Conservative Volumetric Visibility with Occluder Fusion

Gernot Schaufler   Julie Dorsey
Laboratory for Computer Science
Massachusetts Institute of Technology

Xavier Decoret   François X. Sillion
iMAGIS
GRAVIR/IMAG — INRIA

Abstract

Visibility determination is a key requirement in a wide range of graphics algorithms. This paper introduces a new approach to the computation of *volume visibility*, the detection of occluded portions of space as seen from a given region. The method is conservative and classifies regions as occluded only when they are guaranteed to be invisible. It 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. Our method efficiently detects and represents the regions of space hidden by such occluders. It is the first one to use the property that occluders can also be extended into empty space provided this space is itself occluded from the viewing volume. This proves extremely effective for computing the occlusion by a set of occluders, effectively realizing *occluder fusion*. An auxiliary data structure represents occlusion in the scene and can then be queried to answer volume visibility questions. We demonstrate the applicability to visibility preprocessing for real-time walkthroughs and to shadow-ray acceleration for extended light sources in ray tracing, with significant acceleration in both cases.

1 Introduction

Determining visibility is central in many computer graphics algorithms. If visibility information were available in advance, scanline renderers would not need to rasterize hidden geometry, and ray-tracers could avoid tracing shadow rays from points in shadow and testing objects that could not be hit. However, computing and storing all possible view configurations for a scene — the aspect graph [20] — is impractical for complex scenes. Even calculating all the visual events in a scene has very high complexity [9] and poses numerical stability problems.

It is generally easier to conservatively overestimate the set of potentially visible objects (PVS [1, 26]) for a certain region of space (referred to as a “viewcell” throughout this paper). While effective methods exist to detect occlusions in indoor scenes [1, 26] and terrain models [24], in more general types of complex scenes previous approaches [4, 6, 21] consider single convex occluders only to determine objects, or portions of space, that are completely hidden from the viewcell. This is known as volume visibility.

In many cases, objects are hidden due to the combination of many, not necessarily convex, occluders. This situation is exacerbated by the lack of large polygons in today’s finely tessellated models. Figure 7 in Section 4.3 compares the number of occlusions detected using single convex occluders to the number detected with our method. Combining the effect of multiple, arbitrary occluders is complicated by the many different kinds of visual events that occur between a set of objects [9] and by various geometric degeneracies.

As a new solution to these problems, this paper proposes to calculate volume visibility on a conservative discretization of space. Occlusion is explicitly represented in this discretization and can be queried to retrieve visibility information for arbitrary scene objects — either static, dynamic or newly added.

We use opaque regions of space as blockers and automatically derive them from the scene description instead of expecting large convex occluders to be present in the scene. Our representation decouples the scene complexity from the accuracy and computational complexity at which visibility is resolved.

We show that hidden regions of space are valid blockers and that any opaque blocker can be extended into such regions of space. This effectively combines — *fuses* [32] — one blocker with all the other blockers that have caused this region to be occluded and results in a dramatic improvement in the occlusions detected. Collections of occluders need not be connected or convex.

The rest of the paper is organized as follows. In the next section, we review previous approaches to visibility computation with special emphasis on volume visibility methods. Next, we describe our approach in 2D and then extend it to 3D and 2 1/2 D. We present results for PVS computation and reducing the number of shadow rays in ray-tracing. We conclude with a discussion of our results and suggestions for future work.

2 Previous Work

The central role of visibility has resulted in many previously published approaches. We classify them into the following three categories: exact, point-sampled and conservative visibility computations and focus the discussion on volume visibility approaches. Examples of exact visibility representations are the aspect graph [20] or the visibility skeleton [9] and exact shadow boundaries [3, 8, 23, 27]. As mentioned above, they are impractical for complex scenes.

Point-sampling algorithms calculate visibility up to the accuracy of the display resolution [5, 7, 13]. One sample ray is sent into the scene and the obtained visible surface is reused over an area (e.g. a pixel or solid angle on the hemisphere). Today’s most widely used approach is a hardware-accelerated z-buffer [2] or its variants, the hierarchical z-buffer [14] and hierarchical occlusion maps [32]. Visibility results obtained from these algorithms cannot be extended to volume visibility without introducing error. For volume visibility, projections are not feasible, as no single center of projection is appropriate.

To cope with the complexity of today’s models, researchers have investigated conservative subsets of the hidden scene portion. Airey et al. [1] and Teller et al. [26, 28] propose visibility preprocessing for indoor scenes. They identify objects that are visible through sequences of portals. Yagel et al. [31] apply similar ideas in 2D for visibility in caves. Stewart [24] provides a solution for the case of terrain. Unfortunately, these algorithms do not generalize to volume visibility for more general types of complex scenes.

Conservative, but accurate, volume visibility computations for general scenes are limited to considering one convex occluder at a time for identifying hidden objects. Cohen-Or et al. [6] find hidden buildings in cities. Saona-Vazquez et al. [21] apply a similar strategy to the nodes in an octree. They intersect the PVS as seen from the eight corners of each voxel to obtain the PVS for the voxel. Coorg et al. [4] use supporting planes between the blocker and an occludee to determine occlusion. These planes also allow them to determine when the occluder will no longer hide the occludee. All these single occluder approaches share the difficulties of identifying good occluders, and none performs occluder fusion. Unfortunately, in practice many scenes do not contain any large polygons or convex objects. Durand [11] calculates volume visibility by projecting potential occluders onto planes. He modifies point-sampled projections and convolution to obtain a conservative algorithm.

In volume visibility it seems to be inherently difficult to combine the effects of multiple occluders in a provably accurate and efficient way. We believe that this is due to the nature of the occluders considered — convex polygons or objects — and because the portions of occluded space have not been explicitly represented or used in the computations.

Our visibility algorithm works entirely on a volumetric scene representation. We propose to abandon considering polygons as occluders, and instead let the volumetric nature of opaque objects occlude the space behind them. Several authors [4, 6, 10, 21] have required convex decompositions of arbitrary objects in order for their algorithms, operating on convex blockers, to work. Indeed, volumetric representations, such as octrees, provide such a convex decomposition. They also represent space itself so that efficient blockers can be found using occluded regions as described below. In our approach, we construct shafts around blockers as seen from the viewcell similar to Haines et al.'s shaft culling [16] and Teller et al.'s inter-cell visibility algorithm [28]. The difference is that Haines' shafts lie between the viewcell and the occludee, whereas ours lie behind the occluder as seen from the viewcell.

3 Definitions and Overview

Our goal is to determine occlusion from within a viewcell on a conservative discretization of space employing these definitions:

- A *viewcell* is an axis-aligned box that is either identical to or known to bound a volume of viewpoints of interest.
- An *occluder* (or *blocker*) is an axially-aligned box causing occlusion by opacity or other properties established by the algorithm (see Sections 4.2 and 4.3).
- A *shaft* is the convex intersection of half-spaces constructed from the visual events between the viewcell and the occluder. If the viewcell is considered as an extended light source, the volume inside the shaft is identical to the umbra of the occluder.
- A leaf voxel in the spatial subdivision is labeled *opaque* if it is completely inside an object, *empty* if it is completely outside all objects, or *boundary* [22] if it contains a portion of any object's surface. For this classification we require the blockers of the scene to be water-tight solids.

We seek a tight overestimate of the PVS as seen from a viewcell so that quick visibility queries with respect to any viewpoint in the viewcell are possible. Given a viewcell, these are the necessary steps:

Scene discretization. Rasterize the boundary of scene objects into the discretization of space and determine which voxels of space are completely inside an object and therefore opaque.

Blocker extension. Traverse the discretization of space and find an opaque voxel that is not already hidden. Group this blocker with neighboring opaque voxels to obtain an effective blocker.

Shaft construction. Construct a shaft that encompasses the region of space hidden by this blocker as seen from the viewcell.

Occlusion tracking. Use the shaft to classify the voxels into partially or completely outside the shaft and fully inside and thus occluded. Take note of occluded voxels.

The major contributions of this approach are a conservative scene discretization that decouples the effectiveness of visibility calculations from how the scene was modeled (i.e. presence of large polygons or convex objects, or number of polygons), and the introduction of blocker extension as a means of both finding efficient blockers and performing effective occluder fusion. Finally, we improve on shafts by observing that 3D shafts can be treated entirely in 2D.

We begin by describing our algorithm in the simple setting of two dimensions and note that this case is already suited to solve visibility queries on scenes such as a 2D floor plan or the map of a city.

4 2D Case

Our subdivisions of space are a quadtree in 2D and 2 1/2D, and an octree in 3D. In anticipation of the extension to 2 1/2D and 3D we will uniformly call a node in the tree a voxel. Its shape is always a (2D or 3D) axis-aligned box.

4.1 Scene Discretization

Our method requires that the interiors of objects can be distinguished from their exteriors. Objects need to have a solid volume. The discretization represents a cube of space containing the scene. All voxels containing a surface are marked as boundary voxels[1]. After this step, boundary voxels completely separate empty regions of space from opaque regions. Next we classify the non-boundary voxels into empty and opaque voxels using the odd-parity rule [12] for point-in-polygon or point-in-closed-shape testing.

We accelerate voxel classification by propagating the voxel status with depth-first seed-filling [22] up to the boundary voxels. Figure 1 shows an example taken from a simple test scene used throughout Section 4. Note that the blockers are not axis-aligned. Such an arrangement would improve the efficiency of the spatial discretization, but is not a requirement of the algorithm.

Our algorithm deals with scenes containing non-solid objects in several ways. If a voxel size is known, such that after marking boundary voxels the empty regions of space form a connected set of voxels, a single call to seed-filling will determine all empty voxels. The opaque voxels are the remaining non-boundary voxels. If some objects have holes or interior faces, the algorithm can still run and use the opaque interior of other objects. Objects with holes

[1]Near the surfaces, the spatial hierarchy is subdivided down to a maximum level. This maximizes the number of opaque voxels inside the objects.

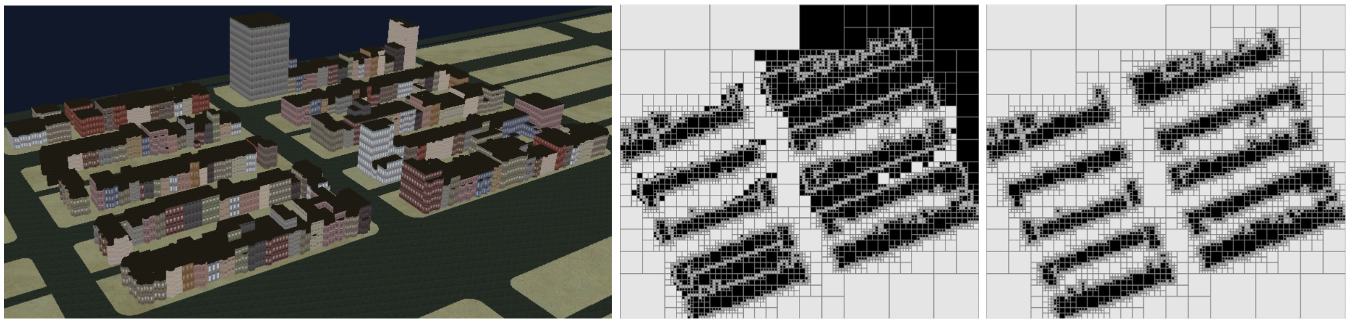


Figure 1: Left: ten blocks of buildings to be projected onto the ground. Right: Marking empty space between the buildings with flood-fill.

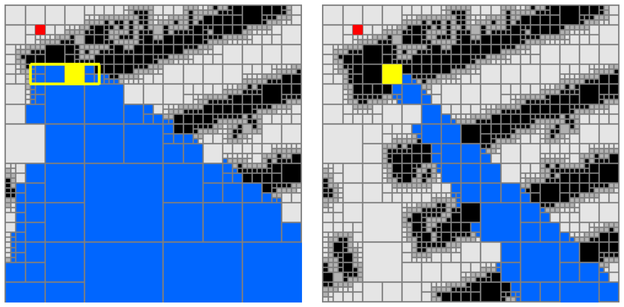


Figure 2: Left: a viewcell in red and a blocker with its extension outlined in yellow. The blocker hides the voxels in blue. Right: occlusion without blocker extension. (Opaque voxels are shown in black.)

have no defined interior and therefore are not used as occluders. Interior faces are ignored by flood-filling only the empty space around objects. Alternatively, degenerate input data can be cleaned up and interior faces can be removed using a method such as the one given by Murali et al. [19].

4.2 Blocker Extension

We recursively traverse the tree until we find an opaque voxel. This is the blocker used with the viewcell to construct the region of occluded space. To maximize occlusion we first *extend* the blocker. This extension must not hinder the ease of constructing the hidden region, so we require the extension to keep the box-shape of the blocker (see left side of Figure 2). Other opaque voxels are shown in black and will be considered as blockers next.

We extend the blocker along the coordinate axis, which maximizes the angle subtended by the blocker. Extension proceeds on both sides in this direction until there is a non-opaque voxel within the width of the blocker. In the cases where two sides of the blocker are visible from the viewcell, the blocker can additionally be extended into an L-shape as shown in Figure 3. From the viewcell, this L-shape appears exactly the same as the big dashed box enclosing the L-shape that is the blocker's final extension.

We use two optimizations to quickly find large hidden regions of space. First, we use the blockers in the order from the viewcell outwards; hence, blockers that subtend a wide solid angle compared to their size get used first. Second, we use large blockers in high levels of our spatial data structure first, as these can be expected to occlude large regions of space quickly. Hidden regions are not considered further for blocker selection.

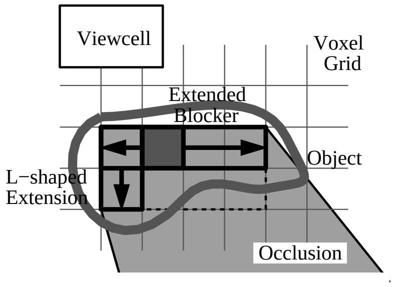
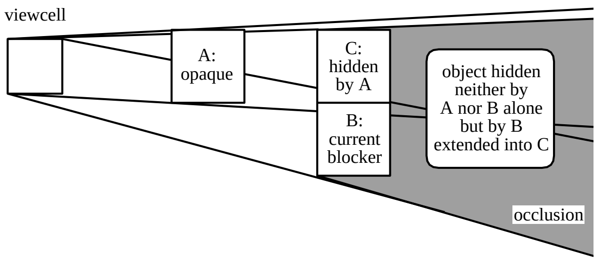


Figure 3: Example of L-shaped blocker extension. First the blocker is extended laterally. If after this step more than one side of the blocker is visible from the viewcell, the blocker is extended along this side away from the viewcell. The resulting occlusion is larger. Only the regions enclosed by the thick lines need to be opaque.


viewcell
A: opaque
C: hidden by A
B: current blocker
object hidden neither by A nor B alone but by B extended into C
occlusion
Figure 4: Because C is hidden by A, blocker B can be extended into C to create a much bigger region of occlusion, which hides the object on the right. Neither A nor B alone would occlude this object as seen from the viewcell.

4.3 Blocker Extension into Hidden Space

Despite its simplicity, the idea of extending blockers into adjacent opaque space has not been used in previous approaches. In addition, we make an even stronger point with blocker extension into *hidden* space, regardless of whether this space is empty or opaque. Extending blockers into hidden regions of space is based on the following observation (see Figure 4).

An observer inside the viewcell is unable to distinguish whether a hidden voxel is opaque or empty. For the sake of finding large occluders for this viewcell, we can assume that the voxel is opaque, and blocker extension can proceed into hidden voxels just as into opaque voxels.

In fact, one can construct different scenes that look exactly the same from within the viewcell by arbitrarily changing hidden parts of the model. One well-known application of this property is the PVS, where all polygons in hidden areas of the scene are removed.

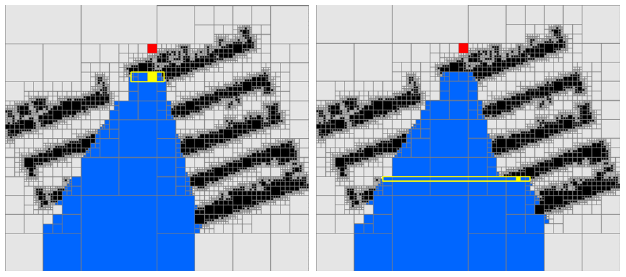


Figure 5: Left: marking the occlusion for one occluder. Right: extending a second occluder through the region hidden by the first one.

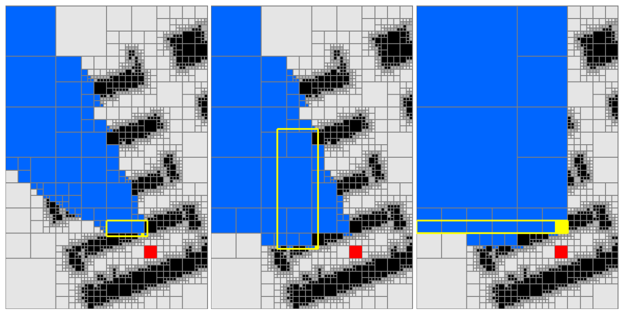


Figure 6: The first three steps of the algorithm as it marks occlusions for the red viewcell. Note how blockers are extended across the streets between the buildings.

It is therefore licit to change hidden voxels into opaque ones, with the benefit that larger blockers can be constructed.

The blocker on the right side of Figure 5 is extended across blocks through empty space connecting one block of buildings to another. Nonetheless, by the argument introduced above, extending the blocker this far is valid. It effectively fuses the current blocker with the blocker shown on the left side of Figure 5. It was this blocker that caused all the voxels between buildings to be hidden.

As shown in Figure 4, occluder fusion occurs if the second blocker overlaps the umbra of the first blocker. In general, any blocker can be arbitrarily extended inside the umbra of another blocker as the umbra is the region of space completely hidden from the viewcell.

Figure 6 shows the progress of the algorithm after using only three extended blockers. In general, especially with scenes containing no major large polygons, occluder fusion is essential as is obvious from the comparison given in Figure 7. It shows the occlusions detected after all blockers have been used for three different approaches from left to right: triangles as single convex occluders, opaque voxels without blocker extension, and opaque voxels with blocker extension. A few large polygons have caused some occlusion to be detected in the upper left of the triangle-based approach. If large polygons are present in the scene database, they can be used to bootstrap our approach to occlusion detection.

4.4 Shaft Construction and Occlusion Tracking

We construct a shaft around the occluded region from the supporting planes[2] between the viewcell and the blocker. Details are given in the appendix. Our implementation differs from the one by Haines et al. [16] in that we replace set manipulations on box corners with table lookups.

[2]A supporting plane contains an edge from one object and a vertex from another object, such that both objects lie on the same side of the plane [4].

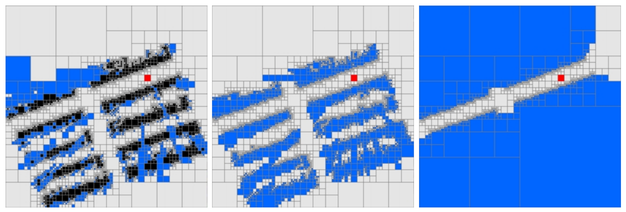


Figure 7: The blocks of buildings from above. Left: occlusions detected using triangles from the database as convex occluders. Middle: occlusions detected without blocker extension. Right: occlusions detected with blocker extension.

Once the shaft has been constructed, a recursive traversal of the spatial data structure flags hidden voxels as occluded in the highest tree node possible. Subtrees outside or inside the shaft are not traversed.

If all the children of a voxel in the spatial data structure are found to be hidden, the parent can be marked as being hidden as well. An exception occurs in the 2 1/2 D case, which will be discussed in Section 6. Propagating visibility up the tree is useful for accelerating traversal of the tree for blocker extension, for marking occluded regions, and for querying occlusion of original objects.

4.5 Querying Occlusion of Scene Objects

Occlusion of the original objects is determined by inserting their bounding boxes into the tree and checking that all the voxels they overlap are hidden. When a bounding hierarchy exists on the input scene — say on the block-, house-, and triangle-level — occlusion queries can be further accelerated by interleaving the traversal of the bounding box hierarchy with the traversal of the tree.

Note that we can also determine the visibility of objects that were not initially inserted into the tree. This allows objects unlikely to cause a lot of occlusion to be ignored when constructing the tree. Also, the visibility status can be determined for moving objects or objects that have been added to the scene.

5 3D Case

Usually visibility algorithms for 2D are difficult to extend to 3D because the number of occlusion boundaries grows from $O(n^2)$ to $O(n^4)$ [10], and because the occlusion boundaries are no longer planar in general. In our case, however, the shaft construction extends to 3D in a straightforward fashion. This is due to our choice of viewcells and blockers as axis-aligned boxes in which case occlusion boundaries remain planar.

5.1 Shafts in 3D

Haines et al. [16] simplify the construction of a shaft's plane equations by noting that a shaft around axis-aligned boxes consists entirely of planes, the normals of which have at least one zero coordinate. In addition, a 3D shaft can be more efficiently treated as the intersection of three 2D shafts, namely the shaft's projections onto the three coordinate planes as shown in Figure 8. A bounding box is then inside the shaft if it is inside each of the three 2D shafts. We ensure that planes orthogonal to the coordinate axes are included only in one 2D shaft.

5.2 Blocker Extension in 3D

In 3D, a blocker must be extended along more than one dimension to subtend a large solid angle. We first extend the blocker along one

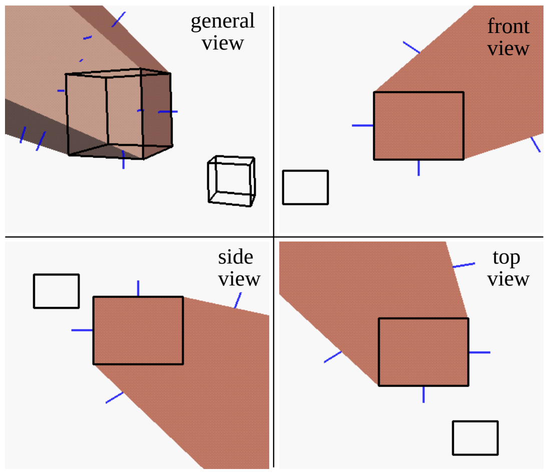


Figure 8: A 3D shaft can be treated as the intersection of three 2D shafts (the shaft's orthographic projection along the coordinate axes).

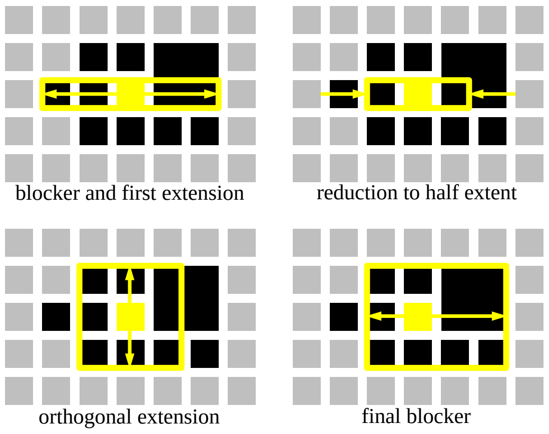


Figure 9: Blocker extension in 3D.

axis, then reduce its size to half its length, then extend it orthogonally to the first extension, and finally, extend again along the first axis as shown in Figure 9.

We have tried to maximize the solid angle subtended by the blocker over different reduction fractions (see second step of Figure 9). We did not observe any noticeable increase in the occlusions identified, but CPU time increased considerably. It is our experience that occluder fusion more than compensates for any suboptimal blocker extension.

Similar to the L-shaped blocker extension in 2D, the blocker is also extended along additionally visible faces as shown in Figure 10. Other than that, occlusion detection in 3D works exactly the same as in 2D: voxels in the tree that are completely inside the shaft of the blocker are marked as occluded.

6 2 1/2D Case

Memory requirements for a 3D octree are sometimes a concern. However, scenes such as terrains or cities can often be dealt with in 2 1/2D. The 2 1/2D quadtree is constructed by recording the height of the highest and lowest points of every primitive falling within a certain voxel (i.e. a square on the ground plane). No ray-casting or seed-filling is necessary. Our tree construction algorithm strictly enforces the 2 1/2D characteristic of the input data set and filters

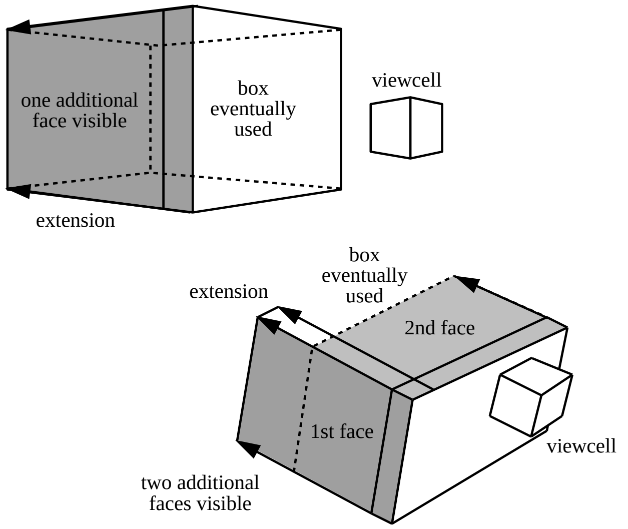


Figure 10: Additional step of blocker extension in the direction orthogonal to the extensions shown in Figure 9. If two additional faces are visible, their minimum extension must be used.

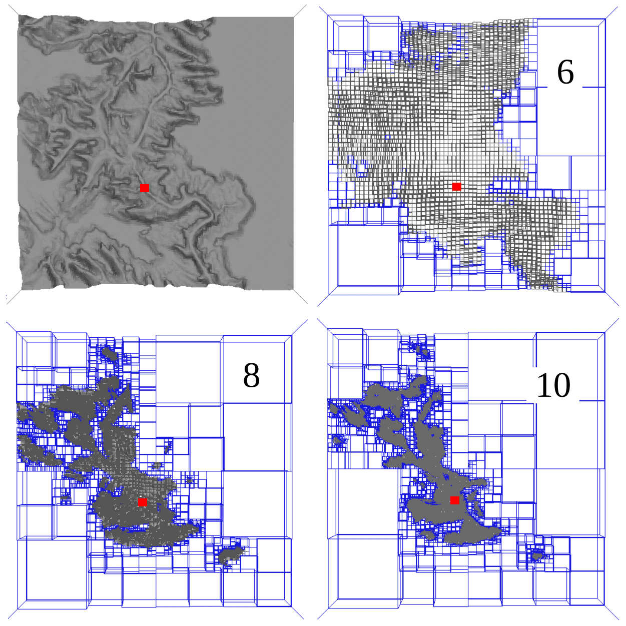


Figure 11: An example of occlusion culling in terrain: Grand Canyon. Top left: geometric model. In the other images the number indicates the maximum quadtree subdivision level down to which the quadtree has been built. Hidden nodes are shown in blue; potentially visible nodes are shown in grey as seen from the red viewcell.

out multiple triangles above a single location on the ground plane. Voxels are occluded if they are inside the shaft up to their maximum height and can be used as an occluder up to their minimum height.

We perform occluder extension the same way as in the 2D case including L-shaped extension; shaft construction is the same as in the 3D case — three 2D shafts are generated. The shaft plane coincident with the ground is ignored. Even though only one shaft in the ground plane and one plane connecting the highest point on the viewcell with the lowest point on the blocker could be used, we found the three 2D shafts to be more effective as they allow easy tracking of occluded height in the 2D projections. We store this height per hidden voxel.

An important difference to the previous two cases of 2D and 3D occlusion detection occurs with the propagation of visibility up the tree. Even if all the children are hidden, the parent is not necessarily hidden everywhere up to its maximum height. Calculating and storing the height up to which a voxel is hidden helps both with visibility propagation up the tree and also with occluder extension, because higher occluders can be extended into a certain voxel.

Figure 11 shows examples of detected occlusion in a terrain model with the maximum tree subdivision of 6, 8 and 10.

7 Applications and Results

We apply occlusion detection to the following rendering algorithms: visibility preprocessing for real-time walkthroughs and shadow-ray culling in ray-tracing. We use the 2 1/2D quadtree to find both viewcells and their PVS in an outdoor city environment. The 3D version is then applied to find the surfaces not receiving direct light from extended light sources in a ray-tracer.

7.1 Visibility Preprocessing for Walkthroughs

Since cities and other outdoor scenes are predominantly 2 1/2D, we use the 2 1/2D quadtree described in Section 6 for occlusion calculations. The model shown in Figure 12 consists of 316,944 triangles that are organized into 665 buildings and 125 blocks complete with streets and sidewalks. Table 1 shows statistics on how long the quadtree takes to build based on the maximum subdivision level in the tree (e.g. subdivision 8 stands for a 256^2 maximum subdivision). Figure 12 shows the tree portions in blue found to be occluded for one viewcell.

Using a moderate tree subdivision level (up to 8 levels) visibility queries are possible at interactive rates (a few tenths of a second). This includes querying objects in the tree. To avoid this expense of computation at runtime, we pre-compute PVSs for those regions of our model reachable during visual navigation.

Our walkthrough system accepts the navigable space as a set of triangles describing the streets or paths. We first determine the PVS for every triangle in the street mesh by constructing a 3D bounding box around it and marking occluded sections of the model in the quadtree. Then we look up the objects such as buildings, terrain and street portions, trees, cars and people in the quadtree and only add those to the triangle's PVS that are not fully occluded.

A total of 2,538 triangles in the streets have been automatically grouped into 700 street sections for database paging as shown in Figure 13. By imposing an upper limit on the difference between the size of the triangles' PVSs in one street section, our greedy merge algorithm limits the amount of overdraw within one section. The walkthrough system pre-fetches the geometry for adjacent street sections as the user moves around so that exploration is possible without interruption. Such predictive database paging is impossible with online point-visibility methods.

Preprocessing this scene took 55 minutes, the vast majority of which was spent to find the PVS for every viewcell. The remaining time is used for building the tree and grouping viewcells into street sections. The average time for finding the PVS of a triangle is less than a second.

tree levels	nodes	memory (kB)	time to build (sec)
7	11,949	574	2.669
8	39,625	1,902	3.704
9	133,577	6,412	6.381
10	472,561	22,683	10.197

Table 1: 2 1/2 D quadtree generation statistics: number of nodes, memory, and build time as a function of the maximum subdivision level in the tree on a MIPS R10k processor running at 250 MHz.

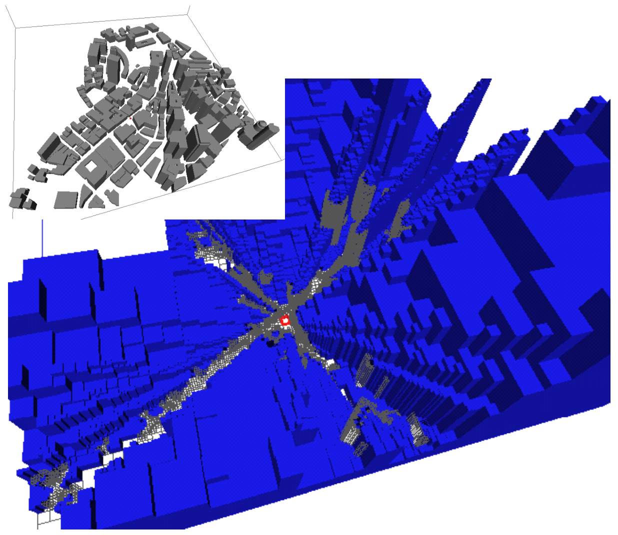


Figure 12: Top: a financial district from above (316,944 triangles). Bottom: occluded tree portion shown as blue boxes in the same overhead view as above. Box heights reflect the hidden height of voxels.

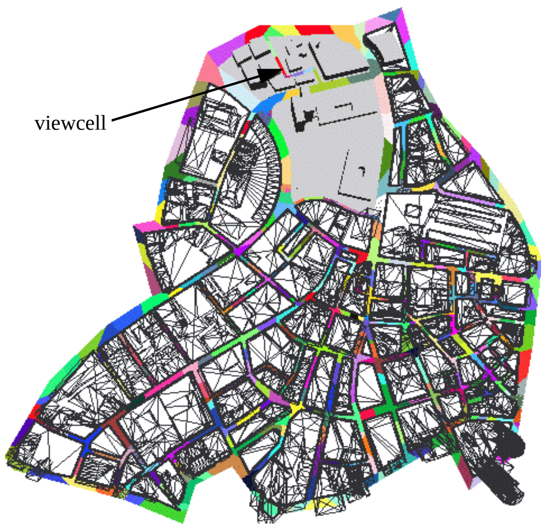


Figure 13: Street triangles for the financial district grouped into sections. For one of them the PVS is shown, the rest of the buildings are shown in wire-frame.

We have retrieved the PVS for every triangle with two different granularities from the tree for comparison: blocks and buildings. For block-based PVS, an average of 33.96 blocks were found potentially visible out of 125 blocks, the minimum count of visible blocks was eight, the maximum 82. It is apparent that querying the bounding box of a whole block in the quadtree results in a rather high over-estimation of the PVS.

For building-based PVS, the average building count was 54 out of a total of 665 buildings, sidewalks, and road segments. The minimum and maximum numbers were 12 and 156. In this case, we queried the smallest bounding boxes available in the bounding-box hierarchy of our model.

We are unaware of a method that could compute an accurate reference solution to this problem. Instead we have tried to compute a good approximation to the true solution using point sampling. From every triangle in the street mesh we took twenty 360 degree

block-based PVS size	total: 125 blocks (about 2,500 tris on avg) our method	point samples	difference
min	8	6	0
max	82	48	47
avg	33.96	20.3	13.64
building-based PVS size	**total: 665 buildings (about 475 tris on avg) our method**	**point samples**	**difference**
min	12	8	0
max	156	90	81
avg	53.87	33.99	19.88

Table 2: Comparison of PVS sizes for block-based (125 blocks) and building-based (665 buildings) PVS computation. Going from block-based to building-based PVSs reduced the average difference between the two methods from 10.91% to 2.98% of the complete model.

256^2 pixel images recording the visible object per pixel into an item buffer. The set union of the objects visible in these images was saved as the PVS for that triangle.

We noted a couple of difficulties in this point-sampled reference solution: narrow gaps between buildings can still be missed and the blocks or buildings visible through these gaps are not reported as visible. Also, in views looking down long straight streets, the sidewalks and streets project to very small areas in screen space and can be missed as well. This supports the need for methods such as the one presented in this paper.

We give the comparison between our PVS and the point-sampled solution in Table 2. Note that the difference is always positive or zero, which demonstrates that our method is conservative.

Finally, Figure 14 shows the difference in drawing time measured on an SGI Infinite Reality system. The left shows results for block-based PVS, the right shows results for building-based PVS compared to IRIS Performer view-frustum culling.

7.2 Shadow Ray Acceleration

When ray-tracing complex scenes with many lights, tracing shadow rays and computing object intersections can account for 95% of the rendering time [15]. It is therefore desirable to minimize the number of rays traced. Methods have been developed to accelerate shadow rays for point sources [15, 30], but the work published for extended lights is either approximate [17] or not directly applicable to a ray tracer [25]. Extended lights are preferable, as they cast soft shadows rather than the unnaturally looking sharp shadows caused by point lights.

Our shadow ray acceleration is a generalization of the method of Woo and Amanatides [30] from point light sources to extended light sources. Figure 15 shows an example of the artifacts to be expected if one tries to apply a point-light acceleration technique to an extended light source. The point light causes a sharp shadow which is contained in the region of penumbra of an area light source at the same position. A point-light acceleration algorithm falsely identifies regions of the penumbra as in shadow and pronounced boundaries appear in the umbra where there should be a continuous light-intensity variation.

The acceleration technique uses the full 3D version of our algorithm with space represented as an octree, as described in Section 5. We find the bounding box for every light source in the model and use it as a viewcell. For every light source we copy and keep the visible portion of the octree[3]. In the case of many light sources, memory consumption is a concern, and only a specific number of top levels in the tree are copied with only a small increase in the

[3]Figure 19 shows that these copies require memory comparable to the initial octree

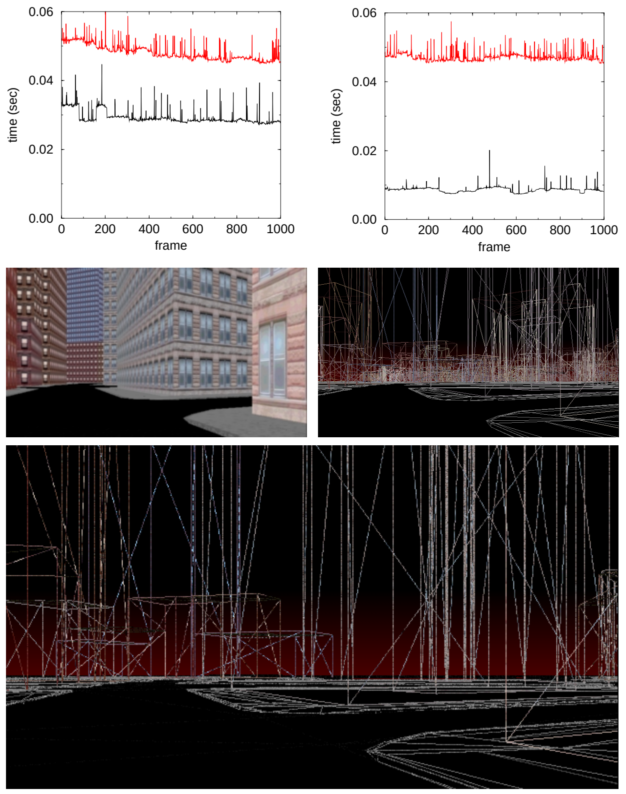


Figure 14: Drawing time without (red) and with (black) occlusion culling in our walkthrough system. Left: block-based PVS. Right: building-based PVS. Below: a frame from the walkthrough and a comparison of the amount of geometry drawn in wireframe.

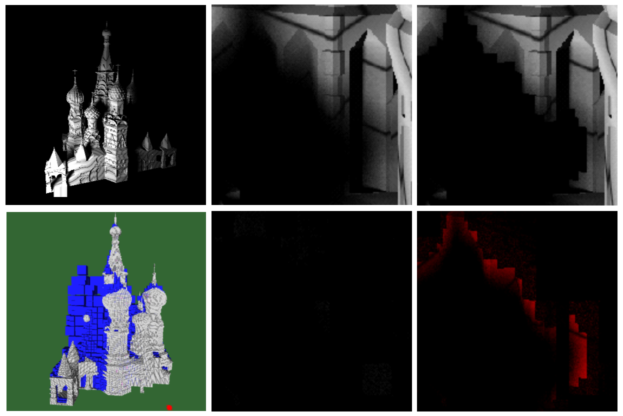


Figure 15: An example of artifacts in the area of soft shadows introduced by an acceleration technique which was not designed for extended light sources (middle: correct shadow, right: artifacts. Geometric model and octree on the left).

number of occlusions that are not detected (as shown in the results below). Our representation of the octree uses eight bytes per node.

As the ray-tracer renders the scene, every light's octree is queried to determine whether the current sample point is potentially visible to the light. Shadow rays need only to be cast for potentially visible points. The top of Figure 16 shows a view of a city block at night with ten street lights around it. Below is a false-color image of the same block from above where every pixel is given a color based on which light sources would be queried with shadow rays. All

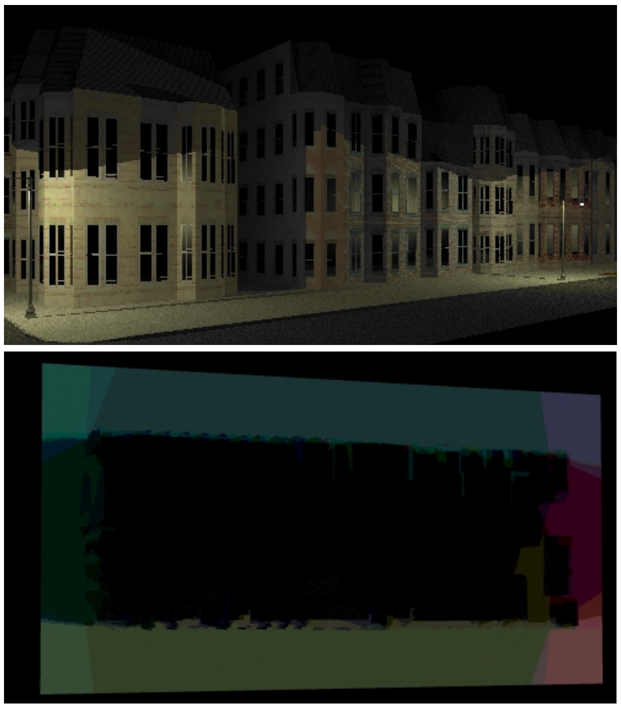


Figure 16: Top: accelerated rendering of a block of houses illuminated by ten street lamps. Bottom: false-color image from above encoding which lamps are queried with shadow rays at each surface point. Note how the corners of the building cause "occlusion boundaries."

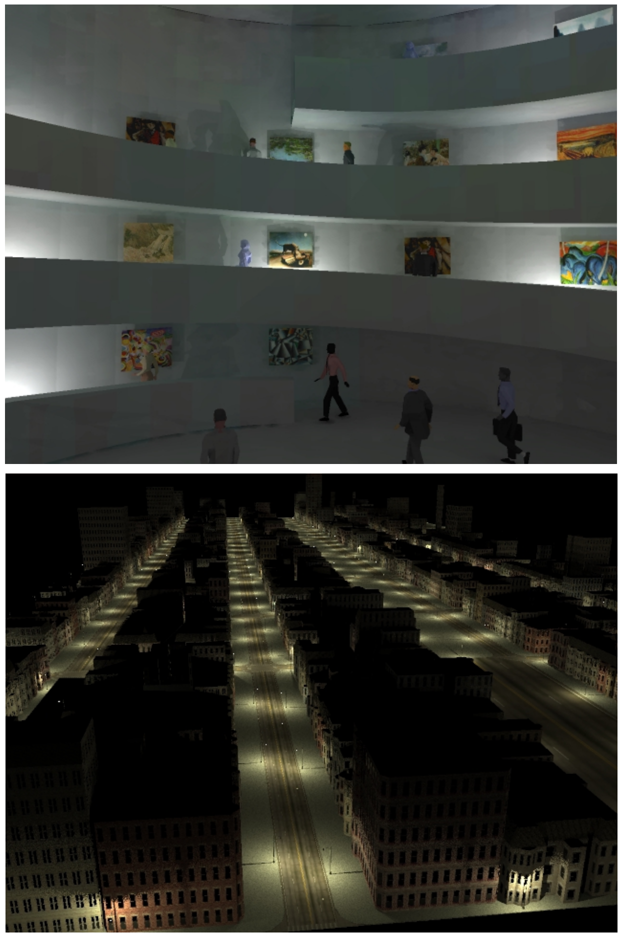


Figure 17: Top: gallery: 17,701 triangles, 23 lights. Bottom: residential area, 616,509 triangles, 459 light sources.

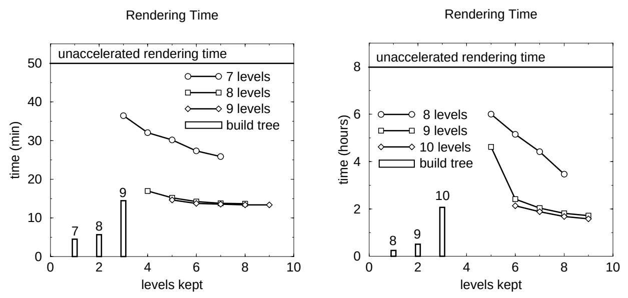


Figure 18: Rendering times for the two models: left: gallery, right: residential area. Different octree depths are shown, the number of levels kept per light varies along the x-axis.

images in this section are rendered with one sample per pixel, and global illumination taken into account using stochastic sampling optimized with irradiance gradient caching [29]. The ray tracer uses hierarchical grids as the general acceleration data structure.

Preprocessing took 53.3 seconds on a Pentium II 400MHz processor, and the time to render this image was reduced from about 250 seconds to 100 seconds (a 60% saving). This is particularly advantageous for animation sequences with static lighting, where the preprocessing is amortized over the full set of frames.

Figure 17 shows more complex scenes: a gallery with paintings and people (17,701 triangles and 23 light sources) and a residential area (616,509 triangles, 459 light sources). The bars on the left of Figures 18 and 19 give tree construction time and initial tree memory usage respectively as a function of the maximum octree subdivision level. The right graph in Figure 18 plots rendering time as a function of the number of levels kept in the octree per light. The right graph in Figure 19 gives the memory requirements for the octree copies. The original tree is no longer needed during rendering.

By considering both charts together, one observes that seven levels in the octree are not sufficient to accurately capture the occluders in the gallery scene (eight in the residential area scene). However, there is no longer a substantial difference between the rendering times using a 256^3 octree or a 512^3 octree (512^3 or 1024^3 for the residential area scene). It is surprising that three or more levels can be discarded from the light source octree without paying a significant price in the number of occlusions missed. However, the savings in memory are quite substantial.

Finally, Figure 20 shows the effectiveness of the method by giving the percentage of shadow rays successfully culled and the percentage of traced shadow rays, which were found to intersect geometry. This is the percentage of rays that was not reported as occluded even though the sample point is hidden from the light source. Note how this number decreases as the octree resolution increases. In the case of the residential area, the percentage of rays blocked although not reported as such remains quite high, because the lamp geometry was not inserted into the octree and, therefore, sample points on the facades at a height above the light could not be found to be in shadow. The results can be summarized as follows: with a doubled memory consumption rendering is accelerated by a factor of three for the gallery scene and by a factor of four for the residential area scene.

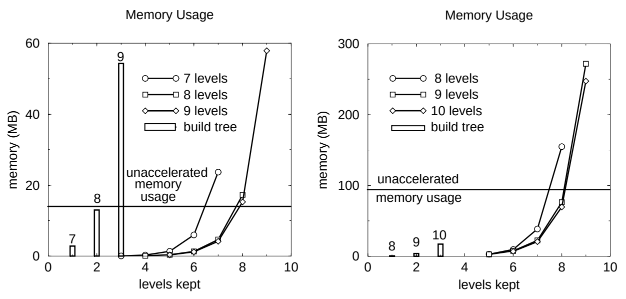


Figure 19: Memory requirements for building the octree (bars on the left) and during rendering: left: gallery, right: residential area as a function of octree levels kept per light. The bars on the left show tree build time as a function of the maximum octree subdivision level allowed.

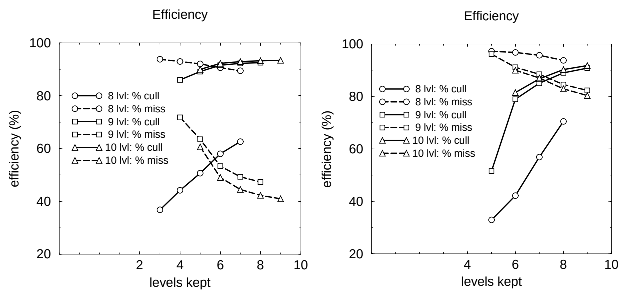


Figure 20: Effectiveness of the octree for eliminating shadow rays: left: gallery, right: residential area. The percent of shadow rays eliminated is shown as a solid line, the number of shadow rays that resulted in an intersection, but were not reported as occluded is shown as a dashed line. The number of levels kept per light varies along the x-axis.

8 Conclusions and Future Work

We have presented a method to compute a conservative approximation of the space hidden as seen from a viewcell. Voxels from a discretization of space are classified into empty, opaque, and boundary. Opaque voxels are used as blockers, and a shaft is constructed to determine the portion of space hidden behind them. We apply blocker extension both into adjacent opaque voxels and hidden voxels to maximize the size of blockers.

Blocker extension into hidden regions of space is motivated by the fact that arbitrary assumptions can be made about scene portions hidden to the viewer. For the sake of blocker extension, we assume them to be opaque and extend blockers into them, thereby fusing blockers with any blocker or group of blockers that caused this region to be hidden.

We have applied the method to visibility preprocessing and have obtained a tight superset of the actual PVS as seen from a region of space. Such information is also useful for subdividing the space of reachable viewpoints into sections for managing on-the-fly paging of geometry.

In the context of ray-tracing, we have successfully eliminated a major fraction of shadow rays cast toward extended lights. These rays are usually necessary to calculate the regions of shadow as cast by the light. Despite the memory requirements for a fine discretization of space during preprocessing, a moderate amount of memory is sufficient during rendering to capture occlusions with high fidelity even for a large number of light sources.

In the future we would like to investigate further applications of the available visibility information, automatic ways of choosing a sufficient octree subdivision level, and other subdivision schemes such as kd-trees or more sophisticated boundary nodes to improve the effectiveness of the method. We also want to investigate the success of blocker extension in other previously proposed visibility methods. Moreover, by rasterizing freeform patches or constructive solid geometry our visibility algorithm could be extended to these modeling approaches.

Acknowledgments

We would like to thank Henrik Wann Jensen for allowing us to build on his *Dali* ray tracer for the results presented in Section 7.2. Max Chen built the city model that appears in Figure 17; Byong Mok Oh contributed shaders to render it.

This work was supported by an NSF Postdoctoral Research Associates award (EIA-9806139), an NSF CISE Research Infrastructure award (EIA-9892229), an NSF-INRIA Cooperative Research award (INT-9724005), and a grant from Intel Corporation. iMAGIS is a joint research project of CNRS, INRIA, INPG and the Joseph Fournier University of Grenoble.

References

[1] Airey, John M., John H. Rohlf and Frederick P. Brooks, Jr., *"Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments,"* Symposium on Interactive 3D Graphics 1990, pp 41-50.

[2] Catmull, Edwin, E., *"Computer Display of Curved Surface,"* IEEE Conference on Computer Graphics, Pattern Recognition and Data Structures, May 1975, pp 11-17.

[3] Chin, Norman and Steven Feiner, *"Fast Object-Precision Shadow Generation for Area Light Sources Using BSP Trees,"* Symposium on Interactive Graphics (1992), pp 21-30.

[4] Coorg, Satyan and Seth Teller, *"Temporally Coherent Conservative Visibility,"* Proc. Twelfth Annual ACM Symposium on Computational Geometry, Philadelphia, PA, May 24-26, 1996, pp 78-87.

[5] Cohen-Or, Daniel and Amit Shaked, *"Visibility and Dead-Zones in Digital Terrain Maps,"* EUROGRAPHICS '95 14 3 (1995) pp 171-180.

[6] Cohen-Or, D., G. Fibich, D. Halperin and E. Zadicario, *"Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes,"* Computer Graphics Forum, 17 (3) 1998, pp 243-253.

[7] Chrysanthou, Y., D. Cohen-Or, and D. Lischinski, *"Fast Approximate Quantitative Visibility for Complex Scenes,"* Computer Graphics International '98, June 1998, pp 220-229.

[8] Drettakis, George, Fiume Eugene L., *"A Fast Shadow Algorithm for Area Light Sources Using Backprojection,"* SIGGRAPH 94, pp 223-230.

[9] Durand, Fredo, George Drettakis and Claude Puech, *"The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool,"* SIGGRAPH 97, pp 89-100.

[10] Durand, Fredo, *"3D Visibility: Analytical Study and Applications,"* PhD Dissertation, Universite Joseph Fournier, Grenoble, France.

[11] Durand, Fredo, George Drettakis, Joelle Thollot and Claude Puech, *"Conservative Visibility Preprocessing using Extended Projections,"* SIGGRAPH 2000.

[12] Foley, James, Andries van Dam, Steven Feiner and John Hughes, "*Computer Graphics: Principles and Practice,*" Addison-Wesley Publishing Co., ISBN 0-201-12110-7, 1990.

[13] Greene, Ned "*Approximating Visibility with Environment Maps,*" Graphics Interface '86, pp 108-114.

[14] Greene, Ned, Michael Kass and Gavin Miller, "*Hierarchical Z-Buffer Visibility,*" SIGGRAPH 93, pp 231-238.

[15] Haines, A. Eric and Donald P. Greenberg, "*The Light Buffer: A Ray Tracer Shadow Testing Accelerator,*" IEEE Computer Graphics and Applications, 6 9, 1986, pp 6-16.

[16] Haines, A. Eric and John R. Wallace, "*Shaft Culling for Efficient Ray Cast Radiosity,*" Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering), Springer-Verlag, New York, 1994, pp 122-138.

[17] Hart, David, Philip Dutre and Donald P. Greenberg, "*Direct Illumination with Lazy Visibility Evaluation,*" SIGGRAPH 99, pp 147-154.

[18] Hudson, Tom, Dinesh Manocha, Jonathan Cohen, Ming Lin, Kenneth E. Hoff III, Hansong Zhang, "*Occlusion Culling using Shadow Volumes,*" Proceedings of 13th Symposium on Computational Geometry, Nice, France, June 4-6 1997, pp 1-10.

[19] Murali T.M., and Thomas A. Funkhouser, "*Consistent Solid and Boundary Representations from Arbitrary Polygonal Data,*" Symposium on Interactive 3D Graphics), 1997, pp 155-162.

[20] Plantinga, Harry. and Charles Dyer, "*Visibility, Occlusion and the Aspect Graph,*" International Journal of Computer Vision, 5(2) 1990, pp 137-160.

[21] Saona-Vazquez, Carlos, Isabel Navazo and Pere Brunet, "*The Visibility Octree. A Data Structure for 3D Navigation,*" TR LSI-99-22-R, Universitat Politecnica de Catalunya, Spain.

[22] Samet, Hanan, "*Applications of Spatial Data Structures,*" Addison-Wesley, Reading, MA, ISBN 0-201-50300-X, 1990.

[23] Stewart, James and S. Ghali, "*Fast computation of shadow boundaries using spatial coherence and backprojections,*" SIGGRAPH 94, pp 231-238.

[24] Stewart, James, "*Hierarchical Visibility in Terrains,*" Eurographics Rendering Workshop, June 1997, pp 217-228.

[25] Tanaka, Toshimitsu, and Tokiichiro Takahashi, "*Fast Analytic Shading and Shadowing for Area Light Sources,*" Computer Graphics Forum, Vol. 16, 3, 1997, pp 231-240.

[26] Teller, Seth J. and Carlo H. Sequin, "*Visibility Preprocessing For Interactive Walkthroughs*", SIGGRAPH 91, pp 61-69.

[27] Teller, Seth, "*Computing the Antipenumbra of an Area Light Source*, SIGGRAPH 92, pp 139-148.

[28] Teller, Seth, and Pat Hanrahan, "*Global Visibility Algorithms for Illumination Computation,*" SIGGRAPH 94, pp 443-450.

[29] Ward, Gregory J. and Paul Heckbert, "*Irradiance Gradients,*" Third Eurographics Workshop on Rendering, May 1992, pp 85-98.

[30] Woo, Andrew and John Amanatides, "*Voxel Occlusion Testing: A Shadow Determination Accelerator for Ray Tracing,*" Graphics Interface '90, pp 213-219.

[31] Yagel, Roni, and William Ray, "*Visibility Computation for Efficient Walkthrough of Complex Environments,*" PRESENCE, Vol.5, No. 1, Winter 1996, pp 1-16.

[32] Zhang, Hansong, Dinesh Manocha, Tom Hudson and Kenneth E. Hoff, "*Visibility Culling using Hierarchical Occlusion Maps,*" SIGGRAPH 97, pp 77-88.

Appendix

In 2D the shaft between two boxes is delimited by lines through the corners of the boxes and a subset of the blocker faces. Table 3 lists the box corners that need to be connected. Figure 21 gives a pictorial overview of the cases.

Case	X: 0	X: 1	X: 2	X: 3
Y: 0	1,2	3,2	3,0	1,0
Y: 1	1,3	-	2,0	all
Y: 2	0,3	0,1	2,1	2,3
Y: 3	0,2	all	3,1	-

Table 3: Box corners to be connected based on the relative position of the boxes. Cases where four connections must be made are marked "all", but are only relevant for the 2 1/2D and 3D extensions if viewcells are limited to empty space.

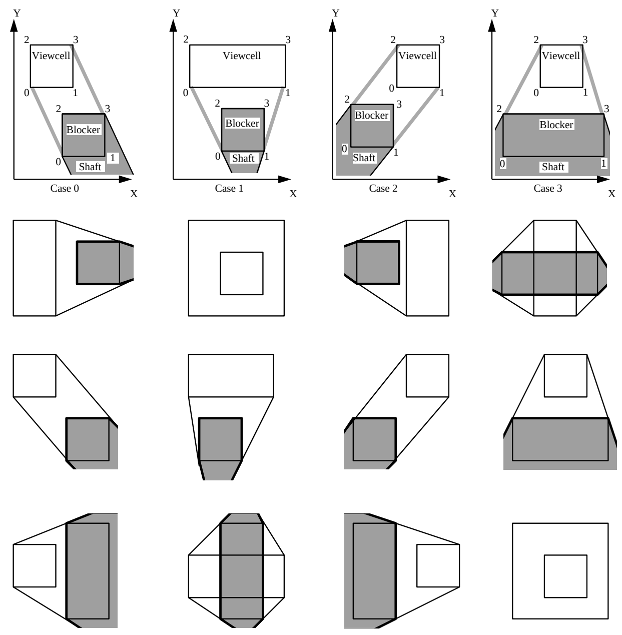


Figure 21: First row: cases of mutual positions between blocker and viewcell along the X-axis. The shafts are shown in grey. The rest of the figure gives a pictorial impression of the contents of Table 3.