Visibility Computations in Polyhedral Three-Dimensional Environments

Seth J. Teller
Carlo H. Séquin
University of California at Berkeley[‡]

Abstract

The cell-to-cell visibility preprocessing and query algorithms for 2D axial floorplans (presented at SIGGRAPH '91) have been extended to cell-to-object visibility in 3D polyhedral environments, and an efficient implementation for 3D axial environments is demonstrated on a fully furnished architectural floor model with 250,000 polygons.

The building is subdivided along the major walls into cells, connected through transparent portals. During preprocessing, cell-to-cell visibility is established for all cell pairs connected by at least one sight line through the intervening portals. Even for a source cell containing a generalized observer, often only a small portion of other cells is visible. Only objects whose bounding boxes intersect these "visible volumes" become part of the cell-to-object visibility set of the source cell.

In the real-time walk-through simulation, cell-to-object visibility is further pruned with the view frustum from the observer's current position, and only the remaining objects, which still form an uncompromised superset of the truly visible polygons, are sent to the renderer. In our architectural floor model, this process removed on average about 95% of all polygons, and accelerated rendering speed by about a factor of seventeen.

CR Categories and Subject Descriptors: [Computer Graphics]: I.3.5 Computational Geometry and Object Modeling – *geometric algorithms, languages, and systems*; I.3.7 Three-Dimensional Graphics and Realism – *visible line/surface algorithms, color, shading, shadowing, and texture.*

Additional Key Words and Phrases: architectural simulation, linear programming, superset visibility.

[‡]Computer Science Department, Berkeley, CA 94720

1 Introduction

Realistic, interesting environmental models may consist of several million polygons. Today's workstations cannot render scenes of this complexity in a fraction of a second, as is necessary for smooth interactive "walkthroughs" or simulations of the model. We are investigating the extent to which we can pay "precomputation" time and space to achieve interactive frame rates during simulation.

Some scenes, such as architectural models, typically consist of large connected elements of opaque material (e.g., walls), so that from most vantage points only a small fraction of the model can be seen. The model can be spatially subdivided along major opaque surfaces [10], producing polyhedral *cells* joined by polygonal *portals*, and the model partitioned into sets of polygons attached to each cell [1,13,25]. Some information about the subdivision and opacity relationships present in the model can then be computed offline and associated with the cell for later use in an interactive rendering phase. In [25], we presented a visibility preprocessing scheme that computed mutual visibility among the cells in a two-dimensional floorplan.

In this paper, we describe a solution to the visibility problem for both axial and general three-dimensional polyhedral environments. Our approach still centers on partitioning the model into a spatial subdivision using major opaque surfaces, and computing and storing observer-independent visibility information about each cell. We have extended it considerably, however, by developing visibility computations of a much finer "grain" than simple links between cells. Specifically, we show how to compute a "visibility funnel" of "virtual light" that shines from a source cell into some other cell through a sequence of portals. We then use this funnel to discard the detailed objects (e.g., cups, phones, books) in the latter cell that are invisible to any observer in the source. This *cell-to-object* determination is performed statically for each cell and object of the subdivision, and stored via a simple modification of the stab tree (which previously stored only the cell-to-cell visibility). Similarly, we extend the real-time computation phase by defining an *eye-to-object* visibility determination, which typically includes a much smaller set of objects than its coarser analogue, the eye-to-cell visibility set.

Enhancing rendering speed with spatial subdivision techniques and precomputation has been a long-standing activity in the graphics and computational geometry communities. Section 2 reviews past efforts related to our own approach. Section 3 contains a conceptual overview of our new approach, reviewing some concepts introduced a year ago [25] where necessary for understanding. Section 4.1 revisits the

notions of sightlines, cell-to-cell visibility, and stab trees, and Section 4.2 introduces the notion of *cell-to-object* visibility. Section 5.1 describes an optimal eye-to-cell visibility computation performed during the interactive walkthrough phase. Quantitative experimental results from use of the visibility computations on a real, furnished axial three-dimensional building model are given in Section 6.

The paper first concentrates on the axial 3-D case, for didactical reasons (also, the algorithms described become particularly efficient in this case). Section 7 then extends our approach to the case of general (i.e., non-axial) polyhedral models in three dimensions. Section 8 discusses the appropriateness and efficacy of these techniques for real-world environments.

2 Previous Work

Several techniques based on spatial subdivision have been proposed to accelerate rendering. We broadly refer to these methods as "visibility precomputations," since by performing work ahead of time they reduce the effort involved in solving the hidden-surface problem during actual scene rendering. Some researchers have directed attention to computing *exact* visibility (e.g., [6,16,20,24,28]), that is, computing a precise description of the visible elements of the scene for qualitatively distinct regions of viewpoints. Such descriptions may be combinatorially complex and difficult to implement [20,23], even for highly restricted viewpoint regions.

The binary space partition (BSP) tree data structure [10] obviates the hidden-surface computation by producing a back-to-front ordering of polygons from any viewpoint. This technique has the disadvantage that every polygon must lie in a splitting plane and, for an n-polygon scene, the splitting operations needed to construct the BSP tree generate $O(n^2)$ new polygons in the worst case [21]. Moreover, all polygons in the scene must be explicitly processed for each rendered frame.

Fixed-grid and octree spatial subdivisions [11,14] accelerate ray-traced rendering by efficiently answering queries about rays propagating through ordered sets of parallelepipedal cells. The advantage of these schemes is that they proceed "forward" along the query ray, terminating at the first polygon hit. These raytracing techniques are not yet efficient enough for use in interactive real-time simulations.

The availability of fast graphics hardware has facilitated algorithms that generate some *potentially visible set* or PVS [1] of polygons with object-space culling methods, then solve the hidden-surface problem for this set in screen-space. One such approach involves intersecting a view cone with an octree-based spatial subdivision of the input [13]. Although this method always generates a superset of the visible polygons, it has the undesirable property that it can report as visible an arbitrarily large part of the scene when, in fact, only a tiny portion can be seen (Figure 1). The algorithm may also have poor average case behavior for scenes with high average depth complexity, i.e., with many viewpoints for which a large number of overlapping polygons paint the

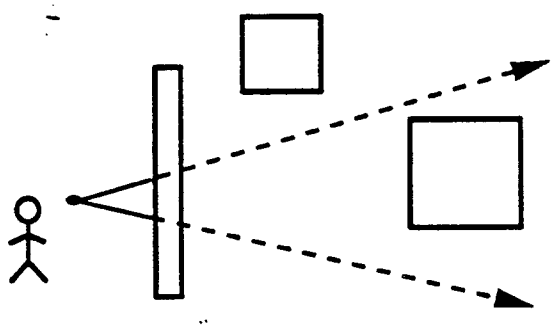


Figure 1: Cone-octree culling: the entire model can be reported visible.

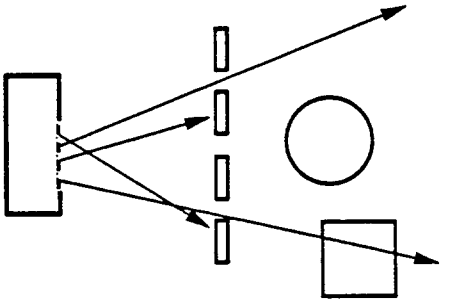


Figure 2: Ray casting: the circular object is not reported visible.

same screen pixel(s).

Another hardware-based method estimates visibility using *discrete sampling*, after spatial subdivision. Conceptually, rays are cast outward from a stochastic, finite point set on the boundary of each spatial cell. Polygons hit by the rays are included in the PVS for that cell [1]. This approach can *underestimate* the cell's PVS by failing to report some visible polygons (Figure 2). In practice, an extremely large number of rays must be cast to overcome this problem.

An object-space overestimation method proposed in [1] finds *portals*, or non-opaque regions, in otherwise opaque model elements, and treats these area light sources. Opaque polygons in the model then cause *shadow volumes* [9] to arise with respect to these light sources. Parts of the model inside the combined shadow volumes can be marked invisible for any observer on the originating portal. The portal-polygon occlusion algorithm has not found use in practice due to implementation difficulties and high computational complexity [1,2].

We contend that for a given model, rather than casting shadows and removing those objects inside them, it is more efficient to propagate *light* from a source cell through a sequence of portals, and *include* in the source's PVS only those objects reached by light rays originating at the source. Moreover, we argue that ignoring small polygons as potential occluders seems worthwhile, since they generally don't obscure much of the generalized observer's view, and their inclusion might greatly increase the computational complexity of our algorithms.

One early researcher proposed a spatial subdivision scheme that did no precomputation, but computed cells visible to the eye by projecting cell portals onto the view plane and solving the generalized polygon clipping problem as each portal was encountered [18]. This approach is roughly equivalent to our computation of eye-to-cell visibility, except that our offline stabbing line computation generally allows the real-time search to terminate more quickly, since it can be known *a priori* that no sightline exists to a candidate cell.

3 Conceptual Overview

3.1 Input

Our system requires as input a detailed 3D model, represented as a collection of *objects*, each of which is comprised of one or more *polygons* organized in a hierarchy of detail [7,12]. The architectural model is processed by a series of filters that correct coincident polygons, improper face intersections, inconsistent face orientations, etc. The model is then "populated" with detailed descriptions of furniture and other objects that might be found in a real building. During subdivision, however, we ignore these detailed polygons; they are considered non-occluding and typically do not greatly decrease visibility. Treating any polygon as non-occluding (i.e., ignoring it) is allowable within our framework, since we seek to compute a visibility *overestimate*; removing a potential occluder can never decrease visibility.

We employ figures depicting both two- and three-dimensional data throughout the paper. However, all of the algorithms we describe have been fully implemented for a real three-dimensional model, as we describe later in the paper. Also, we note that, although many of our examples are framed in terms of architectural walkthrough simulations, the visibility computations we describe here may serve to accelerate other rendering or simulation processes, for example ray-tracing and radiosity methods, flight simulators, and object-space animation and shadowing algorithms.

3.2 Subdivision Requirements

The visibility algorithms we describe assume the existence of a spatial subdivision that consists of *convex cells*, and that supports *point location*, *portal enumeration* on cell boundaries, and *neighbor finding*. Any spatial subdivision supporting these "primitive" operations is suitable for our purposes.

3.3 Subdivision Method

The input or *scene data* consists of n axial faces (non-axial faces are not considered for the purposes of subdivision). We use a k-D tree [5] for spatial subdivision, each node of which is a parallelepipedal *extent* whose sides are parallel to the principal planes. If a node is not a tree leaf, its extent contains the union of the extents of its children, recursively. A balanced k-D tree supports logarithmic-time point location and linear-time neighbor queries.

The k-D tree root cell's extent is initialized to the bounding box of the input, and splitting planes are recursively introduced along the major opaque elements in the model, namely the walls, door frames, floors, and ceilings (details are given in [25]). The subdivision terminates when all sufficiently large, axial opaque elements in the

model are coplanar with an axial boundary plane of at least one subdivision leaf cell. Although the subdivision criteria we use are not optimal under any objective metric, we have found that they do yield a tree whose cell structure reflects the "rooms" of our architectural model. We have recently become aware of an optimal $\Theta(n^{\frac{3}{2}})$ time and space algorithm that produces a minimal-size subdivision [22]; we are implementing this algorithm to evaluate its appropriateness for our system.

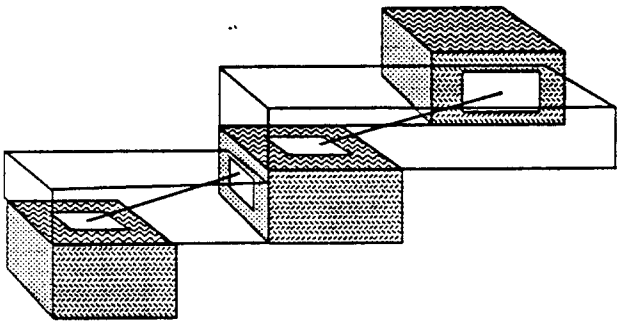


Figure 3: Axial cells and portals in three dimensions.

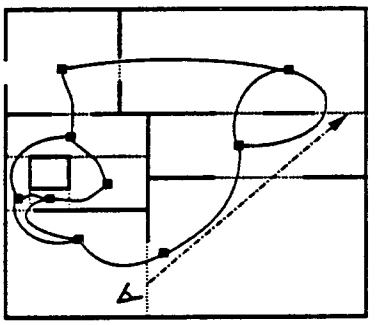


Figure 4: A final subdivision, with portals, adjacency graph, and a sightline.

After subdivision, cell *portals* (i.e., the transparent portions of shared boundaries) are identified and stored with each leaf cell, along with an identifier for the neighboring cell to which the portal leads (Figure 3). Enumerating the portals in this way amounts to constructing an *adjacency graph* over the leaf cells of the subdivision; two leaves (vertices) are adjacent (share an edge) if and only if there is a portal connecting them (Figure 4). All the visibility computations to be described exploit the adjacency graph data structure.

3.4 Visibility Preprocessing

Determining cell-to-cell visibility in three dimensions is substantially harder than in two dimensions. We present some algorithms for computing this visibility that are of practical use. Conceptually, we treat each cell of our subdivision as a light source, and trace light outwards to find potentially visible objects. The approach constructs successively "finer-grain" overestimations of the polygons visible to an increasingly restricted observer, with the constant goal that the overestimations are usefully modest, i.e., not overly large in comparison to the observer's true visibility, and can be tractably computed.

We refine the cell-based visibility computation to the level of individual objects by constructing "antipenumbral" regions of light emitted from a virtual light source on the boundary of a given cell. Only cells or objects reached by light rays from this source are visible from the source. To distinguish between visible and invisible objects, we compute a "visibility funnel" of light that is narrowed whenever a portal

is encountered. Because the narrowing of the volume is substantial, propagating light rather than shadow volumes seems conceptually attractive.

We have implemented all of the techniques described, for the case of *axial* polyhedral models (general polyhedral objects are allowed, but are considered non-occluding). Using a 250,000 polygon, fully furnished AutoDesk [4] model of one floor of a planned computer science building at Berkeley as a test data set (Figure 11-a), we have completed a first version of a system that supports interactive walkthroughs (we expect the fully-furnished seven-floor model to contain well over 1M polygons). We analyze the effectiveness of the visibility culling scheme using both software- and hardware-based polygon counting methods.

4 Precomputed Visibility Sets

By assuming an unrestricted observer able to move freely about a particular source cell, we can precompute which cells can possibly be seen from the source cell (the cell-to-cell visibility), and a superset of the objects potentially visible from the source cell (the cell-to-object visibility).

4.1 Cell-to-Cell Visibility

Once the spatial subdivision has been constructed, we compute and store *cell-to-cell visibility* for each leaf cell. This is the set of cells visible to a *generalized observer* able to look in all directions from any position within the cell. The cell-to-cell visibility for a cell C contains exactly those cells to which an unobstructed *sightline* leads from C. Such a sightline must be disjoint from any opaque elements and must intersect, or *stab*, a portal in order to pass from one cell to the next. Sightlines connecting cells that are not immediate neighbors must traverse a *portal sequence*, each member of which lies on the boundary of an intervening cell (as in Figure 3). We have implemented a procedure that finds sightlines through axial portal sequences, or determines that no such sightline exists, in $O(n \lg n)$ time, where n is the number of portals in the sequence [17]. Two $O(n)$ time stabbing algorithms are known [3,19], although neither has yet been implemented.

We compute the cell-to-cell visibility while constructing a *stab tree* for each leaf cell C of the subdivision (Figure 5). Each node of the stab tree corresponds to a cell visible from C; each edge of the stab tree corresponds to a portal stabbed as part of a portal sequence originating on a boundary of C. The stab tree is constructed incrementally using a constrained depth-first search (DFS) on the adjacency graph. As each cell is encountered by the DFS, it is effectively marked "visible" by its inclusion into the source cell's stab tree. For any source cell C, we say that a cell R is *reached* if R is in C's cell-to-cell visibility set.

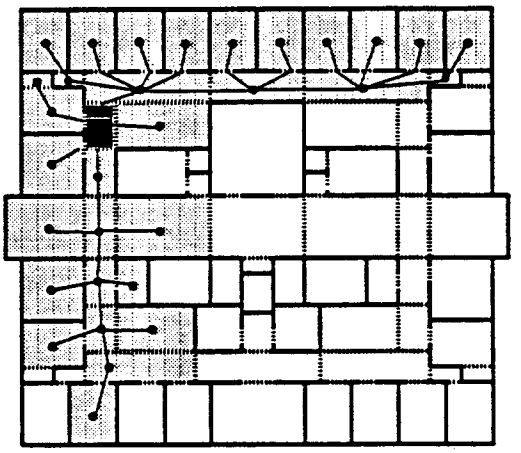


Figure 5: Cell-to-cell visibility and stab tree.

4.2 Cell-to-Object Visibility

Immediate neighbors of the source cell are entirely visible to a generalized observer within, since the observer's eyepoint can be placed on the shared portal. Cells farther from the source, however, are in general only partially visible to an observer in the source cell (Figure 6). This is due to the fact that, as the length of a portal sequence increases, the collection of lines stabbing the entire sequence typically narrows.

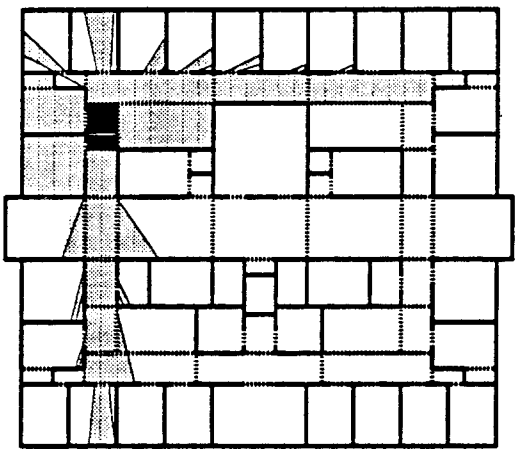


Figure 6: In general, only a fraction of most reached cells is visible to the source.

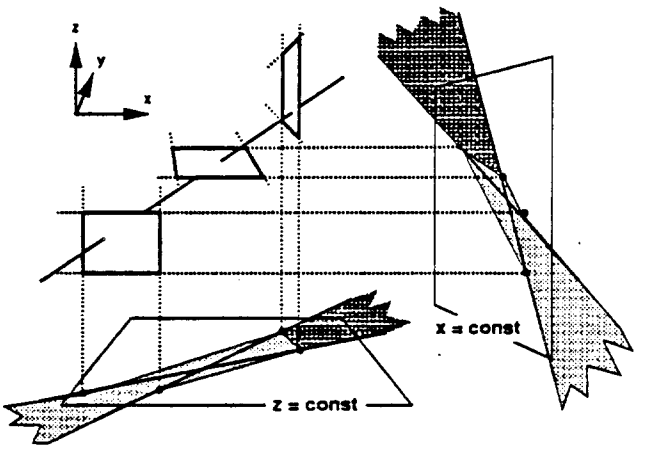


Figure 7: An axial portal sequence decomposed into axial "bowtie" constraints.

Casting the sightline search as a graph traversal yields a simple method for computing the partially visible portion of each reached cell. The traversal *orients* each portal encountered, since the portal is traversed in a known direction. In the plane, each portal contributes a "lefthand" and a "righthand" constraint to the set of sightlines stabbing the sequence. The result, after stepping through n portals, is a bowtie-shaped bundle of lines that stabs every portal of the sequence, and which "fans out" beyond

the final portal into an infinite wedge or *visibility funnel*. This wedge can then be clipped to the boundary of the reached cell (Figure 6).

Axial portals in three dimensions can be similarly oriented, then decomposed into at most three pairs of bowtie constraints, one from each collection of portal edges parallel to the x, y, and z axes. Figure 7 depicts one such portal sequence in 3-D, and the x axis-parallel and z axis-parallel constraints arising from its decomposition. These have been projected into "lefthand" and "righthand" points onto x = constant and z = constant planes, respectively. (The y constraints and projections are not shown.) These lines imply an *hourglass*-shaped volume in 3-D bounding all stabbing lines for the sequence. When this volume is cut at the entrance portal to the reached cell, a half-infinite volume that we call a *visibility funnel* is produced. This funnel is, in general, bounded by quadric surfaces; however, we can compute an efficient linear boundary enclosing it simply by intersecting three constant-size polyhedral wedges, one from each of the axis-parallel decompositions. We call the intersection of the funnel with the reached cell the source's *visibility volume* in the reached cell.

Suppose the stab-tree computation reaches a cell via some portal sequence of length n. We decompose the sequence into three sets of at most n axial lines each. From each set, we can find the two bounding planes in $O(n)$ time. We then compute, in constant time, the common intersection of the bounding planes with the parallelepipedal extent of the reached cell, using a 3-D convex hull algorithm. The resulting volume contains all lines stabbing the portal sequence, and completely or partially contains all objects visible from the source cell. Figure 11-b depicts a source cell, and the visibility volumes constructed within each reached cell, one for each occurrence of the cell in the source's stab tree.

We define *cell-to-object* visibility as the set of *objects* that can be seen by a generalized observer constrained to a given source cell C. For each reached cell R, we compute (and store with C) a superset of C's cell-to-object visibility in R by determining those objects in R incident on the visibility volume(s) originating at C. One special case exists: all objects in C's neighbor cells are tagged as visible from C without any visibility funnel computations.

Figure 8 depicts this process in two dimensions, using a simplified floorplan of our three-dimensional test model. The source's cell-to-object visibility (the filled squares in the figure) are associated with the source and reached cells in a compacted representation of the stab tree. Later, in the interactive walkthrough phase, this object set will be retrieved and culled dynamically based on the observer's position and view direction.

We are actively examining several issues concerning visibility volumes. Rather than polyhedral bounds, we might use the exact, quadratic funnel representation for object culling. Also, by expending more effort during the cell-to-object computation, we can produce even "finer-grain" knowledge of the constituent *polygons* potentially visible to the source, rather than merely the objects. Objects straddling the funnel boundary might be subdivided adaptively, and subportions found outside the funnel discarded. Similarly, any polygon whose negative half-space contains a portal cannot be

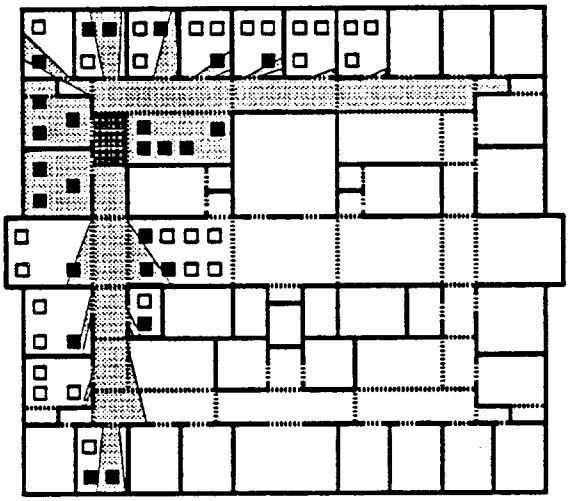


Figure 8: Cell-to-object visibility; filled squares represent potentially visible objects.

"seen" through that portal by any observer. Finally, although it does not yet appear computationally practical, the individual polygons could themselves be partitioned along exact funnel boundaries, and the resulting fragments classified.

5 Visibility Pruning During Walkthrough

Durting the interactive walkthrough phase, the precomputed visibility sets are further pruned based on the momentary view position and direction of the observer.

5.1 Eye-to-Cell and Eye-to-Object Visibility

The cell-to-cell visibility set contains those *cells* visible to a *generalized observer* in a particular cell; that is, one able to look simultaneously in all directions from all positions inside the cell. Similarly, the cell-to-object visibility set contains those *objects* potentially visible to the generalized observer.

During an interactive walkthrough phase, however, the observer is *restricted* to a known point, and has vision limited to a *view cone* emanating from this point, typically defined by left, right, top, and bottom clip planes. We can therefore cull the visible object set further. We defined the *eye-to-cell* visibility in [25] as the set of cells partially or completely visible to an observer with a specified view cone; here, we incorporate the notion of object visibility by extending the definition of eye-to-cell visibility to mean all *objects* incident on cells partially or completely visible to an observer with a specified view cone (e.g., all squares in Figure 5).

Conceptually, we define the *eye-to-object* visibility as the set of objects visible to an observer with a given view cone. We compute a superset of the eye-to-object visibility

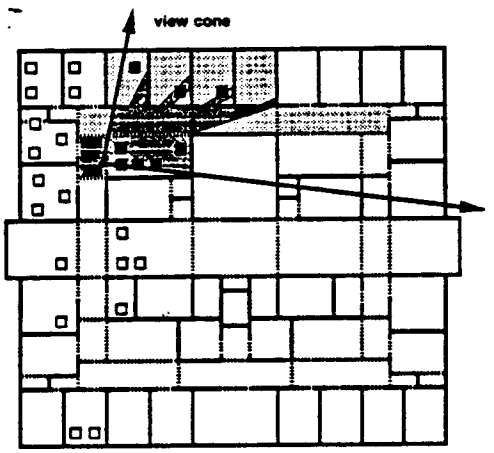


Figure 9: Eye-to-object visibility, among the filled squares of Figure 8.

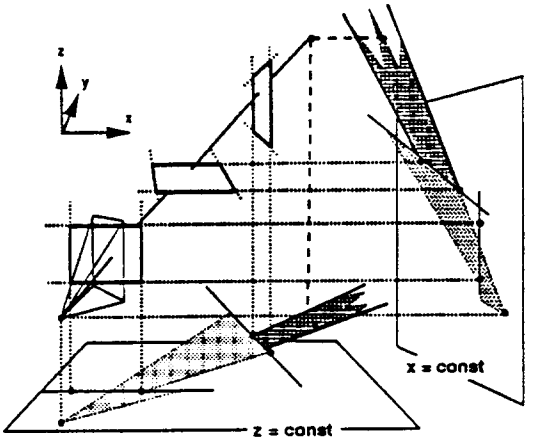


Figure 10: Decomposing portals into axial linear constraints.

by *intersecting* the eye-to-cell and cell-to-object sets (Figure 9). That is, we find the set of objects visible to the generalized observer, and incident on cells visible to the restricted observer. Note that this *intersection set* is *not* the set of objects stabbed by a sightline through the eye, as one might expect (i.e., the objects incident on the dark area in Figure 9). Finding this latter set could be an expensive computation; it cannot be precomputed, since its result depends on the eye position. On the other hand, the cell-to-object and eye-to-cell sets can be retrieve and intersected quickly, producing an object set that is typically a modest superset of the true visibility. For example, in Figure 9, only one square lies in a cell visible to the observer and can be seen from some point inside the cell containing the observer, but is not visible from the observer's current viewpoint.

For a cell to be visible, some portal sequence to that cell must admit a sightline that lies inside the view cone and contains the eyepoint. Retaining the stab tree $\mathcal{S}$ permits an efficient implementation of this sufficient criterion. We root a depth-first-search of the stab tree at the source cell, and upon encountering each portal, again decompose it into its constituent axial constraints (Figure 10). Projecting the set of feasible lines onto each principal plane yields a half-infinite "wedge" of lines, beginning at the plane of the final portal.

Two extremal lines are maintained on each of the three principal planes. As each portal is encountered, it is decomposed, and the set of extremal lines is suitably narrowed in the appropriate axial plane (if the extremal lines exclude the portal entirely, the DFS is immediately terminated). Together, the three resulting pairs of lines imply six constraint planes in three-space. These six planes are combined with the four planes defining the view pyramid to produce at most ten linear constraints on the existence of a stabbing line through the eye. These constraints can be cast as a three-dimensional linear program. If the k^{th} plane equation has normal $\mathbf{n}_k$, any viable stabbing line

through the eye have a direction vector $\mathbf{v}$ such that

$$\mathbf{n}_k \cdot \mathbf{v} \geq 0, \qquad \text{for all } k. \tag{1}$$

We examine the ten linear constraints for a feasible solution in constant time using a linear programming algorithm. If the linear program fails to find a stabbing line through the eye, the most recent portal is impassable, and the active branch of the DFS terminates.

The depth-first nature of the search ensures that portal sequences are assembled incrementally. Further, each newly encountered portal can be examined for a solution in constant time. Thus, a portal sequence of length n can be processed in $O(n)$ time, at the expense of a few scores of arithmetic operations per portal.

5.2 Frame-to-Frame Coherence and Data Management

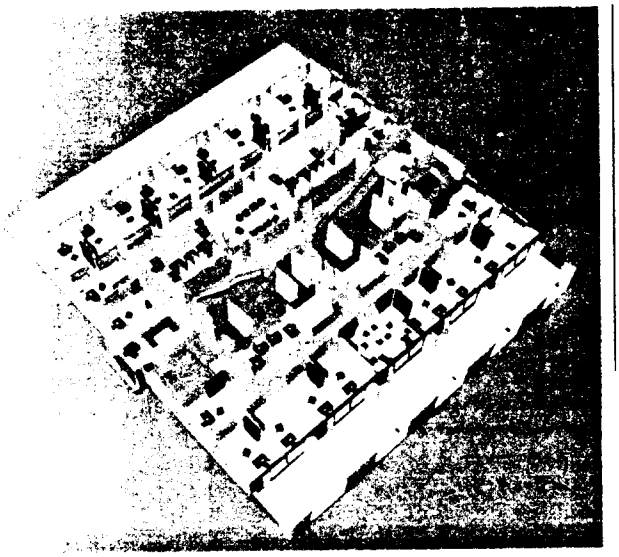
A real observer's motion through the simulation will present considerable *frame-to-frame* coherence, which can be exploited in the dynamic visibility computation. During smooth observer motion, the observer's view point will typically spend several frame-times in each cell it encounters. Consequently, we "cache" the stab tree for that cell in fast memory while the observer remains in the cell. Moreover, the cell adjacency graph allows us substantial predictive power over the observer's motion. We know, for instance, that a physically realistic observer exiting a known cell must leave through a portal, and emerge in the associated neighbor cell. Therefore, we keep an envelope around the cell currently containing the observer, and typically prefetch all polygons visible to those cells *before* the observer's arrival, minimizing or eliminating the waiting times associated with relatively high-latency disk storage.

An additional rendering speedup is achieved by representing individual objects at appropriate levels of detail. Currently, we use static distance information embedded in the adjacency graph to select successively lower "levels of detail" to display objects farther from the observer. This process is independent of visibility determination and has been described elsewhere [12].

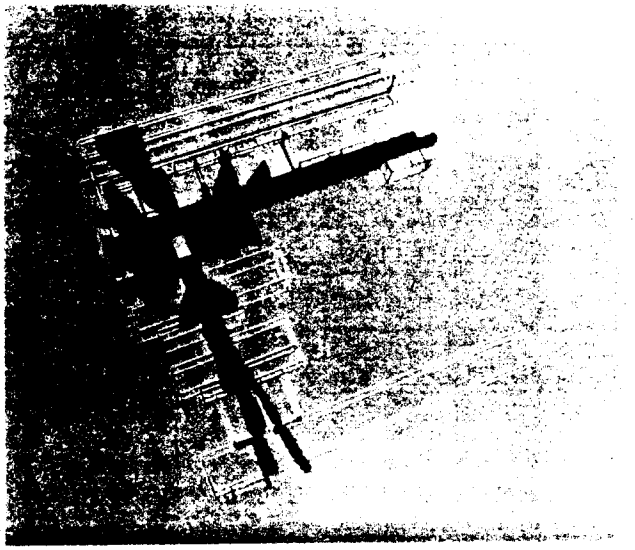
6 Experimental Results

We have implemented the algorithms described for axial 3-D environments. The subdivision and visibility computation routines comprise roughly ten thousand lines of C code, embedded in an interactive visualization program written for a dual-processor. 50-MIP, 10-MFLOPS graphics superworkstation, the Silicon Graphics 320 VGX, configured with 64 Mb of main memory and about one Gb of disk space. The application program is described more fully in [12].

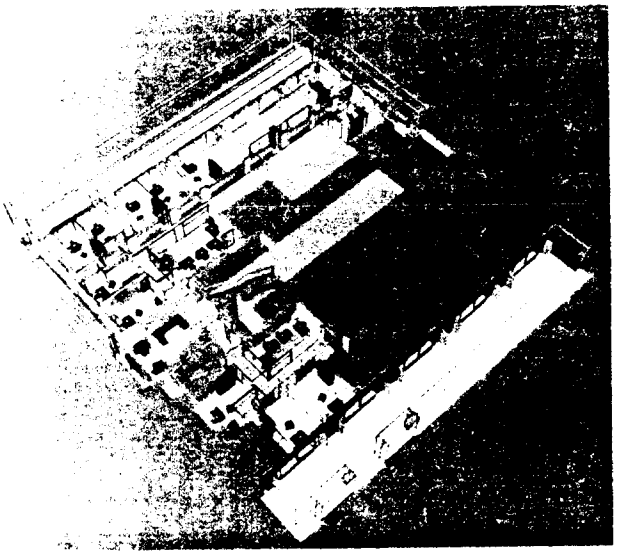
Our test model was one floor of a planned computer science building at UC Berkeley. modeled with roughly 250,000 polygons (Figure 11-*a*), requiring 48 Mb of storage.



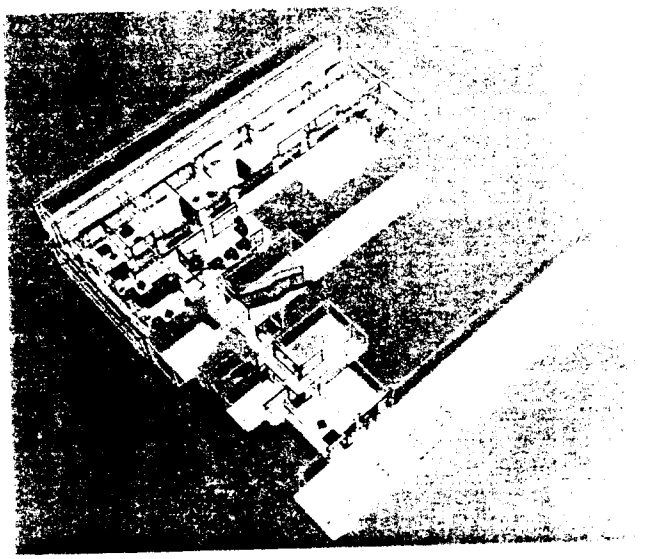
(a) The entire model of 250,000 polygons, drawn with backfacing polygons removed.



(b) One of the 1,280 source cells, its cell-to-cell visibility, and visibility volumes.



(c) The same source cell, and 109,000 polygons in its cell-to-cell visibility.



(d) The same source cell; the 40,475 polygons in its cell-to-object visibility set have been drawn.

Figure 11: An axial model with 250,000 faces, 2000 splitting faces, and 1280 leaf cells.

Subdividing the k-D tree to termination using the 2,000 major faces among the building walls, floors, and ceilings took 1 CPU minute, allocated 4 Mb of main memory, and produced 1280 leaf cells. Portal enumeration produced about 3600 portals; generating these and the cell adjacency graph required about 1 CPU minute. Populating all objects into the subdivision took about 45 CPU minutes and absorbed 42 Mb of memory. The cell-to-cell visibility computation was then performed for every leaf cell, requiring 6 CPU hours and 12 additional Mb of storage.

We exercised the visibility computations in two ways. First, we chose a "bad," highly cluttered sample viewpoint for which the true visibility and the stab tree complexity were considerably higher than average. Table 1 tabulates the number of polygons potentially visible from this viewpoint, under four culling methods, as well as the time (in seconds) required to render the polygons surviving each successive cull. It also includes a new and relevant measure that we call "surviving polygons"; this is the number of polygons that contribute at least one pixel to the final framebuffer. [At press time, we are still instrumenting our renderer to gather this statistic reliably. We found that we cannot compute this number by simply examining the frame buffer after rendering. Higher accuracy is required, and will be achieved by rendering successive portions of the view plane at high resolution].

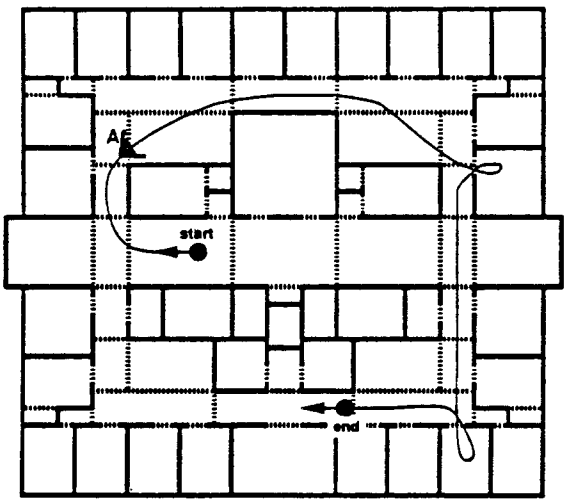
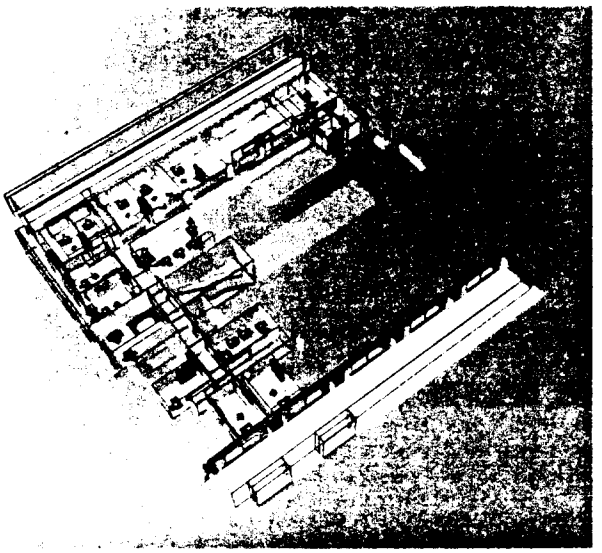


Figure 12: Test path through the building model.

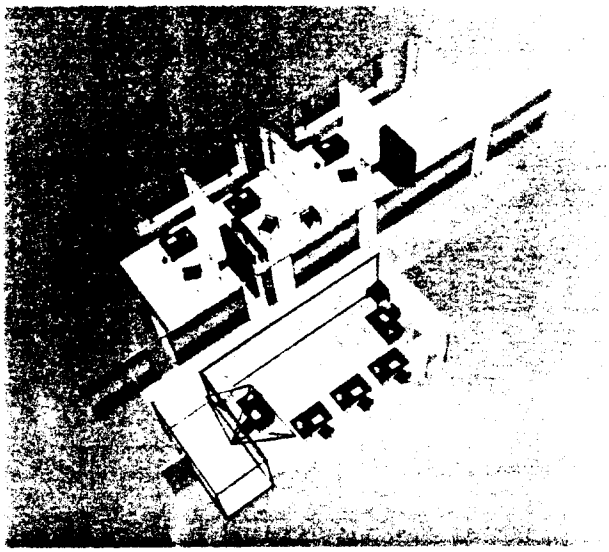
We also performed an average-case analysis by recording 300 viewpoints of an actual model walkthrough, and instrumenting our application to log rendering complexity and timing statistics along the path. The path is about 300 feet long, and a realistic physical walk along it should take approximately one minute (Figure 12). The results are shown in Table 2.

From the "bad" viewpoint A marked in Figure 13, the eye-to-object cull removed about 92% of the model polygons from the view of the observer, and accelerated rendering by a factor of about 11.

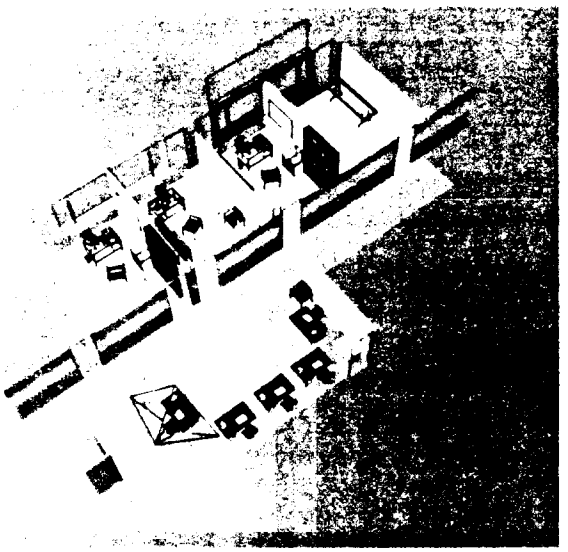
As expected, the eye-to-object cull performs better over the sample path, culling



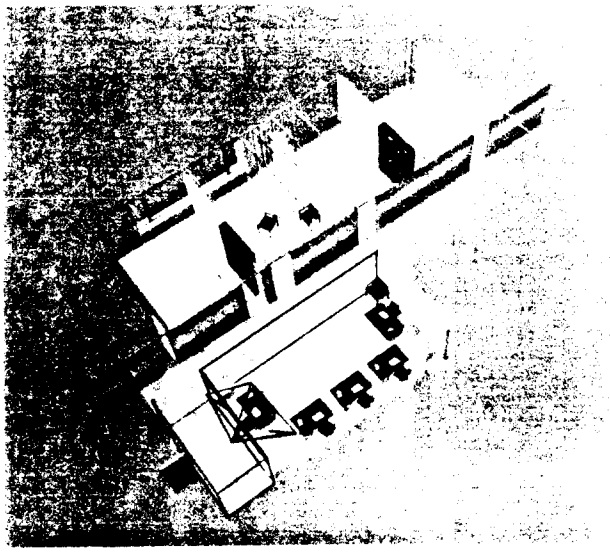
(a) Cell-to-cell (green) and eye-to-cell (blue) visibility sets for viewpoint A in text.



(b) Here, all 30,265 polygons incident on visible cells have been drawn.



(c) Polygons not in the cell-to-object visibility set are drawn in gray.



(d) The 18,927 polygons in the eye-to-object intersection set for this view.

Figure 13: Progressive visibility culling performed at viewpoint A.

Culling Method	# Objs. Drawn	# Polys Drawn	Draw Time (Sec.)	# Polys Survived	% of Model Drawn	Overest-imation (%)
Entire model	2,320	242,668	3.77	·	100%	·
Cell-to-cell	1,065	109,227	1.77	·	45%	·
Cell-to-object	558	40,475	0.65	·	17%	·
Eye-to-cell	241	30,265	0.52	·	12%	·
Eye-to-object	165	18,927	0.33	·	7.8%	·

Table 1: Visibility cull results from viewpoint A in Figure 12.

on average more than 95% of the model polygons before rendering, and producing a rendering-time speedup of about 17.

Culling Method	# Objs. Drawn	# Polys Drawn	Draw Time (Sec.)	# Polys Survived	% of Model Drawn	Overest-imation (%)
Entire model	2,320	242,668	3.66	·	100%	·
Cell-to-cell	778	78,475	1.22	·	32%	·
Cell-to-object	440	36,921	0.59	·	15%	·
Eye-to-cell	207	20,657	0.34	·	8.5%	·
Eye-to-object	141	13,701	0.23	·	5.6%	·

Table 2: Visibility cull results averaged over the test walkthrough.

7 General Three-Dimensional Models

Thus far, we have described visibility computations in terms of axial data sets. However, there is no conceptual obstacle to extending the techniques presented to general polyhedral environments. To process such environments, we need: 1) a suitable spatial subdivision and portal-finding method; 2) a general stabbing line algorithm; 3) a general visibility funnel computation; and 4) a general eye-to-cell determination. The remainder of this section describes these computations, all of which except the first have been implemented.

7.1 Spatial Subdivision and Portal Enumeration

The BSP-tree [10] data structure, with some straightforward modification, satisfies the requirements of Section 3.2 and may be suitable as a spatial subdivision technique for general environments. The leaf cells of the BSP tree are convex polyhedra, since they are the common intersection of a set of 3-D halfspaces. Point location and neighbor finding are simple operators in this context, and portal enumeration requires only a robust package for CSG operations on planar polygons such as that implemented and described in [1]. Finally, a $\Theta(n^2)$ time algorithm exists for generating BSP trees over n polygonal faces in three dimensions [21], although it has not yet been implemented to our knowledge.

In this general environment, portals are no longer axial rectangles, but are instead planar non-convex regions formed by (convex) cell boundaries minus unions of opaque faces on the boundary planes. General non-convex portals can be accommodated by partitioning them into convex fragments. Alternatively, any portal can be replaced with the 2-D convex hull of its vertices. Increasing the area of a portal can only increase the computed cell-to-cell visibility estimation, ensuring that it remains a superset of the actual visibility.

7.2 Stabbing Portal Sequences

Sightlines may be found by stabbing sequences of oriented convex polygons (Figure 14), in analogy to the axial three-dimensional case. To accomplish this, we have implemented a novel algorithm that determines sightlines through such sequences [27]. Briefly, the algorithm operates in a dual space in which the problem reduces to inspecting the edges of a five-dimensional polytope for intersections with a quadric surface, each intersection requiring constant time. For a sequence with n edges total, the polytope has worst-case complexity $O(n^2)$ [15], and can be computed in optimal $O(n^2)$ time by a randomized algorithm [8]. Thus, general portal sequences can be stabbed in $O(n^2)$ time.

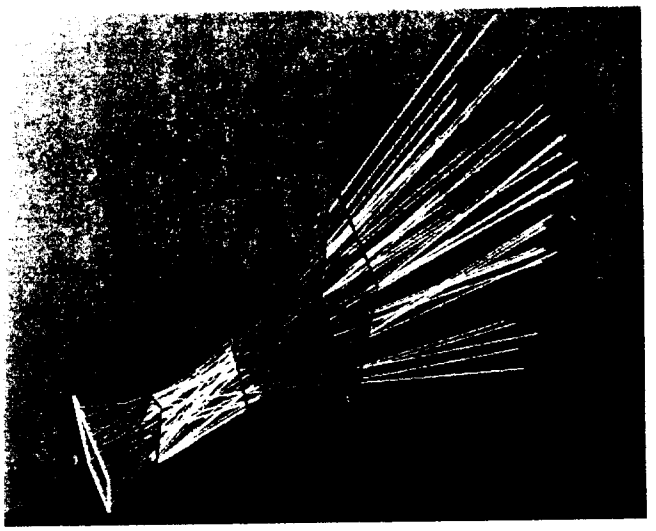


Figure 14: Stabbing a general sequence of convex portals in 3-D.

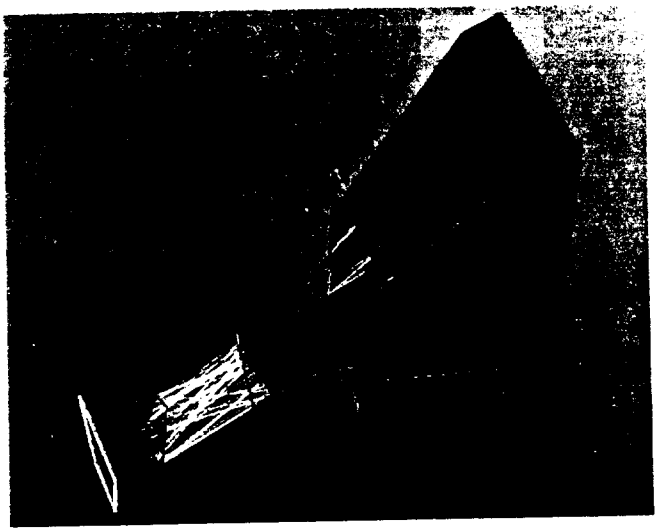


Figure 15: The antipenumbra of a sequence of convex portals in 3-D.

7.3 Cell-to-Object Visibility Funnels

Once the existence of a stabbing line establishes cell-to-cell visibility via some portal sequence, the cell-to-object visibility must be computed for the reached cell. When the portal sequence consists only of axial rectangles, we showed that the visibility funnel in the reached cell can be bounded with a polyhedron of constant complexity. If the portals are general convex polygons, however, the visible portion may be considerably more complex. In fact, this portion (which we call the *antipenumbra* cast by the first portal of the sequence) is in general non-convex and non-connected, and bounded by portions of quadric surfaces (Figure 15). We have implemented an $O(n^2)$ time algorithm that computes the antipenumbra of a general portal sequence with total complexity n [26]. Briefly, the algorithm maps the boundary computation into a linear, high-dimensional space and computes the boundary in that space subject to a single effective quadratic constraint, then remaps the high-dimensional solution to its quadratic three-space form.

7.4 Eye-to-Cell Visibility

Assuming a rectangular display for rendering, culling against a three-dimensional view pyramid is a direct extension of the axial culling methods we described. When the observer's position is known, each portal edge contributes a linear constraint (i.e., a plane through the eye and the edge, oriented to contain the portal interior) on the eye-to-cell visibility. The view pyramid implies four additional linear constraints; one each for the left, right, top, and bottom clipping planes. The observer sees through a given portal sequence if and only if the combined halfspaces admit a feasible point other than the eye. Determining such a point is a linear-time linear programming problem (Section 5.1, Equation 1). Thus, having built up a portal sequence of total edge complexity n (including the most recently encountered portal), the newest portal can be examined for an eye-centered stabbing line in $O(n)$ time. Moreover, incrementally examining a

series of portals with complexity n (by checking sequences of length 1,2, etc.) requires $O(n^2)$ time.

7.5 Comparisons

It would be tedious to catalogue in prose the conceptual and practical algorithmic improvements and extensions that distinguish this work from our previous effort [25]. Table 3 summarizes the advances; entries in italic signify the inclusion of the technique in [25], whereas entries in bold signify techniques introduced or proposed for use in this paper.

Visibility Technique	*Axial 2-D*	Axial 3-D	**Polyhedral 3-D**
Spatial Subdivision	*k*-D *tree* (*I*)	*k*-D *tree, no size bound* (I) *k*-D tree, $O(n\sqrt{n})$ size (P [22])	BSP tree, $O(n^2)$ size (P [21])
Stabbing (Cell-to-cell)	$O(n)$ (*I*)	$O(n \lg n)$ (I) $O(n)$ (P [3,19])	$O(n^2)$ (I [27])
Funnels (Cell-to-object)	*n/a (no objects)*	$O(1)$ size, $O(n)$ time **polyhedral bounds** (I)	$O(n^2)$ time **exact bounds** (I [26])
Eye-to-cell (Per portal)	$O(1)$ (*I*)	$O(1)$ (I)	$O(n)$ (I)

Table 3: Complexity and status of implemented (I) or proposed (P) techniques.

We classify the techniques as *implemented* (I) if they are actually part of a working installation in our system, or *proposed* (P) if they are described in the theoretical literature and we can reasonably expect them to be implemented in the near future. The latter class of techniques includes two linear-time rectangle-stabbing algorithms recently presented in [3] and [19] (both improvements over our previous $O(n \lg n)$ algorithm [17]), as well as efficient BSP-tree algorithms for the axial and general cases described in [22] and [21]. Finally, we note that our current implementation eliminates a restrictive assumption from our earlier work that polygon coordinates fall on a floating-point "grid"; a cleaner model and improved modeling and filtering tools have made the grid unnecessary.

8 Discussion

The methods described here are particularly appropriate for scenes with somewhat restricted "true" visibility (e.g., many architectural models). However, if the model has

many "holes", many distinct portals will be produced along cell boundaries, confounding cell-to-cell and cell-to-object computations with a combinatorially explosive set of sightlines and halfspaces. This problem can be ameliorated by allowing at most one portal to exist per cell boundary, either by subdividing cells or by coalescing portals.

It may occur that subdivision on the scene's major structural elements alone does not sufficiently limit cell-to-cell visibility. In this instance, further refinement of the spatial subdivision might help if it indeed reduces visibility, or hurt if it leaves visibility unchanged but increases the combinatorial complexity of finding sightlines. We may avoid this dilemma by exploiting the hierarchical *coherence* inherent in the spatial subdivision and its infused visibility information: after a leaf cell is subdivided, its children can see only a subset of the cells seen by their parent, since no new exterior portals are introduced (and the childrens' freedom of vision is reduced). Thus each child's sightline search is heavily constrained by its parent's portal/visibility list. Typically, the subdivision will further restrict eye-to-cell visibility during the walkthrough phase. We are studying the potential of adaptive cell and object subdivision, based on the results of the cell-to-cell and cell-to-object visibility precomputations.

We are optimistic about the use of such cell-based visibility algorithms in large architectural models. One can see intuitively that the complexity of most architectural models tends to "top out" at a level determined more by local effects (e.g., furnishings, realistic detail, room style) than by global effects (e.g., the total size of the model). For instance, from most points inside a typical apartment or office, an observer can not visually determine whether the apartment is a singleton or part of a block, or whether the office is a single floor or part of a huge skyscraper.

Applying the visibility computations described here to general polygonal scenes will require a considerable amount of computational and storage capacity, but appears achievable for moderately complex general environments with existing hardware.

Conclusion

We have implemented and analyzed an efficient and effective visibility preprocessing and query algorithm for real, axial architectural models in three dimensions. The algorithm's effectiveness depends on a decomposition of the models into rectangular or parallelepipedal cells in which significant parts of cell boundaries are opaque, on average.

The cell-based visibility determination relies on an efficient search for sightlines connecting pairs of cells through non-opaque portals. Once cells are linked to a given source, a more exacting analysis discards objects invisible from the source. We show that, when relevant portal sequences are retained, determining viewpoint-based visibility for axial, three-dimensional models can be reduced to a series of constant-sized linear programming problems. Finally, we present an extension of the axial-based algorithm to the general case of an arbitrary collection of polygons.

We present empirical evidence of rendering speedups for an axial three-dimensional environment derived from the working model of a planned building. The visibility computation can be performed at reasonable preprocessing and storage costs, and, for most viewpoints, significantly reduces the number of polygons that must be processed by the renderer. Tests on our model showed average rendering speedups of about a factor of thirty for a single floor of the building. As we expand the model to incorporate all seven planned floors, we expect the speedup factor to become even higher, since there is little visibility between floors. That is, we expect to be able to negotiate the complete, fully furnished building simulation as smoothly as we can move about a single floor.

Acknowledgments

Our special thanks go to Jim Winget, who has always held an active, supportive role in our research. Tom Funkhouser and Delnaz Khorramabadi did an enormous amount of work modeling the database and building the walkthrough application program. In particular, Tom's help in instrumenting the application was indispensable. Allan Wilks and Allen McIntosh supplied code to compute d-dimensional simplicial Delaunay decompositions. Michael Hohmeyer contributed much valuable insight and an implementation of Raimund Seidel's randomized linear programming algorithm.

Finally, we gratefully acknowledge the support of the State of California's MICRO program and the support of Silicon Graphics, Inc. We particularly thank Paul Haeberli for his help in preparing the color plates.

References

[1] John M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations.* PhD thesis, UNC Chapel Hill, 1990.

[2] John M. Airey, John H. Rohlf, and Frederick P. Brooks Jr. Towards image realism with interactive update rates in complex virtual building environments. *A CM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24(2):41–50, 1990.

[3] Nina Amenta. Finding a line traversal of axial objects in three dimensions. In *Proc. 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (to appear)*, 1992.

[4] AutoDesk, Inc. Autocad reference manual release 10, 1990.

[5] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[6] B. Chazelle and L.J. Guibas. Visibility and intersection problems in plane geometry. In *Proc. 1st ACM Symposium on Computational Geometry*, pages 135–146, 1985.

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

[8] Kenneth L. Clarkson, Kurt Melhorn, and Raimund Seidel. Four results on randomized incremental constructions. *In preparation*, 1991.

[9] Frank C. Crow. Shadow algorithms for computer graphics. *Computer Graphics (Proc. SIGGRAPH '77)*, 11(2):242–248, 1977.

[10] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics (Proc. SIGGRAPH '80)*, 14(3):124–133, 1980.

[11] Akira Fujimoto and Kansei Iwata. Accelerated ray tracing. *Computer Graphics: Visual Technology and Art (Proc. Computer Graphics Tokyo '85)*, pages 41–65, 1985.

[12] Thomas A. Funkhouser, Carlo H. Séquin , and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *To appear in Proc. 1992 Workshop on Interactive 3D Graphics*, 1992.

[13] Benjamin Garlick, Daniel R. Baum, and James M. Winget. Interactive viewing of large geometric databases using multiprocessor graphics workstations. *SIGGRAPH '90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation)*, 1990.

[14] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[15] B. Grünbaum. *Convex Polytopes*. Wiley-Interscience, New York, 1967.

[16] John E. Hershberger. *Efficient Algorithms for Shortest Path and Visibility Problems*. PhD thesis, Stanford University, 1987.

[17] Michael E. Hohmeyer and Seth J. Teller. Stabbing isothetic rectangles and boxes in $O(n \lg n)$ time. Technical Report UCB/CSD 91/634, Computer Science Department, U.C. Berkeley, 1991. Also to appear in *Computational Geometry: Theory and Applications*, 1992.

[18] C.B. Jones. A new approach to the 'hidden line' problem. *The Computer Journal*, 14(3):232–237, 1971.

[19] N. Megiddo. Stabbing isothetic boxes in deterministic linear time. *Personal communication to Nina Amenta*, 1991.

[20] Joseph O' Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[21] Michael S. Paterson and F. Frances Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete and Computational Geometry*, 5(5):485–503, 1990.

[22] M.S. Paterson and F.F. Yao. Optimal binary space partitions for orthogonal objects. In *Proc.* 1^{st} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 100–106, 1989.

[23] W. H. Plantinga and C. R. Dyer. An algorithm for constructing the aspect graph. In *Proc.* 27^{th} *Annual IEEE Symposium on Foundations of Computer Science*, pages 123–131, 1986.

[24] Michael Ian Shamos and Franco P. Preparata. *Computational Geometry: an Introduction.* Springer-Verlag, 1985.

[25] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proc. SIGGRAPH '91)*, 25(4):61–69, 1991.

[26] Seth J. Teller. Computing the antipenumbra cast by an area light source. Submitted to *Computer Graphics (Proc. SIGGRAPH '92);* also available as UC Berkeley Computer Science Dept. Technical Report UCB/CSD 91/666, 1991.

[27] Seth J. Teller and Michael E. Hohmeyer. Stabbing oriented convex polygons in randomized $O(n^2)$ time. In , 1992. Submitted to *Proc.* 8^{th} *Annual ACM Symposium on Computational Geometry.*

[28] Gert Vegter. The visibility diagram: a data structure for visibility problems and motion planning. In *Proc.* 2^{nd} *Scandinavian Workshop on Algorithm Theory*, pages 97–110, 1990.