Fast and Exact Direct Illumination

F. Mora* L. Aveneau†

SIC - University of Poitiers
France

Abstract

Rendering high quality soft shadows from area light sources is necessary to increase the level of realism. Quality soft shadows are closely related to the visibility computation. An accurate visibility information will improve the shadows realism. However, this remains a challenging problem since even small visibility approximations can lead to unacceptable errors in a picture. We propose a new approach to this problem, based on an exact visibility pre-computation, done in the Plücker space. It takes advantage of this first step to provide an exact from-point visibility query algorithm. We propose several results in a ray tracing application, where the direct illumination at any given point is provided with a fast computation, and with a high degree of quality.

Keywords: exact visibility, soft shadows, direct illumination

1 Introduction

In realistic rendering, soft shadows due to direct illumination are of great importance for the image quality. In particular, the shadows cast by an area light source represent a challenging problem. Many works have been done to compute them efficiently.

In interactive rendering, there are two general algorithms for computing soft shadows : the shadow map and the shadow volume, which allow to compute the penumbra from a linear or area light source [9] [1]. Even if these techniques are fast, and can be implemented in a GPU, they can present some drawbacks when it comes to soft shadows estimation.

In realistic rendering, the classical technique consists in computing the shadows due to the direct illumination from area light sources, using a sampling approach [13]. Of course, this may introduce noise in the results, and imply high computation times. Recently, Szécsi proposed a solution based on correlated sampling [15], to replace the importance sampling when the light source is fully visible, allowing to reduce the image noise.

Since realistic soft shadows depend on the visibility between geometric primitives, previous works attempt to take advantage of an accurate visibility information to enhance the rendering quality. Durand [6, 8] built a visibility skeleton encoding the visibility events in a scene. He uses this information to improve rendering in a radiosity context. However this construction has difficulties to handle visual events due to degenerated geometric configurations. A robust version of the visibility skeleton was developed by Duguet [5]. This new algorithm introduces an epsilon-imprecision and was applied to compute accurate shadows with point or directional light source.

In this paper, we present a method that has a great potential to enhance high quality rendering. Based on a particular solution to the exact polygon-to-polygon visibility problem [10], we derive an algorithm to extract an exact point-to-polygon visibility representation. As a first application, we apply our method to compute the direct illumination in a ray tracing rendering tool. The first results demonstrate that our method allows to render scene fast, while soft shadows remain exact.

In the second section of this paper, we recall the fundamentals allowing to compute an exact visibility information. The third section presents the structure we use to encode the visibility data between two polygons. Section four explains how to take advantage of this information to efficiently extract a from-point visibility information. At last, the section five shows how this can be applied to a ray tracing rendering tool and proposes some results which are discussed.

2 Preliminaries

An important part of the work presented in this paper relies on the ability to compute an exact description of the visibility between two polygons. The four dimensional nature of the visibility in 3D environments has prevented for a long time from leading to tractable solutions. Nirenstein [12] and Bittner [2] have recently published the two first solutions. They both solve the problem by using five dimensional CSG operations in the Plücker space. In this section, we recall the fundamentals on the Plücker space and the main principles to compute exact visibility between polygons.

2.1 The Plücker space

The Plücker space [14] is a five dimensional projective space $\mathbb{P}^5$. It provides an elegant parametrisation for dealing with directed lines in $\mathbb{R}^3$. Each line l passing through the point (p_x, p_y, p_z) and next through (q_x, q_y, q_z) is defined in $\mathbb{P}^5$ by $\pi_l = (\pi_0, \pi_1, \pi_2, \pi_3, \pi_4, \pi_5)$, with :

$$\begin{array}{ll} \pi_0 = q_x - p_x & \pi_3 = q_z p_y - q_y p_z \\ \pi_1 = q_y - p_y & \pi_4 = q_x p_z - q_z p_x \\ \pi_2 = q_z - p_z & \pi_3 = q_y p_x - q_x p_y \end{array}$$

Notice that (π_0, π_1, π_2) is the direction of l while (π_3, π_4, π_5) encodes its location.

Next, let us consider the dual mapping within $\mathbb{P}^5$: each $\pi \in \mathbb{P}^5$ can be associated to a dual hyperplane h_π defined by :

$$\begin{array}{lll} h_\pi = \{x \in \mathbb{P}^5 & | & \pi_3 x_0 + \pi_4 x_1 + \pi_5 x_2 \\ & & +\pi_0 x_3 + \pi_1 x_4 + \pi_2 x_5 = 0\} \end{array}$$

Given two lines l_1 and l_2 and their Plücker mapping π_{l_1} and π_{l_2}, a crucial property is : l_1 and l_2 are incident if and

*e-mail: mora@sic.univ-poitiers.fr
†e-mail: aveneau@sic.univ-poitiers.fr

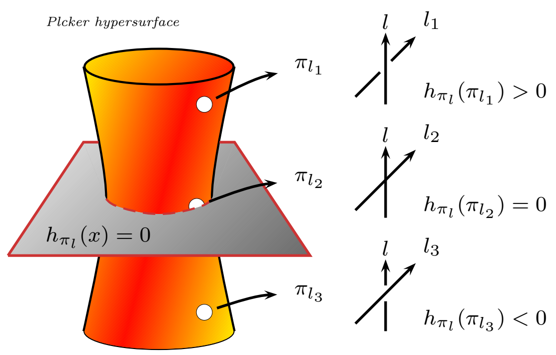


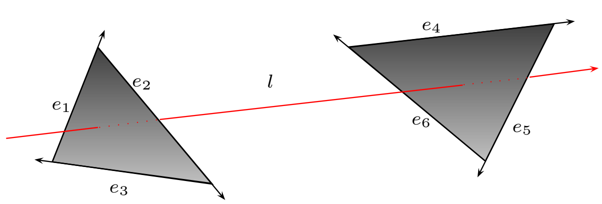
Figure 1: Line orientation in the Plücker space. There are three different cases for an oriented line to pass another: l_1 passes on the left of l_0, l_2 is incident on l_0 and l_3 passes l_0 on the right. The Plücker mapping of l_1, l_2, l_3 will respectively lie above, on and below the dual hyperplane of l_0.

only if π_{l_1} lies on the dual hyperplane of π_{l_2} (and vice versa). If $h_{\pi_{l_1}}(\pi_{l_2}) \neq 0$, the sign of $h_{\pi_{l_1}}(\pi_{l_2})$ determines the relative orientation of l_1 and l_2 as illustrated on figure 1.

At last, each line in $\mathbb{R}^3$ maps to a point in $\mathbb{P}^5$ but each point in $\mathbb{P}^5$ does not map to a line in $\mathbb{R}^3$. The mapping of all real lines in $\mathbb{P}^5$ forms a four-dimensional quadric surface called the *Plücker hypersurface*.

2.2 Exact From Polygon Visibility Principle

Previous definitions are useful to characterise the set of lines stabbing convex polygons. In the Plücker space, these lines form a connected set of points on the hypersurface. For computational convenience, it is easier to deal with a polyhedral representation of this subset by using the dual hyperplane mapping of the polygon edges. The intersection of this polyhedral structure with the Plücker hypersurface gives exactly the set of lines stabbing each polygons. Such an approach was already used by Teller [16] to compute the anti-penumbra of an area light source through a sequence of polygons. Figure 2 illustrates a two triangles example, the



$h_{\pi_{e_1}}(\pi_l) \geq 0 \quad h_{\pi_{e_2}}(\pi_l) \geq 0 \quad h_{\pi_{e_3}}(\pi_l) \geq 0$
$h_{\pi_{e_4}}(\pi_l) \geq 0 \quad h_{\pi_{e_5}}(\pi_l) \geq 0 \quad h_{\pi_{e_6}}(\pi_l) \geq 0$

Figure 2: Lines stabbing two polygons : The Plücker mapping of the polygons edges induces the hyperplane representation of a polytope. Its intersection with the Plücker hypersurface is the set of all lines stabbing the two polygons.

simplest case in a context of polygon to polygon visibility. More generally, if A and B are two polygons with n and m edges $e_1, ..., e_{n+m}$ consistently oriented, all the lines l passing through A then B satisfy :

$$\forall \text{ i} \in [1..n+m],\ h_{\pi_{e_i}}(\pi_l) \geq 0$$

This system of inequations is the hyperplane representation of an unbounded polyhedron in the Plücker space. Additional constraints are added to obtain a closed polyhedron : a polytope. The polytope representation allows to limit computations to the zone of the Plücker hypersurface.

Let P_{AB} be the polytope that represents the set of lines stabbing A and B. Figure 3 gives a 2D illustration of the process that removes from P_{AB} the set of lines blocked by an occluder. This has to be applied to each occluder. The remaining parts of P_{AB} intersecting the hypersurface are exactly the set of lines that stabs A and B without stabbing any occluders. If such a part does not remain, A and B are not visible.

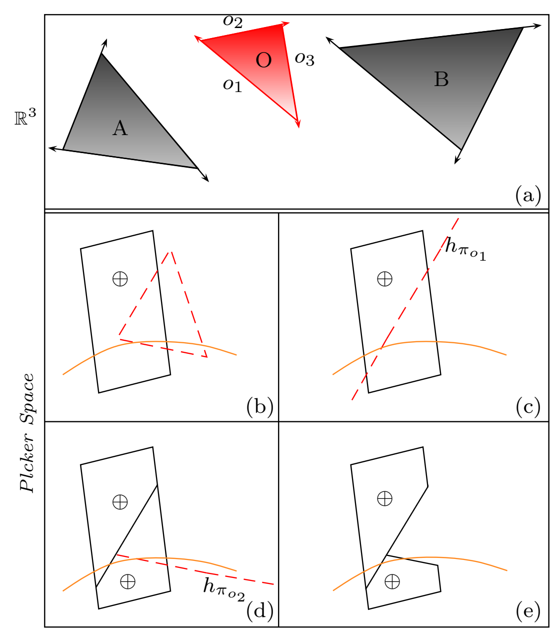


Figure 3: (a) An occluder O blocks some visibilities between two polygons A and B. (b) The Plücker representation of lines stabbing A and B and lines stabbing O. (c) (d) : To remove the subset of blocked lines, P_{AB} is successively split in sub-polytopes using the hyperplanes associated to the occluder edges. (e) The sub-polytope corresponding to blocked lines is removed.

CSG computation in the Plücker space allows to encode all visibilities, whereas a visibility skeleton provides only visual events (i.e lines that are incident to polygons vertices or polygons edges). To make a geometric comparison, the visual events are related to the intersections of the polytopes boundaries with the Plücker hypersurface, whereas the full visibility information is provided by the whole polytope intersection with the hypersurface.

Both Nirenstein algorithm and Bittner algorithm rely on the previous principles. However, they do not provide the same information. On the one hand, Bittner algorithm builds a hierarchical occlusion tree that provides a structured visibility information for a source polygon facing a scene. Bittner occlusion tree [2] is a $5D$ BSP tree that classifies all the rays emerging from the source polygon. A leaf represents a set of unoccluded or occluded rays. In the latter case, the corresponding occluder is associated to the leaf. To provide a depth information, the polygons are processed in a front-to-back order. On the other hand, Nirenstein algorithm is computationally suited for polygon-to-polygon query [11]. However, its purpose is to query if at least one visibility exists between the polygons. The visibility data is maintained as a non-organised set of polytopes, and dropped as soon as the visibility or invisibility is determined.

To render exact soft shadows, our approach requires an exact visibility description between two polygons. As a consequence, we use an algorithm similar to the Nirenstein's one, but modified to provide a structured visibility information like the Bittner's one.

3 From-polygon visibility query

This section presents our approach to compute an exact representation of the visibility between two polygons. Our motivation is to minimize the number of polytopes that describe the visibility. The first reason is to reduce the storage cost of the visibility data. The second reason is to increase the information coherence and to allow a more efficient use after its computation. This will be detailed in the next section.

3.1 The history tree

Given two polygons, our approach builds a binary, tree called a "history tree" since it can be understood as the history of the successive splitting operations. Each inner node stores an oriented edge that was used to create a splitting hyperplane in the Plücker space. Each leaf represents either a set of visibilities between the two queried polygons, or a set of blocked lines. We now detail the history tree construction and refer to the raw algorithm 1.

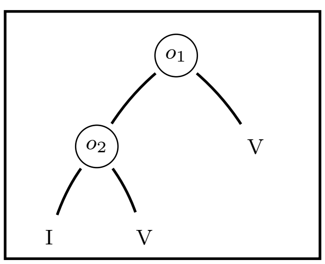


Figure 4: A simple history tree example : This is the history tree corresponding to the figure 3 example. o_1 and o_2 are the occluder edges mapped to a splitting hyperplane in the Plücker space.

During the process, a leaf is marked as *visible*, *invisible*, *undefined* or *rejected*. In the latter case, this means that the associated polytope can not be affected anymore by the current occluder removal. The algorithm starts with a *visible* root node associated to the initial polytope that represents all the lines stabbing the two queried polygons (line 25). Next, each potential occluder is removed using the same process : each occluder edge is mapped to a hyperplane in the Plücker space (line 28). If a hyperplane splits a polytope associated to a leaf marked *visible* or *undefined* (lines 29,30), this one becomes an inner node that stores the oriented edge mapped to the hyperplane. Left and right children are added and respectively marked *undefined* and *rejected*. They are associated with the negative and positive parts of the polytope that was split (line 31). After each occluder removal, the *undefined* leaves can be marked *invisible* (line 35), while the *rejected* leaves can be reset to *visible* (line 36). The algorithm ends with the last occluder removal. Figure 4 shows the history tree that would be produced with the figure 3 example.

Contrary to an occlusion tree, a history tree does not provide a depth information. In our context, this is not necessary, since we are only interested in unoccluded rays between the two polygons.

```
1  function Init (Polygon B, L) return HTree
2  Polytope polytope
3  HTree res
4  begin
5      polytope ← newPolytope(B; L)
6      res ← newHTree (polytope)
7      setMark (res ; visible)
8      return res
9  end

10 procedure SplitLeaf (HTree H; Plucker hplane;
   Edge edge)
11 Polytope polytope⁺, polytope⁻
12 begin
13     polytope⁺ ← H.polytope ∩ hplane⁺
14     polytope⁻ ← H.polytope ∩ hplane⁻
15     H.orientedEdge ← edge
16     H.posTree ← newHTree (polytope⁺)
17     H.negTree ← newHTree (polytope⁻)
18     setMark (H.posTree, rejected)
19     setMark (H.negTree, undefined)
20 end

21 function P2PQuery (Polygon B, L) return HTree
22 HTree H
23 Plucker hplane
24 begin
25 |   H ← Init (B, L)
26 |   foreach occluders O between B and L do
27 |   |   foreach edge of O do
28 |   |   |   hplane← map2Plucker (edge)
29 |   |   |   foreach leaf l of H marked as visible or
   |   |   |   undefined do
30 |   |   |   |   if hplane ∩ l.polytope ≠ ∅ then
31 |   |   |   |   |   SplitLeaf(l, hplane, edge)
32 |   |   |   |   end
33 |   |   |   end
34 |   |   end
35 |   |   Mark all undefined leaves of H as invisible
36 |   |   Mark all rejected leaves of H as visible
37 |   end
38 |   return H
39 end
```

Algorithm 1: Pseudocode for building the history tree (P2PQuery) associated to two polygons B and L. Notice this is a raw algorithm.

3.2 Visibility data improvement

The history tree provides some optimisations facilities to minimize and improve the visibility information. Inner nodes with both visible or both invisible children are replaced by one visible or invisible leaf. Moreover, all splitting operations are not necessary. For example, a hyperplane can split a polytope without affecting its intersection with the plcker hypersurface. Such a split adds unnecessary information in the history tree. As the history of the successive splitting operations, the history tree can be easily restored to its previous configuration. The algorithm 1 presents a simplified pseudocode that does not take into account such optimisations. They are detailed in [10] and help to restrict the oriented edges stored in the tree to the ones having a real impact on the visibility between the two polygons. This is depicted by the figure 5.

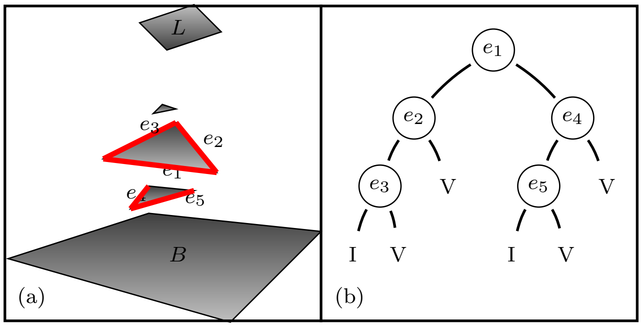


Figure 5: An improved history tree : The oriented edges stores in the nodes are only those having an impact on the visibility between the top polygon L and the bottom one B.

4 From-point visibility query

Given two polygons A and B and their associated history tree, this section presents a simple algorithm to compute efficiently the visible parts of A from any point on B (or vice versa).

Obviously, the visibility from a point on one polygon is a subset of the visibility data between the two polygons. According to the same observation, the occluders edges that define visual events from the point with the queried polygon, are a subset of the occluders edges stored in the history tree.

As a consequence, our solution relies on the history tree traversal to isolate the edges having an impact on the visibility between the point and the queried polygon, as depicted by figure 6. The recursive algorithm 2 details the process. It assumes that the history tree of two polygons B and L has been computed (line 41) and that we want to extract the visible parts of L from a point on B (line 42,43). It starts from the root node using a copy of L. For each inner node met (line 51), the 3D plane defined by the point and the oriented edge stored in the node is computed (line 52). Then the polygon is tested against this plane (line 54). If it lies in the positive (line 55) or negative (line 57) half-space of the plane, the algorithm continues respectively in the right or left subtree. If it is split by the plane (line 59), the algorithm continues in both subtrees with the two relevant parts of the polygon. Fragments that reach a visible leaf (line 48) are exactly the visible parts of the polygon from the point. Those that reach an invisible leaf can not be seen from the point.

We can notice that a from-point algorithm such as [3] could be used to compute exactly the visible parts of a polygon. However, a BSP tree has to be built for each query with a complexity that depends on the occluders number between the point and the polygon. In comparison, the same history tree can be used for any point on one polygon. Moreover, the complexity of a query is restricted to the nodes number

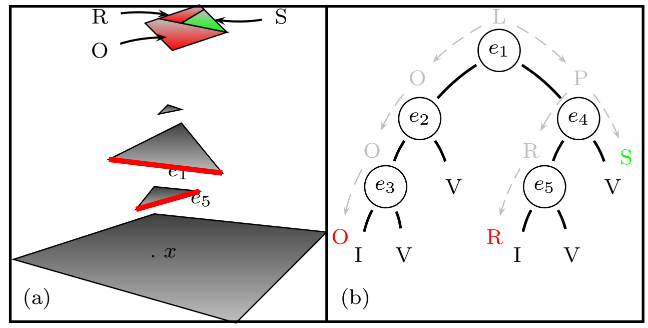


Figure 6: From-point visibility using the history tree : From the root node, the queried polygon L is first split by the plane defined by e_1 and x. The same process continues in the both subtree with O and P, the relevant fragments of L. The visibility of a fragment is determined as it reaches a visible or an invisible leaf.

in the history tree, which depends on the visual complexity between the two polygons. The test of the edges that do not affect the two polygons visibility can be avoided.

40 Polygon B, L
41 HTree $H \leftarrow$ `P2PQuery` (B, L)
42 Point $P \leftarrow$ a point on B
43 Polygon $Poly \leftarrow$ `Copy` (L)
44 Set $VFragments \leftarrow \emptyset$
45 **procedure** `FPQuery` (HTree H; Polygon $Poly$)
46 3DPlane $orientedPlane$
47 **begin**
48 **if** `visibleLeaf` (H) **then**
49 $VFragments \leftarrow VFragments \cup Poly$
50 **else**
51 **if** `innerNode` (H) **then**
52 $orientedPlane \leftarrow$ `newPlane` (P,
53 $H.orientedEdge$)
54 **switch** $Poly \cap orientedPlane$ **do**
55 **case** $POSITIVE$
56 `FPQuery` ($H.posTree$, $Poly$)
57 **case** $NEGATIVE$
58 `FPQuery` ($H.negTree$, $Poly$)
59 **case** $SPLIT$
60 `FPQuery` ($H.posTree$, $Poly^+$)
61 `FPQuery` ($H.negTree$, $Poly^-$)
62
63 **end**
64 **end**
65 **end**
66 **end**

Algorithm 2: Using the history tree of two polygons B and L, FPQuery algorithm compute the visible parts of L from a point on B

This explains why it is worth minimizing the history tree size and trying to compute a visibility information as coherent as possible.

5 Rendering Exact Soft-Shadows

This section shows how to take advantage of the history tree in order to render high quality soft shadows. First, we

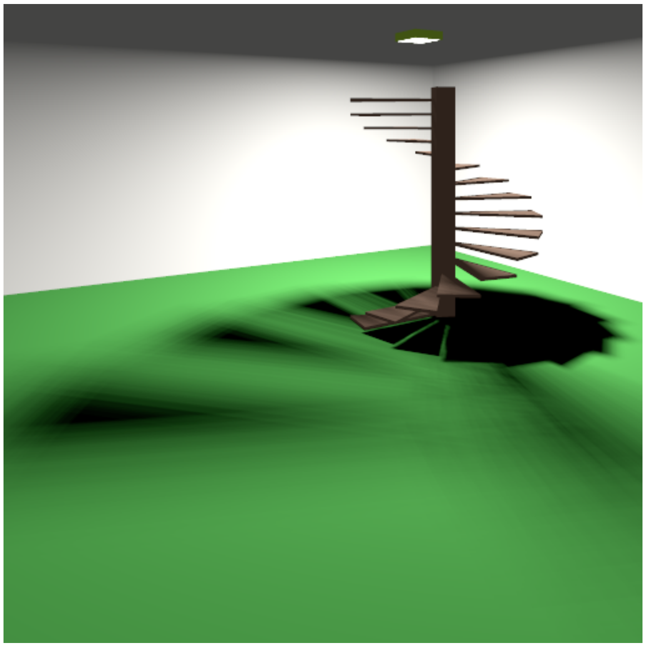


Figure 7: Classical Ray-tracing image with the stairs scene.

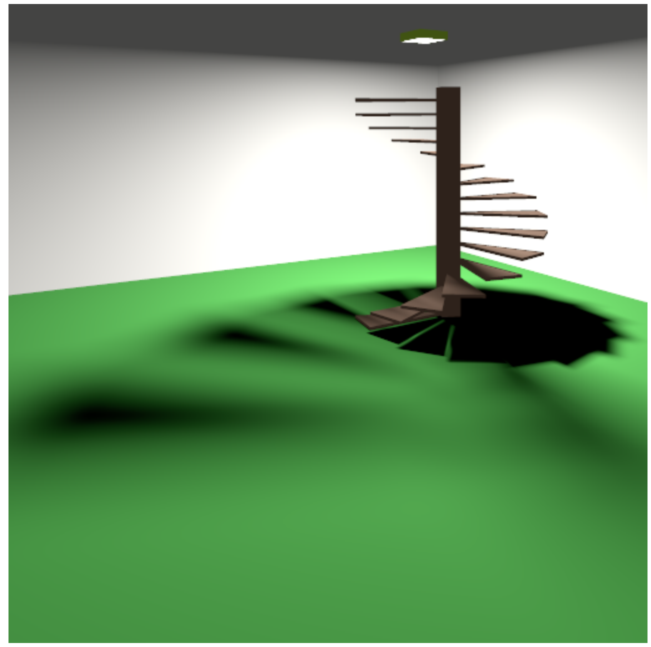


Figure 8: The stairs scene computed with from-point visibility query.

explain our approach before presenting several results for different scenes. Then, we propose a brief discussion on the perspectives.

5.1 Application

The from-point visibility query is applied in a ray-tracing software to compute the soft shadows due to direct illumination. This contribution can be written using the following integral [13] :

$$L^r(\vec{x},\omega) = \int_{S_e} L^e(\vec{y},\omega_{\vec{y}\to\vec{x}}).f_r(\omega_{\vec{y}\to\vec{x}},\vec{x},\omega). \\ v(\vec{x},\vec{y}).G(\vec{x},\vec{y})d\vec{y}$$

where S^e is the set of light source points, L^e is the irradiance emitted by $\vec{y}$ in the direction towards $\vec{x}$, f_r is the BRDF at point $\vec{x}$, v is the visibility function and G is the geometric factor.

Since there is no analytic solution to this integral, it is usually approximated using a Monte-Carlo approach, for instance with importance sampling or coherent sampling [15]. This has two drawbacks : first, the visibility query is usually done by a ray casting approach, leading to an over computation time ; secondly, for a complex visibility between the area light source and the point $\vec{x}$, a good evaluation of the light illumination implies a high number of samples, increasing again the computation time, even for a diffuse light source.

Our approach consists in replacing the visibility computation by a from-point visibility query. The visibility from each area light source to each polygon of a given scene, are precomputed and stored in a graph. We do not use any discontinuity meshes, only the raw polygons of the scene. As our method does not depend on the view point, this is done once in a pre-treatment step, and then stored into a file.

Each time the direct illumination needs to be computed at point $\vec{x}$, we use the visibility graph data for a from-point visibility query. This gives a list of the area light source fragments visible from $\vec{x}$. Each fragment is sampled depending on the area light source BRDF, in order to give a precise evaluation of the direct illumination contribution. Moreover, for a diffuse area light source, we use a faster approach : we compute the ratio between the area of the light source and the sum of the visible light source fragments areas, and use only one light source point set at the light source barycenter.

An important remark is that we never have to evaluate the visibility factor v, since we only choose light samples in visible fragments. This is the key point of the computation time reduction factor presented in 5.2.

Notice that Durand also uses the visibility skeleton to compute exact point-to-polygon form factors. However, since only the visual events are encoded, only the polygons vertices can be queried. In comparison, our point-to-polygon query has no restriction.

5.2 Results

For comparison purpose, we use a ray-tracing software that computes soft shadows with a stratified sampling strategy. Each direct illumination contribution is computed with 128 random points on each area light sources. The software uses a regular 3D grid to speed up the intersection tests.

Table 1: Computation time for the three test scenes, with a classical ray tracing (RT) and the from-point visibility query technique (FPVQ) ; the column PC shows the pre computation time for the visibility graph construction ; the column Size indicates the visibility graph memory size. All the images are computed with 16 samples per pixel.

Scene	RT	FPVQ	PC	Size
Stairs	26’ 09”	35”	6”	22Ko
Table	16’40”	27”	2’37”	230Ko
Grids	18’43”	30”	3”	33 Ko

Figure 9: Classical Ray-tracing image with the table scene.

Figure 10: The table scene computed with from-point visibility query.

We choose three different test scenes. In spite of their small polygons number, they present various visibility configurations. The first one (see figures 7 and 8) contains stairs and is made of 244 triangles. It allows to emphasize the interest of our technique, since even with 128 light source samples, the penumbra are not properly estimated in a classical ray tracing image. The from-point visibility query gives exact soft shadows, with a non negligible improvement of the computation time, as shown on Table 1.

The second test scene is made of 1348 triangles, including a table, four chairs and a detailed house plant. It allows to observe the robustness of the visibility graph construction (see figures 9 and 10). On the one hand, the reduction of the computation time is once again clear. On the other hand, the quality of both images is similar. This is due to the fact that 128 light source samples are sufficient to compute the soft shadows with a classical ray tracing approach.

The last test scene (see figures 11 and 12) is very simple (154 triangles), but emphasizes complex visibility events because of the two grids. Using our method the rendering produces a convincing image. On the contrary, the classical approach using 128 random points on area light sources clearly does not give good results. Moreover, as with the two previous scenes, we have a computation time reduction.

5.3 Discussion

Results demonstrate that our approach produces images with high quality soft shadows. The scenes used present a sufficient visibility complexity to test the correctness of our approach. Nevertheless, it is clear that we have to develop our work to handle scenes with higher polygon numbers. This raised several questions about the scalability of our method. The pre-computation step increases with the scene size. It grows linearly with the polygons number and each polygon-to-light query is related to the average visual complexity between a polygon and an area light source. Since the classical ray tracing complexity is sub-linear as the polygons number increases, it turns out that the whole computation time (i.e the sum of the FPQV and PC times) could become more important with our technique than without it. However we believe that important scenes can be efficiently rendered before reaching this critical point. Moreover, interesting ideas can be taken from the Nirenstein framework [11] to speed up the pre-computation step. Unless a geometric modification in the scene is performed, this preprocess step only needs to be done once.

Another question is the storage cost of the visibility data. The table 1 does not exhibit prohibitive storage costs. The storage complexity of a history tree is related to the average visibility complexity between two polygons. A node of such a tree can be implemented using only 4 integers, the two first ones for the edge vertices pointers inducing a visibility event, and the two others for the children. Our results show that the storage cost is not an important limitation.

A last question is about the point-to-polygon visibility query efficiency. A high complexity between two polygons could lead to a history tree with an important size. This could slow down the point-to-polygon query. Even with complex objects, the visibility complexity depends on the area size of the queried polygon. Some solutions can solve this problem. An obvious idea is to use a discontinuity meshing algorithm. We plane to test this solution in the future. Another solution is given by Bittner. He uses shafts to balance its occlusion tree according to the visibility complexity. The same idea can be applied to bound the size of a history tree.

Nevertheless, the fact is that the soft shadows are greatly enhanced by using our technique. These results are very promising and show that an exact visibility satisfies a high degree of quality. It can be useful in realistic rendering or to provide reference images to valuate approximate solutions. As a future work, we plane to include it in a Monte-Carlo approach, since clearly we think that it will reduce the image noise, allowing to decrease the computation time as for the ray tracing.

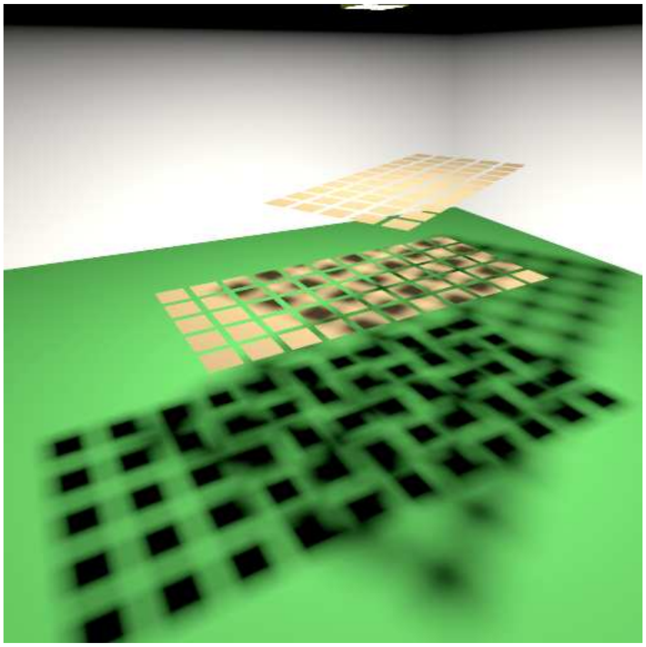


Figure 12: The grids scene computed with from-point visibility query.

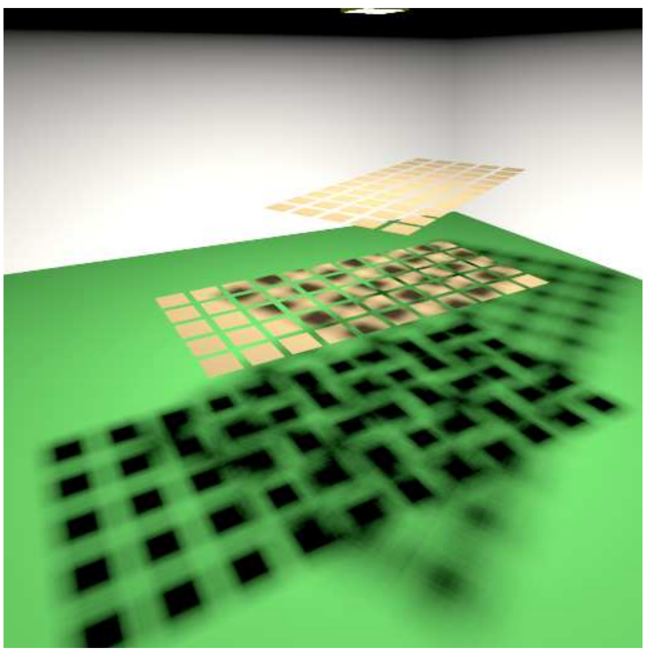


Figure 11: Classical Ray-tracing image with the grids scene.

6 Conclusions

This paper has shown the potential of exact visibility to compute high quality shadows due to a direct illumination from an area light source. We have described how an exact polygon-to-polygon visibility is encoded in a history tree. Next, we have presented a point-to-polygon visibility algorithm that takes advantage of the history tree to compute the visible parts of a polygon. The first results have demonstrated that soft shadows can be exactly computed. In addition, the preprocessed visibility information helps to speed up the rendering. In the future, we will continue to improve our ray tracing solution. But, since the visibility computation is a common problem in all the rendering methods, we also think of enhancing other techniques such as a Monte Carlo approach.

References

[1] U. Assarsson and T. Akenine-Mller. Occlusion culling and z-fail for soft shadow volume algorithms. *The Visual Computer*, 20(8-9), November 2004.

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

[3] J. Bittner, V. Havran, and P. Slavík. Hierarchical visibility culling with occlusion trees. In *Proceedings of Computer Graphics International '98 (CGI'98)*, pages 207–219. IEEE, 1998.

[4] George Drettakis and Eugene Fiume. A fast shadow algorithm for area light sources using backprojection. In *Computer Graphics*. ACM SIGGRAPH, July 1994. Proceedings of Siggraph'94.

[5] F. Duguet and G. Drettakis. Robust epsilon visibility. In *Computer Graphics*, July 2002. Proceedings of ACM Siggraph 2002.

[6] F. Durand. *3D Visibility: Analytical Study and Applications.* PhD thesis, Universite Joseph Fourier, Grenoble, France, July 1999.

[7] Frédo Durand, George Drettakis, and Claude Puech. The visibility skeleton: a powerful and efficient multi-purpose global visibility tool. *Computer Graphics*, 31(Annual Conference Series):89–100, 1997.

[8] Frédo Durand, George Drettakis, and Claude Puech. Fast and accurate hierarchical radiosity using global visibility. *ACM Transactions on Graphics*, 18(2):128–170, 1999.

[9] W. Heidrich, Stefan Brabec, and H-P. Seidel. Soft shadow maps for linear lights. In *Proceedings of the Eurographics Workshop on Rendering'00*, 2004.

[10] F. Mora, L. Aveneau, and M. Mriaux. Coherent and exact polygon-to-polygon visibility. In *Proceedings of WSCG'05*, 2005.

[11] S. Nirenstein. *Fast and accurate visibility preprocessing.* PhD thesis, University of Cap Town, South Africa, October 2003.

[12] S. Nirenstein, E. Blake, and J. Gain. Exact from-region visibility culling. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 191–202. Eurographics Association, 2002.

[13] P. Shirley, C. Wang, and K. Zimmerman. Monte carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(3):1–36, 1996.

[14] Sommerville. *Analytical Geometry in Three Dimension.* Cambridge University Press, 1959.

[15] László Szécsi, Mateu Sbert, and László Szirmay-Kalos. Correlated and importance sampling in direct light source computation and environment mapping. In *Proceedings of Eurographics'04*, 2004.

[16] Seth J. Teller. Computing the antipenumbra of an area light source. *Computer Graphics*, 26(2):139–148, 1992.