Visual Comput (2007) 23: 773–782
DOI 10.1007/s00371-007-0129-4

Sylvain Charneau
Lilian Aveneau
Laurent Fuchs

Exact, robust and efficient full visibility computation in Plücker space

Published online: 1 June 2007
© Springer-Verlag 2007

S. Charneau (✉) · L. Aveneau · L. Fuchs
SIC Laboratory E.A. 4103,
University of Poitiers,
Bât. SP2MI, Téléport 2,
Bvd Marie et Pierre Curie, BP 30179,
86962 Futuroscope Chasseneuil Cedex,
France
{charneau,aveneau,fuchs}@sic.univ-poitiers.fr

Abstract We present a set of new techniques to compute an exact polygon-to-polygon visibility in Plücker space. The contributions are based on the definition of the minimal representation of lines stabbing two convex polygons. The new algorithms are designed to indicate useless computations, which results in more compact visibility data, faster to exploit, and in a reduced computation time. We also define a simple robust and exact solution to handle degeneracies, where previous methods proposed aggressive solutions.

Keywords Plücker space · Polygon-to-polygon visibility · Exact solutions · Visibility precomputation · BSP-tree for CSG

1 Introduction

This paper presents a set of new techniques for a theoretically exact polygon-to-polygon visibility. The polygon-to-polygon visibility characterisation is very useful information, which can have several applications in computer graphics, from soft shadow computation to fast from-region visibility queries.

Most tractable methods [1, 4, 7, 10] that fully or partially represent the visibility between two polygonal faces use the Plücker space of lines in $\mathbb{R}^3$ [11, 13]. In this space, the visibility computation consists in performing CSG operations to solve the selective stabbing problem [9].

Nirenstein et al. [10] propose the first exact and tractable solution to the from-region visibility problem. It is used to precompute an exact PVS (potentially visible set) for an interactive walkthrough application. Haumont et al. [4] present a low-dimensional framework that improves the Nirenstein method. Notice that these two works do not compute a complete representation of the visibility. They aim to find at most one visibility, as fast as possible. Therefore, these algorithms can process large scenes, but do not characterise how the polygons are *seen* from other ones.

Bittner [1] computes a complete representation of the visibility from a polygon facing a scene. The visibility data are encoded in a BSP-tree in the Plücker space and, as in the work by Nirenstein and Haumont, used to preprocess the PVS. Mora et al. [7] take advantage of both Nirenstein's and Bittner's approaches to compute a BSP representation of the complete visibility between two polygons. Both Bittner and Mora entirely compute the visibility, as opposed to Nirenstein and Haumont. The downside is the inability to process complex scenes due to an important data volume.

Actually, all these methods fail to determine efficiently which operations can be avoided. The main reason is the difficult handling of the Plücker space, which does not allow for easy geometrical reasoning. Likewise, they fail to process degeneracies exactly. Previous solutions are aggressive and do not result in an exact visibility description.

The aim of this paper is to obtain a robust method to compute a full polygon-to-polygon visibility, which highly reduces the visibility data by limiting useless splittings. This will enable the computation of the visibility in larger scenes.

This paper is organised as follows: Sect. 2 recalls the previous works and their limitations. Section 3 presents our four contributions to the visibility computation. These contributions first concern a new rejection test which indicates useless splittings, with a low computation cost. Second, a new silhouette processing algorithm is discussed, which leads to a large occluders detection based on the number of hidden objects, instead of their relative size, allowing to greatly optimise the silhouette processing. Finally, the contributions concern a straightforward and exact method to handle degeneracies. In Sect. 4, our visibility computation method is compared to previous methods with some test scenes, to evaluate the reduction of the visibility data and of the computation time.

2 Visibility computation in Plücker space

This section recalls the general principles of the visibility computation and representation in Plücker space. Then, optimisations and limitations of previous works are detailed.

2.1 Plücker space

Plücker space [13] is an elegant parameterisation of directed lines in $\mathbb{R}^3$. Lines are represented by vectors in $\mathbb{R}^6$, assuming that two vectors $\boldsymbol{a}$ and $\lambda\boldsymbol{a}$ represent the same element, for any $\boldsymbol{a}$ in $\mathbb{R}^6$ and λ in $\mathbb{R}^{*+}$. This corresponds to a five-dimensional projective space $\mathbb{P}^5$. The representation is achieved in the following way: let l be a line passing through the points $P:(p_x, p_y, p_z)$ and $Q:(q_x, q_y, q_z)$. This line is mapped to a 6-tuple $l^* = (\pi_0, \pi_1, \pi_2, \pi_3, \pi_4, \pi_5)$, lying on a quadric in $\mathbb{P}^5$, called the Plücker quadric $\mathcal{Q}$ (see Fig. 1). The 6-tuple l^* is called the Plücker coordinates of the line l. It is defined in $\mathbb{P}^5$ by:

$$(\pi_0, \pi_1, \pi_2) = Q - P = \boldsymbol{PQ}$$
$$(\pi_3, \pi_4, \pi_5) = P \times Q.$$

Representing lines by vectors in a vector space is of great interest, because it allows for the expression of geometric relations between lines by computations on vectors, like the relative orientation of two directed lines. This relative orientation of two lines can be interpreted as the way the first line rotates around the second one, as illustrated in Fig. 1.

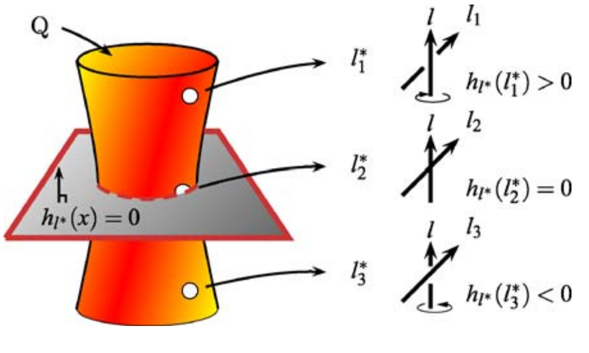


Fig. 1. Relative orientation of lines in the Plücker space. There are three different cases for an oriented line to pass another: l_1 passes on the left of l_0, l_2 is incident to l_0 and l_3 passes l_0 on the right. Their maps l_1^*, l_2^*, l_3^* lie above, on, and below $h_{l_0^*}$, respectively

A dual hyperplane h_{l^*} can be associated to each element l^* in $\mathbb{P}^5$ representing a line l in $\mathbb{R}^3$. This hyperplane is defined by:

$$h_{l^*} = \left\{x \in \mathbb{P}^5 \mid h_{l^*}(x) = 0\right\}$$

where

$$h_{l^*}(x) = \pi_3 x_0 + \pi_4 x_1 + \pi_5 x_2 + \pi_0 x_3 + \pi_1 x_4 + \pi_2 x_5.$$

Such a hyperplane induces a positive half space $h_{l^*}^+ = \{x \in \mathbb{P}^5 \mid h_{l^*}(x) > 0\}$ and a negative half space $h_{l^*}^- = \{x \in \mathbb{P}^5 \mid h_{l^*}(x) < 0\}$ in the Plücker space. While the hyperplane h_{l^*} is the set of lines incident to l, each half space determines a set of lines missing l, with an opposite relative orientation according to the half space to which they belong.

2.2 Polygon-to-polygon visibility

In this section, the general principles of Nirenstein's solution to the visibility computation are recalled from [9]. Let A and B be two convex polygons with n_1 and n_2 vertices. Let $\{e_1, \ldots, e_{n_1+n_2}\}$ be the oriented lines defined by two consecutive vertices of A or B. There exists a unique orientation for the lines $(e_i)_{i=1,\ldots,n_1+n_2}$ such that every line l stabbing A and B satisfies:

$$\forall\, \mathrm{i} \in [1, \ldots, n_1+n_2],\ h_{e_i^*}(l^*) \geq 0. \qquad (1)$$

This system of inequations is the hyperplane representation of an unbound polyhedron $\mathcal{P}$ in the Plücker space. In practice, some constraints are added in order to bound $\mathcal{P}$. The result is a convex polytope P_{AB}, which represents the same set of stabbing lines, i.e. $\mathcal{P} \cap \mathcal{Q} = P_{AB} \cap \mathcal{Q}$.

Computing the visibility consists in removing the lines stabbing an occluder O from the remaining lines stabbing A and B. Denoting $\{o_1, ..., o_m\}$ as the oriented edges of O, a similar reasoning as in Eq. 1 gives the set of lines stabbing A and B but not O:

$$P_{AB} - \bigcap_{1}^{m} h_{o_i^*}^+.$$

By splitting P_{AB} against the hyperplanes $h_{o_i^*}$, the subset of lines blocked by the occluder is computed and then removed, as shown in Fig. 2. Processing every occluder in the same way results in some polytopes representing exactly the set of lines stabbing A and B and omitting every occluder.

However, the naïve solution, which consists in computing a hyperplane arrangement by splitting every polytope by each hyperplane, results in an excessive amount of splittings. Pu [12] used such a solution and was limited to processing about fifteen occluders.

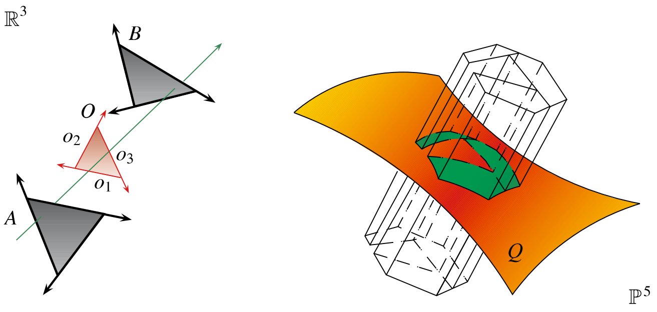


Fig. 2. Occluder removal. The set of lines blocked by an occluder O is computed using CSG operations in the Plücker space. The remaining parts of the polytope represent the visibility between A and B

Actually, many splittings can be avoided. Ideally, a splitting has to be computed, for an occluder O, if and only if the intersection $P_{AB} \cap P_{AO} \cap \mathcal{Q}$ is non-empty. The previous works of Nirenstein [10], Bittner [1] and Mora [7] aimed at limiting this splitting in different ways.

2.3 Optimisations and limitations in previous methods

The previous techniques used to limit the visibility splitting and to handle degeneracies are now presented with their limits.

2.3.1 The minimal polytope

The first solution to limit the splitting is to have the bounds of the polyhedron $\mathcal{P}$, defined by Eq. 1, as close as possible to the quadric $\mathcal{Q}$, which is the only area of interest. This will result in a small initial polytope P_{AB} which can avoid splittings resulting in polytopes containing only points out of the quadric. A minimal candidate for P_{AB} should be defined as the smallest polytope containing all the points on the Plücker quadric representing lines stabbing the polygons A and B.

In [4, 9], the proposed solution consists in translating points in $\mathbb{P}^5$, corresponding to the lines from a vertex of A to a vertex of B, in a particular direction. This method is equivalent to the addition of two parallel hyperplanes to have the polyhedron bound, but it does not define the minimal polytope. Moreover, they do not evaluate the *distance* between their initial polytope and the Plücker quadric, which prevents the determination of the number of splittings that should be avoided with a smaller polytope.

To avoid this problem, Mora et al. [7] use a back-splitting technique to cancel a splitting if the resulting polytope does not intersect with the Plücker quadric, requiring back-ups of intermediate results.

2.3.2 Determining non-empty polytope intersections

Another solution to limit the splitting of the visibility is to determine whether the intersection $P_{AB} \cap P_{AO}$ is empty or not, ignoring the quadric for a while, since one cannot compute directly with it.

Bittner's solution consists in representing the visibility by a BSP-tree [1]. The test is then achieved by *inserting* an occluder in the BSP-tree. This insertion corresponds to a standard polytope insertion algorithm in a BSP-tree [8]. The polytope P_{AO} is split by the hyperplanes contained in the tree inner nodes. If no polytope remains at the end, the intersection is empty. Nevertheless, this method requires a huge amount of intermediate computations (splittings of P_{AO}) and is sensitive to numerical imprecision.

In [4, 7, 10], the authors use instead an implicit representation of P_{AO}, given by the hyperplanes corresponding to the occluder edges. The test consists in rejecting the vertices of P_{AB} against those hyperplanes. If one hyperplane rejects all the vertices, the intersection is empty. However, this test is insufficient. Indeed, it must also reject the vertices of P_{AO} against the hyperplanes of P_{AB}, as illustrated in Fig. 3. The authors do not perform this second test because they do not have a representation of those vertices.

Thus, it appears that either these techniques lead to numerous intermediate computations or they cannot avoid some useless splittings due to an incomplete rejection test.

2.3.3 Ordering occluders

A critical way to accelerate the computation and to reduce the visibility splitting is to determine a good order to subtract occluders, according to their importance in the occlusion of B from A.

Two solutions appear in [9]. The first is a heuristic on the solid angle of the occluders. The second, taken back in [4], consists in shooting rays into the remaining visibility computed between A and B and measuring, for each remaining occluder, the number of rays intersecting it. The *biggest* occluder is the one which is intersected by most of the rays.

However, whereas the first technique does not return good order in all cases, the second only gives the first *biggest* occluder. So, this process must be repeated before each occluder subtraction. Moreover, these two methods do not take into account the number of objects that the large occluders hide, in order to avoid processing those hidden objects.

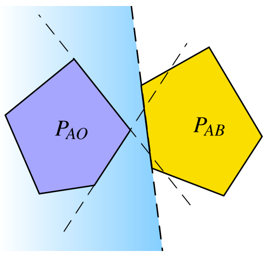


Fig. 3. Polytope rejection test. Two tests must be performed, first vertices of P_{AB} against hyperplanes of P_{AO} then vertices of P_{AO} against hyperplanes of P_{AB}

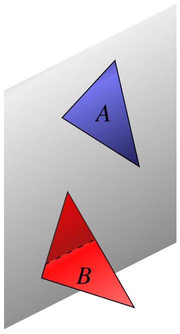


Fig. 4. A degeneracy case. The plane that support A intersects B

2.3.4 Degeneracies

A degeneracy occurs when the plane supporting the face A intersects the plane supporting the face B on B itself or on an edge or a vertex of B (and vice versa). Figure 4 shows a frequently occurring degeneracy.

Previous methods fail in dealing with such cases, since they use an unsuitable projection of vectors in $\mathbb{R}^6$. Recall that all lines in $\mathbb{R}^3$ are mapped to vectors in $\mathbb{R}^6$ whereas Plücker space only has five dimensions. In [4, 9], vectors are projected on a hyperplane in the following way (the projection only depends on a change of coordinate system):

$$(\pi_0, \pi_1, \pi_2, \pi_3, \pi_4, \pi_5) \mapsto \left(1, \frac{\pi_1}{\pi_0}, \frac{\pi_2}{\pi_0}, \frac{\pi_3}{\pi_0}, \frac{\pi_4}{\pi_0}, \frac{\pi_5}{\pi_0}\right).$$

This projection is a direct adjustment of the homogeneous coordinates used in the usual three-dimensional space. It allows to retrieve only five dimensions but fails for $\pi_0 = 0$ (which corresponds to points at infinity) or leads to numerical imprecisions when $\pi_0 \simeq 0$. Yet, points at infinity appear with degeneracies. The previous solution is to split and truncate the intersected face along the intersection (the dashed line in Fig. 4). Therefore, this method is no more exact but aggressive and numerically unstable due to points almost at infinity. We consider this solution as unacceptable for a theoretical exact visibility computation technique.

3 New techniques for computing visibility

Our method is now presented in three parts. All our contributions are based on the definition of the minimal convex polytope representing the lines stabbing two polygons. The definition and its interests are discussed below.

3.1 Visibility representation and occluder rejection

Our representation of the visibility follows the solution proposed in [1, 7]: a set of polytopes (or polytope complex) is represented by a BSP-tree in the Plücker space. Each upper node to a leaf contains a hyperplane determining one boundary of the polytope which is contained in the leaf. Two kinds of leaves can be distinguished: visible leaves, containing a polytope representing a remaining part of the visibility between A and B, and invisible leaves, representing occlusions. The polytope in the leaves is represented only by its vertices and edges, following [4].

The originality of our algorithm resides in the occluder insertion in the BSP-tree, to take into account the occlusion, which is based on a coherent definition of the minimal polytope.

3.1.1 Definition of the minimal polytope

The definition of the minimal polytope is given in [3] by the following theorem.

Theorem 1. *The set of lines stabbing two convex polygons A and B in $\mathbb{R}^3$ is the intersection of the Plücker quadric with the convex hull of the lines going through one vertex of A and one vertex of B if and only if the support planes of A and B do not intersect in A or B.*

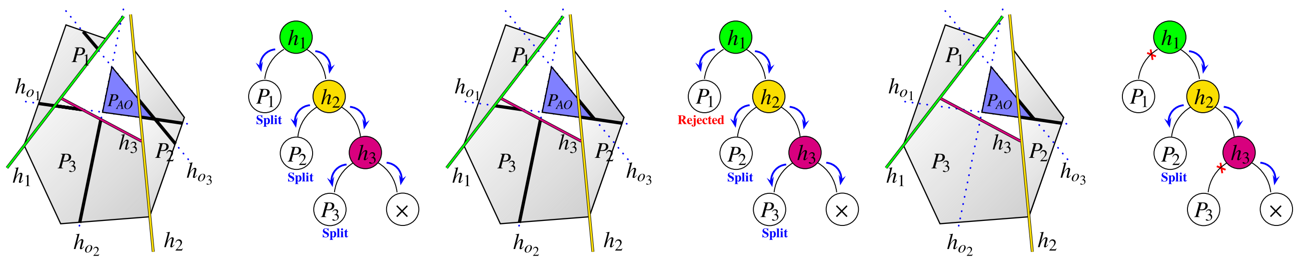
This says that a convex polytope does not exist in degeneracy cases except when the intersection of A and B is either on an edge or a vertex. But, any degeneracy can be reduced to one of these two particular degeneracies by splitting the intersected face along the intersection, but without truncating it, like in previous methods.

The major advantage of Theorem 1 is the ability to obtain the vertices of the minimal polytope, with few computations, from the vertices of A and B. It is of use in two parts of our application. First, a convex hull incremental algorithm [2] is used to compute the initial polytope P_{AB} from its set of vertices. This algorithm is here adapted to a convex hull in an oriented projective space and to a set of points in any position. Second, from the vertex representation of P_{AO} which can now be easily determined for each occluder O, the second rejection test, missing in previous methods, can be performed.

3.1.2 Occluder insertion

The first use of the new rejection test concerns the occluder insertion in a BSP-tree. Thanks to Theorem 1, the vertices of P_{AO} are computed for each occluder O. Then, the occluders are inserted by proceeding with the rejection of the P_{AO} vertices against the polytopes boundaries contained in the tree. Recall that these boundaries are the hyperplanes in the tree inner nodes.

To understand the importance of this rejection test, Fig. 5 shows the result of an occluder insertion obtained without rejection, with a partial or with a complete rejection test. The polytope P_{AO} represents lines stabbing A and the occluder O, whereas the three polytopes P_1, P_2 and P_3 represent the remaining visibility, encoded in a BSP-tree.



a Without rejection test **b** Partial rejection test **c** Total rejection test

Fig. 5a–c. Results of the polytopes splitting after the tree traversal. **a** Without rejection test. **b** With partial rejection test. **c** With total rejection test. The drawings (on the *left*) show how the polytopes are split, and the trees (on the *right*) show which nodes are visited during the algorithm, and which leaves are split or rejected

Figure 5a corresponds to a hyperplane arrangement (without any rejection) and leads to nine polytopes. This is equivalent to Pu's solution [12]. The second Fig. 5b shows the insertion with a partial rejection as used in [4, 7, 10] (see Sect. 2.3). It creates here six polytopes. The rejection of the vertices of $(P_i)_{i=1,2,3}$ against the hyperplanes of P_{AO} enables the rejection of the polytope P_1 by the hyperplane h_{o_2}. A complete rejection test (Fig. 5c) successively rejects the polytope P_{AO} by the hyperplanes h_1 and h_3. It is not inserted in the corresponding subtrees. The occluder insertion leads to four polytopes by splitting the lone polytope P_2.

In practice, this rejection test is achieved on each occluder edge separately, instead of the whole occluder. This determines which edges really change the visibility.

Algorithm 1 presents our occluder insertion: First, the occluder edges are inserted in the BSP-tree root (procedure `InsertOccluder`). If this root contains a hyperplane, the edges are rejected against it (function `RejectEdges`, lines 6, 17). If the root is a leaf, a second rejection test is achieved with the polytope vertices against the hyperplanes of the occluder O (function `isSplit`, line 11). This second test corresponds to the partial rejection performed in previous works [4, 7, 10].

Algorithm 1. *Occluder insertion in the BSP-tree algorithm, using an exhaustive rejection test in inner nodes.*

```
1   Polygon $A$, $B$, $O$
2   procedure InsertOccluder (VBSP tree ; EdgeList $O$; Polygon $A$)
3   begin
4       if isInnerNode (tree) then
5           EdgeList posList, negList
6           (posList, negList) $\leftarrow$ RejectEdges ($O$, $A$, tree.hp)
7           InsertOccluder (tree.posTree, posList, $A$)
8           InsertOccluder (tree.negTree, negList, $A$)
9       else
10          if isVisibleLeaf (tree) then
11              if isSplit (tree.polytope, $O$) then
12                  addSubtree (tree, tree.polytope, $O$)
13              end
14          end
15      end
16  end
17  function RejectEdges (EdgeList $O$; Polygon $A$; Hyperplane $H$)
18  EdgeList posList, negList
19  begin
20      foreach edge in $O$ do
21          $L \leftarrow$ lines from vertices of $A$ to vertices of the edge
22          foreach line $l$ in $L$ do
23              if $H(l*) > 0$ and edge $\notin$ posList then
24                  Add edge in posList
25              else
26                  if $H(l*) < 0$ and edge $\notin$ negList then
27                      Add edge in negList
28                  end
29              end
30          end
31      end
32      return (posList, negList)
31  end
```

Our supplementary test is described in the function `RejectEdges`. This test rejects each occluder edge on one side or on both sides of a hyperplane according to the side where the lines L lie (line 21). Those lines are obtained from the vertices of the source polygon and the vertices of the edge to be rejected. This rejection technique is consistent thanks to the definition of the minimal polytope. By interpreting the hyperplane H by a line l_H in $\mathbb{R}^3$, this rejection corresponds to detecting the side of l_H where the penumbra wedge associated to the edge lies. Recall that the penumbra wedge is the shadow volume determined by the edge and the light source A [5].

The function `RejectEdges` returns two lists, corresponding to the edges to be inserted in the positive subtree and in the negative subtree of the root respectively (lines 7 and 8). If no edge remains, the polytopes under the root are either fully in or fully out of P_{AO}. The second rejection test determines which case occurs.

The proposed new test, performed in the inner nodes and missing in previous works, consequently reduces the visibility splitting. Moreover, this test requires few computations: only some Plücker maps of known points in $\mathbb{R}^3$ and some lines relative orientation computations. The consequences of this extra rejection on the visibility splitting are discussed in Sect. 4.

3.2 Using extended silhouettes

The interest to process only silhouette edges, instead of every occluder, is to limit the computation to elements which potentially change the visibility. Haumont was the first to include a silhouette processing in exact from-polygon visibility computation [4]. However, his technique for silhouette extraction is time-consuming due to the propagation process and is not well-suited to our algorithm since Haumont's algorithm only tests the visibility.

Our occluder insertion in the BSP-tree rejects edges instead of polygons. Algorithm 1 is then adapted to insert edges as occluders, allowing to deal with silhouette edges.

From a viewpoint, a silhouette edge is an edge connected to two faces, with the viewpoint lying in two different half spaces delimited by the oriented planes supporting the two faces. This definition conforms to the classical definition of silhouette edges [5] and is illustrated in Fig. 6. The edge e on the figure is a silhouette from every viewpoint lying in the $+-$ or the $-+$ subspaces.

Since the silhouette is computed from viewpoints on A and B, this definition can be extended to silhouette edges from multiple viewpoints: an edge is a silhouette from a set of viewpoints if it is a silhouette from at least one viewpoint in this set. Occluders for which every edge belongs to the silhouette are considered as silhouette faces and are processed as previously described. Notice that, to handle general scenes like objects with holes for example, edges only connected to one face are also considered as elements of the silhouette.

The visibility computation is then achieved in two steps. First, every silhouette element (edge or polygon) is inserted in the tree, using Algorithm 1 modified to handle edges like faces. This first process makes all necessary splits to separate the lines passing over some occluders and the ones missing all of them, except for some silhouette singularities which are handled during the second process. This process does not determine which leaves, in the BSP-tree, contain a polytope representing occluded lines.

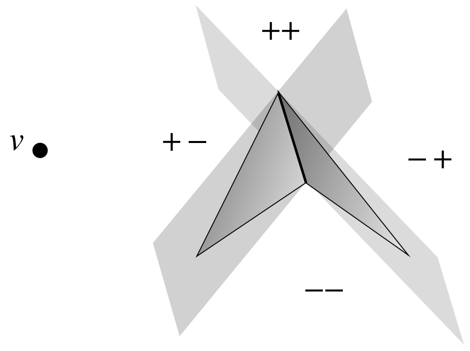


Fig. 6. Determining silhouette edges from a viewpoint v

This leads to the second step which consists in inserting in the tree each occluder which does not belong to the silhouette. It determines which leaves are *visible* or *invisible* and when there is a singularity.

A singularity consists of two edges such that lines from A to B rejected on some side of the two edges lie both on $h^+_{l^*_\infty}$ and $h^-_{l^*_\infty}$, where l_∞ is the line at infinity determined by the two edges directions. This line has coordinates $l^*_\infty = (\mathbf{0}, d_1 \times d_2)$ where $\mathbf{0}$ is the zero vector in $\mathbb{R}^3$, and d_1 and d_2 are the directions in $\mathbb{R}^3$ of the two edges. The simplest singularity is composed of two consecutive occluder edges form a plane intersecting A and B.

Recall that the vertices of P_{AO} are the lines from a vertex of A to a vertex of O. A singularity occurs in the silhouette if the vertices of P_{AO}, lying in $h^+_{l^*_\infty}$ (resp. $h^-_{l^*_\infty}$) are rejected by the two edges, whereas the vertices of P_{AO} in $h^-_{l^*_\infty}$ (resp. $h^+_{l^*_\infty}$) are not rejected. In this case, the leaf represents occluded lines on the side the occluder is not rejected and visible lines on the side it is rejected. In order to separate these lines, the leaf must be split by the hyperplane associated to l^*_∞.

3.2.1 Large occluders

To avoid computations on many silhouette edges which are hidden by few large occluders, a large occluder detection is described. Contrary to previous works [4, 7, 10] which also define an occlusion measure of occluders (see the occluder ordering in Sect. 2.3), our technique is not based on a heuristic but on the effective number of objects hidden by an occluder. The large occluders must then be processed before the silhouette elements.

A detection based on the number of hidden objects is much more efficient than any heuristics in this case. More precisely, consider two boxes, a large one which is empty and a small one containing many small objects. It is of course preferable to consider the faces of the small box as large occluders, since they hide all the objects the box contains. While on the contrary, a heuristic based on the size of the polygons should choose the faces of the large box as larger occluders, even if these faces do not simplify the silhouette.

The detection of hidden objects is now explained. An edge E (or a whole polygon) is hidden by an occluder O if and only if each line from a vertex of A to a vertex of E intersects O (see Fig. 7). If $\boldsymbol{e_1}$ and $\boldsymbol{e_2}$ are the vertices of the edge e, $(a_i)_{i=1,\dots,n}$ the vertices of A and $(e_{o_i})_{i=1,\dots,m}$ the edges of the occluder O, we have, according to Eq. 1:

$$\forall i \in [1; \dots; n],\ j \in [1; 2],\ k \in [1; \dots; m], \quad h_{(a_i,e_j)^*}(e^*_k) \geq 0.$$

This large occluder detection consequently reduces the number of silhouette edges to process in many situations, and gives better results than detections based on some heuristics. Some typical examples are shown in Sect. 4.

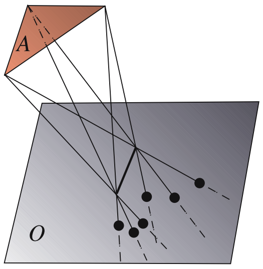
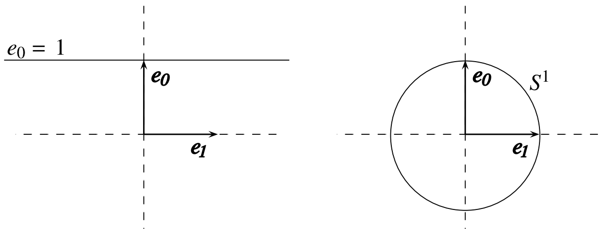


Fig. 7. Detection of large occluders O which "hide" edges according to a source A



a Projection of vectors on the hyperplane $e_0 = 1$ **b** Projection of vectors on the hypersphere S^1

Fig. 8. Different projections in a vector space

3.3 Dealing with degeneracies

Handling degeneracies first needs a correct definition of the initial polytope P_{AB}. The minimal polytope presented in Sect. 3.1.1 is suitable since it is also constructible when a degeneracy occurs. Second, it needs to be able to represent points at infinity (see Sect. 2.3). Actually, such points can easily be represented by using a projection of vectors on the hypersphere S^5 in $\mathbb{R}^6$.

Figure 8 shows an example in $\mathbb{P}^1$, represented by vectors in $\mathbb{R}^2$. In Fig. 8a, it is clear that the vector $\boldsymbol{e_1}$, parallel to the hyperplane $\boldsymbol{e_0} = 1$, cannot be projected on this hyperplane. This vector is a point at infinity. As a consequence, projecting vectors of $\mathbb{R}^6$ on a hyperplane, as in previous works, prevents representing such points in $\mathbb{P}^5$. On the contrary, those vectors can be projected on a hypersphere (see Fig. 8b), where e_1 can be projected on the circle S^1 (the one-dimensional hypersphere). This projection represents points at any direction, even at infinity.

Since the Plücker space is an oriented projective space (i.e. for all λ in $\mathbb{R}^{*+}$, two vectors v and λv in $\mathbb{R}^6$ represent the same element of $\mathbb{P}^5$), computing the visibility with λv instead of v does not change the final result. As a consequence, we do not have to pay attention to the change of normalisation.

Thanks to this new projection, degeneracies can be handled transparently, while keeping a theoretically exact visibility computation technique.

4 Discussion

In this section, we analyse the impacts of our new algorithms and optimisations and discuss some possible improvements.

4.0.1 Analysis context

The study first concerns the impact of the new occluder insertion algorithm, including the rejection test in the inner nodes of the BSP-tree (see Sect. 3). Second, it evaluates the importance of the silhouette processing, and comparisons are made between computation times, BSP-trees sizes and memory requirements.

All tests were performed on an Athlon 64, running at 2.8 GHz and with 3 Gb RAM and 2 Gb swap. The GNU multiple precision arithmetic library (GMP) [14], is used for the Plücker data. It performs precise computations but also decreases the computation speed on coordinates by about a factor of 50 compared to a double precision arithmetic. However, since our method is used to precompute the visibility, accuracy is more important than speed.

Notice that the comparisons are not between the Nirenstein and Haumont results and our results, since they only compute a partial visibility representation until they find the existence of at least one visibility, and our method aims at computing the full visibility. Therefore, the comparisons are between our rejection technique and their partial technique.

4.0.2 Comparison scenes

This study uses five test scenes presented in Fig. 9. The visibility is computed from each light source for every polygon in the scene. The first three scenes (Deformed Sphere, Three Pens and One Pen) contain few polygons but represent various visual complexities.

The two other scenes (Room and Building) show our optimisations behaviour in virtual scenes containing objects with many different visual complexities. The first one presents only one object with a high visual complexity (a plant in Fig. 9e). The second contains many more polygons and lights (about 20 k polygons and 15 lights, leading to 300 k computed BSP-trees) and many tessellated objects. Notice that the last scenes results do not present statistics for the partial rejection test. Indeed, we were unable to process it with this partial test, due to an excessive amount of splittings which require too much memory.

4.1 Visibility computation results

Table 1 shows the results obtained with the different methods. The first remark on the scene Three Pens is that our method reduces the number of inner nodes by about a factor of 2, in spite of the low visual complexity. The difference is all the more important for scenes presenting

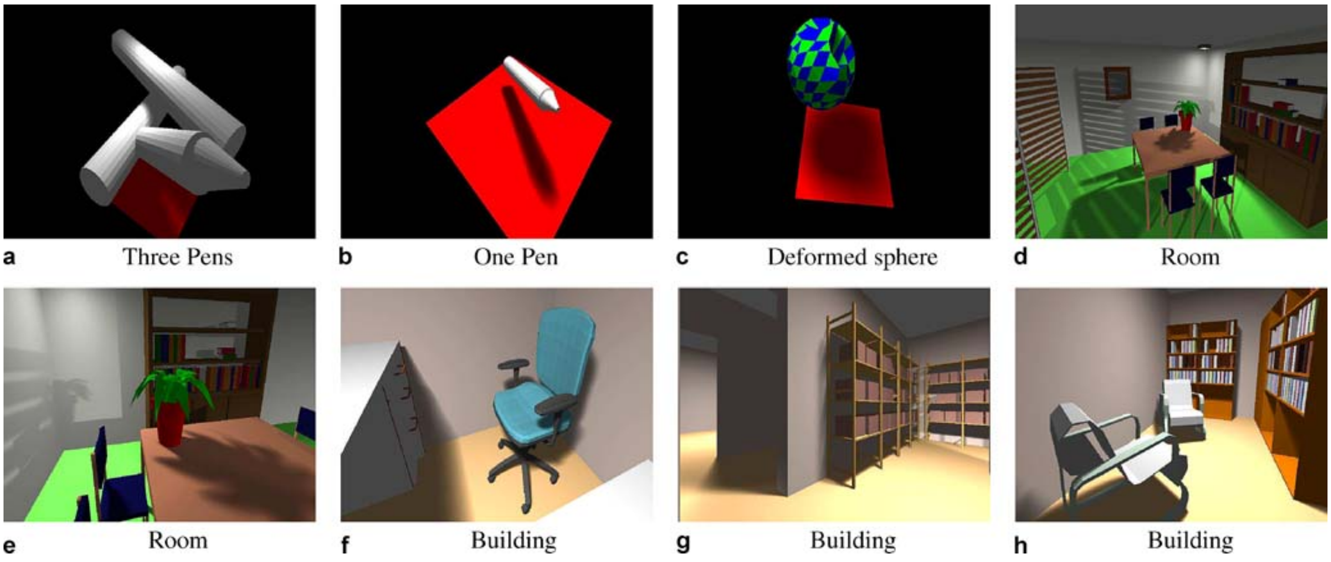


Fig. 9. Test scenes used for comparisons. The visibility data are applied here to compute soft shadows in an interactive ray-tracer

Table 1. Visibility computations results for different scenes, using GMP. These results are followed by ratios, taking the first column as reference

Name	Size*	Silhouettes and total rejection				Total rejection				Partial rejection			
		IN**	L**	CT**	MM**	IN	L	CT	MM	IN	L	CT	MM
Three Pens	1011 (1)	1432	620	7 min 57 s	19 Mb	1960 ×1.369	596 ×0.961	7 min 11 s ×0.904	12 Mb ×0.632	2470 ×1.725	747 ×1.205	7 min 47 s ×0.979	12 Mb ×0.632
One Pen	339 (1)	1811	754	3 min 25 s	137 Mb	3264 ×1.802	875 ×1.160	5 min 59 s ×1.751	185 Mb ×1.350	47851 ×26.422	18113 ×24.023	1 h 36 min 44 s ×28.312	2461 Mb ×17.964
Def. Sphere	323 (1)	6108	2079	6 min 57 s	197 Mb	9359 ×1.530	2586 ×1.244	20 min 56 s ×3.012	370 Mb ×1.878	83708 ×13.705	29502 ×14.190	2 h 50 min 02 s ×24.465	1849 Mb ×9.386
Room	2940 (2)	6912	3337	22 min 08 s		10220 ×1.478	3512 ×1.052	12 min 05 s ×0.546		24494 ×3.543	9003 ×2.698	27 min 02 s ×1.221	
Building	18880 (15)	4803	1930	33 min 17 s		6343 ×1.321	1883 ×0.976	26 min 44 s ×0.803		–	–	–	> 5 Gb

* Number of polygons (number of light sources)
** Average values per light source
IN number of inner nodes, L number of visible leaves, CT computation time, *MM* maximum memory usage during the visibility computation, if available

high visual complexities: One Pen, which contains many short edges in the pen's lead, and Deformed Sphere, which presents a moderately tessellated object. For both scenes, regarding only the rejection test impact, the BSP-tree sizes and the computation times are decreased by more than 90% with our method.

The memory usage concerns the maximal memory requirement during the visibility computation. It consequently depends on the sizes of both the BSP-trees and the polytopes. For applications, only the BSP-trees are required, so the memory requirement in those applications only depend on the number of inner nodes (six coordinates per inner nodes, for the hyperplane), which is significantly lower. Concerning the size of the polytopes, the more a polytope is split, the more its size grows due to the increasing number of vertices and edges necessary to represent it. It explains the high memory requirement for One Pen with a partial rejection test. This proves that our method is more stable in terms of memory consumption and enables us to deal with bigger scenes and more complex objects.
Concerning the silhouette processing, it can increase the computation time compared to the same method without silhouette. However, this method generally reduces the computation time compared to the partial rejection method, and it always reduces the number of inner nodes, whatever the method, complete or partial rejection. The result is a more balanced BSP-tree which accelerates its traversal in applications.

Finally, the Building shows our ability to process larger scenes with many tessellated objects, whereas before we were unable to process it with a partial rejection test, due to the excessive amount of memory required. This scene, composed of many rooms, also shows the advantage of our large occluders detection. Where the solid angle measure returns inappropriate results, our method efficiently detects the correct large occluders, which avoids processing the complete silhouette of complex objects, such as the chair in Fig. 9f, when those objects are hidden from lights by large walls.

4.2 Application

Our visibility computation method is applied in an interactive ray-tracer to compute soft shadows. The visibility data are precomputed and saved into files, which are then used to render soft shadows, based on Mora's visibility query [6], at an interactive frame rate. To save memory, the BSP-trees are loaded with single precision arithmetic values. The ray-tracer software uses a coherent approach based on Wald's previous work [15] to render 800×600 pixels images. Rays are intersected with the triangles four at a time, using SSE instructions.

On the scene Building, using the trees computed with the silhouette, the ray-tracer allows to trail round the rooms with about 2 to 13 fps, with an average of 3.5 fps. In the One Pen scene and in the worst case, when the camera moves closer to the lead shadow, the frame rate using the trees computed without the first rejection is about seven times slower (0.09 fps) than using the reference tree with silhouettes and a complete rejection test (0.65 fps).

This application shows the high speed of the tree traversals, thanks to their compactness, which maintains an interactive soft shadows rendering.

5 Conclusion

In this article, we propose new algorithms to compute the visibility between pairs of convex polygons in $\mathbb{R}^3$. Our algorithms offer some improvements to existing methods which can be easily plugged into the previous algorithms. First, we define a robust and exact technique to deal with degenerate cases. Second, our algorithms implement a new rejection test, based on the minimal representation of the visibility by a polytope in the Plücker space, which significantly reduces the visibility splitting. Third, we propose an algorithm to process only the silhouettes instead of the whole objects.

The splitting reduction has a direct impact on the use of the visibility data in any application: the compactness of the visibility BSP-tree increases the speed of the tree traversal, and the data extraction.

This visibility data extraction speed is of interest. Exact visibility precomputation can be, for example, used as an accelerating structure, in place of a classical kd-tree, in ray-tracing methods.

However, the current methods usability is limited to static scenes. We are now looking at computing soft shadows in animated scenes, for non-deformable objects, by computing the visibility in the bounding boxes of the objects in the scene and by transforming the hyperplanes in the BSP-trees (translate and rotate them) according to the position and orientation of the objects in the scene. We are also looking for a way to use the visibility data to determinate which polygon is intersected by a ray, without performing any intersection tests, but simply by traversing a BSP-tree containing the adequate data.

References

1. Bittner, J.: Hierarchical Techniques for Visibility Computations. Dissertation, Czech Technical University in Prague (2002)
2. Boissonnat, J.D., Yvinec, M.: Algorithmic Geometry (translated by Hervé Brönnimann). Cambridge University Press, Cambridge UK (1998)
3. Charneau, S., Aveneau, L., Fuchs, L.: Plücker space and polygon to polygon visibility computation with geometric algebra. Tech. rep., Laboratoire SIC, University of Poitiers, http://www.sic.sp2mi.univ-poitiers.fr/charneau (2006)
4. Haumont, D., Mäkinen, O., Nirenstein, S.: A low dimensional framework for exact polygon-to-polygon occlusion queries. In: Proceedings of the Eurographics Symposium on Rendering, pp. 211–222 (2005)
5. Laine, S., Aila, T., Assarsson, U., Lehtinen, J., Akenine-Möller, T.: Soft shadow volumes for ray tracing. ACM Trans. Graph. **24**(3), 1156–1165 (2005)
6. Mora, F., Aveneau, L.: Fast and exact direct illumination. In: Proceedings of CGI'2005, pp. 191–197. IEEE Computer Society, Stony Brooks University, New York, USA (2005)
7. Mora, F., Aveneau, L., Mériaux, M.: Coherent and exact polygon-to-polygon visibility. In: Proceedings of WSCG'05, pp. 87–94. UNION Agency – Science Press, University of West Bohemia, Plzen, Czech Republic (2005)
8. Naylor, B., Amanatides, J., Thibault, W.: Merging BSP trees yields polyhedral set operations. In: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques, pp. 115–124. ACM, New York (1990)
9. Nirenstein, S.: Fast and Accurate Visibility Preprocessing. Dissertation, University of Cape Town (2003)
10. Nirenstein, S., Blake, E., Gain, J.: Exact from-region visibility culling. In: Proceedings of the 13th Eurographics Workshop on Rendering, pp. 191–202. Eurographics Association, Aire-la-Ville, Switzerland, (2002)
11. Plücker, J.: On a new geometry of space. Phil. Trans. R. Soc. London **155**, 725–791 (1865)
12. Pu, F.-T., Mount, D.M.: Binary space partitions in Plücker space. In: Selected Papers from the International Workshop on Algorithm Engineering and Experimentation, vol. 32, pp. 94–113. ALENEX'99, Springer, Berlin Heidelberg New York (1999)
13. Sommerville, D.M.L.Y.: Analytical Geometry of Three Dimensions. Cambridge University Press, Cambridge, UK (1959)
14. The Free Software Foundation: GMP: The GNU Multiple Precision Arithmetic Library. http://www.swox.com/gmp (Cited 2006)
15. Wald, I.: Realtime Ray Tracing and Interactive Global Illumination. Dissertation, Saarland University (2004)

Sylvain Charneau has been a PhD student at the SIC laboratory, Université de Poitiers since October 2004. He received his MSc degree in Computer Science from Université de Poitiers. For his master's thesis, he implemented a library about the geometric algebras, using the Objective Caml language, and showed the advantages of a functional programming language to implement such algebras. He is now studying the advantages of the Grassmann algebras to formulate visibility computation in any dimension.

Lilian Aveneau has been an assistant professor at the University of Poitiers since 2000, and manages its research in the SIC Laboratory. He received his MSc degree in Computer Science from the University of Bordeaux. He received his PhD in Computer Science from the University of Poitiers in 1999. His main research interests include fast rendering, from polygon visibility computations and their applications, and radiocommunication propagation.

Laurent Fuchs is an assistant professor at the Université de Poitiers, SIC Laboratory. He graduated in Mathematics from the Université de Strasbourg. He received his PhD in Computer Science from the Université de Strasbourg. His main research interests include topological based modeling, formal computer graphics, and geometric algebras.