Hierarchical Visibility Culling with Occlusion Trees

Jiří Bittner Vlastimil Havran Pavel Slavík

Department of Computer Science and Engineering, Czech Technical University,
Karlovo náměstí 13, 121 35 Praha 2, Czech Republic
E-mail: {bittner,havran,slavik}@fel.cvut.cz

Abstract

In the scope of rendering complex models with high depth complexity, it is of great importance to design output-sensitive *algorithms, i.e., algorithms with the time complexity proportional to the number of visible graphic primitives in the resulting image. In this paper an algorithm allowing efficient culling of the invisible portion of the rendered model is presented. Our approach uses a spatial hierarchy to represent the topology of the model. For a current viewpoint a set of polygonal* occluders *is determined that are used to build the* occlusion tree*. In the occlusion tree* occlusion volumes *of the selected occluders are merged. Visibility from the viewpoint is determined by processing the spatial hierarchy and classifying the visibility of its regions. In this process the occlusion tree is used to determine the viewpoint-to-region visibility efficiently. The algorithm is well-suited for complex models where large occluders are present.*

Keywords: visibility, occlusion culling, spatial partitioning, real time rendering, virtual reality, BSP [1] .

1. Introduction

Visibility determination is an important task in computer graphics. The goal of visibility determination (also known as hidden surface removal) is to efficiently determine visible parts of the model, given a viewpoint and a viewing direction.

Many algorithms to solve the hidden surface removal have been developed, two of them most commonly used. The *Binary Space Partitioning* (BSP) preserves the topological information about the model in a binary tree [5]. It is an example of an algorithm resolving visibility in object space. Knowing the viewpoint, rendering order can be determined by appropriate traversal of the BSP tree. The *z-buffer* is an image-space oriented algorithm, which solves the visibility problem for each pixel of the screen. This is simple to implement in hardware, thus commonly used in today's rendering systems. Both these algorithms are not *output-sensitive*, since they may spend significant time processing parts of the model actually invisible.

To achieve the output-sensitivity of the visibility algorithm we exploit the idea of *visibility culling*. Visibility culling is used to quickly determine a subset of occluded objects. These need not be considered for exact visibility determination (e.g. z-buffer rendering). The use of *occlusion trees* is a novel approach in the context of visibility culling. In the next two sections work related to this paper is presented followed by an overview of our algorithm.

1.1. Related Work

Some algorithms attempt to solve the visibility problem by building data structures allowing fast *exact visibility* queries. For example, the aspect graph [13] subdivides the space into $O(n^9)$ regions where visibility does not change qualitatively. For each such region the visible portion of the model can be determined. Unfortunately, the high complexity restricts this approach to models containing only a few objects. Recently Durand et al. [4] introduced the 3D visibility complex, which captures the visibility in line-space. The authors claim that its worst case space complexity $O(n^4)$ is much better in practice. However, its contribution to the real time rendering of complex models seems unclear.

The *conservative visibility* identifies the superset of the visible portion of the model. Only this superset is used to solve exact visibility. Often, this can be carried out using a hardware z-buffer.

Teller and Séquin [16, 17] use the concept of *potentially visible sets* (PVS). Potentially visible regions are determined for each region of the spatial subdivision using a region-to-region visibility through the set of transparent polygonal *portals*. Another algorithm based on PVS was introduced by Luebke and Georges in [11]. These methods achieve good performance if applied to static densely occluded environments with a particular structure, such as models of architectural interiors. However, for less structured models they can face a combinatorial explosion of complexity. Yagel and Ray [19] present an algorithm, which uses a regular spatial subdivision. Although it is not sensitive to the structure of the model in terms of complexity, its efficiency can be significantly lower compared with the portal based methods.

Recently, algorithms based on the idea of fast *hierarchical visibility culling* were published. The hierarchical z-buffer [8] algorithm uses a z-pyramid to represent occlusion. It exploits spatial coherence by processing an object hierarchy through the z-pyramid. Although it is a very promising approach if using hardware resources, the simulation of the z-pyramid in software would cause a significant overhead. Similar methods, which use an image space representation of the selected occluders, appeared in [20, 7]. While taking advantage of hardware rendering, these methods can suffer if the rendering support is insufficient.

The work presented in this paper is closely related to object space occlusion culling algorithms presented in [9] and [3]. In [9] a shadow frustum is constructed for each of the selected occluders. These frusta are used to detect the invisible regions of the spatial hierarchy. A possible drawback with this method is the independent visibility testing against each frustum. Therefore, the occlusion caused by multiple occluders is not discovered.

1.2. Algorithm Overview

The algorithm presented addresses the problem of conservative visibility from a point (viewpoint). It identifies a superset of objects visible from the viewpoint. For complex models, where many objects are not visible from a given viewpoint, this superset is only a fraction of the whole model. The exact visibility is solved by simply rendering the superset of visible objects using the z-buffer algorithm. Assume we are able to determine visibility of a region from the viewpoint. This visibility reaches one of the following states: *fully visible, partially visible, invisible*. We can apply the visibility test to all bounding volumes of objects in the model. The superset to be considered for exact visibility consists solely of objects classified as fully visible or partially visible. Nevertheless, for a complex model the visibility testing of all objects would be very time consuming.

We can exploit the spatial coherence of visibility by grouping close objects together. Applying this step recursively, we can build a spatial hierarchy, keeping links to the objects in its leaf nodes[2]. Each node of the hierarchy corresponds to certain spatial region. Starting from the root node of the hierarchy, visibility of each node can be determined as follows: If a node is found fully visible, all of its descendants are fully visible (assuming that the spatial hierarchy meets certain criteria, as it will be mentioned in Section 2.2). Similarly, if a node is found invisible, all its children are invisible. Descendants of nodes classified as partially visible must be further tested to refine their visibility. When the visibility of all leaves is known, objects from fully visible and partially visible leaves are gathered and rendered using a low-level exact visibility solver (hardware z-buffer).

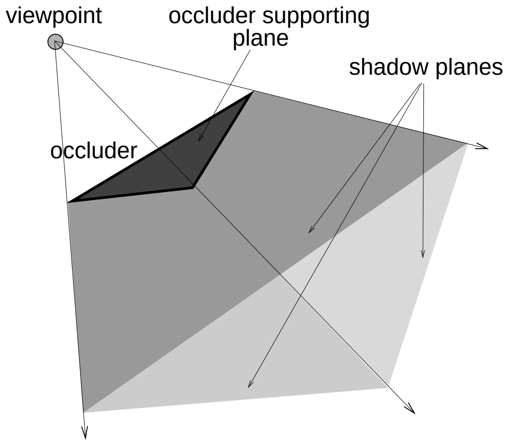

Figure 1. Occlusion volume of a polygon. The occlusion volume is formed by three shadow planes and the supporting plane of the polygon.

It remains to show how to determine the visibility of a region from the viewpoint. It is often the case that most of the occlusion is due to a few large objects (occluders) close to the viewpoint. In this paper we require the occluders to be convex polygons. Assume we are able to identify several such occluders for each viewpoint. For each polygon the occlusion volume (frustum) can be determined. It is an intersection of $(e+1)$-half-spaces, where e is number of edges of the polygon. The half-spaces are formed by planes passing through the viewpoint and the particular edge and the supporting plane of the polygon (see Figure 1). We merge these occlusion volumes into a unified data structure – the

[2]Such grouping corresponds to so called bottom-up approach. Here it is used for explanation purposes. In our implementation the hierarchy is actually built in top-down fashion, as mentioned in Section 2.2.

occlusion tree, that is a variant of the *shadow volume BSP* tree introduced by Chin and Feiner [2].

We show that the visibility of a closed polyhedral region can be determined by combining visibility states of its faces. Assuming the faces are convex polygons, these tests are performed efficiently using the occlusion tree. In particular, the regions of our spatial hierarchy are axis-aligned boxes (parallelepipeds), that are closed polyhedra with six convex faces. We also present a *modified occlusion tree* (MOBSP). With this data structure visibility of a region can be established without testing the visibility of its boundaries (faces). The only operation involved in the visibility test is the determination of the position of a region relative to a plane. Although this method can identify an invisible region as partially visible (with respect to the selected occluders), we observed its good performance in practice.

The paper is organized as follows: Section 2 describes preprocessing of the model that includes the algorithm building the spatial hierarchy. The occluder selection is outlined in Section 3. In Section 4 we discuss the motivation for building a unified data structure representing the merged occlusion volumes of the selected occluders. In Section 5 the occlusion tree and algorithms of its construction and traversal are presented. The modified occlusion tree is introduced in Section 6. In Section 7 we present results obtained on several different models and discuss the behaviour of the algorithm. Finally, Section 8 concludes and in Section 9 we point out some topics for future work.

2. Preprocessing

2.1. Occluder Identification

Previous methods of hierarchical visibility culling [3, 9] attempt to create an *occluder database* in preprocessing. They subdivide the space into a set of non-overlapping regions (cells). Within each cell a certain number of polygonal occluders are determined and stored.

We do not attempt to build such occluder database. Instead, we only identify and mark *potential occluder* polygons. In our implementation these are identified taking advantage of the knowledge of the model structure. Preprocessing and visibility culling have been applied usually on models of architectural interiors. A typical such model consists of walls, ceilings, floors, and detailed objects. All polygons belonging to detailed objects (flowers, chairs, ...) are considered non-occluding. All remaining polygons are marked as potential occluders (assuming these are walls, ceilings and floors). These potential occluders are used in the algorithm of dynamic occluder selection (Section 3).

2.2. Spatial Hierarchy

As we already mentioned, the hierarchical visibility algorithm assumes that a spatial hierarchy is built over all objects of the model. In the case of static scenes this can be done in preprocessing. There is an important requirement imposed on the hierarchy. Regions corresponding to descendants of any node of the hierarchy must be completely contained in the region corresponding to that node. Otherwise, no assumptions of the visibility of the node's descendants could be made based on knowledge of the visibility of their parent. In previous work bounding volume hierarchies and hierarchical spatial subdivisions (octree, BSP tree) were used.

We use an axis-aligned BSP tree [10] (sometimes referred to as kD-tree), because of its high flexibility and simplicity of building and traversal. This selection implies that the regions corresponding to nodes of the hierarchy are parallelepipeds. Naturally, the BSP tree meets the criterion mentioned above.

The most important step during the building of the BSP tree is the choice of the splitting plane. This plane subdivides the current node into two descendants. Objects are distributed into the descendants according to their position to the splitting plane. Initially the root node of the BSP tree corresponds to the bounding box of the model. Applying the algorithm recursively, the whole BSP tree is built. The recursion is terminated when the number of objects in the current node falls under the specified threshold or the specified maximum depth of the hierarchy is reached.

In certain cases an object lies on both sides of the plane (i.e. in both positive and negative half-spaces induced by the plane). Such objects must be "duplicated" in both new nodes. We want the object duplications in leaf nodes of the tree to be minimized while keeping a well-balanced tree. To achieve this goal the following strategy of the splitting plane selection was used:

For the current node we identify the axis with the largest extent of the parallelepiped corresponding to the node. We search for a splitting plane perpendicular to the selected axis. We identify boundaries of object bounding boxes located within a certain distance from the spatial median of the node's parallelepiped. Each identified boundary induces one *boundary plane*. We evaluate a number of objects split by each boundary plane. The boundary plane with the lowest number of split objects is selected as the splitting plane.

The binary tree structure can be easily used to simulate irregular quad-trees and octrees in the scope of the hierarchical visibility algorithm.

3. Dynamic Occluder Selection

The goal of the dynamic occluder selection is to obtain a specified number of occluders, given a viewpoint and a viewing direction. The algorithm uses the *area-angle* measure [3] to estimate the quality of an occluder. The area-angle is expressed as:

$$M = \frac{-A(\vec{N} \cdot \vec{V})}{\|\vec{D}\|^2} \tag{1}$$

where A is the area of the occluder, $\vec{N}$ denotes the occluder normal, $\vec{V}$ the viewing direction and $\vec{D}$ corresponds to the vector from the viewpoint to the center of the occluder ($\|\vec{N}\| = \|\vec{V}\| = 1$).

As mentioned in the previous section, the potential occluders are identified in preprocessing. The dynamic occluder selection is performed after each change of the viewpoint or the viewing direction. The set of occluders obtained in the current frame is used to perform visibility culling in the next frame.

The dynamic occluder selection proceeds as follows: We identify all visible or partially visible leaves of the hierarchy, whose centers are located within a certain distance λ of the viewpoint. For each potential occluder referred in these leaves the area-angle is computed. These values are used to select k occluders with the largest area-angle, that form the desired occluder set for the next rendering frame.

The distance λ has an impact on the time spent by the occluder selection. In our implementation it is a multiple of the observer's step size ($\lambda = 100 \cdot step$). The number of selected occluders (k) influences the size of the occlusion tree, the time of visibility determination as well as its efficiency. We have used k between 6 and 256. More details about this selection are given in Section 7.

4. Representation of Occlusion Information

In this section we discuss the crucial part of the visibility culling algorithm – representation of the occlusion caused by a set of convex polygonal occluders for a given viewpoint. In many cases treating the occlusion volumes separately identifies the invisible region as partially visible.

Unlike previous methods used in the field of visibility culling, we build an additional object space data structure for the current viewpoint. It efficiently merges the occlusion volumes of the selected occluders. This allows us to discover occlusion caused by multiple connected occluders and even occluders completely disjunct in space.

It is obvious that merging occlusion volumes requires an additional time to build the appropriate data structure. We observed that by exploiting the spatial coherence of occluders in this data structure, the additional time is recovered during the visibility queries.

As a basis for our research we adopted the concept of *shadow volume BSP trees* (SVBSP) introduced by Chin and Feiner [2]. They used the SVBSP for fast generation of shadows cast by polygonal objects in scenes with point light sources.

Next we present a brief overview of the SVBSP data structure and the algorithm suitable for the generation of shadows. Further, we discuss how to adapt the concept of SVBSP for the viewpoint-to-region visibility determination.

4.1. Generation of Shadows with SVBSP

The SVBSP is a variant of the BSP tree for representing polyhedra [12, 18]. It represents a union of shadow volumes cast by convex polygons (occluders) facing a point light source. Each internal node of the tree is associated with a *shadow plane* passing through the light source and an edge of the occluder.

The direction of the shadow plane normal is used to determine a half-space in which the occluder and its shadow are located. The normals are oriented so that the shadow volume and the occluder itself lie in the negative half-space (back side) of the plane. Each leaf node of the tree corresponds to a semi-infinite polyhedral cell (frustum). The leaves are classified as *in* or *out*. A leaf is marked as in-leaf if the corresponding cell lies in shadow. Similarly, an out-leaf indicates that the corresponding cell is lit by the light source. Thus the shadow volume is a union of all cells corresponding to in-leaves. An example of the SVBSP tree is depicted in Figure 2.

Assume that the polygons (occluders) are ordered in a front-to-back manner with respect to the light source. We know that farther polygons cannot cast shadows on polygons lying closer to the light source. The SVBSP can be constructed by incrementally processing the polygons in the given order. The contribution of one polygon to the SVBSP is determined as follows:

- Lit fragments of the polygon are determined.
- The SVBSP is enlarged by shadow volumes cast by these fragments.

The lit fragments of the polygon are determined by *filtering* the polygon down the SVBSP. The filtering is applied recursively on certain nodes of the SVBSP, starting from the root. For the current node the position of the polygon with respect to node's shadow plane is determined. If the polygon is located completely on the back or front side of the node's shadow plane, it is filtered down the back or front child of the node, respectively. Otherwise, the polygon is split by the shadow plane into two fragments. These fragments are filtered down the both children of the node.

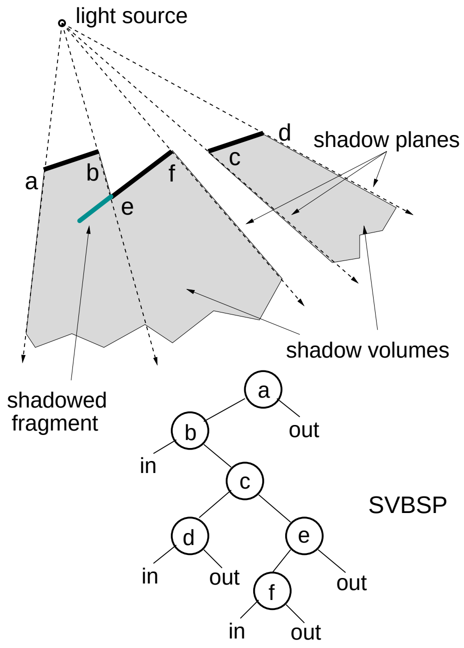

Figure 2. A 2D example of three polygons facing a point light source and the corresponding SVBSP tree.

Reaching leaf nodes a set of convex polygonal fragments is obtained. These are either lit (out-leaves) or shadowed (in-leaves). In the appropriate out-leaves the tree is enlarged by the shadow volumes cast by the lit fragments. For each lit fragment e new nodes are used to replace the corresponding leaf (e is the number of edges of the fragment). The algorithm including shadows of one polygon into the SVBSP is given in Figure 3.

For shadow generation purposes, the lit and shadowed fragments can be stored within the original polygon. During rendering the colour of these fragments can be set accordingly.

The front-to-back ordering of polygons can be achieved by building a BSP tree and its appropriate traversal [5]. Alternatively, the *feudal priority tree* [1] could be used.

4.2. Visibility Determination and SVBSP

The SVBSP tree is a hierarchical data structure allowing fast incremental determination of lit and shadowed fragments of scene polygons.

We will refer to the algorithm described previously as the *original SVBSP algorithm*. All lit fragments obtained by the original SVBSP algorithm are visible from the light source. Similarly, all fragments located in shadow are invisible from the light source. From now on assume the position

```
Algorithm FilterDown(Node, Polygon, Viewpoint)
begin
  if Node is leaf then
    if Node is out-leaf then
      replace Node by
      OcclusionVolume(Polygon, Viewpoint)
    else
      do nothing
  else
    case Split(Polygon, Node.Plane, Back, Front) of
     FRONT : (* pass the polygon to the front subtree *)
          FilterDown(Node.FrontChild, Polygon, Viewpoint);
     BACK  : (* pass the polygon to the back subtree *)
          FilterDown(Node.BackChild, Polygon, Viewpoint);
     SPLIT : (* pass fragments to apropriate subtrees *)
          FilterDown(Node.FrontChild, Front, Viewpoint)
          FilterDown(Node.BackChild, Back, Viewpoint);
    end
end
```

Figure 3. Pseudo-code of the algorithm processing a polygon through the SVBSP tree.

of the point light source to be a viewpoint. Instead of lit and shadowed, the terms visible and invisible (occluded) will be used in the following text.

Assume the SVBSP is built with respect to the selected set of occluders and the current viewpoint. Given a polyhedral region (cell) we want to quickly determine if this region is:

- fully visible,
- partially visible,
- invisible.

We will use the term *visibility algorithm*, referring to the traversal of the SVBSP, which appropriately classifies the visibility of a region.

The differences between the desired visibility algorithm and the original SVBSP algorithm can be summarized as follows:

- Only selected occluders are used to build the tree. In the scope of the visibility algorithm the tree is not modified any more. The original SVBSP algorithm assumes all polygons to be processed and the tree to be always updated accordingly.
- The subject of the visibility algorithm is a polyhedral region, whereas in the original SVBSP algorithm the subject is a convex polygon.

- The region of which the visibility is determined may lie in front of some occluders. Hence the front-to-back ordering is not satisfied in the visibility algorithm.

- Only one of the three visibility states is the result of the visibility algorithm. In the original SVBSP algorithm the goal is to obtain lit and shadowed fragments of a polygon.

With these differences in mind, we designed the concept of *occlusion trees* (OBSP) and appropriate visibility algorithms (i.e. algorithms of their traversal).

5. Occlusion Tree

An *occlusion tree* is a BSP tree built with respect to a set of occluders and a viewpoint. By relevant traversal of the OBSP we determine visibility of a polyhedral region with respect to the selected occluders either exactly or *conservatively*. Conservatively means that a region with any of its part visible is never classified as invisible, but invisible regions can be classified as partially visible.

To meet the criteria mentioned above we build the OBSP as follows: The selected occluders are used to build a BSP tree. This tree is used to establish the front-to-back order of the occluders. The occluders are processed in this order and their occlusion volumes are used to enlarge the OBSP. The OBSP construction process is essentially the same as the one of SVBSP. Additionally, in each in-leaf we store a link to a fragment occluding the frustum, which corresponds to this leaf. These links are needed in the visibility algorithm to determine if a polyhedron tested for visibility lies behind the occluder.

It remains to show how to determine visibility of a polyhedral region using the OBSP. For a closed polyhedron it is sufficient to combine visibility of its faces appropriately. We assume that these faces are convex polygons. In the next section we describe how to classify visibility of a convex polygon using the OBSP. In Section 5.2 we present the visibility algorithm for a convex polyhedron. Both these algorithms classify the visibility *exactly* with respect to the occluders the OBSP was built for. Since these occluders are only a subset of all objects in the model, the hierarchical visibility algorithm presented later gives conservative results.

5.1. Visibility of Polygon

The visibility of a polygon can be determined filtering it down the occlusion tree. When a leaf is reached, the visibility of the current fragment of the polygon is classified. For out-leaves the fragment is fully visible. The visibility in the in-leaves can reach any of the three possible states, hence, an additional test must be applied. This test will be explained later in this section.

If there is no fragment of the polygon which is fully visible, the polygon is invisible. Similarly, if no invisible fragment exists, the polygon is fully visible. In all other cases the polygon is partially visible with respect to the occluders the tree was built for.

Given a polygon the OBSP is traversed by *depth first search* (DFS). In each internal node of the OBSP the position of the polygon with respect to the plane referred in the node is determined. It is essentially the same procedure as the polygon filtering in the construction of the OBSP. If the polygon lies completely in front or back of the plane, the visibility algorithm is applied on the appropriate child of the current node. Otherwise the polygon is split in two fragments and the algorithm is applied on both children using the relevant fragments. In this case, the visibility states of the fragments must be combined to classify visibility of their union (see Table 1).

Fragment A	Fragment B	A ∪ B
F	F	F
I	I	I
P	X	P
X	P	P

Table 1. Combining visibility states of fragments. Abbreviations: I – Invisible; P – Partially visible; F – Fully visible; X – any of the I,P,F states.

Thus, in each node reached by the DFS the visibility of the corresponding fragment of the polygon is computed. The visibility of the whole polygon corresponds to a visibility state of the root node of the OBSP. Nevertheless, the DFS can be terminated whenever a fragment is found partially visible. It follows from the fact that if a fragment of the polygon is partially visible, the polygon itself is partially visible (see Table 1). This constraint can accelerate the visibility algorithm significantly. The speedup is particularly remarkable for large polygons, which are likely to be partially visible.

As already mentioned, the polygon tested for visibility need not lie behind all occluders. Therefore, reaching an in-leaf node the additional test must be applied. We use the link to the occluder-fragment occluding the frustum corresponding to the leaf. The supporting plane of the occluder-fragment is used to establish visibility of the fragment, which reached the leaf during the DFS. If the fragment is completely in front of the plane, it is fully visible. If it is completely on the back side of the plane, it is invisible. Otherwise, it lies on both sides of the plane and it is partially visible.

The visibility algorithm for a polygon with respect to the occlusion tree is summarized in Figure 4.

```
Algorithm Visibility(Node, Polygon)
begin
  if Node is leaf then
    if Node is out-leaf then
      Visibility ← VISIBLE
    else
      Visibility ← visibility state based on
      FragmentIntersection(Node.Fragment, Polygon);
  else
    case Split(Polygon, Node.Splitter, Back, Front) of
     FRONT : (* pass the polygon to the front subtree *)
          Visibility ← Visibility(Node.FrontChild, Polygon);
     BACK  : (* pass the polygon to the back subtree *)
          Visibility ← Visibility(Node.BackChild, Polygon);
     SPLIT : (* pass fragments to apropriate subtrees *)
          Visibility ← Visibility(Node.FrontChild, Front)
          if Visibility <> PARTIALLY then
          begin
            aux ← Visibility(Node.BackChild, Back)
            Visibility ← CombineVisibility(aux, Visibility);
          end
    end
end
```

Figure 4. Pseudo-code of the polygon visibility algorithm using the OBSP.

5.2. Visibility of Polyhedron

In this section we show how the visibility of a closed polyhedron from a viewpoint is determined. The polyhedron visibility test will be used extensively during the hierarchical visibility culling.

As already mentioned the visibility state of a closed polyhedron can be determined by combining the visibility of its faces. Assuming these faces are convex polygons, the above presented polygon visibility algorithm can be applied. The visibility of the polyhedron is refined incrementally, processing its faces one by one. We call the *current visibility* of the polyhedron the visibility of the union of those polyhedron faces which were already processed. Visibility of a face of the polyhedron which is facing the viewpoint is computed using the polygon visibility algorithm. If it is not the first face processed, the visibility of the polyhedron is updated using the method given in Table 1. The current polyhedron visibility is combined with the visibility of the face recently processed. Whenever the current polyhedron visibility reaches the partially visible state, the algorithm can be terminated. Otherwise, it proceeds with a next face until all faces are visited.

In following sections the polyhedron visibility test will be applied on regions (cells) of the spatial hierarchy. In our case these cells are parallelepipeds. To determine visibility of such a cell at most three rectangular polygons must be tested for visibility (these polygons can be determined by a table lookup).

5.3. Hierarchical Visibility Culling

The visibility algorithms mentioned above are used during the hierarchical visibility culling. Starting from the root node of the hierarchy, the visibility of each node is determined using the polyhedron visibility algorithm. Recall that if a node is found fully visible, all its descendants are fully visible. Similarly, if a node is found invisible, all its children are invisible. Descendants of nodes classified as partially visible are further tested to refine their visibility (see Figure 5).

The view-frustum culling can be easily merged into the algorithm. Before the visibility test, the position of the polyhedron with respect to the view-frustum is computed. If the polyhedron is lying outside the frustum, the corresponding node is marked invisible. When the visibility of all leaves is known, objects from fully visible and partially visible leaves are gathered and rendered using a low-level exact visibility solver (hardware z-buffer).

To avoid visibility testing of hierarchy nodes where only few objects are contained, we use a *node-cost*, which is determined during preprocessing. The node-cost is compared with a certain threshold (minimum cost). If the node-cost is lower than the threshold, the node is simply classified fully visible. The node-cost is based on costs of objects contained in the region corresponding to the node. As the cost of an object we use the number of polygons forming the object.

The hierarchical visibility algorithm can be applied to various kinds of spatial hierarchies. For example, the hierarchy of polyhedral bounding volumes could be used without any modification of the algorithm.

5.4. Temporal Coherence

The visibility algorithms presented in this paper make good use of the spatial coherence. However, the temporal coherence has not been exploited so far.

Suppose the observer (viewpoint) moves smoothly. It is of high probability, that hierarchy nodes classified as partially visible remain partially visible in the following visibility test. This holds especially for nodes at the top of the spatial hierarchy.

During the hierarchical visibility determination we mark nodes where the visibility is ambiguously determined (partially visible). When the visibility of any node is unambiguously determined, its parent node is unmarked. When a leaf

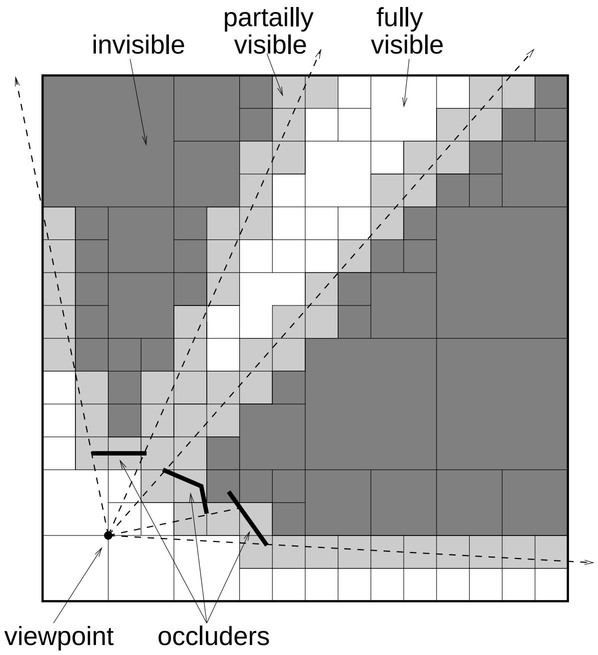

Figure 5. A 2D example of the hierarchical visibility culling. Regions classified invisible are shown in dark; partially visible ones are marked lighter. The spatial subdivision is assumed to be an octree.

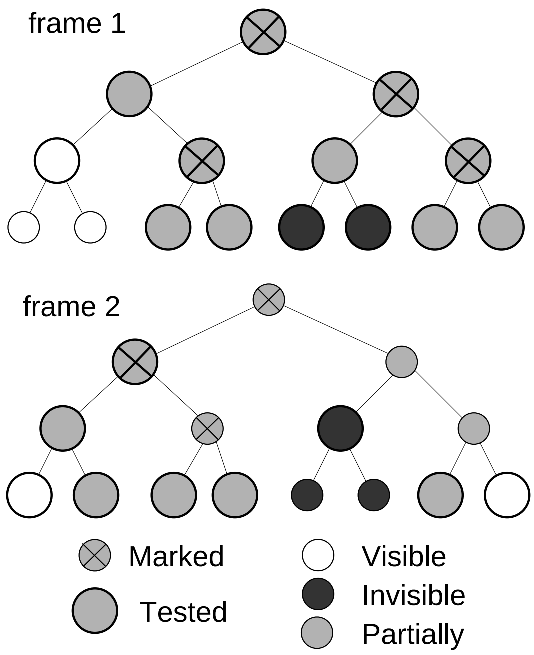

Figure 6. Illustration of the temporal coherence heuristics. In the first frame no nodes were skipped and four nodes were marked. In the second frame the four nodes were skipped and three nodes were marked. These three will be skipped in the third frame.

node is reached it is always unmarked, regardless of its visibility. When processing the hierarchy in the next frame the visibility tests are not applied on nodes marked in the previous pass of the algorithm. The effect of this modification is illustrated in Figure 6.

Suppose all nodes of the hierarchy are classified as partially visible. In the next pass of the algorithm all inner nodes are skipped. The visibility test is applied only on the leaf nodes. Hence, for a binary tree hierarchy the modification saves up to 50% of the visibility tests which would be applied on the inner nodes.

6. Modified Occlusion Tree

In this section we introduce a *modified occlusion tree* (MOBSP) and the algorithm of its construction. Further, we present a fast conservative visibility algorithm using the MOBSP.

The OBSP data structure is used extensively in the hierarchical visibility algorithm. The elementary operation taking place in both the tree construction and the visibility algorithms is a *polygon splitting*. Recall that the splitting operation determines fragments of the polygon lying in negative and positive half-spaces induced by a plane.

Although the splitting can be implemented quite efficiently, the overhead of fragment allocation remains when the polygon is split by the plane. The splitting operation was also a reason that the polyhedra visibility could not be efficiently determined by processing the polyhedra itself. This is due to the complexity of the splitting operation for polyhedra (parallelepipeds) as well as maintaining its fragments. Therefore, the decomposition of polyhedra and the combination of visibility states of its faces has been used.

Motivated by the idea of the visibility algorithm (tree traversal) without the necessity of the polygon splitting, we developed the concept of the MOBSP. It is based on observation that carefully removing some of the nodes (shadow planes) of the OBSP, the new data structure still contains all the information about the occlusion volumes. Additionally such a data structure can be traversed easier using the *modified visibility algorithm*. The algorithm is conservative with respect to the selected occluders (occluders used to built the MOBSP). Recall that the algorithms presented so far determine the visibility state exactly with respect to the selected occluders. Although the modified visibility algorithm is only conservative, we observed its very good performance in practice.

6.1. Construction of the MOBSP

The MOBSP is constructed similarly to the OBSP. Assume an occlusion volume of a polygon is being merged to the MOBSP. The polygon splitting operation used during filtering is enriched by *flagging* (marking) edges of the polygon embedded to the shadow planes. If new fragments are created by the splitting operation, the edge of both fragments embedded to the shadow plane that splits the polygon is flagged as well.

When an out-leaf is reached during the polygon filtering, only unflagged polygon edges are used to create shadow planes enlarging the MOBSP. Shadow planes that would have been created by flagged edges must be already present in the MOBSP (otherwise the edges would not be flagged). If an out-leaf is reached and all edges of the filtered fragment are flagged, no internal nodes are added to the tree. Instead, the out-leaf is replaced by an in-leaf and the link to the fragment in the new in-leaf is set. The difference between an occlusion tree and its modified version is illustrated in Figure 7.

It is obvious that the MOBSP contains lower or equal number of nodes than the OBSP built using the same set of occluders. It remains to show that the visibility algorithm as described before gives a correct answer if applied to the MOBSP. Let us focus again on the structure of the MOBSP. Assume an out-leaf is reached during filtering and the corresponding fragment contains flagged edges. Some shadow planes are not added to the tree because the corresponding edges were flagged.

Ignoring some shadow planes the occlusion volume of a fragment is enlarged. This larger occlusion volume is merged to the tree by replacing certain out-leaf. Obviously, each node of the tree implies a unique path from the root. During the polygon visibility algorithm, the filtered polygon is clipped by all the planes on the path to a node. In the subtree of the node it is not necessary to clip the position of the filtered fragment to any plane which already occurred on the path. The flagged edges would generate exactly such planes. Hence, ignoring these planes does not effect the correctness of the visibility algorithm.

In the next section we present a modified visibility algorithm for the MOBSP. In the previously presented algorithms, the polygon visibility algorithm has been used to determine the visibility of a polyhedron. The modified algorithm determines the visibility of regions of various shapes directly. It only uses the classification of the position of the region with respect to a plane.

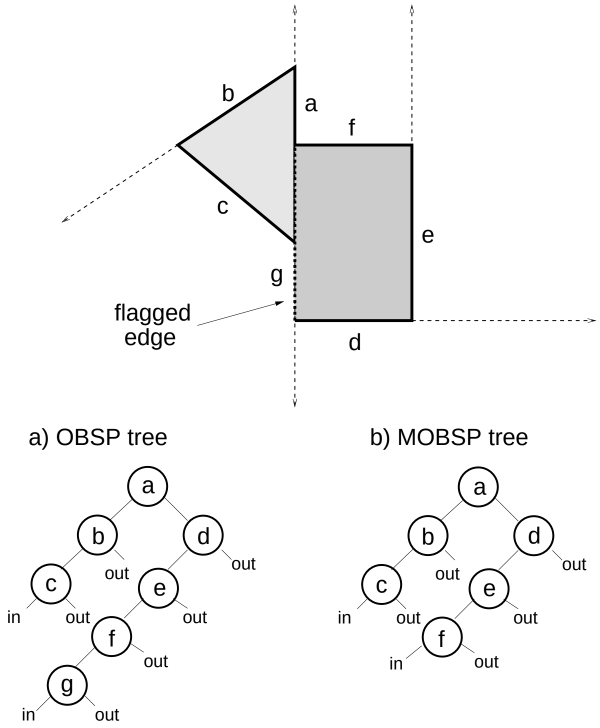

Figure 7. The difference between the OBSP and the MOBSP. Both trees are constructed with respect to the same occluders. The occluders are shown as seen from the viewpoint. Node g is not present in the MOBSP, since the corresponding occluder-edge was flagged.

6.2. Conservative Visibility of a Region

Assume that the position of a region with respect to a splitting plane can be determined. It indicates if the region lies in negative (back), positive (front), or both half-spaces.

Given this operation, the MOBSP is traversed similarly to the OBSP polygon visibility algorithm. In each internal node of the tree we determine the position of the region with respect to the shadow plane and apply the algorithm recursively on appropriate subtrees. No attempt to split the region is made even if it lies on both sides of the plane.

The modified visibility algorithm is conservative only. It implies that the results (i.e. visibility classification of leaves) can vary comparing to the exact algorithms presented. Reaching an in-leaf, the region is tested for position to the supporting plane of the occluder-fragment referred in the leaf. Call this leaf the current leaf. As mentioned previously, if the region lies on both sides (half-spaces) of the supporting plane, it would be classified partially visible. It is possible that a part of the region crossing the supporting plane is actually occluded by another occluder-fragment than the one referred in the current leaf.

Such a situation can be discovered by testing the region for an intersection with the occluder-fragment. If they do not intersect, the region is classified invisible with respect to the current leaf. In other words it is invisible in the frusta induced by the viewpoint and the fragment. Note that the region can still be found partially visible when the visibility of all leaves reached during the visibility algorithm is combined. In the case of an axis-aligned box, a fast algorithm can be used for the box/polygon intersection[6]. In general, it is unclear if the possible improvement in correctness of the algorithm is worth the time spent by the additional intersection test.

Here we mention another possible modification of the algorithm. It is possible to apply similar intersection tests also in the inner nodes of the MOBSP. Then the algorithm always determines visibility exactly with regard to the selected occluders. However, such a traversal can loose several desired features of the modified visibility algorithm. These are the simplicity of traversal, generality, and speed. Due to lack of space we decided not to present the modification in detail.

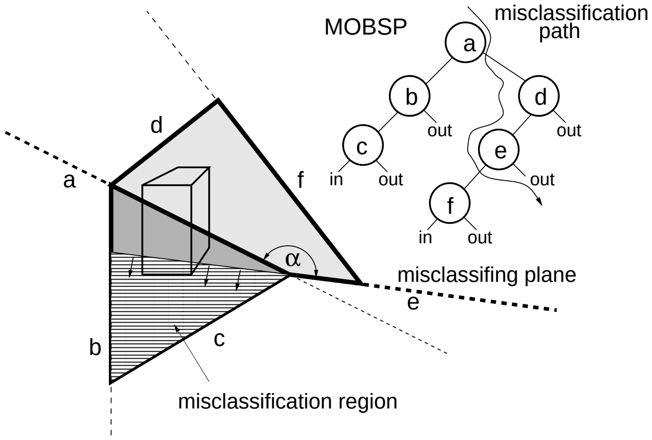

Figure 8. An example of disadvantageous configuration of occluders. The shadow planes of the MOBSP are shown by thin lines. The invisible polyhedra intersecting both the shadow plane a and the marked region is said to be partially visible.

We have mentioned that the modified visibility algorithm is conservative. The situation when a region is misclassified as partially visible is depicted in Figure 8. In general, it is difficult to give a probabilistic analysis of the number of cases when the algorithm fails to give an exact result. Assume the configuration of occluders as given in the figure. It can be seen that the algorithm is likely to give an imprecise (conservative) result if the angle α between the highlighted shadow planes gets larger. Nevertheless, in most cases the algorithm performs well in practice as documented in the next section.

7. Results

In this section we document the behaviour of the algorithm presented in this paper. Several models of architectural interiors were used to compare efficiency of the algorithms. As a reference we used the *hierarchical frustum culling* [14] with no visibility processing. The results are summarized in Table 2. Each line corresponds to average values obtained in the scope of one walk-through.

The *Time* field is an average frame time. The *Overhead* field depicts an additional overhead of the visibility culling algorithms. This includes the dynamic occluder selection, building of the BSP of occluders, building the OBSP (MOBSP), and the hierarchical visibility culling. The *Speedup* is a fraction of the average frame time of the pure view-frustum culling and the actual average frame time. The *Rendered polygons* field contains the average number of polygons rendered in one frame. The other two parameters shown in the table are user-specified constants influencing behaviour of the algorithm. The number of occluders used to build the OBSP (MOBSP) is shown in the *Occluders* field. The field called *Method* represents the algorithm used for visibility culling. Its meaning is explained below the table. In all visibility culling algorithms the temporal coherence heuristics was used. The minimum cost of the node to be tested for visibility was 50. Recall that the cost of a node expresses the number of polygons referred in the node. For each measurement 100 detailed objects were spread randomly in the scene. We used a virtual plant consisting of 644 polygons.

Plots of frame times and numbers of rendered polygons measured during a walk-through of an architectural model (*big-7*) are shown in Figures 9 and 10. The speedup of the rendering achieved for the tested models varies between 1.75 and 3.75. We observed that the speedup is not linearly proportional to the number of occluders used for the visibility culling. Increasing the number of occluders used by factor of two, the speedup is usually increased much less. Important is that in such a situation the overhead of visibility culling is also increased less than two times.

Informally, we explain the behaviour of the algorithm as follows: Firstly, assume the occlusion tree contains occluders that occlude large portion of the view. It is of high probability that another occluder is found invisible. In this case the occlusion tree is not enlarged. Secondly, the occlusion tree inherits the logarithmic search properties of hierarchies. Therefore, the number of steps of the polyhedron visibility algorithm with the occlusion tree is usually much

Scene	Method	Occluders	Rendered polygons	Overhead [ms]	Time [ms]	Speedup
soda-5	F	—	18192	—	276.5	1.00
soda-5	FME	8	9466	3.7	157.3	1.75
soda-5	FME	16	7390	5.9	139.0	1.99
soda-5	FME	24	6537	7.9	116.0	2.38
soda-5	FME	32	5941	9.8	109.8	2.52
soda-5	FME	50	5490	14.2	109.5	2.53
soda-5	FSE	16	6512	12.7	120.5	2.29
soda-5	FSE	24	5569	16.0	110.5	2.50
soda-5	FSE	32	4725	19.1	100.1	2.76
soda-5	FMC	16	7988	5.4	135.9	2.03
soda-5	FMC	24	7362	7.3	129.3	2.14
big-7	F	—	12587	—	182.0	1.00
big-7	FME	16	3641	8.8	71.5	2.55
big-7	FME	24	2286	10.2	54.6	3.33
big-7	FME	32	1818	11.6	48.5	3.75
big-7	FSE	24	2214	19.9	63.4	2.87
big-7	FMC	16	3640	8.4	74.3	2.44
big-7	FMC	24	2263	9.7	51.4	3.54
soda	F	—	15297	—	315.2	1.00
soda	FME	24	4744	10.1	119.7	2.63
soda	FSE	24	4388	23.7	128.0	2.46

F – view-frustum culling
S – OBSP tree + standard visibility algorithm
M – MOBSP tree + conservative visibility algorithm
E – exact occluder-fragment/parallelepiped intersection test
C – conservative occluder-fragment/parallelepiped intersection test

Table 2. Results of the hierarchical visibility culling. The table shows the average number of polygons rendered, the average frame-time and the speedup over the view-frustum culling for different scenes and methods of the visibility culling. Measured on SGI O_2, 64MB RAM.

lower than the number of occluders.

It follows from Table 2 that the best results were achieved by the FME method. It is the combination of the MOBSP and the conservative visibility algorithm with the leaf-fragment-intersection test. Under certain circumstances, we observed that the FSE method led to better results. Particularly, when a detailed object was found invisible by FSE (and not by FME), the time saved in rendering exceeded the time needed for more complex visibility determination.

8. Conclusion

In this paper we have introduced the concept of occlusion trees. It was shown that the occlusion trees can be used to determine the viewpoint-to-region visibility efficiently by exploiting the spatial coherence of occluders. We presented an algorithm determining the visibility of a polyhedra exactly with respect to the selected occluders. Further, a fast conservative visibility algorithm applicable to regions of more general shape was described. Although the precision of the conservative algorithm is generally lower than the exact one, it performs well in practice.

The concept of occlusion trees was used in the hierarchical visibility culling for the rendering of complex models. It was shown that, using the occlusion tree, we can efficiently avoid rendering of invisible parts of a model. For models with high depth complexity, these savings are significant. Since the visibility is determined in a hierarchical fashion, the spatial coherence is exploited in the algorithm. We also presented the temporal coherence heuristics for the hierarchical visibility culling.

9. Future Work

Many themes for future work have been encountered in the paper. Some of the themes are related to the occlusion tree data structure itself, others to the application of occlusion trees in the scope of hierarchical visibility culling.

We assumed occluders to be sufficiently large convex polygons. This assumption is often false in practice. We propose model simplification for the purposes of visibility algorithms. The simplified model description should consist solely of convex polygons which should keep occlusion properties of objects they were generated from.

Another topic for future work concerns the occluder selection. It can be advantageous to determine the set of potential occluders in preprocessing [9, 3]. We currently develop an algorithm that computes the potential occluders for a certain region. The algorithm uses a sophisticated sampling scheme to determine the asset of an occluder.

We used a user-specified constant (minimum cost) to decide if a node of the spatial hierarchy should be tested for visibility. The minimum cost could be determined automatically based on the ratio of the speed of the rendering subsystem and the speed of the processor. It could also reflect how the algorithm succeeds in the visibility culling for a particular model.

Regarding the MOBSP, we are going to give a probabilistic analysis of the conservative visibility algorithm. Based on this analysis, we want to increase the precision of the algorithm. It should be achieved by inserting additional nodes to the MOBSP, where the probability of visibility misclassification is high.

Currently, the occlusion tree is built from a scratch in each rendering frame. We study the possibility of exploiting the temporal coherence in the construction of the occlusion tree. Finally, we would like to apply the occlusion trees to accelerate the casting of shadow rays in the ray tracing algorithm.

Acknowledgements

This work has been supported by a grant number 1252/1998 from the Grant Agency of the Ministry of Education of the Czech Republic.

References

[1] H. Chen and W. Wang. The feudal priority algorithm on hidden-surface removal. In *Proceedings of SIGGRAPH '96*, pages 55–64, Aug. 1996.

[2] N. Chin and S. Feiner. Near real-time shadow generation using BSP trees. In *Proceedings of SIGGRAPH '90*, pages 99–106, Aug. 1990.

[3] S. Coorg and S. Teller. A spatially and temporally coherent object space visibility algorithm. Technical Report TM-546, Department of Computer Graphics, MIT, Feb. 1996.

[4] F. Durand, G. Drettakis, and C. Puech. The 3D Visibility Complex, a new approach to the problems of accurate visibility. In *Eurographics Workshop on Rendering*, June 1996.

[5] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *Proceedings of SIGGRAPH '80*, pages 124–133, July 1980.

[6] N. Greene. Detecting intersection of a rectangular solid and a convex polyhedron. In *Graphics Gems IV*, pages 74–82. Academic Press, Boston, 1994.

[7] N. Greene. Hierarchical polygon tiling with coverage masks. In *Proceedings of SIGGRAPH '96*, pages 65–74, Aug. 1996.

[8] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In *Proceedings of SIGGRAPH '93*, pages 231–238. ACM Press, Aug. 1993.

[9] 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.

[10] M. Kaplan. Space-tracing: A constant time ray-tracer. In *SIGGRAPH '85 State of the Art in Image Synthesis seminar notes*, pages 149–158. Addison Wesley, July 1985.

[11] D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *1995 Symposium on Interactive 3D Graphics*, pages 105–106. ACM SIGGRAPH, Apr. 1995.

[12] B. Naylor, J. Amanatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. In *Proceedings of SIGGRAPH '90*, pages 115–124, Aug. 1990.

[13] H. Plantinga and C. Dyer. Visibility, occlusion, and the aspect graph. *International Journal of Computer Vision*, 5(2):137–160, 1990.

[14] J. Rohlf and J. Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In *Proceedings of SIGGRAPH '94*, pages 381–395, July 1994.

[15] S. Teller and P. Hanrahan. Global visibility algorithms for illumination computations. In *Proceedings of SIGGRAPH '93*, pages 239–246, 1993.

[16] S. J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, Dept. of Computer Science, University of California, Berkeley, 1992. Also available as Technical Report UCB//CSD-92-708.

[17] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Proceedings of SIGGRAPH '91*, pages 61–69, July 1991.

[18] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *Proceedings of SIGGRAPH '87*, volume 21, pages 153–162, July 1987.

[19] R. Yagel and W. Ray. Visibility computation for efficient walkthrough of complex environments. *Presence: Teleoperators and Virtual Environments*, 5(1), 1995.

[20] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. Technical Report TR97-004, Department of Computer Science, University of North Carolina - Chapel Hill, Feb. 21 1997.

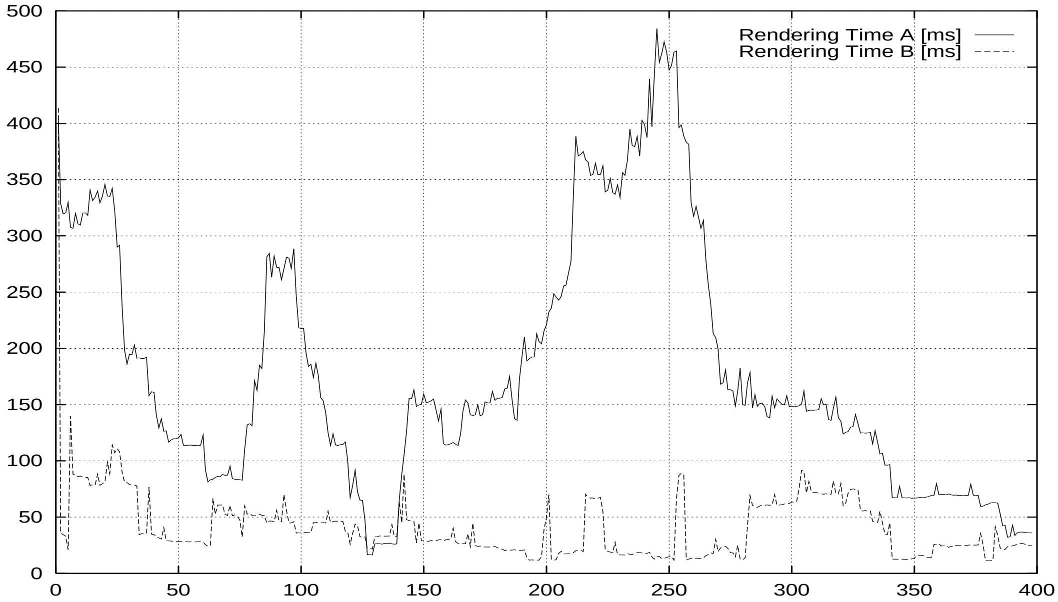

Figure 9. Rendering times obtained for a sequence of 400 frames during walkthrough of the big-7 scene. Curve A corresponds to view-frustum culling only; curve B includes the hierarchical visibility culling using 32 occluders (method FME).

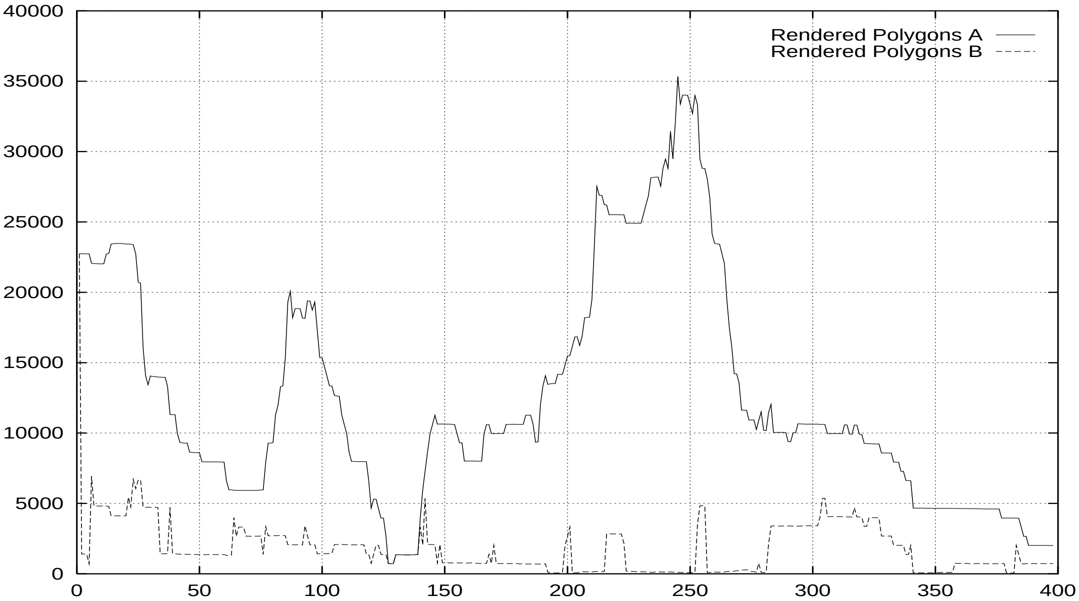

Figure 10. The amount of polygons rendered during a sequence of 400 frames during walkthrough the big-7 scene. Curve A corresponds to view-frustum culling only; curve B includes the hierarchical visibility culling using 32 occluders (method FME).