A Survey of Visibility for Walkthrough Applications

Daniel Cohen-Or[1,*] Yiorgos Chrysanthou[2,†] Cláudio T. Silva[3,‡]

[1]Tel Aviv University [2] University College London [3]AT&T Labs-Research

Abstract

The last few years have witnessed tremendous growth in the complexity of computer graphics models as well as network-based computing. Although significant progress has been made in the handling of specific types of large polygonal datasets (i.e., architectural models) on single graphics workstations, only recently have researchers started to turn their attention to more general solutions, which now include network-based graphics and virtual environments. The situation is likely to worsen in the future since, due to technologies such as 3D scanning, graphical models are becoming increasingly complex. One of the most effective ways of managing the complexity of virtual environments is through the application of smart visibility methods.

Visibility determination, the process of deciding what surfaces can be seen from a certain point, is one of the fundamental problems in computer graphics. It is required not only for the correct display of images but also for such diverse applications as shadow determination, global illumination, culling and interactive walkthrough. The importance of visibility has long been recognized, and much research has been done in this area in the last three decades. The proliferation of solutions, however, has made it difficult for the non-expert to deal with this effectively. Meanwhile, in network-based graphics and virtual environments, visibility has become a critical issue, presenting new problems that need to be addressed.

In this survey we review the fundamental issues in visibility and conduct an overview of the work performed in recent years.

1 Introduction

The term *visibility* is very broad and has many meanings and applications in various fields of computer science. Here, we focus on visibility algorithms in support of virtual reality applications. For a more general survey see [23] (also appears in [14]). For those interested in the computational geometry literature, see [21, 20, 22]. Zhang's thesis [80] contains a short survey of computer graphics visibility work. Moller and Haines [50, Chapter 7] cover several aspects of visibility culling.

[*]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel, daniel@math.tau.ac.il

[†]Department of Computer Science, University College London, Gower Street, London WC1E 6BT, United Kingdom, y.chrysanthou@cs.ucl.ac.uk

[‡]AT&T Labs-Research, 180 Park Ave., PO Box 971, Florham Park, NJ 07932; csilva@research.att.com.

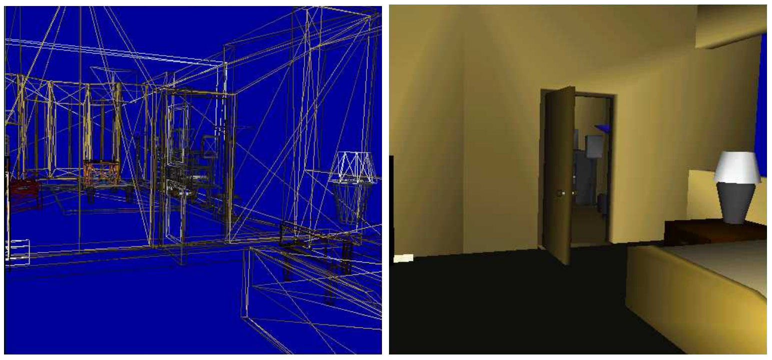

(a) (b)

Figure 1: With indoor scenes often only a very small part of the geometry is visible from any given viewpoint. Courtesy of Craig Gotsman, Technion.

We deal primarily with algorithms related to walkthrough applications where we assume that a scene consists of a very large number of primitives. Moreover, we assume that models keep getting larger and more complex and that user appetite will never be satisfied with the computational power available. For very complex models we can usually do better with a smart rendering algorithm than with faster machines.

One of the most interesting visibility problems in this context is the one of selecting a set of polygons from the model that is visible from a given viewpoint. More formally (after [21]), let the scene, $\mathcal{S}$, be composed of modeling primitives (*e.g.*, triangles) $\mathcal{S} = \{\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_n\}$, and a viewing frustum defining an eye position, a view direction, and a field of view. The visibility problem encompasses finding the visible fragments within the scene, that is, connected to the eyepoint by a line segment that meets the closure of no other primitive. One of the obstacles to solving the visibility problem is its complexity. For a scene with $n = O(|\mathcal{S}|)$ primitives, the complexity of the set of visible fragments might be as high as $O(n^2)$ (i.e., quadratic in the number of primitives in the input).

What makes visibility an interesting problem is that for large scenes, the number of visible fragments is usually much smaller than the total size of the input. For example, in a typical urban scenes, one can see only a very small portion of the entire model, regardless of one's location. Such scenes are said to be *densely occluded*, in the sense that from any given viewpoint, only a small fraction of the scene is visible [15]. Other examples include indoor scenes, where the walls of a room occlude most of the scene, and in fact, from any viewpoint inside the room, one may only see the details of that room or those visible through the *portals*, see Figure 1. A different example is a copying machine, shown in Figure 2, where from the outside one can only see its external parts. Although intuitive, this information is not available as part of the model representation, and only a non-trivial algorithm can determine it automatically. Note that one of its doors might be open.

Visibility is not an easy problem, since a small change in the viewpoint might cause large changes in the visibility. It means that solving the problem at one point does not help much in solving it at a nearby point.

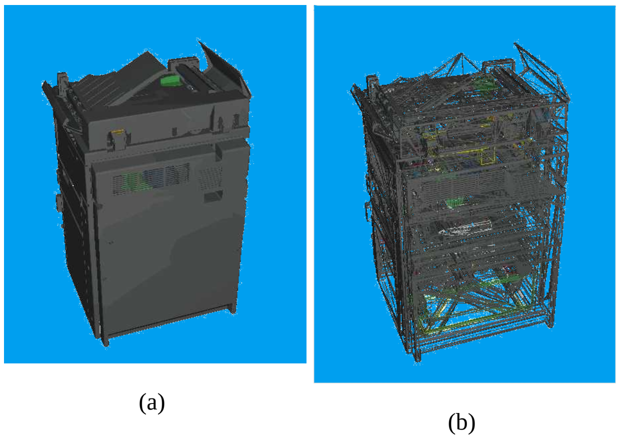


Figure 2: A copying machine; only a fraction of the geometry is visible from the outside. Courtesy of Craig Gotsman, Technion.

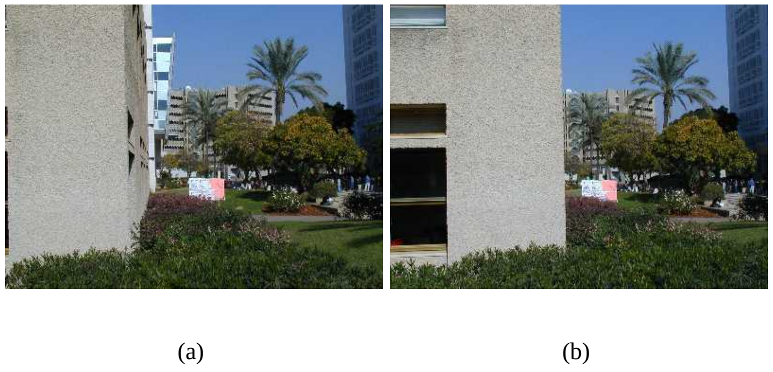


Figure 3: A small change in the viewing position can cause large changes in the visibility.

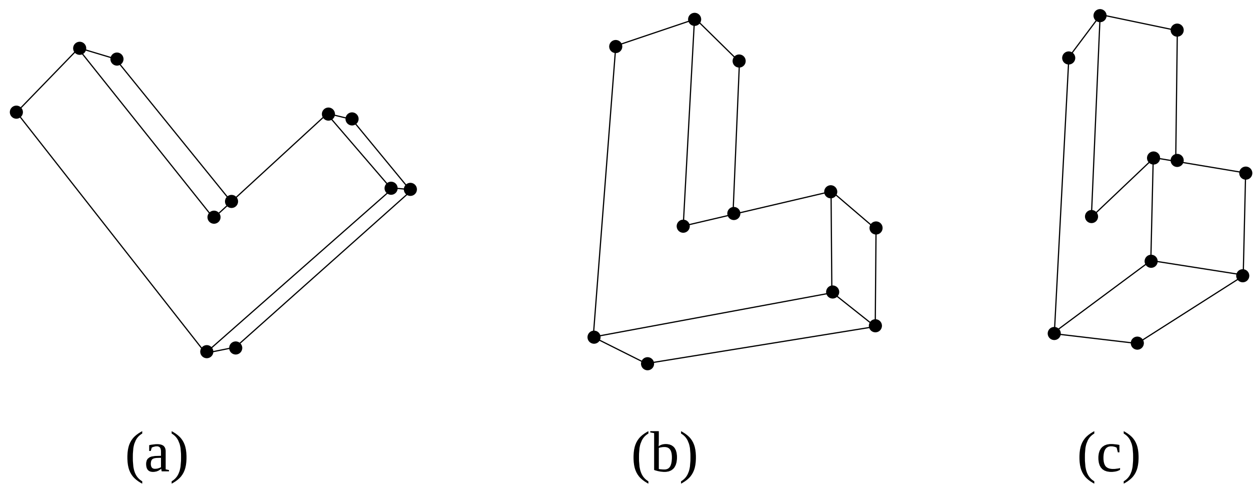


Figure 4: Two different view directions of an object have the same *aspect* if and only if the corresponding Image Structure Graphs are isomorphic. Note that (a) and (b) have the same aspect, which is different to (c).

An example of this can be seen in Figure 3. The *aspect graph*, described in Section 2, and the *visibility complex* (described in [23]) sheds light on the complex characteristics of visibility.

The rest of this paper is organized as follows. We first give a short description of the aspect graph, which is a fundamental concept in visibility, in Section 2. Then, we briefly review some 3D graphics hardware features which are important for visibility culling (Section 4). Next, we present a taxonomy of visibility culling algorithms in Section 5. This introductory part is then followed by a more detailed description and analysis of recent visibility-culling algorithms.

2 The aspect graph

When dealing with visibility, it is useful to consider an important theoretical concept called the aspect graph [26]. Let us look at the two isomorphic graphs in Figure 4. They are a projection of a 3D object; however, we treat them as 2D entities. First, let us define the *Image Structure Graph* (ISG) as a planar graph, defined by the outlines of an image created by projecting a polyhedral object in a certain view direction. Then two different view directions of an object have the same *aspect* if and only if their corresponding ISGs are isomorphic. Now we can partition the viewspace into maximal connected regions in which the viewpoints have the same view or aspect. This partition is the VSP - *the visibility space partition*, where the boundary of a VSP region is called a *visual event* as it marks a change in visibility (see Figure 5).

The term, aspect graph, refers to the graph created by assigning a vertex to each region of the VSP, where the edges connect adjacent regions.

Figures 5 and 6 show a visibility space partition in 2D, which is created by just two and three segments (the 2D counterparts of polygons), respectively. One can observe that the number of aspect regions is already large, and in fact, can be shown to grow quite rapidly.

Plantinga and Dyer [57] discuss aspect graphs and their worst-case complexity, including algorithms for efficiently computing aspect graphs. The worst complexity of aspect graphs is quite high, and in three

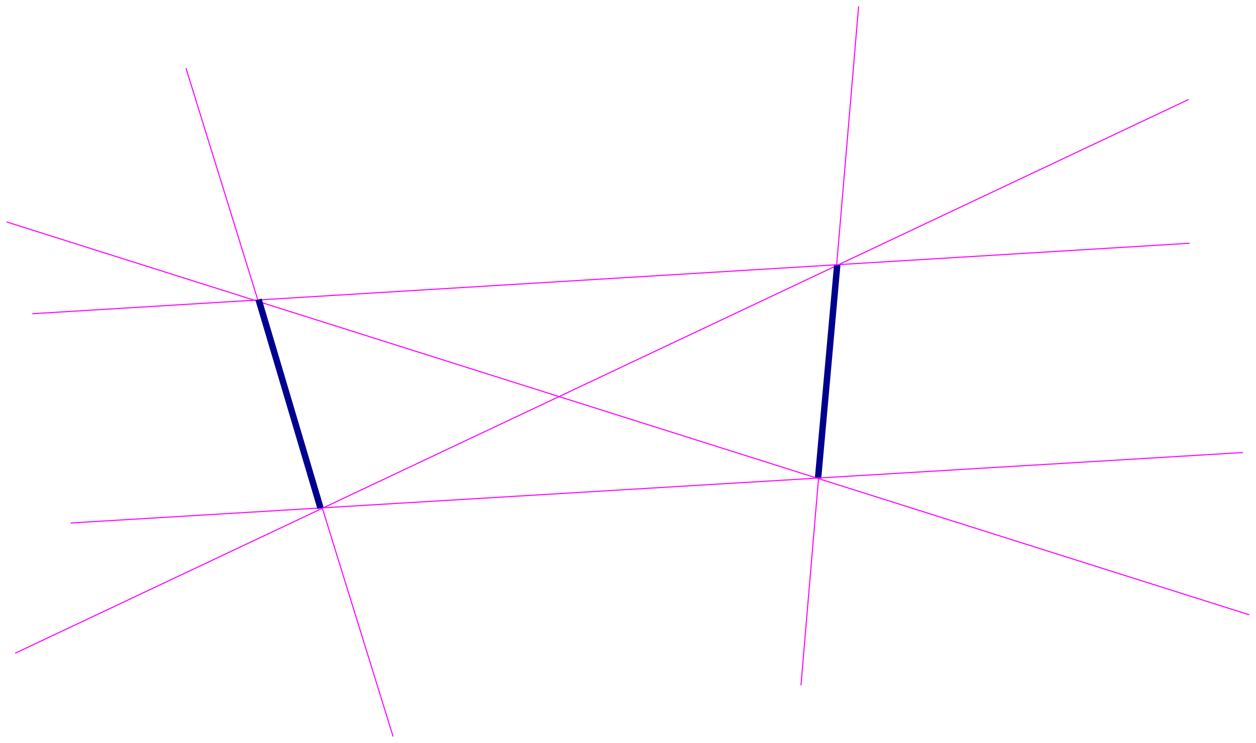


Figure 5: 2 polygons - 12 aspect regions.

dimensions, can be as large as $O(n^9)$. For a typical number of segments (say tens of thousands), in terms of space and time it turns out that computing the aspect graph is computationally impractical. (Plantinga [58] proposes an early conservative visibility algorithm based on his aspect graph work.)

However, as can be seen in Figure 7, different aspect regions can have equal sets of visible polygons. This means that there are far fewer different regions of different visibility sets than different aspects.

Looking once again at the aspect partition of the two segments in Figure 8, we can treat one as an occluder and the other as the occludee, defining their endpoint connecting lines as *supporting lines* and *separating lines*. These lines partition the space into three regions: (i) the region from which no portion of the occludee is visible, (ii) the region from which only a portion of the occludee is visible, and (iii) the region from which the occluder does not occlude any part of the occludee [17].

The 3D visibility complex [23] is another way of describing and studying the visibility of 3D space by a dual space of 3D lines, in which all the visibility events are described. This structure is global, spatially coherent and complete, since it encodes all the visibility relations in 3D. It allows efficient visibility computations, such as view extraction, computation of the aspect graph, discontinuity meshing and form-factor computation.

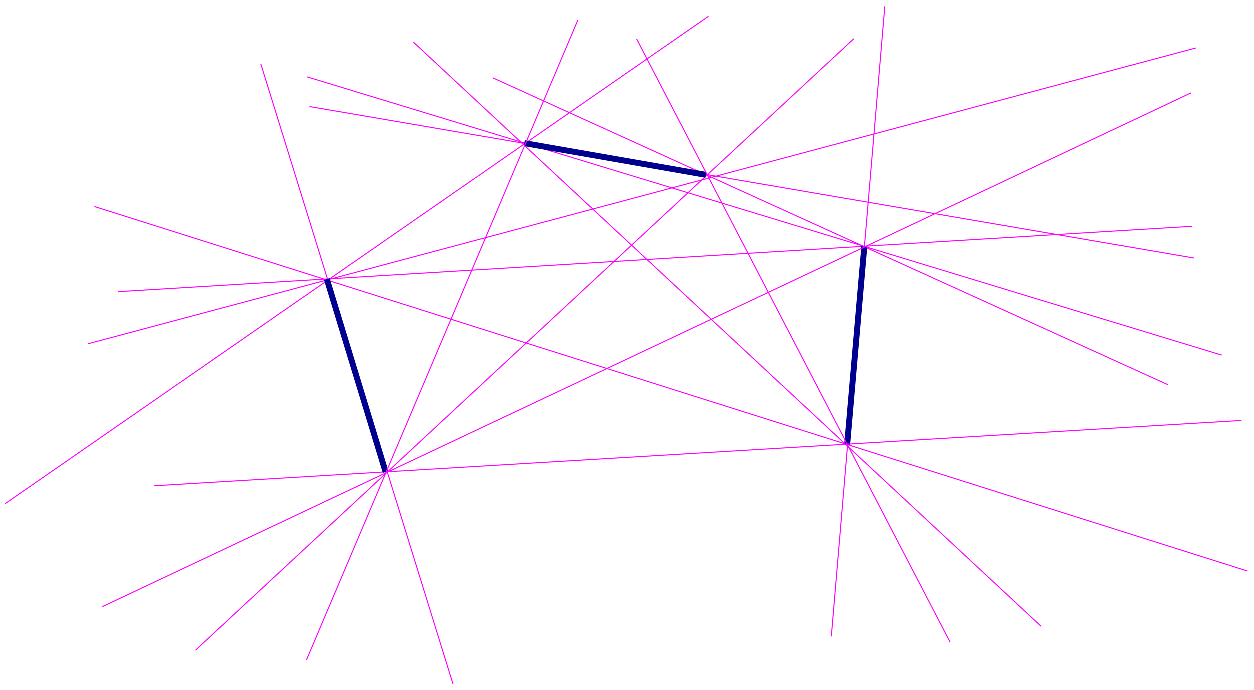


Figure 6: 3 polygons - “many” aspect regions.

3 Hidden-surface removal methods

As mentioned in the introduction, one of the fundamental visibility problems in computer graphics is the determination of the visible parts of the scene, the so-called *hidden-surface removal* (HSR) (also known as visible-surface determination) algorithms. Assuming the scene is composed of and represented by triangles, these algorithms not only define the set of visible triangles, but also the exact portion of each visible triangle that has to be drawn into the image.

An early classification was proposed by Sutherland et al. [69]. Later it was reviewed in [28] and also in the computational geometry literature [22]. The HSR methods can be broadly classified into three groups:

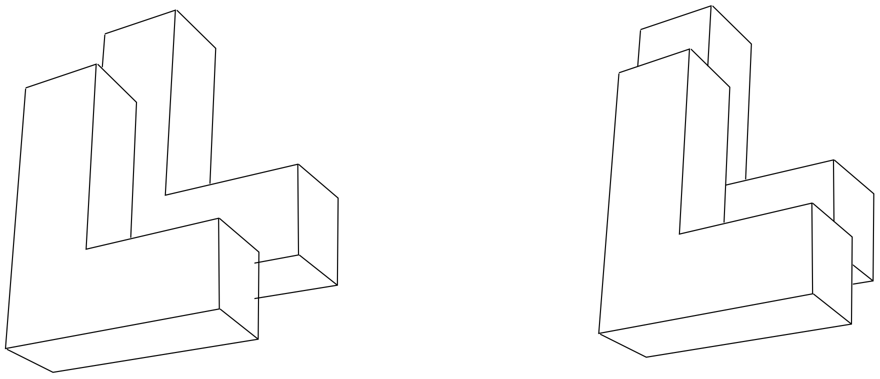


Figure 7: Different aspect regions can have equal sets of visible polygons.

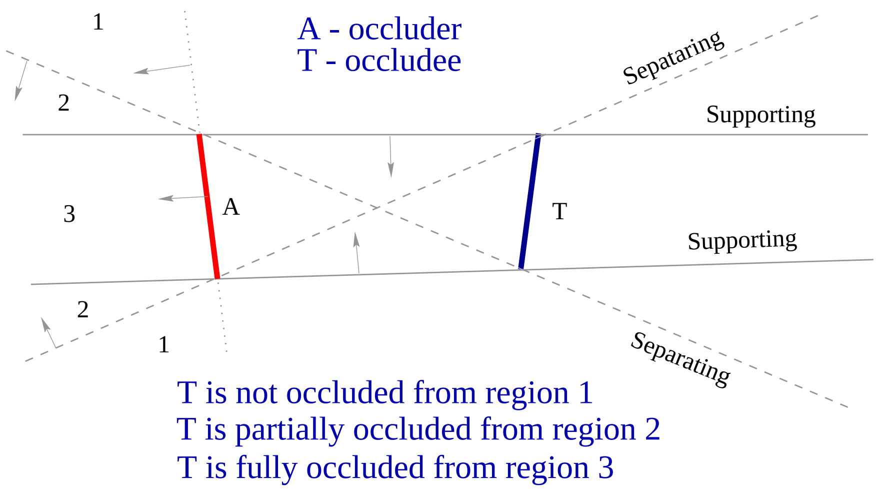

1
A - occluder
T - occludee
Sepataring
2
Supporting
3
A
T
Supporting
2
1
Separating
T is not occluded from region 1
T is partially occluded from region 2
T is fully occluded from region 3

Figure 8: Supporting and separating planes.

object precision methods, image precision methods and hybrid methods. Object precision methods compare objects to decide exactly which parts of each one is visible in the image. One of the first examples of this class was presented by Weiler and Atherton [74]. They used a general clipping method to partition polygons which were further away from the viewpoint using the boundaries of those closer, discarding the regions where they overlapped. Object precision algorithms can be considered as a continuous solution (to the extent that machine precision allows) but often suffer from scalability problems as the size of the environment grows, and are difficult to implement robustly.

Image precision algorithms on the other hand operate on the discrete representation of the image. The overall idea is to produce a solution at the resolution of the required image by determining the visible object at each pixel. Ray casting is one example of this class [3]. Other examples are the scan-line methods [9, 73], variations of which are popular in games and flight simulators, and the z-buffer [10] whose implementation in hardware has made it the de-facto standard HSR method today.

Finally, in the third class, are hybrid methods that combine object and image precision operations. Of most interest are the so-called list-priority algorithms. Their underlying idea is to quickly determine a partial order list of all polygons such that for any given pair p, q, if p can occlude some part of q, then p comes earlier in the list. In other words, if q is after p in the list, q cannot occlude p. Then during rendering, the ordered polygons are drawn back-to-front, thus occluding polygons are correctly drawn into the image, covering only those parts that are occluded. Some of the early methods were those of Schumacker et al. [61] and Newell et al. [54] and later Fuchs et al.'s BSP trees [29]. One of the additional features of the list-priority techniques is that they are able to correctly handle the rendering of transparent objects. Although the methods were originally designed for depth ordering of individual polygons, some of their ideas have been used in occlusion methods (i.e., [34]).

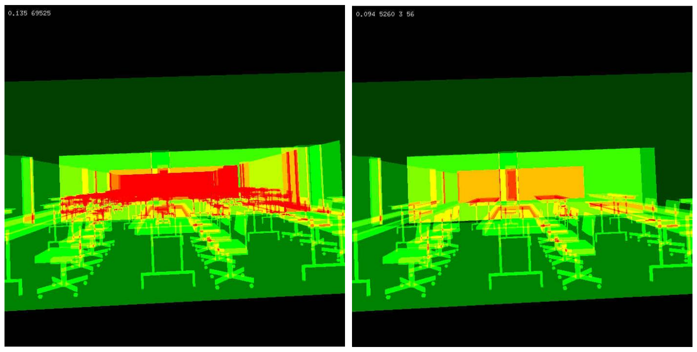


Figure 9: Depth complexity of the scene as rendered by (a) view-frustum culling, (b) a conservative occlusion culling technique. The depth complexity ranges from light green (low) to bright red (high). If the occlusion-culling algorithm were “exact”, (b) would be completely green.

4 3D graphics hardware

In this section, we briefly review some common features of modern 3D graphics hardware which are helpful in visibility calculations.

We do not cover the important topic of efficient use of specific graphics hardware, in particular, the optimization of specific applications to specific hardware. A good starting point is the text by Moeller and Haines [50]. The interested reader should also consult the OpenGL tutorials given every year at Siggraph.

Hardware features for specific visibility calculations are usually bare-bones, because of the need for graphics hardware to be streamlined and very simple. Most often, by careful analysis of the hardware, it is possible to combine a software solution which exploits the basic hardware functionality, but at the same time also improves it considerably.

4.1 Graphics pipeline

The graphics pipeline is the term used for the path a particular primitive takes in the graphics hardware from the time the user defines it in 3D to the time it actually contributes to the color of a particular pixel on the screen. At a very high level, given a primitive, it must undergo several simple tasks before it is drawn on the screen.

Often, such as in the OpenGL graphics pipeline, a triangle primitive is first transformed from its local coordinate frame to a world coordinate frame; then it is transformed again to a normalized coordinate frame, where it is clipped to fit the view volume. At this point, a division by w is performed to obtain non-homogeneous normalized coordinates, which are then normalized again to be in screen-space. Depending on a set of user-defined state flags, the hardware can reject the primitive based (among other things) on the direction of its normal. This is called back-face culling, and is a very primitive form of visibility culling.

Once a primitive has passed all these phases, the rasterization phase can start. It is here that the colors (and other properties) of each pixel are computed. During rasterization, we usually refer to the primitives as "fragments". Modern graphics architectures have several per-fragment operations that can be performed on each fragment as they are generated.

As fragments are computed, they pass through further processing, and the hardware will incrementally fill several buffers in order to compute the image. The actual image we see on the screen is only one of these buffers: the color buffer. Other buffers include the stencil buffer and the depth (or z-) buffer. There are other buffers, such as the accumulation buffer, etc., but we do not use them in the rest of this paper. In OpenGL, updates to the different buffers can be toggled by a set of function calls, *e.g.* `glEnable(GL_DEPTH_TEST)`.

One view of the OpenGL buffers is as a simple processor with little memory (just a few bytes), and a limited instruction set. Recently, techniques for performing general computations using the OpenGL pipeline have been proposed. Two such examples are Peercy et al. [56] and Trendall and Stewart [71].

4.2 Stencil buffer

The stencil buffer is composed of a small set of bits (usually more than 4) that can be used to control which areas of the other buffers,)textite.g. color buffer), are currently active for drawing. A common use of the stencil buffer is to draw a piece of static geometry once (the cockpit of an airplane), and then mask the area so that no further changes can be made to those pixels.

But the stencil buffer is actually much more flexible, since it is possible to change the value of the pixels on the stencil buffer depending on the outcome of the test performed. For instance, a very useful computation that uses the stencil buffer is to compute the "depth-complexity" of a scene. For this, one can simply program the stencil buffer as follows:

```
glStencilFunc(GL_ALWAYS, ~0, ~0);
glStencilOp(GL_KEEP, GL_INCR, GL_INCR);
```

which essentially means the stencil buffer will get incremented every time a pixel is projected onto it. Figure 9 shows a visual representation of this. The stencil buffer is useful in several types of visibility computations, such as real-time CSG calculations [31], occluder calculations [25], and so on.

4.3 Z-buffer

The z-buffer is similar to the stencil buffer, but serves a more intuitive purpose. Basically, the z-buffer saves its "depth" at each pixel. The idea is that if a new primitive is obscured by a previously drawn primitive, the z-buffer can be used to reject the update. The z-buffer consists of a number of bits per pixel, usually 24 bits in most current architectures.

The z-buffer provides a brute-force approach to the problem of computing the visible surfaces. Just rendering each primitive, and the z-buffer will take care of not drawing in the color buffer of those primitives that are not visible. The z-buffer provides a great functionality, since (on fully hardware-accelerated architectures) it is able to solve the visibility problem (up to screen-space resolution) of a set of primitives in the time it would take to scan-convert them all.

As a visibility algorithm, the z-buffer has a few drawbacks. One drawback is that each pixel in the z-buffer is touched (potentially) as often as its depth complexity, although one simply needs the top surface of each pixel. Because of this potentially excessive overdrawing a lot of computation and memory bandwidth is wasted. A visibility pre-filtering technique, such as back-face culling, can be used to improve the speed of rendering with a z-buffer.

There have been several proposals for improving the z-buffer, such as the hierarchical z-buffer [35] (see Section 7.1 and related techniques). A simple, yet effective hardware technique for improving the performance of the visibility computations with a z-buffer has been proposed by Scott et al. [62], see Section 7.5.

5 Visibility culling algorithms

Visibility algorithms have recently regained attention in computer graphics as a tool for handling large and complex scenes, which consist of millions of polygons. In the early 1970s hidden surface removal (HSR) algorithms (see Section 3) were developed to solve the fundamental problem of determining the visible portions of the polygons in the image. In light of the Z-buffer being widely available, and exact visibility computations being potentially too costly, one idea is to use the Z-buffer as a filter, and design algorithms that lower the amount of overdraw by computing an approximation of the *visible set*. In more precise terms, define the visible set $\mathcal{V} \subset \mathcal{S}$ to be the subset of primitives which contribute to at least one pixel of the screen.

In computer graphics, visibility-culling research mainly focuses on algorithms for computing (hopefully tight) estimations of $\mathcal{V}$, then using the Z-buffer to obtain correct images.

5.1 View frustum and back-face culling

The simplest examples of visibility culling algorithms are back-face and view-frustum culling [28]. Back-face culling algorithms avoid rendering geometry that faces away from the viewer, while viewing-frustum culling algorithms avoid rendering geometry that is outside the viewing frustum. These culling operations can be left to the graphics hardware without affecting the final image. However, that comes at a great cost since the polygons will be processed through most of the pipeline only to be rejected just before scan converting.

Back-facing polygons can be identified with a simple dot product, since their normal points away from the view-point. On average we expect half the scene polygons to be back-facing, so ideally we would like to avoid processing all of them. Kumar et al. [45] present a method which has a sub-linear number of polygons. The input model is partitioned into a hierarchy of clusters based on both similarity of orientation and physical proximity of the polygons. The viewspace is also partitioned with respect to the clusters. At each frame the viewpoint position is hierarchically compared with the clusters in order to quickly reject the bulk of the back-facing polygons. Frame-to-frame coherence is further used to accelerate the process.

View frustum culling is usually performed using either a hierarchy of bounding volumes or a spatial data structure, such as a KD-tree, octree or BSP tree. This is hierarchically compared with the view frustum to quickly reject parts of the scene that are clearly outside [13].

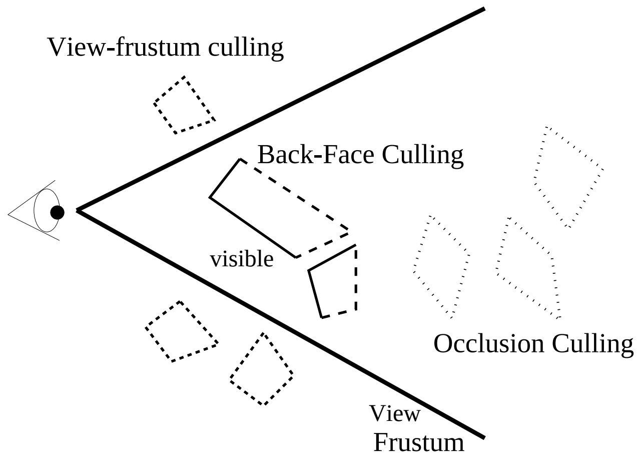


Figure 10: Three types of visibility culling techniques: (i) view frustum culling, (ii) back-face culling and (iii) occlusion culling.

Slater et al. [65] present an alternative approach which makes heavy use of frame-to-frame coherence. It relies on the fact that the sets of objects that are completely outside, completely inside, or intersect the boundary of the view volume, change slowly over time. This coherence is exploited to develop an algorithm that quickly identifies these three sets of objects, and partitions those completely outside into subsets which are probabilistically sampled according to their distance from the view volume. A statistical object representation scheme is used to classify objects into the various sets. The algorithm is implemented in the context of a BSP tree.

Very recently, Assarsson and Möller [4] proposed a new view-frustum culling technique. Their work is based on shrinking the view frustum to enable the use of point-containment queries to efficiently accept or reject primitives.

5.2 Occlusion culling

Even though both of the above techniques are very effective in culling geometry, more complex techniques can lead to substantial improvements in rendering time. The term *Occlusion culling* is used for visibility techniques that avoid rendering primitives that are occluded by some other part of the scene. This technique is global as it involves interrelationship among polygons and is thus far more complex than local visibility techniques. The three kinds of visibility culling can be seen in Figure 10.

It is important to note the differences between occlusion culling and HSR. HSR algorithms determine which portions of the scene need to be drawn on the screen. These algorithms eventually remove the occluded parts, but in doing so, are expensive, since they usually have to touch all the primitives in $\mathcal{S}$ (and actually have a running time that is superlinear in the size of $\mathcal{S}$). Occlusion-culling techniques are supposed to be *output sensitive*, that is, their running time should be proportional to the size of $\mathcal{V}$, which for most complex scenes, is a small subset.

Let us define the following notation for a scene consisting of polygons.

- The *exact visibility set*, $\mathcal{V}$, is the set of all polygons which are at least partially visible, and only these polygons.
- The *approximate visibility set*, $\mathcal{A}$, is a set that includes most of the visible polygons plus maybe some hidden ones.
- The *conservative visibility set*, $\mathcal{C}$, is the set that includes at least all the visible objects plus maybe some additional invisible objects. It may classify an invisible object as visible, but may never classify a visible object as invisible.

5.3 Conservative visibility

A very important concept is the idea of *conservative visibility*. The idea is to design efficient output-sensitive algorithms for computing $\mathcal{C}$, then to use a standard HSR as a back-end for computing the correct image.

These methods yield a *potential visibility set* (PVS) which includes all the visible polygons, plus a small number of occluded polygons. Then the HSR processes the (hopefully small) excess of polygons included in the PVS. Conservative occlusion culling techniques have the potential to be significantly more efficient than the HSR algorithms. Conservative culling algorithms can also be integrated into the HSR algorithm, aiming towards an output sensitive algorithm [35].

To reduce the computational cost, the conservative occlusion culling algorithms usually use a hierarchical data structure where the scene is traversed top-down and tested for occlusion against a small number of selected occluders [18, 39]. In these algorithms the selection of the candidate occluders is done before the online visibility calculations. The efficiency of these methods is directly dependent on the number of occluders and their effectiveness. Since the occlusion is tested from a point, these algorithms are applied in each frame during the interactive walkthrough.

5.4 A taxonomy of occlusion culling techniques

In order to roughly classify the different visibility-culling algorithms, we will employ a loosely-defined taxonomy:

- *Conservative vs. Approximate.*

  Few visibility-culling algorithms attempt to find the exact visible set, since they are mostly used as a front-end for another hidden-surface removal algorithm, most often the Z-buffer. Most techniques described in this paper are conservative, that is, they overestimate the visible set. Only a few approximate the visible set, but are not guaranteed of finding all the visible triangles, e.g., PLP [43, 42] (there is also a conservative version of PLP which is described in [40]). Others can be tuned to be conservative or approximate depending on the time constraint and available resources. In an attempt to accelerate the culling step they might actually miss small visible primitives, such as HOM [81, 80], and also the OpenGL assisted occlusion culling of Bartz et al. [6, 5].

- *Point vs. Region.*

  The major difference here is whether the particular algorithm performs computations that depend on the exact location of the viewpoint, or performs bulk computations which can be re-used anywhere in a region of space.

  Obviously, from-region algorithms perform their visibility computations on a region of space, that is, while the viewer is inside that region, these algorithms tend to render the same geometry. The strength of the from-region visibility set is that it is valid for a number of frames, and thus its cost is amortized over a number of frames (see Section 8).

  Most other algorithms attempt to perform visible-set computations that depend on the exact location of the viewpoint.

- *Precomputed vs. Online.*

  Most techniques need some form of preprocessing, but what we mean by "precomputed" are the algorithms that actually store visibility computations as part of their preprocessing.

  Almost all of the from-region algorithms should be classified as "precomputed". A notable exception is [44].

  In general, the other algorithms described do their visibility computation "online", although much of the preprocessing might have been performed before. For instance, HOM [81, 80], DDO [7], Hudson et al. [39], Coorg and Teller [18], perform some form of occluder selection which might take a considerable amount of time (in the order of hours of preprocessing), but in general have to save very little information to be used during rendering.

- *Image space vs. Object space.*

  Almost all of the algorithms use some form of hierarchical data structure. We classify algorithms as operating in "image-space" versus "object-space" depending on where the actual visibility determination is performed.

  For instance, HOM [81, 80] and HZB [35, 36] perform the actual occlusion determination in image-space (e.g., in HOM, the occlusion maps are compared with a 2D image projection and not the 3D original representation.). Other techniques that explore image-space are DDO [7] (which also explores a form of object-space occlusion-culling by performing a view-dependent occluder generation) and [6, 5].

  Most other techniques work primarily in object-space.

- *Software vs. Hardware.*

  Several of the techniques described can take further (besides the final z-buffer pass) advantage of hardware assistance either for its precomputation or during the actual rendering.

  For instance, the from-region technique of Durand et al. [25] makes non-trivial use of the stencil buffer; HOM [81, 80] uses the texture hardware to generate mipmaps; [6, 5] uses the OpenGL selection mode; and Meissner et al. [49] uses the HP occlusion-culling test.

The HP occlution-culling test [62] is not actually an algorithm on its own, but a building block for further algorithms. It is also exploited (and expanded) in [40].

- *Dynamic vs. Static scenes.*

  A few of the algorithms in the literature are able to handle dynamic scenes, such as [68] and HOM [81].

  One of the main difficulties is handling changes to object hierarchies that most visibility algorithms use. The more preprocessing used, the harder it is to extend the algorithm to handle dynamic scenes.

- *Individual vs. Fused occluders.*

  Given three primitives, A, B, and C, it might happen that neither A nor B occlude C, but together they do occlude C. Some occlusion-culling algorithms are able to perform *occluder-fusion,* while others are only able to exploit single primitive occlusion. citeCohen-Or:1998:CVA,Coorg:1997:ROC,ct-tccv-96 give examples of techniques that use a single (fixed number of) occluder(s). Papers [79, 35, 42] support occluder fusion.

5.5 Related problems

There are many other interesting visibility problems, for instance:

– **Shadow algorithms.** The parts that are not visible from the light source are in the shadow. So occlusion culling and shadow algorithms have a lot in common and in many ways are conceptually similar [78, 12]. It is interesting to note that conservative occlusion culling techniques have not been as widely used in shadow algorithms.

– **The Art Gallery Problem.** One classic visibility problem is that of positioning a minimal number of guards in a gallery so that they cover all the walls of the gallery. This class of problem has been extensively studied in computational geometry, see, for instance, O'Rourke [55].

  In this context, "cover" can have a different meaning. Much is known about this problem in 2D, but in 3D, it gets much harder. Fleishman et al. [27] proposes an algorithm for automatically finding a set of posing cameras which cover a 3D environment. Stuerzlinger [66] proposes a technique for a similar problem.

– **Radiosity solutions.** This is a much more difficult problem to compute accurately. In radiosity, energy needs to be transfered from each surface to every other *visible* surface in the environment [32, 37]. This requires a from-region visibility determination to be applied at each surface or patch. Exact solutions are not practical, and techniques such as clustering [64] are often used.

6 Object-space culling algorithms

Work on object-space occlusion culling dates back at least to the work of Teller and Sèquin [70] and Airey et al. [1] on indoor visibility.

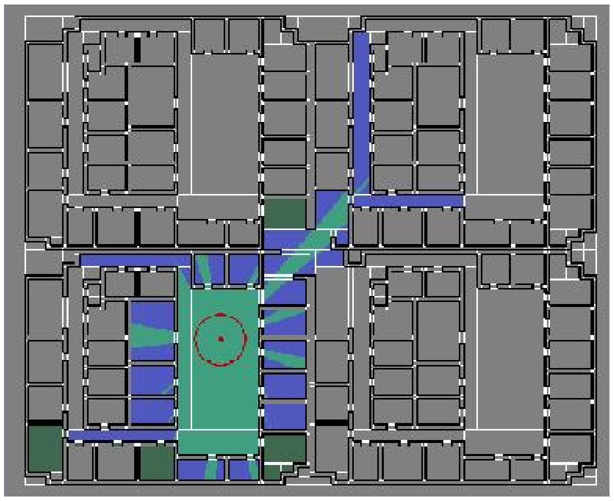


Figure 11: Results from [70] showing the potentially visible set from a given cell. Courtesy of Seth Teller, UC, Berkeley.

The work of Teller and Sèquin is mostly based on 2D, since it deals with computing potentially visible sets for cells in an architectural environment. Their algorithm first subdivides space into cells using a 2D BSP tree. Then it uses the connectivity between the cells, and computes whether straight lines can hit a set of "portals" (mostly doors) in the model. They elegantly model the stabbing problem as a linear programming problem, and in each cell save the collection of potentially visible cells. Figure 11 shows one of the results presented in their paper. The linear programming solution computes cell-to-cell visibility, which does not constrain the position of a viewer inside the cell, nor the direction in which he is looking, and thus is far too conservative. They also propose techniques which further constrain the PVS by computing eye-to-cell visibility, which take into consideration the view-cone emanating from the viewer.

Another technique that exploits cells and portals in models is described in Luebke and Georges [48]. Instead of precomputing the visibility, Luebke and Georges perform an on-the-fly recursive depth-first traversal of the cells using screen-space projections of the portals to overestimate the portal sequences. In their technique they use a "cull box" for each portal, which is the axial 2D bounding box of the projected vertices of the portal. Any geometry which is not inside a cull box of the portal cannot be visible. The basic idea is then to clip the portal1s cull boxes as the cells are traversed, and only to continue the traversal into cells which have a non-zero (intersection) portal-sequence. Their technique is simple and quite effective; the source code (an SGI Performer library) is available for download from David Luebke's web page. [1]

6.1 Coorg and Teller

Coorg and Teller [17, 18] have proposed object-space techniques for occlusion culling. The technique in [18] is most suitable for use in the presence of large occluders in the scene. Their algorithm explores the visibility relationships between two convex objects as in Figure 12. In brief, while an observer is between

[1]Pfportals can be obtained at http://pfPortals.cs.virginia.edu.

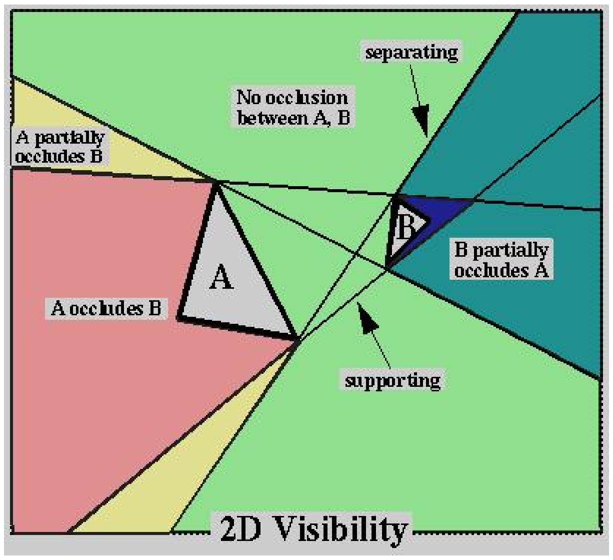


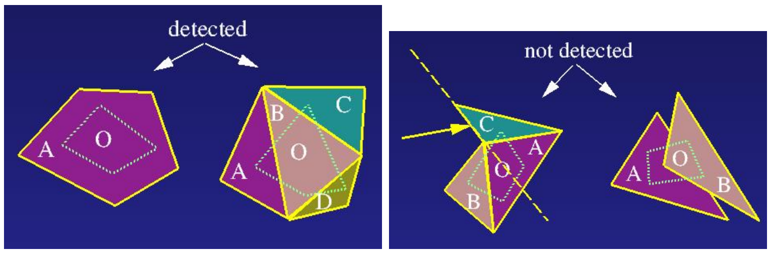
Figure 12: The figure highlights the visibility properties exploited by the algorithm of Coorg and Teller [17, 18]. While an observer is between the two supporting planes to the left of A, it is never possible to see B. Courtesy of Satyan Coorg, MIT.

the two supporting planes to the left of A, it is never possible to see B. The Coorg and Teller technique uses simple concepts such as this to develop a technique based on tracking visibility events among objects as the user moves and the relationships among objects change. The algorithm proposed in [17] is conservative, and explores temporal coherency as it tracks the visibility events.

In [17], Coorg and Teller give sufficiency conditions for computing the visibility of two objects (that is, whether one occludes the other), based on tracking relationships among the silhouette edges supporting and separating the planes of the different objects. They build an algorithm which incrementally tracks changes in those relationships. There, they also show how to use object hierarchies (based on octrees) to handle the potential quadratic complexity computational increase. One drawback of this technique (as pointed out by the authors in their subsequent work [18]) is precisely the fact that it needs to reconstruct the visibility information for a continuous sequence of viewpoints.

In [18], Coorg and Teller propose an improved algorithm. (It is still based on the visibility relationship shown in Figure 12.) Instead of keeping a large number of continuous visibility events, in [18], they dynamically choose a set of occluders, which is used to determine which portions of the rest of the scene cannot be seen. The scene is inserted into an object hierarchy, and the occluders are used to determine which portions of the hierarchy can be pruned, and not rendered.

Coorg and Teller [18] develop several useful building blocks for implementing this idea, including a simple scheme to determine when the fusion of multiple occluders can be added together (see Figure 13), and a fast technique for determining supporting and separating planes. They propose a simple metric for



(a) (b)

Figure 13: The figure illustrates that the algorithm described in [18] can perform occlusion fusion if the occluders combine to be a larger “convex” occluder. Courtesy of Satyan Coorg, MIT.

identifying the dynamic occluders which is based on approximating the solid angle an object subtends:

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

, where A is the area of the occluder, $\vec{N}$ the normal, $\vec{V}$ the viewing direction, and $\vec{D}$ the vector from the viewpoint to the center of the occluder.

6.2 Culling using shadow frusta

The work described by Hudson et al. in [39] is in several ways similar to the work of Coorg and Teller [18]. Their scheme also works by dynamically choosing a set of occluders, then using those occluders as the basis for culling the rest of the scheme. The differences between the two works lie primarily in the details. In [39], the authors propose extra criteria for choosing the occluders. Besides the Coorg and Teller solid-angle heuristic, they also propose taking into account the depth complexity and coherence of the occluders. They use a spatial partition of the scene, and for each cell, identifying the occluders that will be used anytime the viewpoint is inside that cell, and store them for later use.

A separate data structure, a hierarchy of bounding volumes is used for the occlusion culling. The way Hudson et al. determine which parts of the hierarchy are occluded is different to that of Coorg and Teller. For each of the *n* best occluders that fall within the view frustum, the authors build a shadow frustum using the viewpoint as the apex and passing through the occluder silhouette. The scene hierarchy is tested top-down against each of these shadow frusta. If a node of the hierarchy is found to be totally enclosed by one of the frusta then it is occluded and hence discarded (for this frame). If it is found not to intersect any of them then it totally visible and all the objects below it are rendered. If however it partially overlaps even one of them then its children need to be further tested. Interference detection techniques are used for speeding up the tests.

6.3 BSP tree culling

The method described in Hudson et al. [39] can be improved using BSP trees. Bittner et al. [8] combine the shadow frusta of the occluders into an *occlusion tree*. This is done in a very similar way to the SVBSP tree of Chin and Feiner [11]. The tree starts as a single *lit* (visible) leaf and occluders are inserted, in turn, into it. If an occluder reaches a *lit* leaf then it augments the tree with its shadow frustum; if it reaches a *shadowed* (invisible) leaf then it is just ignored since it means it already lies in an occluded region. Once the tree is built the scene hierarchy can be compared with it. The cube representing the top of the scene hierarchy is inserted into the tree. If it is found to be fully visible or fully occluded then we stop and act appropriately, otherwise its children are compared with the occlusion tree recursively. This method has an advantage over [39] in that instead of comparing the scene with each of the N shadow frusta, it is compared with one tree of depth (potentially) O(N).

The above technique is conservative; an alternative *exact* method was proposed much earlier by Naylor [53]. That involved a merging of the occlusion tree with the BSP tree representing the scene geometry.

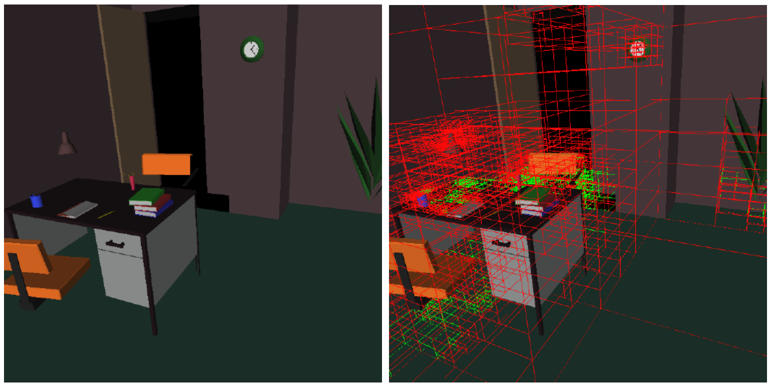
6.4 Prioritized-layered projection

Prioritized-Layered Projection (PLP) is a technique for fast rendering of high-depth complexity scenes. It works by *estimating* the visible polygons of a scene from a given viewpoint incrementally, one primitive at a time. On its own, PLP is not a conservative technique, but instead is suitable for the computation of partially correct images for use as part of time-critical rendering systems. At a very high level, PLP amounts to the modification of a simple view-frustum culling algorithm. However, it requires the computation of a special occupancy-based tessellation, and the assignment of a *solidity* value to each cell of the tessellation, which is used to compute a special ordering on how primitives get projected.

The core of the PLP algorithm consists of a space-traversal algorithm, which prioritizes the projection of the geometric primitives in such a way as to avoid (actually delay) projecting cells that have a small likelihood of being visible. Instead of explicitly overestimating, the algorithm works on a budget. At each frame, the user can provide the maximum number of primitives to be rendered, a polygon budget, and the algorithm will deliver what it considers to be the set of primitives which maximizes the image quality (using a solidity-based metric).

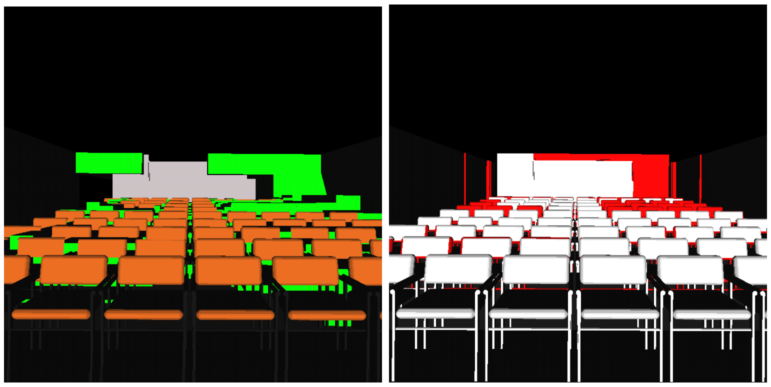
PLP is composed of two parts. First, PLP tessellates the space that contains the original input geometry with convex cells. During this one-time preprocessing, a collection of cells is generated in such a way as to roughly keep a uniform density of primitives per cell. The sampling leads to large cells in unpopulated areas, and small cells in areas that contain a lot of geometry. Using the number of modeling primitives assigned to a given cell (*e.g.*, tetrahedron), a *solidity* value ρ is defined. The accumulated solidity value used throughout the priority-driven traversal algorithm can be larger than one. The traversal algorithm prioritizes cells based on their solidity value. Preprocessing is fairly inexpensive, and can be done on large datasets (about one million triangles) in a couple of minutes.

The rendering algorithm traverses the cells in roughly front-to-back order. Starting from the seed cell, which in general contains the eye position, it keeps carving cells out of the tessellation. The basic idea of



(a) (b)

Figure 14: The Prioritized-Layered Projection Algorithm. PLP attempts to prioritize the rendering of geometry along layers of occlusion. Cells that have been projected by the PLP algorithm are highlighted in red wireframe and their associated geometry is rendered, while cells that have not been projected are shown in green. Notice that the cells occluded by the desk are outlined in green, indicating that they have not been projected.



(a) (b)

Figure 15: The input geometry is a model of an office. (a) snapshot of the PLP algorithm highlights the spatial tessellation used. The cells which have not been projected in the spatial tessellation are highlighted in green. (b) This figure illustrates the accuracy of PLP. Shown in red are the pixels which PLP misses. In white, we show the pixels PLP renders correctly.

the algorithm is to carve the tessellation along *layers of polygons.* We define the layering number $\zeta \in \aleph$ of a modeling primitive $\mathcal{P}$ in the following intuitive way. If we order each modeling primitive along each pixel by its positive (assume, without loss of generality, that $\mathcal{P}$ is in the view frustum) distance to the eye point, we define $\zeta(\mathcal{P})$ as the smallest rank of $\mathcal{P}$ over all the pixels to which it contributes. Clearly, $\zeta(\mathcal{P}) = 1$ if and only if $\mathcal{P}$ is visible. Finding rank 1 primitives is equivalent to solving the visibility problem. Instead of solving this difficult problem, the PLP algorithm uses simple heuristics. The traversal algorithm *attempts* to project the modeling primitives by layers, that is, all primitives of rank 1, then 2, and so on. We do this by always projecting the cell in the front $\mathcal{F}$ (we call *the front*, the collection of cells that are immediate candidates for projection) which is least likely to be occluded according to its solidity value. Initially, the front is empty, and as cells are inserted, we estimate its accumulated solidity value to reflect its position during the traversal. Every time a cell in the front is projected, all of the geometry assigned to it is rendered.

PLP is very effective in finding the visible polygons. For more details about PLP, including comprehensive results, see [43, 42].

7 Image-space occlusion culling

As the name suggests image-space algorithms perform the culling in the viewing coordinates. The key feature in these algorithms is that during rendering of the scene the image gets filled up and subsequent objects can be culled away quickly by the already-filled parts of the images. Since they operate on a discrete array of finite resolution they also tend to be simpler to implement and more robust than the object-space ones, which tend to have numerical precision problems.

Since testing each individual polygon against the image is too slow, almost all the algorithms that we will describe here, use conservative tests. They place a hierarchy on the scene, with the lowest level usually being the bounding boxes of individual objects, and they perform the occlusion test on that hierarchy. Approximate solutions can also be produced by some of the image-space algorithms by classifying as occluded geometry parts which are visible through an insignificant pixel count. This invariably results in an increase in running speed.

When the scenes are composed of many small primitives without well-defined large occluders then performing the culling in image-space becomes more attractive. The projections of many small and individually insignificant occluders can be accumulated on the image using standard graphics rasterizing hardware, to cover a significant part of the image which can then be used for culling. Another advantage of these methods is that the occluders do not have to be polyhedral; any object that can be rasterised can be used.

7.1 Hierarchical Z-buffer

The Hierarchical Z-buffer (HZB) [35, 36] is an extension of the popular HSR method, the Z-buffer. In this method, occlusion is determined by testing against the *Z-pyramid.* The Z-pyramid is a layered buffer with different resolution at each level. At the finest level it is just the content of the Z-buffer, each coarser level is created by halving the resolution in each dimension and each element holding the furthest Z-value in the corresponding 2x2 window of the finer level below. This is done all the way to the top, where it is just one value corresponding to the furthest Z-value in the buffer. During scan-conversion of the primitives, if the contents of the Z-buffer change then the new Z-values are propagated up the pyramid to the coarser levels.

In [35] the scene is arranged into an octree which is traversed top-down front-to-back and each node is tested for occlusion. If at any point a node is found to be occluded then it is skipped; otherwise any primitives associated with it are rendered and the Z-pyramid is updated. To determine whether a node is visible, each of its faces is tested hierarchically against the Z-pyramid. Starting from the coarsest level, the nearest Z value of the face is compared with the value in the Z-pyramid. If the face is found to be further away then it is occluded; otherwise it recursively descends down to finer levels until its visibility can be determined.

To allow for real-time performance, a modification of the hardware Z-buffer is suggested that allows for much of the culling processing to be done in the hardware. In the absence of the custom hardware the process can be somewhat accelerated through the use of temporal coherence, by first rendering the geometry that was visible from the previous frame and building the Z-pyramid from its Z-buffer.

7.2 Hierarchical occlusion map

The hierarchical occlusion map method [80] is similar in principle to the HZB, though, it was designed to work with current graphics hardware and also supports approximate visibility culling; objects that are visible through only a few pixels can be culled using an opacity threshold. The occlusion is arranged hierarchically in a structure called the *Hierarchical Occlusion Map* (HOM) and the bounding volume hierarchy of the scene is tested against it. However, unlike the HZB, the HOM stores only opacity information while the distance of the occluders (Z-values) is stored separately. The algorithm then needs to independently test objects for overlap with occluded regions of the HOM and for depth.

During preprocessing, a database of potential occluders is assembled. Then at run-time, for each frame, the algorithm performs two steps: construction of the HOM and occlusion culling of the scene geometry using the HOM.

To build the HOM, a set of occluders is selected from the occluder database and rendered into the frame-buffer. At this point only occupancy information is required; therefore texturing, lighting and Z-buffering are all turned off. The occluders are rendered as pure white on a black background. The result is read from the buffer and forms the highest resolution in the occlusion map hierarchy. The coarser levels are created by averaging squares of 2x2 pixels to form a map which has half the resolution on each dimension. Texturing hardware can provide some acceleration of the averaging if the size of the map is large enough to warrant the set-up cost of the hardware. As we proceed to coarser levels the pixels are not just black or white (occluded or visible) but can be shades of grey. The intensity of a pixel at such a level shows the opacity of the corresponding region.

An object is tested for occlusion by first projecting its bounding box onto the screen and finding the level in the hierarchy where the pixels have approximately the same size as the extent of the projected box. If the box overlaps pixels of the HOM which are not opaque, it means that the box cannot be culled. If the pixels are opaque (or have opacity above the specified threshold when approximate visibility is enabled) then the object is projected on a region of the image that is covered. In this case a depth test is needed to determine

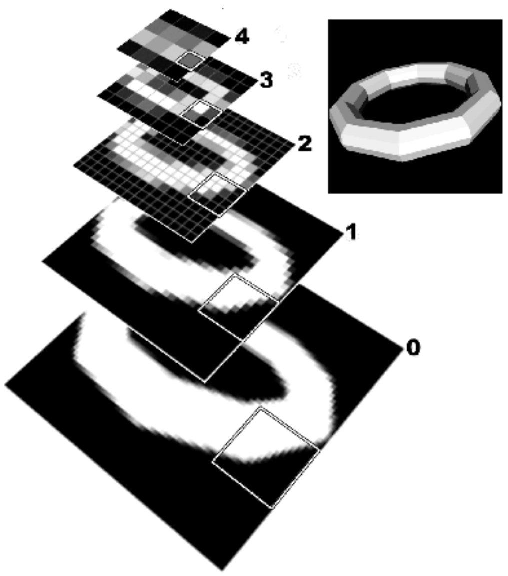


Figure 16: A hierarchy of occlusion maps created by recursively averaging blocks of pixels. Courtesy of Hansong Zhang, UNC.

whether the object is behind the occluders.

In paper [80] a number of methods are proposed for testing the depth of the objects against that of the occluders. The simplest test makes use of a plane placed behind all the occluders; any object that passes the opacity test is compared with this. Although this is fast and simple it can be over-conservative. An alternative is the *depth estimation buffer* where the screen space is partitioned into a set of regions and a separate plane is used for each region of the partition.

7.3 Directional discretized occluders

The Directional discretized occluders (DDOs) approach is similar to the HZB and HOM methods in that it also uses both object- and image-space hierarchies. In their preprocessing stage, Bernardini et al. [7] approximate the input model with an octree and compute simple, view-dependent polygonal occluders to replace the complex input geometry in subsequent visibility queries. Each face of every cell of the octree is regarded as a potential occluder and the solid angles spanning each of the two halfspaces on the two sides of the face are partitioned into regions. For each region, they compute and store a flag that records whether that face is a valid occluder for any viewpoint contained in that region. Each square, axis-aligned face is a view-dependent polygonal occluder that can be used in place of the original geometry in subsequent visibility queries.

The rendering algorithm visits the octree in a top-down, front-to-back order. Valid occluders found during the traversal are projected and added to a two-dimensional data structure, such as a quadtree. Each octree node is first tested against the current collection of projected occluders: if the node is not visible, traversal of its subtree stops. Otherwise, recursion continues and if a visible leaf node is reached, its geometry is rendered.

The DDO preprocessing stage is not inexpensive, and may take in the order of hours for models containing hundreds of thousands of polygons. However, the method does have several advantages if one can tolerate the cost of the preprocessing step. The computed occluders are all axis-aligned squares, a fact that can be exploited to design efficient data structures for visibility queries. The memory overhead of the DDOs is only six bitmasks per octree node. The DDO approach also benefits from *occluder fusion* and does not require any special or advanced graphics hardware. The approach could be used within the framework of other visibility culling methods as well. Culling methods which need to pre-select large occluders, (*e.g. Coorg and Teller* [18]), or which pre-render occluders to compute occlusion maps, (*e.g.* Zhang, *et Al.* [81]), could benefit from the DDO preprocessing step to reduce the overhead of visibility tests.

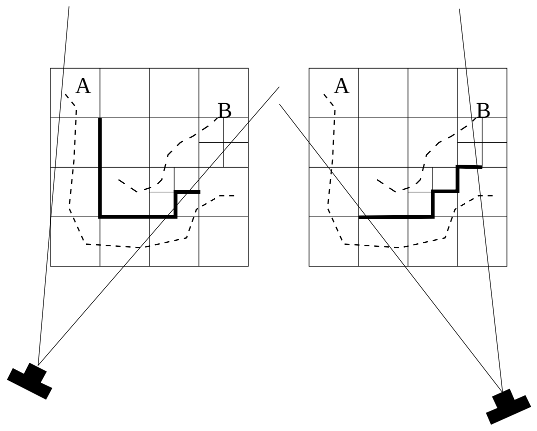


Figure 17: Illustration of the DDO approach. The input geometry, *A* and *B*, is drawn as dashed lines. The valid occluders for the two viewpoints are shown as thick solid lines. Courtesy of James Klosowski, IBM.

Figure 17 is a two-dimensional illustration of the DDO approach. The grid is a discretization of the space surrounding the scene; it represents our octree nodes. The input geometry, *A* and *B*, is shown using dashed lines. For the purpose of occlusion culling, the geometry *A* can be replaced by a simpler object (shown using thick solid lines) which is a subset of the grid edges, that is, the octree faces. The two figures show the same scene from different viewpoints and view directions. Note that the subset of grid edges that can act as occluders (in place of geometry *A*) changes as the viewpoint changes.

7.4 OpenGL-assisted occlusion culling

Bartz et al. in [6, 5] describe a different method of image-space culling. The scene is arranged in a hierarchical representation and tested against the occluded part of the image, which resembles the HZB and the HOM. However, in contrast to these methods, there is no hierarchical representation of the occlusion, rather OpenGL calls are used to query the hardware for visibility information. Both view-frustum and occlusion culling are done in that way.

For view-frustum culling the *OpenGL selection mode* is used. The selection mode can track a certain region of the screen and identify whether a given object is rendered onto it. By setting the tracked region to be the entire screen and rendering hierarchically the bounding volumes of the objects, it can quickly be decided on which to intersect the view volume. Of course the rendering of the bounding volumes here is purely for selecting the objects and does not contribute to the frame-buffer.

To test for occlusion, a separate buffer, the *virtual occlusion buffer*, is associated with the frame-buffer to detect the possible contribution of any object to the frame-buffer. This was implemented with a stencil buffer. The bounding boxes of the scene are hierarchically sent down the graphics pipeline. As they are rasterised, the corresponding pixels are set in the virtual occlusion buffer whenever the z-buffer test succeeds. The frame-buffer and the z-buffer remain unaltered throughout this process.

The virtual occlusion buffer is then read and any bounding box that has a footprint in it is considered to be (at least partially) visible and the primitives within it can be rendered. Since the operation of reading the virtual occlusion buffer can be very expensive, it was proposed to sample it by reading only spans from it. The sampling inevitably makes the algorithm a non-conservative test.

As in the methods above, approximate culling can be implemented if we allow boxes that have a small footprint in the occlusion buffer to be considered invisible. The performance of the algorithm depends on the hardware being used. In low- to mid-range graphics workstations where part of the rendering process is in software, the reduction in rendered objects can provide significant speed-ups. On high-end machines the set-up for reading the buffer becomes a more significant portion of the overall time, reducing the usefulness of the method.

7.5 Hardware assisted occlusion culling

Hardware vendors have started adopting occlusion-culling features into their designs. Greene et al. [35] report that the Kubota Pacific Titan 3000 was an early example of graphics hardware that supported occlusion-culling features.

A hardware feature available on HP machines (which seems quite similar to the Kubota Pacific Titan 3000) makes it possible to determine the visibility of objects as compared to the current values in the z-buffer. The idea is to add a feedback loop to the hardware which is able to check if changes will be made to the z-buffer when scan-converting a given primitive. One possible use of this hardware feature is to avoid rendering a very complex set model by first checking if it is potentially visible. In general this can be done with the HP occlusion-culling extension by checking whether an enveloping primitive (usually the bounding box of the object, but in general it might be more efficient to use an enclosing k-dop [41]) is visible, and only rendering the actual object if the simpler enclosing object is indeed visible.

The actual hardware feature as implemented on the HP fx series graphics accelerators is explained in [62] and [63]. One way to use the hardware is to query whether the bounding box of an object is visible. This can be done as follows:

```
glEnable(GL_OCCLUSION_TEST_HP);
glDepthMask(GL_FALSE);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
DrawBoundingBoxOfObject();
bool isVisible;
```

```
glGetBooleanv(GL_OCCLUSION_RESULT_HP, &isVisible);
glDisable(GL_OCCLUSION_TEST_HP);
glDepthMask(GL_TRUE);
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
```

Clearly, if the bounding box of an object is not visible, the object itself, which potentially could contain a large amount of geometry, must not be visible. This hardware feature is implemented in several of HP's graphics accelerators, for instance, the HP fx6 graphics accelerator. Severson [63] estimates that performing an occlusion-query with a bounding box of an object on the fx6 is equivalent to rendering about 190 25-pixel triangles. This indicates that a naive approach where objects are constantly checked for occlusion might actually hurt performance, and not achieve the full potential of the graphics board. In fact, it is possible to slow down the fx6 considerably if one is unlucky enough to project the polygons in a back-to-front order (because none of the primitives would be occluded).

Meissner et al. [49] propose an effective occlusion culling technique using this hardware test. In a preprocessing step, a hierarchical data structure is built which contains the input geometry. (In their paper, they propose several different data structures, and study their relative performance.) Their algorithm is as follows:

(1) traverse the hierarchical data structure to find the leaves which are inside the view frustum;

(2) sort the leaf cells by the distance between the viewpoint and their centroids;

(3) for each sorted cell, render the geometry contained in the cell **only** if the cell boundary is visible.

In their recent offerings, HP has improved the occlusion-culling features. The fx5 and fx10 hardware can perform several occlusion culling queries in parallel [19]. Also, HP reports that their OpenGL implementations have been changed to use the occlusion-culling features automatically when possible. For instance, before rendering a long display list, HP software would actually perform an occlusion query before rendering all the geometry.

ATI's HyperZ technology [51] is another example of a hardware-based occlusion-culling feature. HyperZ has three different optimizations which they claim greatly improve the performance of 3D applications. The main trust on all the optimizations is on lowering the memory bandwidth required for updating the Z-values (which they claim is the single largest user of bandwidth on their cards). One optimization is a technique for lossless compression of Z-values. Another is a "fast" Z-buffer clear, which performs a lazy clear of depth values. ATI also reports on the implementation of the hierarchical Z-buffer of Greene et al. [35] in hardware. Details of the actual features are sketchy, and at this point ATI has not exposed any of the functionality of their hardware to applications, that is, applications are blind, and should automatically get improved performance.

There are reports that other vendors, including SGI, Nvidia, and so on, are working on similar occlusion-culling features for their upcoming hardware.

7.6 Discussion

There are several other algorithms which are targeted at particular types of scenes. For example, the occluder shadows proposed by Wonka and Schmalstieg [75] specifically target urban environments. In this work the scene is partitioned in a regular 2D grid. During run-time a number of occluders are selected and their 'shadows' - the planes defined by the view-point and the top edge of each occluder - are rendered into an auxiliary buffer called the cull-map. Each pixel in the cull-map (image-space) corresponds to a grid cell of the scene grid (object-space). If the cull-map pixel is not covered then objects in the corresponding scene grid cell are potentially visible.

Hong et al. [38] use an image-based portal technique (similar in some respects to the cells- and-portals work of Luebke and Georges [48]) to be able to fly through a virtual human colon in real-time. The colon is partitioned into cells at preprocessing and these are used to accelerate the occlusion with the help of a Z-buffer at run-time.

One drawback of the techniques described in this section is that they rely on being able to read information from the graphics hardware. Unfortunately, on most current architectures, using any sort of feedback from the graphics hardware is quite slow and places a limit on the achievable frame rate of such techniques. As Bartz et al. [5] show, these methods are usually only effective when the scene complexity is above a large threshold.

There are other shortcomings to these techniques. One of the main problems is the need for preselection of occluders. Some techniques, such as HOM, need to create different versions of the actual objects (through some sort of "occlusion-preserving" simplification algorithm) to be able to generate the occlusion-maps. Another interesting issue is how to deal with dynamic scenes. The more preprocessing used, the more expensive it is to deal with dynamic environments.

The BSP tree method introduced by Naylor [53] already in 1992 can be thought of as somewhere between image-precision and object-precision, since although he used a 2D BSP tree in image-space for culling the 3D scene, this was done using object-precision operations rather than image-precision.

8 From-region visibility

In a typical visibility culling algorithm the occlusion is tested from a point [18, 39]. Thus, these algorithms are applied in each frame during the interactive walkthrough. A promising alternative is to find the PVS from a region or viewcell, rather than from a point. The computation cost of the PVS from a viewcell would then be amortized over all the frames generated from the given viewcell.

Effective methods have been developed for indoor scenes [70, 30], but for general arbitrary scenes, the computation of the visibility set from a region is more involved than from a point. Sampling the visibility from a number of view points within the region [33] yields an approximated PVS, which may then cause unacceptable flickering artifacts during the walkthrough. Conservative methods were introduced in [15, 59] which are based on the occlusion of individual large convex objects.

In these methods a given object or collection of objects is culled away if and only if they are fully occluded by a single convex occluder. It was shown that a convex occluder is effective only if it is larger

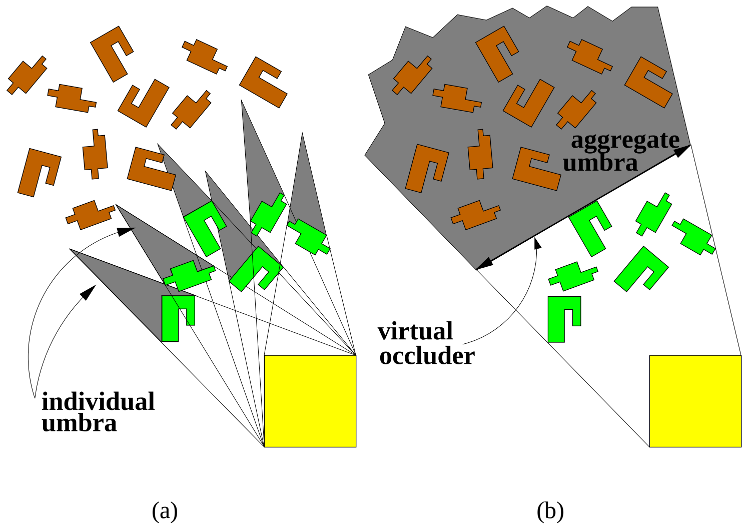


Figure 18: The union of the umbrae of the individual objects is insignificant, while their aggregate umbra is large and can be represented by a single virtual occluder.

than the viewcell [52]. However, this condition is rarely met in real applications. For example, the objects in Figure 18 are smaller than the viewcell, and their umbrae (with respect to the viewcell) are rather small. Their union does not occlude a significant portion of the scene (see in (a)), while their aggregate umbra is large (see in (b)).

Recently, new techniques were developed in which the visibility culling from a region is based on the combined occlusion of a collection of objects (occluder fusion). The collection or cluster of objects that contributes to the aggregate occlusion has to be neither connected nor convex. The effective from-region culling of these techniques is significantly larger than previous from-region visibility methods. Below, four techniques are described followed by a discussion.

8.1 Conservative volumetric visibility with occluder Fusion

Schaufler et al. [60] introduce a conservative technique for the computation of viewcell visibility. The method operates on a discrete representation of space and uses the opaque interior of objects as occluders. This choice of occluders facilitates their extension into adjacent opaque regions of space, in essence, maximizing their size and impact.

The method efficiently detects and represents the regions of space hidden by occluders and is the first to use the property that occluders can also be extended into empty space provided this space itself is occluded from the viewcell. This is proved to be effective for computing the occlusion by a set of occluders, successfully realizing occluder fusion.

Initially, the boundary of objects is rasterized into the discretization of space and the interior of these boundaries is filled with opaque voxels. For each viewcell, the occlusion detection algorithm iterates over

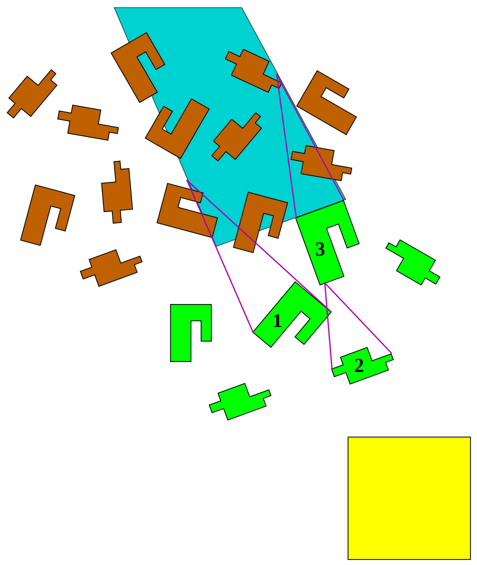


Figure 19: The individual umbrae (with respect to the yellow viewcell) of objects 1, 2 and 3 do not intersect, but yet their occlusion can be aggregated into a larger umbra.)

these opaque voxels, and groups them with adjacent opaque voxels into effective blockers. Subsequently, a shaft is constructed around the viewcell and the blocker to delimit the region of space hidden by the blocker. This classification of regions of space into visible and hidden is noted in the spatial data structure. As regions of space have already been found to be hidden from the viewcell, extension of blockers into neighboring voxels can also proceed into these hidden regions realizing occluder fusion with all the occluders which caused this region to be hidden.

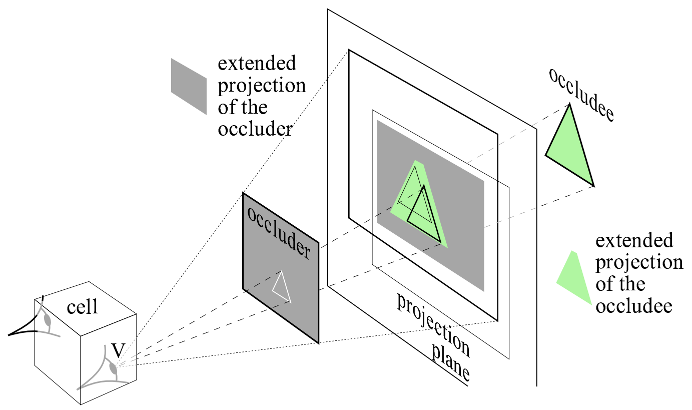
As an optimization, opaque voxels are used in the order from large to small and from front to back. Occluded opaque voxels are not considered further as blockers.

To recover the visibility status of objects in the original scene description, the space they occupy is looked up in the spatial data structure and, if all the voxels intersected by the object are classified as hidden, the object is guaranteed to be hidden as well.

The authors present specialized versions for the cases of 2D and 2 1/2D visibility, and motivate the ease of extension to 3D: because only two convex objects at a time are considered in the visibility classification (the viewcell and the occluder), the usual difficulties of extending visibility algorithms from 2D to 3D, caused by triple-edge events, are avoided. Example applications described in the paper include visibility preprocessing for real-time walkthroughs and reduction in the number of shadow rays required by a ray-tracer (see [60] for details).

8.2 Conservative visibility preprocessing using extended projections

Durand et al. [25] (see also [47]) present an extension of point-based image-space methods such as the Hierarchical Occlusion Maps [81] or the Hierarchical Z-buffer [35] to volumetric visibility from a view-cell, in the context of preprocessing PVS computation. Occluders and occludees are projected onto a plane,


extended projection of the occluder
occludee
occluder
extended projection of the occludee
cell
V
projection plane
(a)

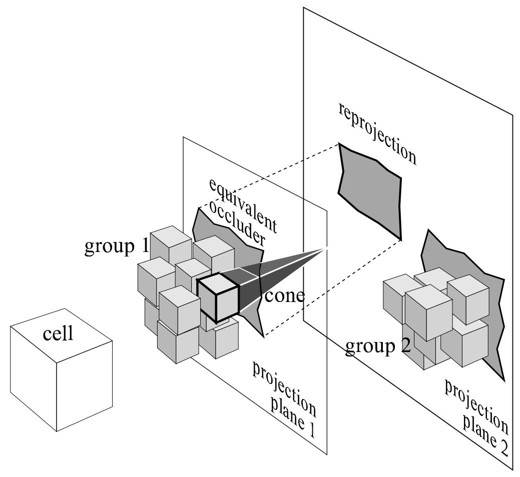


Figure 20: (a) Principle of Extended Projections. The Extended Projection of the occluder is the intersection of its projections from all the points in the viewing cell, while the Extended Projection of the occludee is the union of its projections. (b) If plane 2 is used for projection, the occlusion of group 1 is not taken into account. The shadow cone of the cube shows that its Extended Projection would be void, since it vanishes in front of the plane. The same constraint applies for group 2 and plane 1. We thus project group 1 onto plane 1, then reproject this aggregate projection onto plane 2. Courtesy of Fredo Durand, MIT.

and an occludee is declared hidden if its projection is completely covered by the cumulative projection of occluders (and if it lies behind). The projection is however more involved in the case of volumetric visibility: to ensure conservativeness, the Extended Projection of an occluder underestimates its projection from any point in the view-cell, while the Extended Projection of an occludee is an overestimation (see Figure 20(a)). A discrete (but conservative) pixel-based representation of extended projections is used, called an *extended depth map*. Extended projections of multiple occluders aggregate, allowing occluder-fusion, that is, the cumulative occlusion caused by multiple occluders. For convex view-cells, the extended projection of a convex occluder is the intersection of its projections from the vertices of the cell. This can be computed efficiently using the graphics hardware (stencil buffer) and a conservative rasterization. Concave occluders intersecting the projection plane are sliced (see [25] for details).

A single set of six projection planes can be used, as demonstrated by an example involving a city database. The position of the projection plane is however crucial for the effectiveness of Extended Projections. This is why a reprojection operator was developed for hard-to-treat cases. It permits a group of occluders to be projected onto one plane where they aggregate, and then reproject this aggregated representation onto a new projection plane (see Figure 20(b)). This re-projection is used to define an occlusion-sweep where the scene is swept by parallel planes leaving the cell. The cumulative occlusion obtained on the current plane is reprojected onto the next plane as well as new occluders. This allows the handling of very different cases such as the occlusion caused by leaves in a forest.

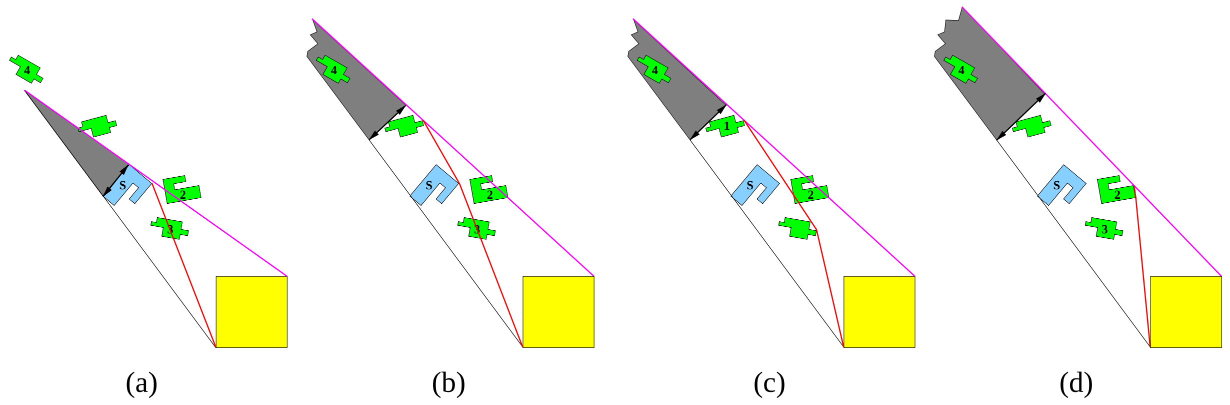


Figure 21: Growing the virtual occluders by intersecting objects with the active separating and supporting lines.

8.3 Virtual occluders

Koltun et al. [44] introduce the notion of from-region virtual occluders. Given a scene and a viewcell, a virtual occluder is a view-dependent (simple) convex object, which is guaranteed to be fully occluded from any given point within the viewcell and which serves as an effective occluder from the given viewcell. Virtual occluders compactly represent the aggregate occlusion for a given cell. The introduction of such view-dependent virtual occluders enables to apply an effective region-to-region or cell-to-cell culling technique and to efficiently compute a potential visibility set (PVS) from a region/cell. The paper presents an object-space technique that synthesizes such virtual occluders by aggregating the visibility of a set of individual occluders. It is shown that only a small set of virtual occluders is required to compute the PVS efficiently on-the-fly during the real-time walkthrough.

In the preprocessing stage several objects are identified as seed objects. For each seed object, a cluster of nearby objects is constructed so that a single virtual occluder faithfully represents the occlusion of this cluster of objects. At first, the cluster is defined to include only the seed object. Then, iteratively, at each step, more objects which satisfy a geometric criterion are added to the cluster of occluders, thus augmenting the aggregate umbra of the cluster. The virtual occluder is placed just behind the furthest object in the cluster, and is completely contained in the aggregate umbra of the cluster (see Figs. 18 and 21).

One virtual occluder is stored at each step of the iteration. As a result, at the end of the process, there is a large and highly redundant group of virtual occluders. This group can be well represented by a small subset of the most effective virtual occluders.

In the real-time rendering stage, the PVS of a viewcell is computed just before the walkthrough enters the viewcell. It is done by hierarchically testing the scene-graph nodes against the virtual occluders. Since only a very small number of them are used, this test is extremely fast.

The 3D problem is solved by a 2.5D implementation, which proves to be effective for most typical scenes, such as urban and architectural walkthroughs. The 2.5D implementation performs a series of slices in the height dimension, and uses the 2D algorithm to construct 2D virtual occluders in each slice. These occluders are then extended to 3D by giving them the height of their respective slices.

8.4 Occluder fusion for urban walkthroughs

Wonka et al. [76] present an approach based on the observation that it is possible to compute a conservative approximation of the umbra for a viewcell from a set of discrete point samples placed on the viewcell's boundary. A necessary, though not sufficient condition that an object is occluded is that it is completely contained in the intersection of all sample points' umbrae. Obviously, this condition is not sufficient as there may be viewing positions between the sample points where the considered object is visible.

However, shrinking an occluder by ε provides a smaller umbra with a unique property: an object classified as occluded by the shrunk occluder will remain occluded with respect to the original larger occluder when moving the viewpoint no more than ε from its original position.

Consequently, a point sample used together with a shrunk occluder is a conservative approximation for a small view cell with radius ε centered at the sample point. If the original view cell is covered with sample points so that every point on the boundary is contained in an ε -neighborhood of at least one sample point, then an object lying in the intersection of the umbrae from all sample points is occluded for the original viewcell.

Using this idea, multiple occluders can be considered simultaneously. If the object is occluded by the joint umbra of the shrunk occluders for every sample point of the viewcell, it is occluded for the whole view cell. In that way, occluder fusion for an arbitrary number of occluders is implicitly performed (see Figure 22 and Figure 23).

8.5 Discussion

When the visibility from a region is concerned, occlusion caused by individual occluders in a general setting is insignificant. Thus, it is essential to take advantage of aggregate occlusion caused by groups of nearby objects. The above four papers address the problem of occlusion aggregation also referred to as occluder fusion.

All four techniques are conservative; they aggregate occlusion in most cases, but not in all possible ones. In some techniques, the criterion to fuse two occluders or to aggregate their occlusions is based on the intersection of two umbrae. However, in [44, 77], a more elaborate criterion is used, which permits aggregation of occlusions even in cases where the umbrae are not necessarily intersected. These cases are illustrated in Figure 19.

To cope with the complexity of the visibility in 3D scenes, all the techniques use some discretizations.

The first method discretizes the space into voxels, and operates only on voxels. This leads to the underestimation of occlusion when the umbra of occluders is relatively small and partially overlaps some large voxels, but does not completely contain any. The advantage of this approach is its generality: it can be applied to any representation of 3D scenes, and not necessarily polygonal.

The second method discretizes the space in two ways. First, it projects all objects onto a discrete set of projection planes, and second, the representation of objects in those planes is also discrete. Moreover, 3D projections are replaced by two 2D projections (see Figure 20), to avoid performing analytical operations on objects in 3D space. The advantage of this algorithm is that, since most operations are performed in

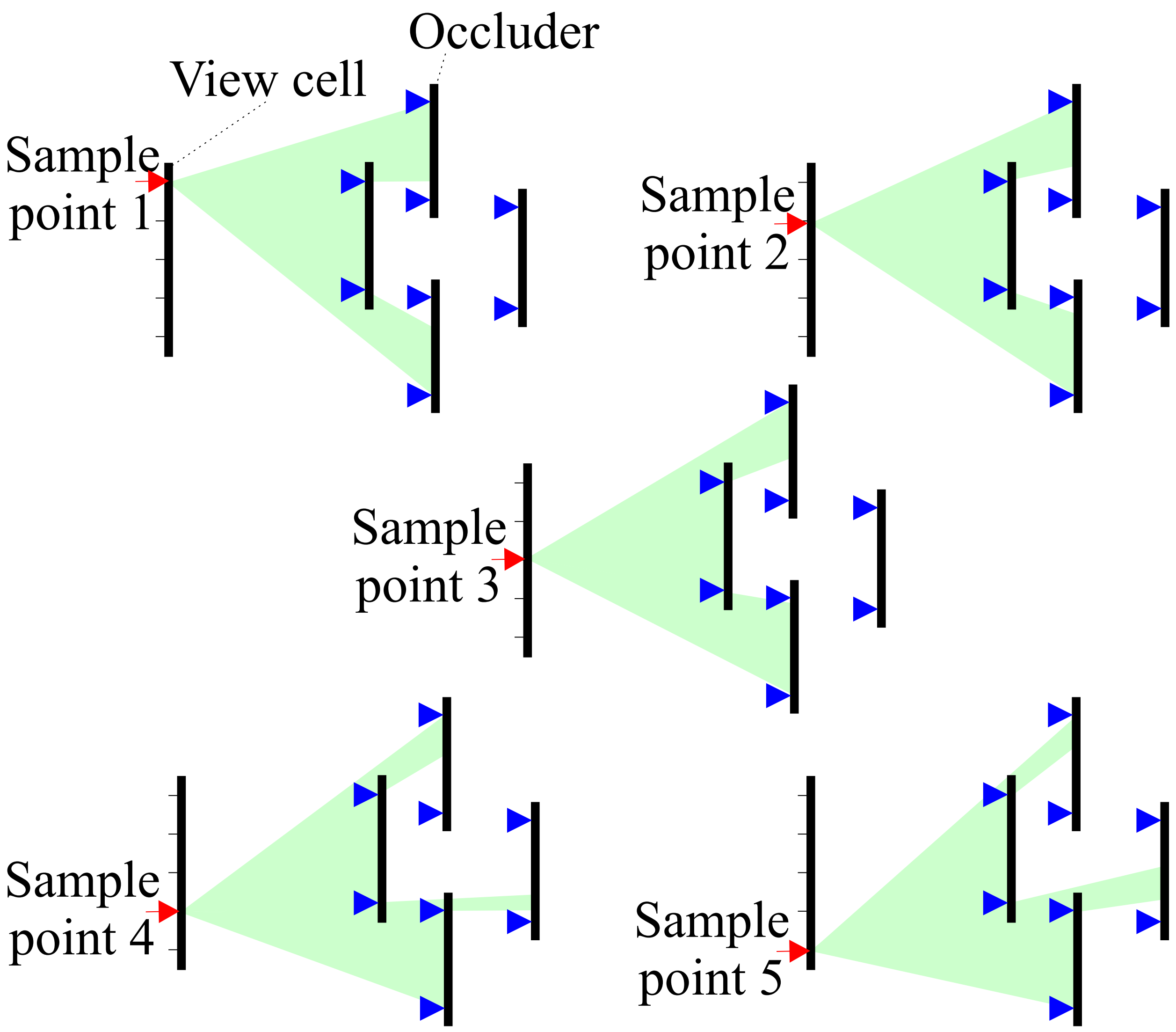


Figure 22: Sampling of the occlusion from five sampling points. Courtesy of Peter Wonka, Vienna University of Technology.

image-space, they can be hardware-assisted to shorten the preprocessing time.

The third method is object-space analytical in the 2D case. It treats the 3D cases as a 2.5D scene and solves it by a series of 2D cases by discretizing the height dimension. It is shown that in practice the visibility of 2.5D entities approximate well the visibility of the original 3D models.

The forth method samples the visibility from a viewcell from a discrete number of sample points. Although it underestimates occlusion, it is also a conservative method. This may be insignificant in the case of close and large occluders, but in cases where the occlusion is created by a large number of small occluders, the approximation might be too crude.

Something that could prove useful when computing visibility from a region is a method for depth-ordering objects with respect to the region. Finding such an ordering can be a challenging task, if at all possible, since it might vary at different sample points in the given region. Chrysanthou in [12] (Section 3.2) suggests a hybrid method based on Graph Theory and BSP trees which will sort a set of polygons as far as possible and report unbreakable cycles where they are found.

8.6 Approximate from-region visibility

In [2] a scheme to combine approximate occlusion culling with levels-of-detail (LOD) techniques is presented. The idea is to identify partially-occluded objects in addition to fully-occluded ones. The assumption is that partially-occluded objects take less space on the screen, and therefore can be rendered using a lower LOD. The authors use the term *Hardly-Visible Set* (HVS) to describe a set consisting of both fully and partially visible objects.

A set of occluders is selected and simplified to a collection of Partially-overlapping boxes. Occlusion culling is performed from the viewcell using these boxes as occluders to find the ”fully-visible” part of the HVS. It is performed considering only occlusion by individual boxes [15, 59]. There is no occlusion fusion, but a single box may represent several connected occluder objects.

To compute partially-visible objects, all the occluders (boxes) are enlarged by a certain small degree, and occlusion culling is performed again using these magnified occluders. The objects that are occluded by the enlarged occluders and not by the original ones are considered to be partially occluded from the viewcell, and are thus candidates to be rendered at a lower LOD.

Several parts of the HVS are computed by enlarging the occluders several times, each time by a different degree, thus, classifying objects with a different degree of visibility. During real-time rendering, the LOD is selected with respect to the degree of visibility of the objects.

It should be noted that this basic assumption of the degree of visibility is solely heuristic, since an object partially occluded from a region does not mean it is partially occluded from any point within the region. It could be fully visible at one point and partially visible or occluded at another.

In [33] another approximate from-region visibility technique is proposed. Casting rays from a five-dimensional space samples the visibility. The paper discusses how to minimize the number of rays cast to achieve a reliable estimate of the visibility from a region.

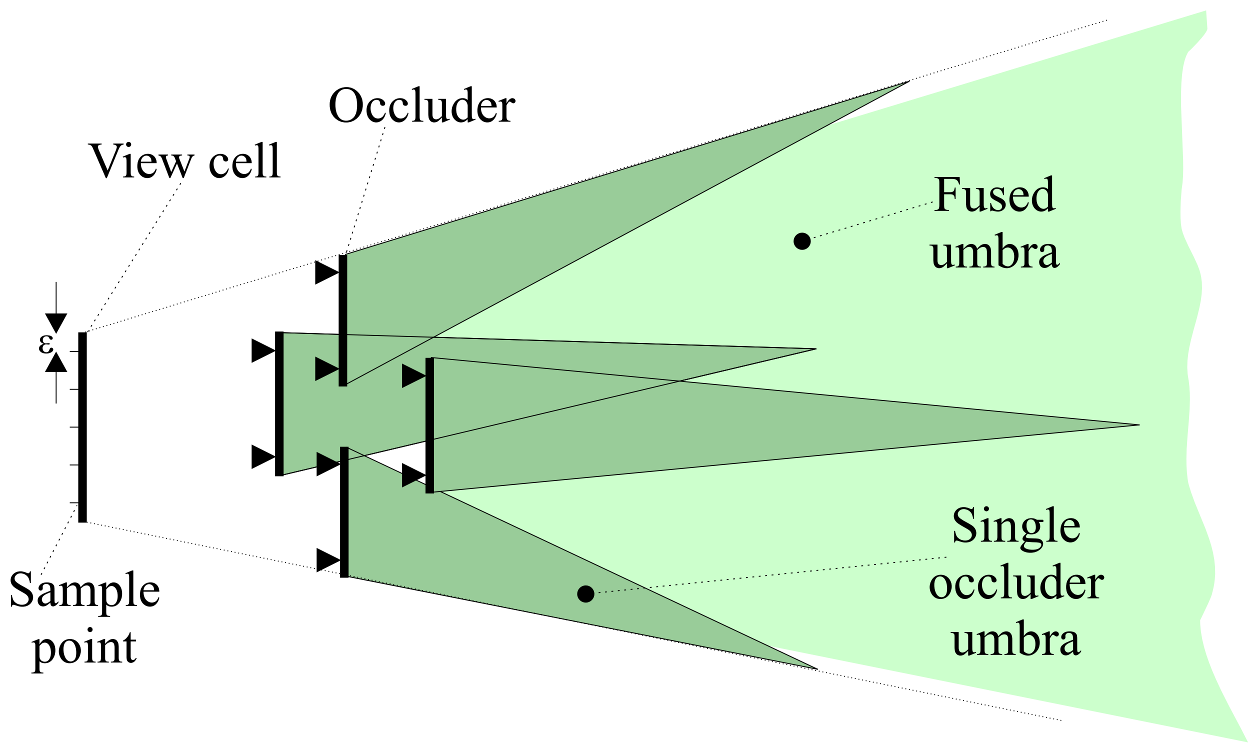


Figure 23: The fused umbra from the five points (in the figure above) is the intersection of the individual umbrae. It is larger than the union of umbrae of the original viewcell. Courtesy of Peter Wonka.

8.7 The PVS storage space problem

Precomputing the PVS from a region requires solving a prominent space problem. The scene is partitioned into viewcells and for each cell a PVS is precomputed and stored readily for the online rendering stage. Since the number of viewcells is inherently large, the total size of all the visibility sets is much larger than the original size of the scene. Aside for a few exceptions this problem has not received enough attention yet. Van de Panne and Stewart [72] present a technique to compress precomputed visibility sets by clustering objects and viewcells of similar behavior. Gotsman et al. [33] present a hierarchical scheme to encode the visibility efficiently. Cohen-Or et al. [16, 15] deal with the transmission of visibility sets from the server to the client and in [15, 52] discuss the selection of the best viewcell size in terms of the size of the PVS.

A completely different approach was taken by Koltun et al. [44]. The PVS of each viewcell does not need to be stored explicitly. An intermediate representation that requires much less storage space than the PVS is created and used to generate the PVS on-the-fly during rendering.

9 Conclusion

In this paper, we have surveyed most of the visibility literature available in the context of walkthrough applications. We see that a considerable amount of knowledge has been assembled in the last decade; in particular, the number of papers in the area has increased substantially in the last couple of years. It is hard to say exactly where the field is heading, but there are some interesting trends and open problems.

It seems further research is necessary into techniques which lower the amount of preprocessing required. Also, memory is a big issue for large scenes, especially in the context of from-region techniques.

It is expected that more hardware features which can be used to improve visibility computations will be available. At present, a major impediment is the fact that reading back information from the graphics boards is very slow. It is expected that this will get much faster, enabling improvements in hardware-based visibility culling algorithms. The efficient handling of dynamic scenes is an open area of research at this point.

Acknowledgements

We would like to thank all the authors who have helped us with material from their work. In particular, we thank Fredo Durand, Vladlen Koltun, Pere Brunet, Seth Teller, James Klosowski, Graig Gotsman, Gernot Schaufler and Peter Wonka, for helping us with text and figures from their papers, and Joseph Mitchell for useful comments on this survey.

This research was partially supported by the Israel Science Foundation founded by The Israel Academy of Sciences and Humasnities-Center of Excellence Program, and by a grant of the Israeli Ministry of Science.

References

[1] J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):41–50, March 1990.

[2] C. Andujar, C. Saona-Vazquez, I. Navazo, and P. Brunet. Integrating occlusion culling and levels of details through hardly-visible sets. *Computer Graphics Forum*, 19(3), 2000.

[3] A. Appel. Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, volume 32, pages 37–45, 1968.

[4] U. Assarsson and T. Moller. Optimized view frustum culling algorithms for bounding boxes. *Journal of Graphics Tools*, 5(1), 2000.

[5] D. Bartz, M. Meiner, and T. Httner. Opengl-assisted occlusion culling for large polygonal models. *Computer & Graphics*, 23(5):667–679, 1999.

[6] D. Bartz, M. Messner, and T. Httner. Extending graphics hardware for occlusion queries in opengl. In *Proc. Workshop on Graphics Hardware '98*, pages 97–104, 1998.

[7] F. Bernardini, J. T. Klosowski, and J. El-Sana. Directional discretized occluders for accelerated occlusion culling. *Computer Graphics Forum*, 19(3), 2000.

[8] J. Bittner, V. Havran, and P. Slavik. Hierarchical visibility culling with occlusion trees. In *Proceedings of Computer Graphics International '98*, pages 207–219, June 1998.

[9] W. Jack Bouknight. A procedure for generation of three-dimensional half-toned computer graphics presentations. *Communications of the ACM*, 13(9):527–536, September 1970.

[10] E. E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces.* Ph.d. thesis, University of Utah, December 1974.

[11] N. Chin and S. Feiner. Near real-time shadow generation using BSP trees. *ACM Computer Graphics*, 23(3):99–106, 1989.

[12] Y. Chrysanthou. *Shadow Computation for 3D Interaction and Animation.* PhD thesis, Queen Mary and Westfield College, University of London, February 1996.

[13] J. H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, October 1976.

[14] D. Cohen-Or, Y. Chrysanthou, C. Silva, and G. Drettakis. Visibility, problems, techniques and applications. SIGGRAPH 2000 Course Notes, July 2000.

[15] D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum*, 17(3):243–254, 1998.

[16] D. Cohen-Or and E. Zadicario. Visibility streaming for network-based walkthroughs. *Graphics Interface '98*, pages 1–7, June 1998.

[17] S. Coorg and S. Teller. Temporally coherent conservative visibility. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 78–87, 1996.

[18] S. Coorg and S. Teller. Real-time occlusion culling for models with large occluders. *1997 Symposium on Interactive 3D Graphics*, pages 83–90, April 1997.

[19] R. Cunniff. Visualize fx graphics scalable architecture. In *presentation at Hot3D Proceedings, part of Graphics Hardware Workshop*, 2000.

[20] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.

[21] D. P. Dobkin and S. Teller. Computer graphics. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 42, pages 779–796. CRC Press LLC, Boca Raton, FL, 1997.

[22] S. E. Dorward. A survey of object-space hidden surface removal. *Internat. J. Comput. Geom. Appl.*, 4:325–362, 1994.

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

[24] F. Durand, G. Drettakis, and C. Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 89–100. ACM SIGGRAPH, Addison Wesley, August 1997.

[25] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative visibility preprocessing using extended projections. *Proceedings of SIGGRAPH 2000*, pages 239–248, July 2000.

[26] D. Eggert, K. Bowyer, and C. R. Dyer. Aspect graphs: State-of-the-art and applications in digital photogrammetry. In *Proc. ISPRS 17th Cong.: Int. Archives Photogrammetry Remote Sensing*, pages 633–645, 1992.

[27] S. Fleishman, D. Cohen-Or, and D. Lischinski. Automatic camera placement for image-based modeling. In *Proceedings of Pacific Graphics 99*, pages 12–20, October 1999.

[28] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990. Overview of research to date.

[29] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. *ACM Computer Graphics*, 14(3):124–133, 1980.

[30] T. A. Funkhouser. Database management for interactive display of large architectural models. *Graphics Interface*, pages 1–8, May 1996.

[31] J. Goldfeather, J. P. M. Hultquist, and H. Fuchs. Fast constructive-solid geometry display in the Pixel-Powers graphics system. *Computer Graphics (SIGGRAPH ’86 Proceedings)*, volume 20, pages 107–116, August 1986.

[32] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (SIGGRAPH ’84 Proceedings)*, volume 18, pages 212–22, July 1984.

[33] C. Gotsman, O. Sudarsky, and J. Fayman. Optimized occlusion culling. *Computer & Graphics*, 23(5):645–654, 1999.

[34] N. Greene. Hierarchical polygon tiling with coverage masks. *SIGGRAPH 96 Conference Proceedings*, Annual Conference Series, pages 65–74. ACM SIGGRAPH, Addison Wesley, August 1996.

[35] N. Greene, M. Kass, and G. Miller. Hierarchical z-buffer visibility. *Proceedings of SIGGRAPH 93*, pages 231–240, 1993.

[36] N. Greene and M. Kass. Error-bounded antialiased rendering of complex environments. *Proceedings of SIGGRAPH ’94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 59–66. ACM SIGGRAPH, ACM Press, July 1994.

[37] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. In Thomas W. Sederberg, editor, *ACM Computer Graphics*, volume 25, pages 197–206, July 1991.

[38] L. Hong, S. Muraki, A. Kaufman, D. Bartz, and T. He. Virtual voyage: Interactive navigation in the human colon. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 27–34. ACM SIGGRAPH, Addison Wesley, August 1997.

[39] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frustra. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 1–10, 1997.

[40] J. T. Klosowski and C. T. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. Technical report. submitted for publication, 2000.

[41] J. T. Klosowski, M. Held, Joseph S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January-March 1998.

[42] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):108–123, April - June 2000. ISSN 1077-2626.

[43] J. T. Klosowski and C. T. Silva. Rendering on a budget: A framework for time-critical rendering. *IEEE Visualization '99*, pages 115–122, October 1999.

[44] V. Koltun, Y. Chrysanthou, and D. Cohen-Or. Virtual occluders: An efficient intermediate pvs representation. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 59–70, June 2000. ISBN 3-211-83535-0.

[45] S. Kumar, D. Manocha, W. Garrett, and M. Lin. Hierarchical back-face computation. *Computers and Graphics*, 23(5):681–692, October 1999.

[46] F.-A. Law and T.-S. Tan. Preprocessing occlusion for real-time selective refinement (color plate S. 221). In Stephen N. Spencer, editor, *Proceedings of the Conference on the 1999 Symposium on interactive 3D Graphics*, pages 47–54, New York, April 26–28 1999. ACM Press.

[47] H. L. Lim. Toward a fuzzy hidden surface algorithm. In *Computer Graphics International*, Tokyo, 1992.

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

[49] M. Meissner, D. Bartz, T. Huttner, G. Muller, and J. Einighammer. Generation of subdivision hierarchies for efficient occlusion culling of large polygonal models. *Computer & Graphics*, To appear.

[50] T. Moeller and E. Haines. *Real-Time Rendering.* A.K. Peters Ltd., 1999.

[51] S. Morein. Ati radeon hyper-z technology. In *presentation at Hot3D Proceedings, part of Graphics Hardware Workshop*, 2000.

[52] B. Nadler, G. Fibich, S. Lev-Yehudi, and D. Cohen-Or. A qualitative and quantitative visibility analysis in urban scenes. *Computer & Graphics*, 23(5):655–666, 1999.

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

[54] M. E. Newell, R. G. Newell, and T. L. Sancha. A solution to the hidden surface problem. *Proc. ACM Nat. Mtg.*, 1972.

[55] J. O'Rourke. *Art Gallery Theorems and Algorithms.* The International Series of Monographs on Computer Science. Oxford University Press, New York, NY, 1987.

[56] M. S. Peercy, M. Olano, J. Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. *Proceedings of SIGGRAPH 2000*, pages 425–432, July 2000.

[57] H. Plantinga and C. R. Dyer. Visibility, occlusion, and the aspect graph. *Internat. J. Comput. Vision*, 5(2):137–160, 1990.

[58] H. Plantinga. Conservative visibility preprocessing for efficient walkthroughs of 3D scenes. In *Proceedings of Graphics Interface '93*, pages 166–173, Toronto, Ontario, Canada, May 1993. Canadian Information Processing Society.

[59] C. Saona-Vazquez, I. Navazo, and P. Brunet. The visibility octree: A data structure for 3d navigation. *Computer & Graphics*, 23(5):635–644, 1999.

[60] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion. Conservative volumetric visibility with occluder fusion. *Proceedings of SIGGRAPH 2000*, pages 229–238, July 2000.

[61] R. Schumacker, B. Brand, M. Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL-TR-69-14, NTIS AD700375, U.S. Air Force Human Resources Lab., Air Force Systems Command, Brooks AFB, TX,, September 1969.

[62] N. Scott, D. Olsen, and E. Gannet. An overview of the visualize fx graphics accelerator hardware. *The Hewlett-Packard Journal*, May:28–34, 1998.

[63] K. Severson. VISUALIZE Workstation Graphics for Windows NT. HP product literature.

[64] F. Sillion and G. Drettakis. Feature-based control of visibility error: A multi-resolution clustering algorithm for global illumination. In Robert Cook, editor, *ACM Computer Graphics*, Annual Conference Series, pages 145–152. ACM SIGGRAPH, Addison Wesley, August 1995. held in Los Angeles, California, 06-11 August 1995.

[65] M. Slater and Y. Chrysanthou. View volume culling using a probabilistic cashing scheme. In S. Wilbur and M. Bergamasco, editors, *Proceedings of Framework for Immersive Virtual Environments FIVE*, December 1996.

[66] W. Stuerzlinger. Imaging all visible surfaces. *Graphics Interface '99*, pages 115–122, June 1999.

[67] B. Smits, J. Arvo, and D. Greenberg. A clustering algorithm for radiosity in complex environments. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 435–442. ACM SIGGRAPH, ACM Press, July 1994.

[68] O. Sudarsky and C. Gotsman. Dynamic scene occlusion culling. *IEEE Transactions on Visualization and Computer Graphics*, 5(1):13–29, January - March 1999.

[69] I. E. Sutherland, R. F. Sproull, and R. A. Schumaker. A characterization of ten hidden surface algorithms. *ACM Computer Surveys*, 6(1):1–55, March 1974.

[70] S. J. Teller and C. H. Sequin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):61–69, July 1991.

[71] C. Trendall and A. James Stewart. General calculations using graphics hardware with applications to interactive caustics. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 287–298, June 2000.

[72] M. van de Panne and J. Stewart. Efficient compression techniques for precomputed visibility. In *Proceedings of Eurographics Workshop on Rendering '99*, 1999.

[73] G. S. Watkins. A real-time visible surface algorithm. Technical Report UTECH-CSc-70-101, University of Utah, Salt Lake City, Utah, 1970.

[74] K. Weiler and K. Atherton. Hidden surface removal using polygon area sorting. *ACM Computer Graphics*, 11(2):214–222, July 1977.

[75] P. Wonka and D. Schmalstieg. Occluder shadows for fast wakthroughs of urban environments. In Hans-Peter Seidel and Sabine Coquillart, editors, *Computer Graphics Forum*, volume 18, pages C51–C60. Eurographics Association and Blackwell Publishers Ltd 1999, 1999.

[76] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 71–82, June 2000. ISBN 3-211-83535-0.

[77] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. Technical Report TR-186-2-00-06, Institute of Computer Graphics, Vienna University of Technology, Karlsplatz 13/186, A-1040 Vienna, Austria, March 2000. human contact: technical-report@cg.tuwien.ac.at.

[78] A. Woo, P. Poulin, and A. Fourier. A survey of shadow algorithms. *IEEE Computer Graphics and Applications*, 10(6):13–31, 1990.

[79] H. Zhang, D. Manocha, T. Hudson, and K. Hoff. Visibility culling using hierarchical occlusion maps. In *Computer Graphics (Proceedings of SIGGRAPH 97*, pages 77–88, 1997.

[80] H. Zhang. *Effective Occlusion Culling for the Interactive Display of Arbitrary Models*. Ph.D. thesis, Department of Computer Science, UNC-Chapel Hill, 1998.

[81] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. Visibility culling using hierarchical occlusion maps. In Turner Whitted, editor, *SIGGRAPH 97 Conference Proceedings*, Annual Conference Series, pages 77–88. ACM SIGGRAPH, Addison Wesley, August 1997.