Efficient Construction of Visibility Maps using Approximate Occlusion Sweep

Jiří Bittner[*]

Center for Applied Cybernetics
Czech Technical University in Prague

Abstract

We present an algorithm that efficiently constructs a visibility map for a given view of a polygonal scene. The view is represented by a BSP tree and the visibility map is obtained by postprocessing of that tree. The scene is organised in a kD-tree that is used to perform an approximate occlusion sweep. The occlusion sweep is interleaved with hierarchical visibility tests what results in expected output sensitive behaviour of the algorithm. We evaluate our implementation of the method on several scenes and demonstrate its application to discontinuity meshing.

Keywords: Visibility, BSP tree, discontinuity meshing

1 Introduction

Computation of visibility maps is related to the problem of visible surface determination. Visible surface algorithms aim to determine a collection of visible surfaces for a given view of the scene. A visibility map contains more information — it captures also the topology of the view. Visibility map is a graph representing the view of a polygonal scene in which vertices, edges and faces are associated with vertices, edges and polygons of the scene. Each element of the visibility map holds information about its adjacent elements. See Figure 1 for an example of a visibility map.

Visibility maps can be used to construct an approximate discontinuity mesh [16, 12] in the context of radiosity global illumination algorithm. Another their application is efficient antialiasing for high resolution rendering [10]. Visibility maps can also guide occluder preprocessing for real time visibility culling [1, 5, 13]. A visibility map provides cues that can help a user to understand the view of the scene [10].

We present an algorithm that efficiently constructs a visibility map for a given view of the scene. The method does not rely on a single projection plane and easily handles a view of 360 degrees that spans the whole spatial angle. The algorithm is exact in the sense that it does not use a discrete representation of the view. The view is represented hierarchically what allows its efficient construction and postprocessing. The algorithm uses an approximate occlusion sweep interleaved with hierarchical visibility tests. This concept results in output sensitive behavior of the algorithm in practice without the necessity of complicated data structures for obtaining exact priority order of scene polygons.

[*]bittner@fel.cvut.cz

2 Related work

A lot of research has been devoted to visibility problems due to their importance in computer graphics, computer vision, and robotics. A recent interdisciplinary survey was published by Durand [6]. Computation of visibility maps is related to the visible surface determination [9]. Traditional visible surface algorithms such as the algorithm of Watkins, Weiler-Atherton, Warnock [7] or Fuchs et al. [8] provide output that can be used for the construction of visibility maps. Unfortunatelly these methods do not scale very well to large scenes with dense occlusion.

The visible surface algorithms are nowadays dominated by z-buffer that is often implemented in hardware. Nevertheless it is difficult to reconstruct a visibility map from the discretized image obtained by the z-buffer algorithm. Another drawback of z-buffer is the lack of output sensitivity of the algorithm. Therefore many recent techniques aim to increase efficiency of z-buffered rendering by visibility culling [1, 5, 11, 13].

Recently computation of visibility maps was studied by Stewart and Karkanis [16]. They propose an interesting algorithm for construction of approximate visibility maps using dedicated graphics hardware. They first render the scene in the *item buffer*. Then they construct a rectilinear graph that is *relaxed* to match the edges and vertices of visible scene polygons. The drawback of the algorithm is that it can fail to correctly relax all features of the visibility map. Grasset et al. [10] dealt with theoretical operations on visibility maps and their applications in computer graphics.

3 Algorithm overview

The proposed algorithm consists of two main steps: Firstly an occlusion tree [1] is constructed for a given view of the scene. The occlusion tree is conceptually a BSP tree

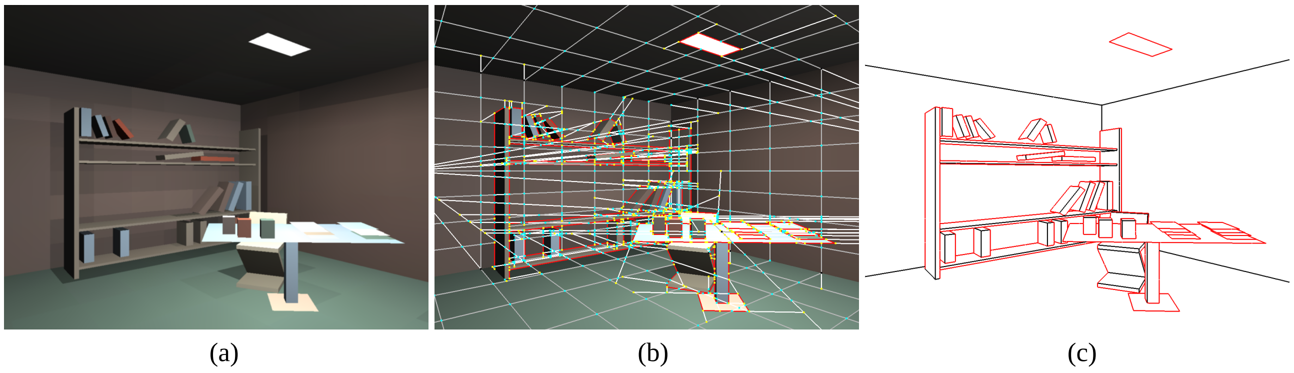


Figure 1: (a) A view of a scene with 556 polygons. (b) The view with the corresponding visibility map. Color coding of the visibility map vertices corresponds to the number of adjacent edges. Edges are colored depending on their semantic classification. (c) A sweep through the visibility map allows to identify only edges corresponding to "corners" (black) and edges forming "shadow" boundaries (red).

representing the view. Secondly the visibility map is constructed by postprocessing of the occlusion tree. The hierarchical structure of the occlusion tree is used for efficient lookups of adjacent elements of the visibility map.

The occlusion tree is constructed using an approximate occlusion sweep with respect to the given viewpoint. Scene polygons are swept in an approximate front-to-back order that is established using a kD-tree. The order is approximate in the sense that a currently processed polygon can be occluded by a constant number of unprocessed polygons. The occlusion tree is constructed incrementally by inserting the currently processed polygon. At each step the occlusion tree represents the view of the scene consisting of already processed polygons. The traversal of the kD-tree is interleaved with hierarchical visibility tests applied on its nodes. The visibility test uses the current occlusion tree to determine visibility of a region corresponding to the given node of the kD-tree. If the region is invisible the corresponding node and its whole subtree are culled.

When the occlusion tree represents the complete view it is used to construct the visibility map. Each non-empty leaf of the occlusion tree corresponds to a fragment of a visible polygon. Visibility map is constructed by inserting the visible fragments and updating adjacency links to the fragments already processed. For each fragment the occlusion tree is used to efficiently locate its neighbor fragments. All subsequent operations are restricted to the located neighbors. When the construction of the visibility map is finished a simple sweep through the map can classify its edges and vertices into several categories. Based on this classification the visibility map can pruned depending on the particular application.

The rest of the paper is organized as follows: The next section outlines the concept of approximate occlusion sweep. Section 5 discusses the construction of occlusion tree. Section 6 briefly discusses the use of occlusion tree for hierarchical visibility tests. In Section 7 we discuss the construction of visibility map by postprocessing of the occlusion tree. Section 8 contains evaluation of our implementation of the proposed method. Finally Section 9 concludes the paper.

4 Approximate occlusion sweep

Traditional list-priority methods for visible surface determination aim to determine strict priority ordering of the scene polygons. A popular approach is the algorithm using an autopartition BSP tree [8] for the scene polygons. By simple traversal of the tree a strict front-to-back or back-to-front order of the polygons can be determined. The disadvantage of the method is that the BSP tree increases the amount of scene polygons due to splitting. Additionally the tree is not well suited to dynamic scenes since the partitioning planes are aligned with the scene polygons.

We use a novel concept of approximate priority ordering: *approximate occlusion sweep*. The approximate occlusion sweep processes the scene polygons in an approximate front-to-back order: a currently processed polygon can be occluded by k unprocessed polygons. In practice k is typically very small and very often $k = 0$. The main advantage of the method is that almost any common spatial index (kD-tree, octree, bounding volume hierarchy, ...) can be used to establish the approximate front-to-back order.

The approximate occlusion sweep is used to construct the occlusion tree and so the tree construction algorithm must be able to process polygons in reverse order ($k > 0$). In our method this case is resolved consistently with a certain performance penalty as discussed in Section 5.

We used kD-tree to organize the scene polygons that is constructed according to the *surface are heuristics* [14]. The tree is built by recursive subdivision until certain termination criteria are met. Leaves of the tree contain references to scene polygons. In practice each leaf contains a small number of references (number of objects per leaf is one of the termination criteria of the tree construction

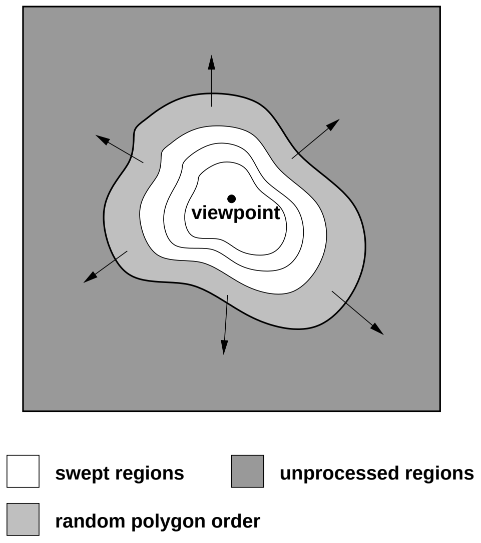


Figure 2: Approximate occlusion sweep. The scene is processed in an approximate front-to-back order. The order is defined by the traversal of a spatial hierarchy. On the figure a breadth-first-like traversal is depicted. In a currently visited region polygons are processed in random order.

algorithm[1]). A simple depth-first traversal of the tree can be used that visits leaves of the tree in a front-to-back order [8]. All polygons associated with a leaf are processed in random order.

Alternatively a priority stack can be used for the kD-tree traversal where the priority of the node is inversely proportional to the minimal distance of the node from the viewpoint. This approach can be used for octree and bounding volume hierarchies. An illustration of the approximate occlusion is depicted in Figure 2.

5 Occlusion tree

Occlusion tree [1] is a BSP tree representing a view of the scene. It is conceptually equivalent to the Shadow Volume BSP tree introduced by Chin and Feiner [4]. A BSP representation of the image was also used by Naylor [15] who proposed an efficient and elegant way of output sensitive rendering of scenes organised in an autopartition BSP tree. Naylor uses the scene BSP tree to obtain a strict front-to-back order of scene polygons and incrementally constructs the image BSP tree. As mentioned in the previous section our method relaxes the constraint of strict front-to-back order what allows to exploit various spatial hierarchies for the construction of the occlusion tree.

We briefly describe the structure of the occlusion tree and the algorithm of its construction. We present a necessary modification of the algorithm that allows to insert polygons in an approximate front-to-back order. More details on occlusion trees and their applications can be found in [1, 2, 3].

[1]In a pathological case, when the scene objects are not separable by an orthogonal plane, there can be as much as $O(n)$ objects per leaf.

5.1 Structure of occlusion tree

Occlusion tree is a BSP tree where each node represents a set of rays $\mathcal{Q}_N$ emanating from the viewpoint. The root of the tree represents the whole view. Each interior node N is associated with a plane π_N passing through the viewpoint. Right child of N represents rays $\mathcal{Q}_N \cap \pi_N^+$, left child $\mathcal{Q}_N \cap \pi_N^-$, where π_N^+ and π_N^- are halfspaces induced by π_N.

Leaves of the tree are classified in or out. If N is an out-leaf, $\mathcal{Q}_N$ represents unoccluded rays. If N is an in-leaf it is associated with a closest scene polygon P that is intersected by the corresponding set of rays $\mathcal{Q}_N$. Further N stores a fragment F_N that is an intersection of the polygon P and $\mathcal{Q}_N$.

It is easier to think about the occlusion tree in a restricted projection to a particular 2D viewport. The root of the tree corresponds to the whole viewport. Each interior node is associated with a line subdividing the current polygonal region in two parts. Leaves of the tree represent either empty region of the viewport or a fragment of a visible polygon.

Occlusion tree constructed for a single polygon P contains interior nodes corresponding to the planes defined by edges of P and the viewpoint. We call such a tree *elementary occlusion tree*, denoted e-OT(P) (see Figure 3).

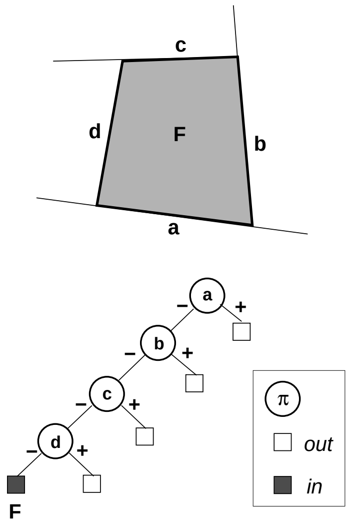


Figure 3: Elementary occlusion tree for a single polygon.

5.2 Construction of occlusion tree

Occlusion tree is constructed incrementally by inserting scene polygons in the order given by the approximate occlusion sweep. The algorithm inserting a polygon P in the tree maintains two variables — the current node N_c and the current polygon fragment F_c. Initially N_c is set to the root of the occlusion tree and F_c equals to P.

The insertion of a polygon in the tree proceeds as follows: If N_c is an interior node we determine the position of F_c and the plane π_{N_c} associated with N_c. If F_c lies in the positive halfspace induced by π_{N_c} the algorithm continues in the right subtree. Similarly if F_c lies in the negative halfspace induced by π_{N_c} the algorithm continues in the left subtree. If F_c intersects both halfspaces it is split by π_{N_c} into two parts F_c^+ and F_c^- and the algorithm proceeds in both subtrees of N_c with relevant fragments of F_c.

If N_c is a leaf node then we make a decision depending on its classification. If N_c is an out-leaf then F_c is visible and N_c is replaced by e-OT(F_c). If N_c is an in-leaf the mutual position of F_c and fragment F_{N_c} associated with N_c is determined. If F_c is behind F_{N_c} it is invisible and no modification to the tree necessary. Otherwise N_c is replaced by e-OT(F_c) and the old fragment F_{N_c} is inserted in the new subtree e-OT(F_c) using the just described polygon insertion algorithm. The nodes corresponding to the edges of the old fragment F_{N_c} are kept in the tree. Consequently the tree is slightly larger than it would be in the case of inserting polygons in a strict front-to-back order. An example of an occlusion tree for three polygons is depicted in Figure 4.

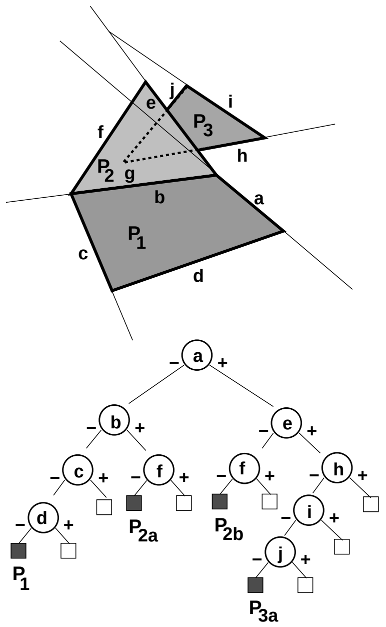


Figure 4: Three polygons and the corresponding occlusion tree. Polygon P_2 is split into two visible fragments. P_3 is partially covered by P_2 and so the tree reflects only its visible part P_{3a}.

6 Hierarchical visibility tests

To increase efficiency of the algorithm, the traversal of the scene kD-tree is interleaved with visibility tests applied on its nodes. If the test determines that the node is invisible the corresponding subtree and all polygons it contains are culled.

We use a conservative visibility test that performs a constrained depth first search on the occlusion tree using a bounding box corresponding to the given kD-tree node. Starting at the root of the occlusion tree the position of the box and the plane associated with the root is determined. If the box intersects only positive halfspace defined by the plane the algorithm recursively continues in the right subtree. Similarly, if the box intersects only negative halfspace the algorithm proceeds in the left subtree. If the box spans both halfspaces both subtrees are processed recursively. Reaching an out-leaf the algorithm identifies the node as visible and terminates. Reaching an in-leaf the position of the box and the fragment associated with the leaf is determined. If the box lies at least partially in front of the fragment it is visible and the algorithm terminates. If the search does not find any visible part of the box the corresponding node can be culled.

7 Construction of visibility map

Visibility map is constructed by postprocessing of the complete occlusion tree. The advantage of this approach is that only visible polygons are considered for the construction of the map. Each in-leaf of the tree corresponds to a visible fragment and each such fragment is incrementally inserted in the visibility map.

Visibility map consist of the following elements:

- *vm-vertex* — a vm-vertex corresponds to a vertex of a scene polygon or an apparent vertex that results from an intersection of edges in the view.
- *vm-polygon* — a vm-polygon corresponds to a fragment associated with a leaf of the occlusion tree.
- *vm-edge* — a vm-edge corresponds to an edge or a part of an edge of a scene polygon. A *contour* vm-edge is associated with a single vm-polygon, other vm-edges are associated with two polygons, each on one side of the edge.

The elements of the visibility map contain the following connectivity information:

- *vm-vertex*
  list of adjacent vm-edges,
  list of adjacent vm-polygons.
- *vm-edge*
  the two vm-vertices it connects,
  the two vm-polygons that share this vm-edge (one is possibly empty).
- *vm-polygon*
  list of vm-edges that bound the polygon,
  list of adjanced vm-vertices.

There is some redundancy in the above described representation, but the redundant information provides some more efficient lookups.

The visibility map is linked with the occlusion tree so that each in-leaf of the tree contains a link to the corresponding vm-polygon. In the following sections we describe how the visibility map is constructed by inserting visible fragments.

7.1 Neighbor location

In the first step of the insertion of a fragment F we create a new vm-polygon P_F. P_F is associated both with F and the corresponding in-leaf of the occlusion tree. Then we locate all already processed *neighbour vm-polygons* that share a boundary with F. The algorithm performs a constrained search on the occlusion tree pruning subtrees that have no intersection with F.

Then for each vertex of F we check if it was already inserted in the map by comparing it with vm-vertices associated with the neighbour vm-polygons. If the corresponding vm-vertex is not found we insert a new vm-vertex in the map. The either found or newly created vm-vertex is then is associated with P_F.

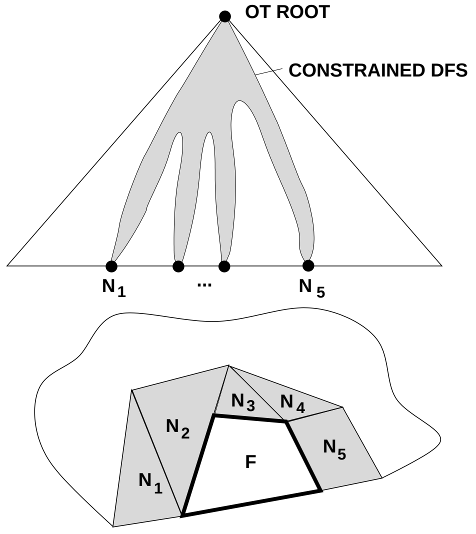


Figure 5: Neighbour location is performed using constrained depth first search on the occlusion tree. The figure depicts a fragment, its five neighbours and symbolic illustration of the search through the occlusion tree.

7.2 Inserting fragment edges

The crucial step of the visibility map construction is the insertion of edges of the currently processed fragment F. For each edge E_i the following steps are performed:

1. Locate all vm-edges of the neighbour vm-polygons that intersect the E_i. Denote the set of such edges $\mathcal{E}$.
2. Create links from these edges of $\mathcal{E}$ that completely overlap E_i to the new vm-polygon P_F.
3. Create new vm-edges for part of E_i that is not covered by any edge from $\mathcal{E}$. These edges are associated with P_F and contain an empty link to the other vm-polygon.
4. Subdivide and update edges of $\mathcal{E}$ that partially overlap E_i.

An illustration of the insertion of a new edge into the visibility map is depicted in Figure 6.

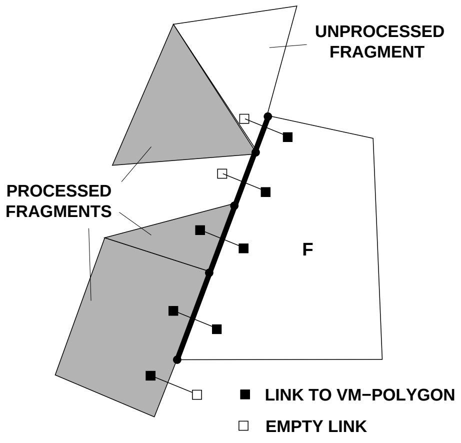


Figure 6: Insertion of a new edge in the visibility map. The figure depicts links corresponding to the updated or newly created vm-edges.

7.3 Classification of edges and vertices

The elements of the visibility map can be classified into several categories depending on the configuration of elements from their neigbourhood. We used the following five categories:

- *contour edge* — an edge that is associated with a single vm-polygon. It forms a part of the contour of the view.
- *shadow edge* — an edge that forms a “shadow” boundary. The two associated polygons do not share the corresponding edge in 3D, i.e. there is a depth discontinuity on the edge. The closer vm-polygon associated with the edge is the *occluder* the farther is the *occludee*.
- *corner edge* — an edge in a corner or on the rim of an object. The edge is shared by two connected vm-polygons that form an angle greater than the predefined *crease angle*.
- *flat edge* — the edge is shared by two connected vm-polygons that form an angle smaller than the predefined crease angle.

scene	polygons [−]	KD nodes [−]	view No.	total time [s]	OT time [s]	vm-vertices [−]	vm-edges [−]	vm-polygons [−]	culled KD leaves [%]
rad	26526	8809	I	0.24	0.15	642	1229	567	99.6
			II	0.14	0.07	555	1029	474	99.5
			III	1.15	0.69	3312	6307	2906	94.0
soda	1685	4735	I	0.02	0.01	39	62	24	98.8
			II	0.04	0.03	109	188	78	98.0
			III	0.09	0.07	201	368	163	95.1
random	10000	47139	I	0.10	0.08	275	472	197	99.9
			II	1.24	0.99	2269	3758	1472	95.0
			III	4.38	2.98	12016	20718	8533	84.1

Table 1: Summary of the results. The table contains the scene name, the number of scene polygons, the number of kD-tree nodes, the total time for construction of visibility map, the time for construction of occlusion tree, the number of visibility map polygons, vertices and edges, and the percentage of leaves of the scene kD-tree culled by the hierarchical visibility tests.

- *bsp edge* — a flat edge shared by vm-polygons that result from splitting of the same scene polygon in the process of construction of the occlusion tree

This classification enables to better understand the structure of the view. Depending on the application only edges of certain classes need to be considered. For example in the context of discontinuity meshing the visibility maps can be constructed with respect to each vertex of a given light source. Then the shadow edges define a subset of vertex-edge (VE) discontinuities due to the light source.

8 Results

We have implemented the proposed algorithm in C++ as a part of larger experimental rendering system. We evaluated the construction of visibility maps using three types of scenes:

- *rad* — a building interior with some detailed objects and finer meshes resulting from the radiosity algorithm. See Figures 1, 9.
- *soda* — a building interior with large walls, see Figure 8-a.
- *random* — random triangles, see Figure 8-b.

The measurements were conducted on a PC with 500MHz CPU, 256MB RAM, running Linux. For each scene we selected several viewpoints and measured the following: the total time for construction of visibility map, the time for construction of occlusion tree only, the number of visibility map vertices, edges and polygons, and the percentage of leaves of the scene kD-tree culled by the hierarchical visibility tests. The measurements are summarized in Table 1.

The first scene contains many polygons that result from meshing due to computation of global illumination using the radiosity method. For view with very restricted visibility (view No. I and II) the computation of visibility map is very fast and majority of the scene (99%) is culled by the hierarchical visibility tests. With less restricted visibility the computational time increases approximately linearly with the number of resulting vm-polygons.

The second scene is a building interior consisting of large polygons. The resulting visibility map is much simpler than for the *rad* scene and computational times are proportionally faster. A view of the *soda* scene is depicted in Figure 8-a.

The third scene contains 10000 randomly generated triangles. Due to the lack of a regular structure the resulting visibility map is rather complex. Additionally, the complexity of the view is increased due to mutual triangle intersections. The first view (random-I) corresponds to a viewpoint located inside of the cluster of triangles. The triangles appear larger and block visibility of many other triangles and consequently the computation is significantly faster in comparison with the other views. The second view (random-II) is depicted in Figure 8-b.

Further we studied the growth of the occlusion tree during its construction. We measured the size of the tree after processing each scene polygon using the approximate occlusion sweep. To better understand the results hierarchical visibility tests were not applied for this test. On the measured curve (Figure 7-b) we can identify two big steps of a sudden increase of the tree size. The first step corresponds to the insertion polygons near the viewpoint, the second to farther polygons visible through the door (see Figure 7-a). We can observe that once the occlusion tree contains all visible polygons its size does not increase.

Figure 9 shows a visibility map from the bird's perspective. We can see that the proposed algorithm efficiently culls invisible part of the scene during the construction of the occlusion tree. Visibility map is then build using only visible polygons.

Figure 10 shows a subset of a discontinuity mesh computed using four visibility maps. The four maps were constructed for views centered at the vertices of the rectangular light source. Shadow vm-edges were then projected on the associated occludees. More discontinuity edges could be identified by searching through vertices of the four visibility maps and matching the information about associated shadow edges [16].

9 Conclusion and future work

We have presented an algorithm that efficiently constructs a visibility map for a given view of the scene. The method does not rely on a single projection plane and easily handles views that span the whole spatial angle. Visibility map is constructed by a two stage algorithm: construction of a hierarchical representation of the view and its postprocessing.

The view is represented by the occlusion tree. The tree is constructed using a novel concept of approximate occlusion sweep that allows efficient incremental construction without complicated data structures for exact priority orders. Occlusion sweep is interleaved with hierarchical visibility tests what results in output sensitive behavior of the algorithm in practice.

Visibility map is constructed by simple postprocessing of the occlusion tree. We presented a classification of the elements of visibility map that helps to better understand the structure of the view. We evaluated our implementation of the technique on several non trivial scenes.

One of the challenging topics is the construction of an approximate occlusion map using the proposed method. This could overcome the problem of overly detailed output of our algorithm in the presence of many very small polygons. Another intersecting topic is the application of occlusion maps in synthesis of occluders for occlusion culling and visibility preprocessing. Visibility maps can provide a lot of structural information that can improve efficiency of these techniques.

Acknowledgements

Many thanks to Daniel Meneveaux for providing the *rad* test scene. This research was supported by the Czech Ministry of Education under Project LN00B096.

References

[1] 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.

[2] Jiří Bittner and Jan Přikryl. Exact regional visibility using line space partitioning. Technical Report TR-186-2-01-06, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2001. Available as ftp://ftp.cg.tuwien.ac.at/pub/TR/01/TR-186-2-01-06Paper.ps.gz.

[3] Jiří Bittner, Peter Wonka, and Michael Wimmer. Visibility preprocessing for urban scenes using line space subdivision. In *Proceedings of Pacific Graphics (PG'01)*, pages 276–284, Tokyo, Japan, 2001. IEEE Computer Society.

[4] Norman Chin and Steven Feiner. Near real-time shadow generation using BSP trees. In Jeffrey Lane, editor, *Computer Graphics (Proceedings of SIGGRAPH '89)*, pages 99–106, 1989.

[5] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 83–90, New York, April27–30 1997. ACM Press.

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

[7] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Co., Reading, MA, 2nd edition, 1990.

[8] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, volume 14, pages 124–133, July 1980.

[9] Charles W. Grant. *Visibility Algorithms in Image Synthesis*. PhD thesis, U. of California, Davis, 1992.

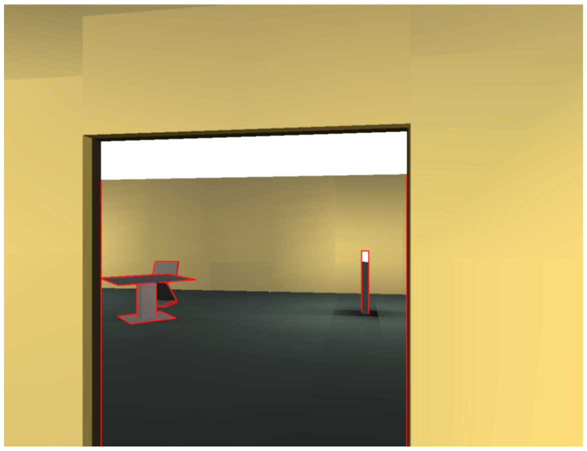
[10] Jérome Grasset, Olivier Terraz, Jean-Marc Hasenfratz, and Dimitri Plemenos. Accurate scene display by using visibility maps. In *Spring Conference on Computer Graphics and its Applications*, 1999.

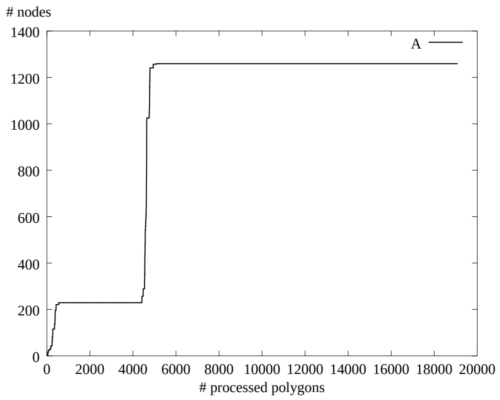
[11] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *Computer Graphics (Proceedings of SIGGRAPH '93)*, pages 231–238, 1993.

[12] Paul S. Heckbert. Discontinuity meshing for radiosity. In *Third Eurographics Workshop on Rendering*, pages 203–216, Bristol, UK, May 1992.

[13] T. Hudson, D. Manocha, J.Cohen, M.Lin, K.Hoff, and H.Zhang. Accelerated occlusion culling using shadow frusta. In *Proceedings of the Thirteenth ACM Symposium on Computational Geometry, June 1997, Nice, France*, 1997.

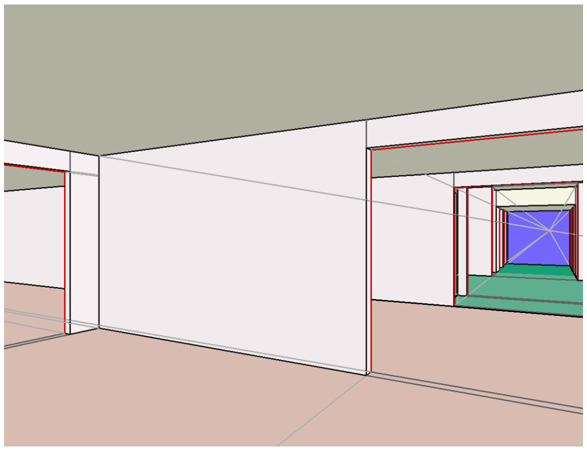
[14] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(6):153–65, 1990. criteria for building octree (actually BSP) efficiency structures.


(a)



(b)

Figure 7: (a) A view of the *rad* scene with shadow vm-edges depicted. (b) The size of the occlusion depending on the number of inserted polygons. Note the two steps corresponding to sudden increase of the tree size. The first corresponds to polygons close to the viewpoint, the second to the farther polygons visible through the door. No hierarchical visibility tests were applied.


(a)

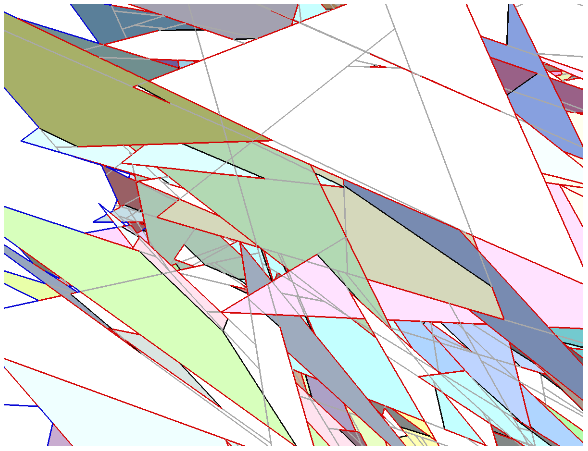
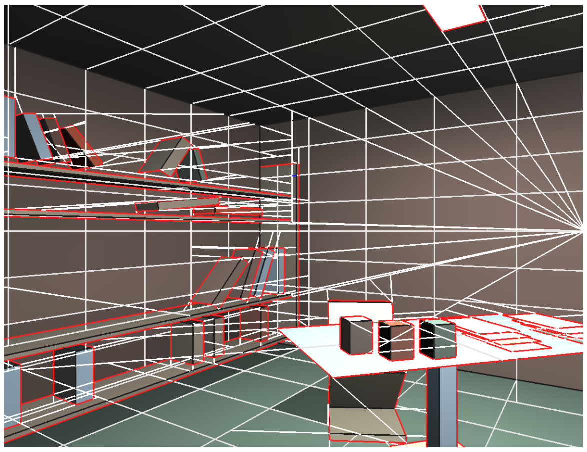

(b)

Figure 8: (a) Visibility map corresponding to the view soda-III in Table 1. (b) Visibility map corresponding to the view random-II in Table 1.

[15] Bruce F. Naylor. Partitioning tree image representation and generation from 3D geometric models. In *Proceedings of Graphics Interface '92*, pages 201–212, May 1992.

[16] A. J. Stewart and T. Karkanis. Computing the approximate visibility map, with applications to form factors and discontinuity meshing. In *Proceedings of the Ninth Eurographics Workshop on Rendering*, pages 57–68, 1998.


(a)

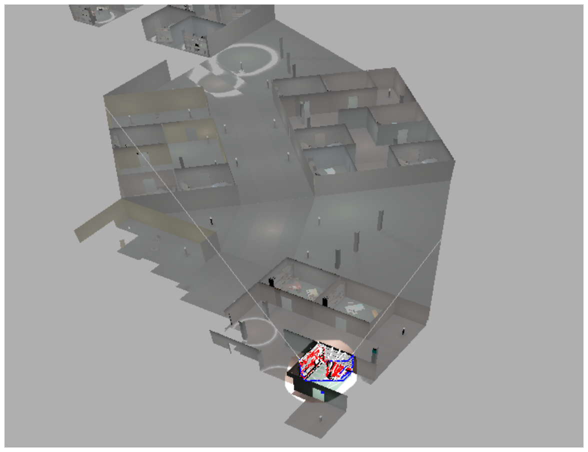

(b)

Figure 9: (a) Visibility map computed in a large scene where most of the scene is invisible. (b) Visibility map from the bird's perspective. The algorithm efficiently culls invisible regions (shown in grey) and considers only the visible part of the scene for visibility map construction. Computation time: 0.3s.

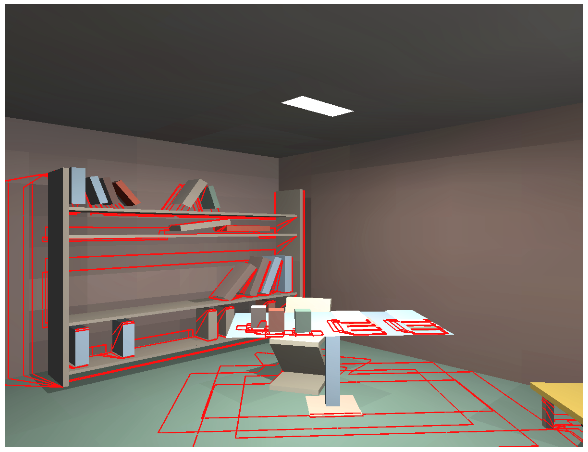


Figure 10: A subset of discontinuity mesh constructed using four visibility maps. The picture shows shadow edges of four visibility maps projected on occludees. The visibility maps correspond to views from the four vertices of a rectangular light source. Computation time: 1.2s.