
7:          select the occluders O of T and L using shaft culling
8:          initialize a BSP tree root node n associated with O
9:          foreach image point xyz on T do
10:             visible_parts ← BSP_query( n, L, xyz)
11:             compute the illumination in xyz using visible_parts
12:         end for
13:     end while
14: end for
```

- 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 (line 1).
- Multiple lights are handled successively and independently (line 2)
- For each visible triangle, an empty BSP tree is created and associated with its set of occluders (line 8).
- Next, for each image point, the algorithm 1 is used to compute the visible parts of the light (line 10). In our framework, we consider area light sources with a uniform emission, therefore we compute direct illumination analytically (line 11) by integrating over the visible parts of the light [NN85].

This framework is designed to allow an efficient implementation. The loop order is chosen to build one BSP tree per light and per visible triangle. Since visible triangles are successively handled, BSP trees are developed successively and independently. This improves the memory coherence and avoids switching many times between BSP trees. This also limits the memory consumption since each BSP tree is deleted as soon as all its related image points are shaded. Moreover, the implementation can be easily multithreaded: A thread gets a visible triangle from the list, shades its image points and starts over until the list becomes empty. In this case, the triangle list access (line 6) has to be protected.

Implementation details
In Section 4, occluders are defined as the geometry intersecting the convex hull of a triangle and a light source. In practice, the occluders selection relies on a shaft culling approach as described in [Hai00]. Let's consider the bounding box of a visible triangle and the bounding box of an area light source. The shaft is defined as the convex hull of the two bounding boxes plus the triangle's plane support and the light's plane support. Any triangle intersecting the shaft is considered as a possible occluder. While this definition can lead to a conservative occluder set, it can be computed efficiently. At last, compared to Algorithm 1, a stack is managed to avoid recursive system calls. Except for this detail, the implementation is straightforward, it uses single floating point precision and does not use SIMD instructions.

5. Results

All tests were run on a 2.67 GHz Intel Core i7 920 processor with 4GB of memory. For comparison purpose, all pictures were rendered at 1280×720 pixels with one primary ray per pixel. Four sets of results are presented, three sets testing the global performance of our soft shadow framework, and a last one giving an insight on Algorithm 1 behaviour.

5.1. Comparisons on time and quality

The first set of results compares our method to ray-traced soft shadows both at comparable time and comparable quality. The ray tracer implementation is similar to [WBWS01]: It relies on an optimized SAH kd-tree, uses SIMD instructions to trace four rays simultaneously, and supports multithreading to render several parts of the picture in parallel. Ray-traced soft shadows are computed using groups of 4 shadow rays, and an uncorrelated stratified sampling of the area light sources. Since both our method and the ray-tracer support multithreading, all computations are run using 4 threads.

We use four scenes to test our method in different configurations. Despite its moderate geometrical complexity (26 673 triangles), the T-Rex scene is challenging for our approach because it presents difficult and complex shadows due to a long rectangular light source. This means that the light source visibility is complex and this is precisely what is computed by our algorithm. The modified Sponza Atrium

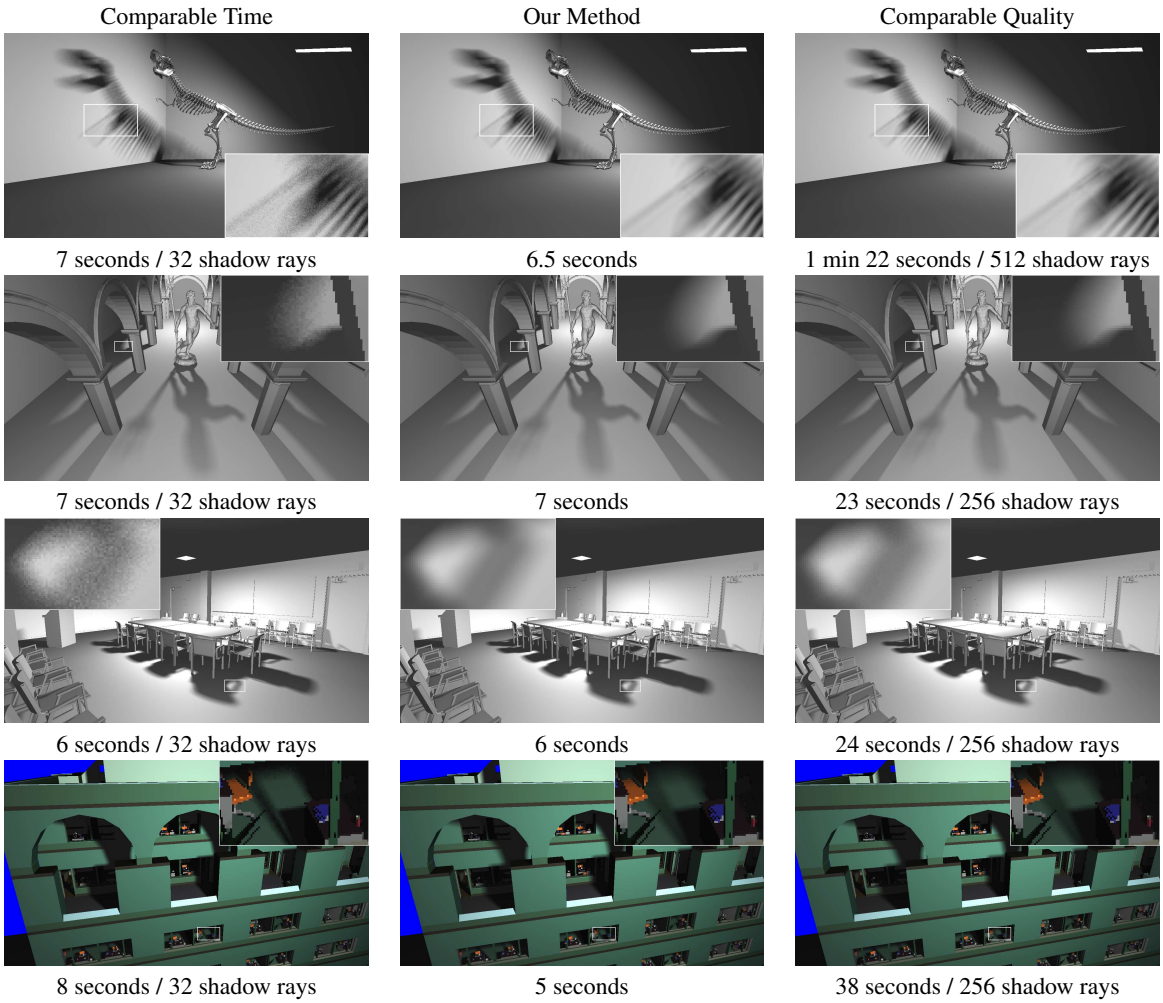

Figure 7: *From top to bottom: T-Rex, Sponza with Neptune, Conference and Soda Hall. The middle column shows the results computed with our algorithm. The left column presents the same pictures computed at comparable time and the right one at comparable quality, both using shadow rays. As an indication of the performance of our comparison method, we have rendered the same pictures under the same circumstances (same computer, 4 threads), using Mental Ray© and obtained the following results: Trex (512 samples - 4"43) Sponza-Neptune (256 samples - 3"52), Conference (256 samples - 2"20), Soda (256 samples - 3"04). These timings concern only the shadow rays.*

with the statue of Neptune (115 737 triangles) and Conference (282 873 triangles) are significant and detailed models with different kinds of shadow complexities. At last, Soda Hall (2 147K triangles) is used to test the scalability of our approach on a massive model with heterogeneous geometry.

Figure 7 details these results, and presents the time spent in soft shadows computation for each method. At comparable time, ray traced soft shadows are always noisy. At comparable quality (*i.e.* the noise is not noticeable anymore in the stochastic shadows), our method is always faster. In addition, this is very noticeable on a complex case such as the T-rex scene, because the stochastic approach requires a very high number of shadow rays to remove almost all the visible noise. In contrast, the exact visibility algorithm produces high quality results in a few seconds.

Figure 8 presents the computation measurements (memory and time consumption) for our exact visibility algorithm. Since we compute a BSP tree in Plücker space for each visible triangle from the camera, the memory consumption varies during the process according to the building of the BSP tree. As a consequence, our results report the maximum memory load reached by our soft shadow framework. In any case, the memory footprint is low. The lazy evaluation of the visibility driven by the visibility queries allows to focus

	Memory	Time				Modified Version	
Scene	Max Size (KB)	Shaft Culling (%)	Init (%)	Queries (%)	Total (s)	MV Time (s)	Falling off
T-Rex	19 916	4.8	1.9	93.3	6.5	314	×48
Sponza	16 246	4.0	2.0	94.0	7	191	×27
Conference	20 758	4.0	4.5	91.5	6	146	×24
Soda Hall	20 991	1.6	3.5	94.9	5	83	×16

Figure 8: *The time and memory consumption using our algorithm. The Max Size column is the maximum memory load reached during the process. The Shaft Culling column gives the time percentage spent to select the occluders. The Init column gives the time percentage spent to initialize each BSP tree. The Queries column gives the time percentage spent to query the BSP trees for shading each image point. The Total column gives the time in seconds for the whole process. MV Time column presents the result obtained using a Modified Version of our framework, where all queries are prevented from taking advantage of the others. The last column gives the falling off factor between our framework (Time) and its modified version (MV Time).*

on the equivalence classes related to the shadows and avoid the computation of useless visibility data. In addition, our framework is designed to avoid building too many BSP trees simultaneously (at most one per thread) allowing to keep the memory consumption low.

The total time can be subdived in three steps: The occluder selection, the BSP tree initialization (including the computation of the hyperplanes and the *vvset* for each occluder), and the visibility queries used for shading the image points. We can notice that the computation time is clearly dominated by the visibility queries *i.e.* calls to the `BSP_query` function (see Algorithm 1) which is the core of our method.

Aside the comparison with the stochastic approach, these results show that we can make the most of from-polygon visibility coherence to design an efficient and robust algorithm, in contrast with previous works on this topic. As an example, we were unable to process the scenes presented in this paper with a method as described in [MA05]. The scenes are too complex for such an approach, which relies on 5D CSG operations. This becomes numerically unstable and leads to degenerate results.

As expected, the sensitivity of our algorithm to the visual complexity of the light source is confirmed. For instance, the T-Rex scene required roughly the same amout of time as the Sponza and Neptune scene or the Conference scene.

About visibility coherence:

A crucial property of our approach is its ability to take advantage of the visibility coherence between image points. To test this ability, the same pictures were rendered again using a modified version of our visibility algorithm: Between each visibility query (*i.e.* between each `BSP_query` call), the related BSP tree is reset to its root node associated with its former occluder set (this additional operation is excluded from the timings). In such a case, each query is "the first one" and we prevent all queries from taking advantage of the previous ones. Figure 8 presents (two last columns of the table) the computation times compared to the timings obtained with the true version: The loss of efficiency is considerable. This demonstrates the capacity of the visibility algorithm to benefit from the visibility coherence, which is a key point to its efficiency.

5.2. Increasing the area of the light

The soft shadows complexity also depends on the size of the area light source. This is intrinsic to the soft shadows problem and it will inevitably affect our algorithm since it relies on the visibility coherence of the light source. If its area increases, the visibility coherence may decrease and lead to a loss of efficiency. As a consequence the second part of our tests investigates our framework's behavior when the area light source increases. These tests are run on the Conference model, selected for its significant number of triangles as well as the wide range of shadows cast in the scene. The tests start using a small area light source whose size is progressively increased until it becomes 100 times larger. For each size of light, the time and maximum memory used by our method are measured. Figure 9 sums up the results. As expected, it shows a loss of efficiency, both in time and memory, as the light source size grows (left and right graph of Figure 9). By extent, there is inevitably a critical light size where the time and the memory consumption would become a problem, in particular the memory since it is a limited resource. However, our tests show that, even with the largest area light source, we are far from such a point. In addition, further comparisons using our ray-tracer show that our approach remains fast. At comparable quality, 1024 samples are required for the largest light and the computation takes 91 seconds against 39 seconds using our algorithm (2.33 times slower). Independently of the increase in the noise, the break-even occurs for a light source 184 times larger than the smallest one (92 versus 93 seconds using 1024 samples).

5.3. Increasing the number of lights

Generally, a scene has several light sources. As detailed by Algorithm 2, our implementation supports multiple lights which are handled successively. Thus, it is interesting to test our framework behaviour in such a case. The Conference

Figure 9: *Increasing the light size. Left: The time consumption. Right: The maximum memory load reached during the computations. Center: A picture from our tests with the largest area light source (100 times larger than the smallest one. This corresponds to a square whose side is exactly the width of the conference table).*

Figure 10: *Increasing the number of lights. Left: The time consumption. Right: The maximum memory load reached during the computations. Center: A picture from our tests with 36 area light sources.*

model is used again and rendered with 2 to 36 area light sources. All the lights have the same size and cast roughly the same "amount" of shadow. Figure 10 presents the results. The left graph shows that the time consumption is linear with respect to the number of area light sources. This is the expected behaviour since our framework evaluates the contribution of each light source one after another. In addition this makes the memory load independent from the number of lights, as shown on the right graph. However we can notice a significant growth of the memory load when the number of lights is more than 12. Indeed, the 12 first lights are above the conference table while the 24 other lights are mainly above the chairs, casting more complex shadows. The memory growth is independent from the number of lights in the scene, but it is coherent with the increase in the visibility complexity.

The previous set of results shows that oversized area light sources can become a limitation for our approach, mainly because of the memory consumption. The present tests give a solution: Any huge light source, even with a critical size, could always be treated as the union of several smaller lights. And the result is always guaranteed noise free, which is specialy interesting with huge light sources since they require a high number of shadow rays using a stochastic method.

5.4. Focus on the `BSP_query` behaviour

Previous results demonstrate the global efficiency and robustness of this work, but they do not highlight the visibility algorithm (Algorithm 1) behaviour, *i.e.* how the visibility data and the computational cost evolve with respect to the visibility queries. Figure 11 gives such an insight. It focuses on the construction of a single BSP tree from the Conference model. This tree was selected because it is representative of the `BSP_query` behaviour. It has 2 896 occluders and a significant number of queries (17 878) are performed. In particular, most of the image points are located in the umbra or penumbra, which represent the most complex cases for any visibility algorithm.

At first, the tree grows quickly because there is no visibility data and the algorithm has to develop it to answer the visibility queries. The timings show the extra computational cost required for this construction. As a second step, the tree growth slows down drastically because the previously computed equivalence classes can be re-used and only need to be completed from time to time. As a consequence the computational cost falls down. This is the global behaviour of the visibility algorithm. In addition, notice that the image points are shaded in the scanline order. This allows handling consecutive points which are likely to share the same visibility data. This is noticeable locally: An "expensive" query is al-

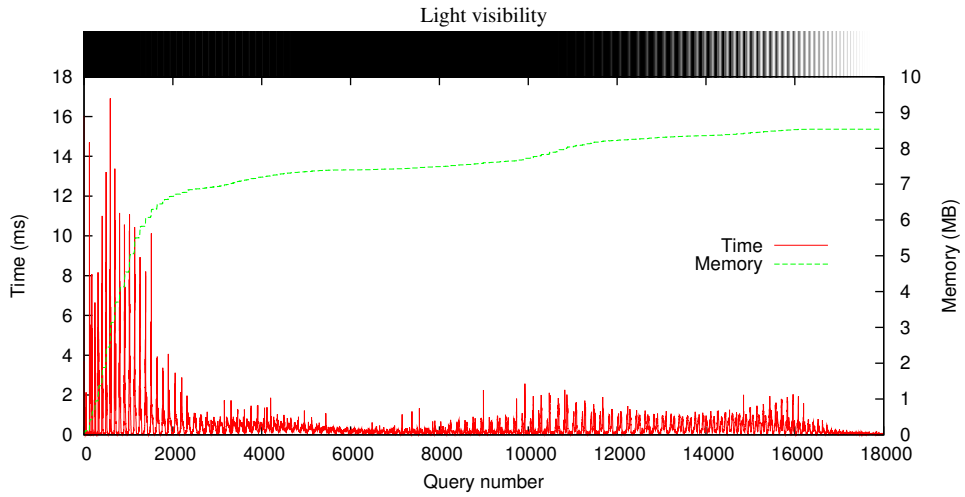

Figure 11: *Lazy construction of a representative BSP tree. The abscissa corresponds to the number of queries. The left ordinate is the time in milliseconds and is related to the continuous curve, while the right one is the memory consumption in MB and is related to the dashed curve. Above the graphic, a half-tone illustration of the light visibility for each query. Black means invisible while white is fully visible.*

ways followed by "cheaper" queries, taking advantage of the previous computational effort.

5.5. Discussions and future work

In this section we discuss our algorithm and point out some limitations and some issues we would like to address in the future.

The lazy construction of the visibility data is an important feature of our algorithm. This requires to query the visibility from some points to drive the computations. On one hand, this allows our approach to fit the image resolution and to focus the computational effort where and when it is needed. On the other hand, it may be inappropriate to solve some visibility problems. For example it is not suitable to prove that the light source is invisible from a polygon or to compute exact Potentially Visible Sets. Indeed, it is not possible to query all the points on a polygon or on the PVS boundaries. However, using an adaptative sampling of the surfaces for example, our approach can still be used to query the visibility from each sample and produce an accurate solution to these problems.

In this paper, a BSP tree is built for each triangle with image points to shade and makes the most of the visibility coherence between those points. This is an easy solution to group image points. However, if the geometric resolution is very high with respect to the image resolution, it can lead to very few image points per triangle and thus, a loss of efficiency, because the visibility data may be dropped before being re-used. To overcome this problem we plane to develop an approach independent from the geometry. We are thinking about mapping image points into a Bounding Volume Hierarchy built to balance the number of image points per leaf. Next, the light source visibility could be computed per leaf, using its bounding box faces to apply our visibility algorithm. This would probably require to solve the self occlusions that may occur inside a bounding box. As a future work this is our main idea since it would solve any problems related to the image versus object resolution. For example, it could even handled micro-polygons.

We also think our implementation could be improved using a parallel computing architecture such as CUDA or OpenCL to benefit from a higher number of threads on the graphic hardware. This will require fixing some technical issues, in particular about memory management since the lazy construction of data structures requires dynamic allocations from the graphic device. When available, they are made from the global memory, which could lead to a loss of efficiency.

In this work, the visibility data is dropped as soon as it is not needed anymore. Instead, we are thinking it could be saved to be used again for other renderings. In addition, we plane to simplify and compress the visibility data. This may further improve the memory footprint and the visibility queries efficiency.

At last, we also plane to introduce depth information in the visibility data, which means grouping lines according to the first triangle they intersect. This would allow extending our approach to other problems and applications such as ambient occlusion.

6. Conclusion

In this paper, we have presented a new approach to compute analytically from-polygon visibility. The core of this method is an original algorithm based on equivalence classes of lines. It lazily encodes the visibility from a polygon, and

allows taking advantage of the visibility coherence between successive visibility queries. In addition, the algorithm implementation is easy, and uses only basic geometric operations. As a consequence, it is fast and robust in contrast to previous works on exact from-polygon visibility.

Using this core algorithm, we have proposed a noise-free solution to analytically compute high quality soft shadows. We have tested our approach in various configurations, including different models, different sizes and different numbers of area light sources. These results demonstrated the reliability of our method.

To conclude, we hope this work illustrates that exact visibility, especially exact from-polygon visibility, is not necessarily incompatible with robustness and performance. Moreover, since visibility is a recurrent question in computer graphics, soft shadows is not the only problem that could benefit from the core of our method. As discussed in the previous section, we are interested in several issues and we intend to continue this work.

Acknowledgments: Sponza Atrium by Marko Dabrovic. Neptune model is provided courtesy of Laurent Saboret (IMATI, INRIA) from the AIM@SHAPE Shape Repository.

7. Appendix

To prove the theorem 1, we use a vector notation for Plücker's coordinates: Given two points $\mathbf{p}$ and $\mathbf{q}$, the Plücker coordinates of (pq) is the vector couple $(\vec{u},\vec{v})$ such as:

$$(pq)\begin{cases}\vec{u}=\overrightarrow{pq} & =(l_0,l_1,l_2)\\ \vec{v}=\overrightarrow{op}\times\overrightarrow{oq} & =(l_3,l_4,l_5)\end{cases}$$

where $\mathbf{o}$ is the origin in 3D space. For simplicity, we will omit the symbol $\vec{}$, simply denoting the vector couple $(\mathbf{u},\mathbf{v})$. Using this notation, we now formulate the side operator as the sum of two dot products. For example, given two lines l and l' with Plücker coordinates $(\mathbf{u},\mathbf{v})$ and $(\mathbf{u}',\mathbf{v}')$ respectively, then : $side(l,l')=\mathbf{u}\cdot\mathbf{v}'+\mathbf{v}\cdot\mathbf{u}'$.

Proof: Let A and B be two convex polygons with n and m vertices in the 3D space. Using barycentric coordinates, any point $\mathbf{a}$ on A (resp. $\mathbf{b}$ on B) can be defined by a linear combination of its vertices $(\mathbf{a_1},\cdots,\mathbf{a_n})$ (resp. $\mathbf{b_1},\cdots,\mathbf{b_m}$):

$$\mathbf{a}=\sum_{i=1}^{n}\alpha_i\mathbf{a_i}\text{ with }\sum_{i=1}^{n}\alpha_i=1,\text{ with }\forall i\ \alpha_i\geq 0,$$

$$\mathbf{b}=\sum_{i=1}^{n}\beta_i\mathbf{b_i}\text{ with }\sum_{i=1}^{n}\beta_i=1,\text{ with }\forall i\ \beta_i\geq 0.$$

$\mathbf{a}$ and $\mathbf{b}$ define any line q stabbing A and B, and the Plücker point of q is $(\mathbf{ab},\mathbf{oa}\times\mathbf{ob})$. Thus the relative orientation of a line q and any line l $(\mathbf{u},\mathbf{v})$ is:

$$side(q,l)=\mathbf{u}\cdot(\mathbf{oa}\times\mathbf{ob})+\mathbf{v}\cdot\mathbf{ab}.$$

Since the orientation of two lines is relative, we can assume without loss of generality that l goes through the origin of the 3D space (otherwise it is always possible to translate l). Thus $\mathbf{v}$ (the last 3 Plücker coordinates of l) becomes zero and:

$$\begin{aligned}side(q,l) &=\mathbf{u}\cdot(\mathbf{oa}\times\mathbf{ob})\\ &=\mathbf{u}\cdot(\textstyle\sum_i^n\alpha_i\mathbf{oa_i}\times\sum_j^m\beta_j\mathbf{ob_j})\end{aligned}$$

Since the cross product is distributive over addition:

$$side(q,l)=\mathbf{u}\cdot(\textstyle\sum_j^m\sum_i^n(\alpha_i\mathbf{oa_i}\times\beta_j\mathbf{ob_j}))$$

Next, by compatibility with scalar multiplication:

$$side(q,l)=u\cdot(\textstyle\sum_j^m\beta_j\sum_i^n\alpha_i(\mathbf{oa_i}\times\mathbf{ob_j}))$$

At last, since the dot product is distributive over addition:

$$\begin{aligned}side(q,l) &=\textstyle\sum_j^m\beta_j\sum_i^n\alpha_i\mathbf{u}\cdot(\mathbf{oa_i}\times\mathbf{ob_j})\\ &=\textstyle\sum_j^m\beta_j\sum_i^n\alpha_i(side((a_ib_j),l))\end{aligned}$$

Since $\alpha_i\geq 0$ and $\beta_j\geq 0$, the sign of $side(q,l)$ depends only on the sign of $side(a_ib_j,l)$. In particular, if they are all positive (resp negative) then $side(q,l)$ is positive (resp negative).

References

[AAM03] ASSARSSON U., AKENINE-MÖLLER T.: A geometry-based soft shadow volume algorithm using graphics hardware. In *SIGGRAPH* (2003), ACM, pp. 511–520. 2

[ACFM11] AVENEAU L., CHARNEAU S., FUCHS L., MORA F.: *Guide to geometric algebra in practice*. Springer, 2011, ch. Applications of Line Geometry. A Framework for n-dimensional Visibility Computations. 4

[AMA02] AKENINE-MÖLLER T., ASSARSON U.: Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *Eurographics workshop on Rendering* (2002), Eurographics, pp. 297–306. 2

[Bit02] BITTNER J.: *Hierarchical Techniques for Visibility Computations*. PhD thesis, Czech Technical University in Prague, October 2002. 2

[BW09] BENTHIN C., WALD I.: Efficient ray traced soft shadows using multi-frusta tracing. In *High Performance Graphics 2009* (2009), ACM, pp. 135–144. 2

[CAF06] CHARNEAU S., AVENEAU L., FUCHS L.: *Plücker space and polygon to polygon visibility computation with Geometric Algebra.* Tech. Rep. 2006-02, SIC Laboratory E.A. 4103 – Univeristy of Poitiers, 2006. 4

[CEG*96] CHAZELLE B., EDELSBRUNNER H., GUIBAS L. J., SHARIR M., STOLFI J.: Lines in space: Combinatorics and algorithms. *Algorithmica 15* (1996), 428–447. 4

[Cro77] CROW F. C.: Shadow algorithms for computer graphics. *SIGGRAPH Computer Graphics 11*, 2 (1977), 242–248. 2

[DDP02] DURAND F., DRETTAKIS G., PUECH C.: The 3d visibility complex. *ACM TOG 21*, 2 (2002), 176–206. 2

[EASW09] EISEMANN E., ASSARSSON U., SCHWARZ M., WIMMER M.: Casting shadows in real time. In *ACM SIGGRAPH Asia Courses* (Dec. 2009). 2

[ED07] EISEMANN E., DÉCORET X.: Visibility sampling on gpu and applications. *Computer Graphics Forum 26*, 3 (2007). 2

[FBP08] FOREST V., BARTHE L., PAULIN M.: Accurate shadows by depth complexity sampling. *Computer Graphics Forum 27*, 2 (2008), 663–674. In proceedings of Eurographics. 1, 2

[GH98] GHAZANFARPOUR D., HASENFRATZ J.: A beam tracing with precise antialiasing for polyhedral scenes. *Computer Graphics 22*, 1 (1998), 103–115. 2

[Hai00] HAINES E.: A shaft culling tool. *Journal of Graphic Tools (http://jgt.akpeters.com/papers/Haines00/) 5*, 1 (2000), 23–26. 7

[HH84] HECKBERT P. S., HANRAHAN P.: Beam tracing polygonal objects. In *SIGGRAPH'84* (1984), ACM, pp. 119–127. 2

[HLHS03] HASENFRATZ J.-M., LAPIERRE M., HOLZSCHUCH N., SILLION F.: A survey of real-time soft shadows algorithms. *Computer Graphics Forum 22*, 4 (dec 2003), 753–774. 2

[HMN05] HAUMONT D., MAKINEN O., NIRENSTEIN S.: A low dimensional framework for exact polygon-to-polygon occlusion queries. In *Eurographics Workshop on Rendering* (2005), pp. 211–222. 2, 7

[JHH*09] JOHNSON G. S., HUNT W. A., HUX A., MARK W. R., BURNS C. A., JUNKINS S.: Soft irregular shadow mapping: fast, high-quality, and robust soft shadows. In *Interactive 3D graphics and games* (2009), ACM, pp. 57–66. 2

[LA05] LAINE S., AILA T.: Hierarchical penumbra casting. *Computer Graphics Forum 24*, 3 (2005), 313–322. 2

[LAA*05] LAINE S., AILA T., ASSARSSON U., LEHTINEN J., AKENINE-MÖLLER T.: Soft shadow volumes for ray tracing. *Transactions on Graphics 24*, 3 (2005), 1156–1165. 2

[LLA06] LEHTINEN J., LAINE S., AILA T.: An improved physically-based soft shadow volume algorithm. *Computer Graphics Forum 25*, 3 (2006). 1, 2

[MA05] MORA F., AVENEAU L.: Fast and exact direct illumination. In *Computer Graphics International* (Juin 2005). Proceedings of CGI'2005, New York, Stony Brooks. 2, 7

[MAM05] MORA F., AVENEAU L., MÉRIAUX M.: Coherent and exact polygon-to-polygon visibility. In *WSCG'05* (2005). 2

[MP99] MOUNT D. M., PU F.-T.: Binary space partitions in plücker space. In *ALENEX '99* (London, UK, 1999), Springer-Verlag, pp. 94–113. 2

[NBG02] NIRENSTEIN S., BLAKE E., GAIN J.: Exact from-region visibility culling. In *Eurographics workshop on Rendering* (June 2002), Eurographics Association, pp. 191–202. 2

[NN85] NISHITA T., NAKAMAE E.: Continuous tone representation of three-dimensional objects taking account of shadows and interreflection. In *SIGGRAPH* (1985), ACM, pp. 23–30. 7

[ORM07] OVERBECK R., RAMAMOORTHI R., MARK W. R.: A real-time beam tracer with application to exact soft shadows. In *Eurographics Symposium on Rendering* (Jun 2007), pp. 85–98. 2

[Pel91] PELLEGRINI M.: Ray-shooting and isotopy classes of lines in 3-dimensional space. In *Algorithms and Data Structures*, Dehne F., Sack J.-R., Santoro N., (Eds.), vol. 519. Springer, 1991, pp. 20–31. 4

[Pel04] PELLEGRINI M.: *Handbook of Discrete and Computanional Geometry - second edition*. Chapman & Hall/CRC Press, 2004, pp. 839–856. Ray shooting and lines in space. 1, 2, 4

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. *ACM TOG 24*, 3 (2005), 1176–1185. 2

[SEA08] SINTORN E., EISEMANN E., ASSARSSON U.: Sample-based visibility for soft shadows using alias-free shadow maps. *Computer Graphics Forum 27*, 4 (June 2008), 1285–1292. 2

[Sho98] SHOEMAKE K.: Plücker coordinates tutorial. *Ray Tracing News* (July 1998). http://tog.acm.org/resources/ RTNews/html/rtnv11n1.html. 3

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. In *Computer Graphics Forum* (2001), Chalmers A., Rhyne T., (Eds.), vol. 20, Blackwell, pp. 153–164. 2, 7

[WIK*06] WALD I., IZE T., KENSLER A., KNOLL A., PARKER S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM TOG* (2006), 485–493. 2