Visibility Preprocessing For Interactive Walkthroughs

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

Abstract

The number of polygons comprising interesting architectural models is many more than can be rendered at interactive frame rates. However, due to occlusion by opaque surfaces (e.g., walls), only a small fraction of a typical model is visible from most viewpoints.

We describe a method of visibility preprocessing that is efficient and effective for axis-aligned or *axial* architectural models. A model is subdivided into rectangular *cells* whose boundaries coincide with major opaque surfaces. Non-opaque *portals* are identified on cell boundaries, and used to form an *adjacency graph* connecting the cells of the subdivision. Next, the *cell-to-cell* visibility is computed for each cell of the subdivision, by linking pairs of cells between which unobstructed *sightlines* exist.

During an interactive *walkthrough* phase, an observer with a known position and *view cone* moves through the model. At each frame, the cell containing the observer is identified, and the contents of potentially visible cells are retrieved from storage. The set of potentially visible cells is further reduced by culling it against the observer's view cone, producing the *eye-to-cell visibility*. The contents of the remaining visible cells are then sent to a graphics pipeline for hidden-surface removal and rendering.

Tests on moderately complex 2-D and 3-D axial models reveal substantially reduced rendering loads.

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*.

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

[‡]Computer Science Department, Berkeley, CA 94720

1 Introduction

Interesting architectural models of furnished buildings may consist of several million polygons. This is many more than today's workstations can render in a fraction of a second, as is necessary for smooth interactive walkthroughs.

However, such scenes 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 scene can be spatially subdivided into *cells*, and the model partitioned into sets of polygons attached to each cell. Approximate visibility information can then be computed offline, and associated with each cell for later use in an interactive rendering phase. This approximate information must contain a superset of the polygons visible from any viewpoint in the cell. If this "potentially visible set" or PVS [1] excluded some visible polygon for an observer position, the interactive rendering phase would exhibit flashing or holes there, detracting from the simulation's accuracy and realism.

1.1 Visibility Precomputation

Several researchers have proposed spatial subdivision techniques for rendering acceleration. We broadly refer to these methods as "visibility precomputations," since by performing work offline they reduce the effort involved in solving the hidden-surface problem. Much attention has focused on computing *exact* visibility (e.g., [5, 12, 16, 20, 22]); that is, computing an exact description of the visible elements of the scene data for every qualitatively distinct region of viewpoints. Such complete descriptions may be combinatorially complex and difficult to implement [16, 18], even for highly restricted viewpoint regions (e.g., viewpoints at infinity).

The binary space partition or BSP tree data structure [8] obviates the hidden-surface computation by producing a back-to-front ordering of polygons from any viewpoint. This technique has the drawback that, for an n-polygon scene, the splitting operations needed to construct the BSP tree may generate $O(n^2)$ new polygons [17]. Fixed-grid and octree spatial subdivisions [9, 11] accelerate ray-traced rendering by efficiently answering queries about rays propagating through ordered sets of parallelepipedal cells. To our knowledge, these ray-propagation techniques have not been used in interactive display systems.

Given the wide availability of fast polygon-rendering hardware [3, 14], it seems reasonable to search for simpler, faster algorithms which may *overestimate* the set of visible polygons, computing a *superset* of the true answer. Graphics hardware can then solve the

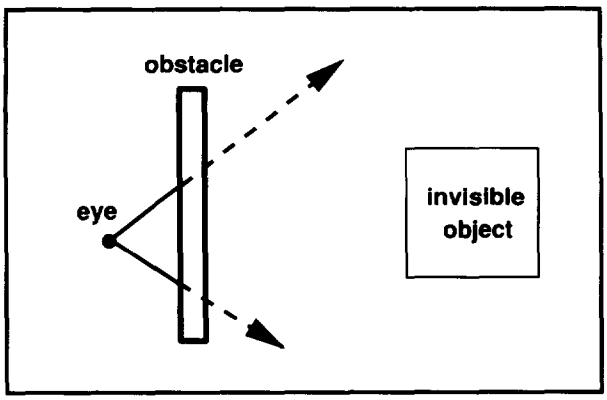


Figure 1: Cone-octree culling: the boxed object is reported visible.

hidden-surface problem for this polygon superset in screen-space. One approach involves intersecting a view cone with an octree-based spatial subdivision of the input [10]. This method 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 depth complexity; i.e., many viewpoints for which a large number of overlapping polygons paint the same screen pixel.

Another overestimation method involves finding *portals*, or non-opaque regions, in otherwise opaque model elements, and treating these as lineal (in 2-D) or areal (in 3-D) light sources [1]. Opaque polygons in the model then cause *shadow volumes* [6] to arise with respect to the light sources; those parts of the model inside the shadow volumes can be marked invisible for any observer on the originating portal. This portal-polygon occlusion algorithm has not found use in practice due to implementation difficulties and high computational complexity [1, 2].

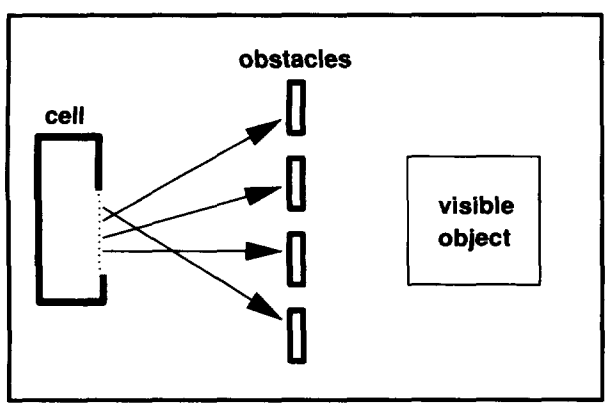


Figure 2: Ray casting: the boxed object is not reported visible.

A third approach 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 visible polygons (Figure 2). In practice, an extremely large number of rays must be cast to overcome this problem.

1.2 Overview

This paper describes a new approach to spatial subdivision and the visibility problem. The scene space is subdivided along its major opaque features; small, detailed scene elements are considered "non-occluding" and are ignored. After subdivision, a maximal set of *sightlines* is found from each cell to the rest of the subdivision. A novel aspect of our algorithm is that sightlines are not cast from discrete sample locations. Instead, *cell-to-cell visibility* is established if a sightline exists from any point in one cell to any point in another. As a consequence, the cells reached by sightlines provably contain a superset of the PVS for any given cell.

The data structure created during this gross visibility determination is stored with each cell, for use during an interactive walkthrough phase. The cell-to-cell visibility can be further dynamically culled against the *view cone* of an observer, again producing a reliable superset of the visible scene data, the *eye-to-cell visibility*. The detailed data contained in each visible cell, along with associated normal, color, texture data etc., are passed to a hardware renderer for removal of hidden surfaces (including, crucially, those polygons invisible to the observer). The two-fold model pruning described admits a dramatic reduction in the complexity of the exact hidden-surface determination that must be performed by a real-time rendering system.

We describe the spatial subdivision along major structural elements in Section 2, and the cell-to-cell visibility computation in Section 3. Section 4 describes the additional culling possible when the position and viewing direction of the observer are known. Some quantitative experimental results are given in Section 5, based on an implementation for axial 2-D models. Section 6 describes work in progress toward a more general algorithm.

2 The Spatial Subdivision

2.1 Assumptions About Input

We make two simplifying assumptions. First, we restrict our attention to "faces" that are axial line segments in the plane; that is, line segments parallel to either the x- or y-axis. These admit a particularly simple subdivision technique, and are useful for visualization and expository purposes. Second, we assume that the coordinate data occur on a grid; this allows exact comparisons between positions, lengths, and areas. Relaxing either assumption would not affect the algorithms conceptually, but would of course increase the complexity of any robust implementation.

Throughout the paper we use example data suggestive of architectural floorplans, since realizing truly interactive architectural and environmental simulations is a primary goal of our research. However, we note that the methods we describe have a modular nature and can be used to accelerate a range of graphics computations, for example ray-tracing and radiosity methods, flight simulators, and object-space animation and shadowing algorithms.

2.2 Subdivision Requirements

We require that any spatial subdivision employed consist of *convex cells*, and support *point location*, *portal enumeration* on cell boundaries, and *neighbor finding*. We will demonstrate the algorithm's correctness for any such spatial subdivision. Its effectiveness, however, depends on the more subjective criterion that cell boundaries in the subdivision be "mostly opaque."

2.3 Subdivision Method

The input or *scene data* consists of n axial faces. We perform the spatial subdivision using a BSP tree [8] whose splitting planes contain the major axial faces. For the special case of planar, axial data, the BSP tree becomes an instance of a k-D tree [4] with $k = 2$. Every node of a k-D tree is associated with a spatial cell bounded by k half-open *extents* $[x_{0,min} \cdots x_{0,max}), \ldots, [x_{k-1,min} \cdots x_{k-1,max})$. If a k-D node is not a leaf, it has a *split dimension* s such that $0 \leq s < k$; a *split abscissa* a such that $x_{s,min} < a < x_{s,max}$; and *low* and *high child* nodes with extents equivalent to that of the parent in every dimension except $k = s$, for which the extents are $[x_{s,min} \ldots a)$ and $[a \ldots x_{s,max})$, respectively. 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 (Fig. 3-*a*). Each input face **F** is classified with respect to the root cell as:

- **disjoint** if **F** has no intersection with the cell;
- **spanning** if **F** partitions the cell interior into components that intersect only on their boundaries;
- **covering** if **F** lies on the cell boundary and intersects the boundary's relative interior;
- **incident** otherwise.

Spanning, covering, and incident faces, but not disjoint faces, are stored with each node. Clearly no face can be disjoint from the root cell. The disjoint class becomes relevant after subdivision, when a parent may contain faces disjoint from one or the other of its children.

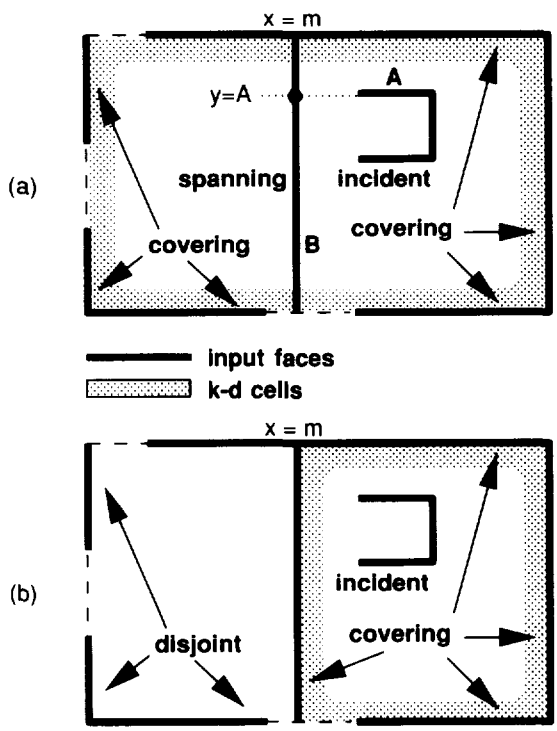


Figure 3: (a): A k-D tree root cell and input face classifications. (b): The right-hand cell and contents after the first split at $x = m$.

We say that face **A** *cleaves* face **B** if the line supporting **A** intersects **B** at a point in **B**'s relative interior (Fig. 3-*a*). We recursively subdivide the root node, repeatedly subjecting each leaf cell of the k-D tree to the following procedure:

- If the k-D cell has no incident faces (its interior is empty), do nothing;
- if any spanning faces exist, split on the median spanning face;
- otherwise, split on a sufficiently obscured minimum cleaving abscissa; i.e., along a face **A** cleaving a minimal set of faces orthogonal to **A**.

"Sufficiently obscured" means that the lengths of the faces at this abscissa sum to more than some threshold. If several abscissae are minimally cleaving, the candidate closest to that of the median face is chosen. Figure 4 depicts four minimally cleaving abscissae in x, marked as **0**; the median abscissa is marked as *.

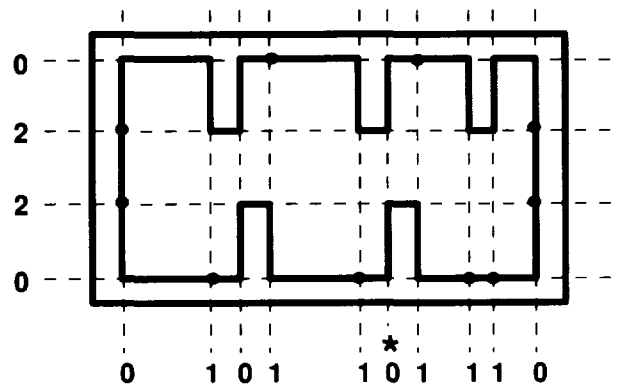


Figure 4: Cleaving abscissae (the split abscissa is marked *).

After each split, the contents of the parent node are reclassified as disjoint, spanning, covering, or incident with respect to each child, and all but the disjoint faces are stored with the child. Figure 3-*a* depicts a k-D tree root node; after this node is split at $x = m$, Figure 3-*b* shows the reclassification of the root's contents with respect to its high (i.e., right-hand) child.

This recursive subdivision continues until no suitable split abscissae are identified. We have found that these criteria, although somewhat naive, yield a tree whose cell structure reflects the "rooms" of the architectural models fairly well. Moreover, the splitting procedure can be applied quickly. At the cost of performing an initial $O(n \lg n)$ sort, the split dimension and abscissa can be determined in time $O(f)$ at each split, where f is the number of faces stored with the node.

After subdivision terminates, the *portals* (i.e., non-opaque portions of shared boundaries) are emumerated and stored with each leaf cell, along with an identifier for the neighboring cell to which the portal leads (Figure 5). Enumerating the portals in this fashion amounts to constructing an *adjacency graph* over the subdivision leaf cells, in which two leaves (vertices) are adjacent (share an edge) if and only if there is a portal connecting them.

3 Cell-to-Cell Visibility

Once the spatial subdivision has been constructed, we compute *cell-to-cell visibility* information about the leaf cells by determining

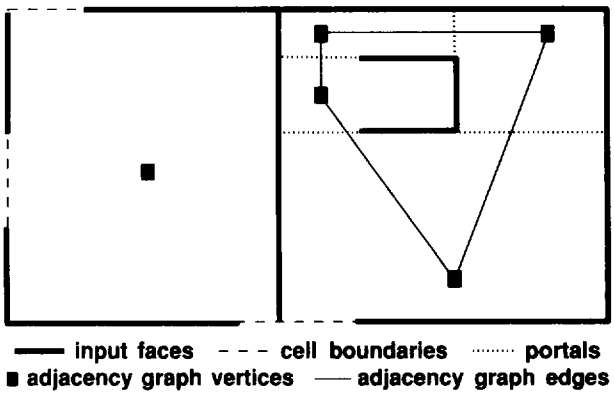


Figure 5: Subdivision, with portals and adjacency graph.

cells between which an unobstructed *sightline* exists. Clearly such a sightline must be disjoint from any opaque faces and thus 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. Observe that it is sufficient to consider sightlines originating and terminating on portals since, if there exists a sightline through two points in two cells' interiors, there must be a sightline intersecting a portal from each cell. The problem of finding sightlines between cell areas reduces to finding sightlines between line segments on cell boundaries.

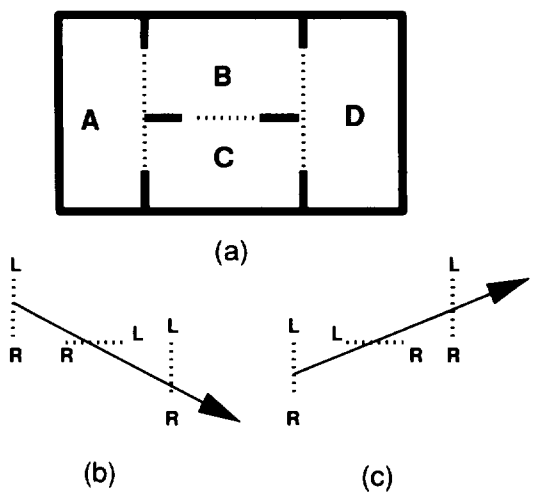


Figure 6: Oriented portal sequences, and separable sets **L** and **R**.

We say that a portal sequence *admits* a sightline if there exists a line that stabs every portal of the sequence. Figure 6 depicts four cells A, B, C, and D. There are four portal sequences originating at A that admit sightlines: $[A/B, B/C, C/D]$, $[A/C, C/B, B/D]$, $[A/B, B/D]$, and $[A/C, C/D]$, where P/Q denotes a portal from cell P to cell Q. Thus A, B, C, and D are mutually visible.

3.1 Generating Portal Sequences

To find sightlines, we must generate candidate portal sequences, and identify those sequences that admit sightlines. We find candidate portal sequences with a graph traversal on the cell adjacency graph. Two cells P and Q are *neighbors* if their shared boundary is not completely opaque. Each connected non-opaque region of this shared boundary is a portal from P to Q. Given any starting cell C for which we wish to compute visible cells, a recursive depth-first search (DFS) of C's neighbors, rooted at C, produces candidate portal sequences. Searching proceeds incrementally; when a candidate portal sequence no longer admits a sightline (according to the criterion described below), the depth-first search on that portal sequence terminates. The cells reached by the DFS are stored in a *stab tree* (see below) as they are encountered.

3.2 Finding Sightlines Through Portal Sequences

The fact that portal sequences arise from directed paths in the subdivision adjacency graph allows us to *orient* each portal in the sequence and find sightlines easily. As the DFS encounters each portal, it places the portal endpoints in a set **L** or **R**, according to the portal's orientation (Figure 6). A sightline can stab this portal sequence *if and only if the point sets* **L** *and* **R** *are linearly separable*; that is, iff there exists a line S such that

$$\begin{aligned} S \cdot L &\geq 0, \quad \forall L \in \mathbf{L} \\ S \cdot R &\leq 0, \quad \forall R \in \mathbf{R}. \end{aligned} \tag{1}$$

For a portal sequence of length m, this is a linear programming problem of $2m$ constraints. Both deterministic [15] and randomized [19] algorithms exist to solve this linear program (i.e., find a line stabbing the portal sequence) in linear time; that is, time $O(m)$. If no such stabbing line exists, the algorithms report this fact.

3.3 The Algorithm

Assume the existence of a routine *Stabbing_Line*(**P**) that, given a portal sequence **P**, determines either a stabbing line for **P** or determines that no such stabbing line exists. All cells visible from a source cell C can then be found with the recursive procedure:

```
Find_Visible_Cells (cell C, portal sequence P, visible cell set V)
    V = V ∪ C
    for each neighbor N of C
        for each portal p connecting C and N
            orient p from C to N
            P' = P concatenate p
            if Stabbing_Line (P') exists then
                Find_Visible_Cells (N, P', V)
```

Figure 7 depicts a spatial subdivision and the result of invoking *Find_Visible_Cells* (cell I, $\mathbf{P}$ = empty, $\mathcal{V} = \emptyset$). The invocation stack can be schematically represented as

```
Find_Visible_Cells (I, P = [ ], V = ∅)
  Find_Visible_Cells (F, P = [I/F], V = {I})
    Find_Visible_Cells (B, P = [I/F, F/B], V = {I, F})
    Find_Visible_Cells (E, P = [I/F, F/E], V = {I, F, B})
      Find_Visible_Cells (C, P = [I/F, F/E, E/C], V = {I, F, B, E})
  Find_Visible_Cells (J, P = [I/J], V = {I, F, B, E, C})
    Find_Visible_Cells (H, P = [I/J, J/H1], V = {I, F, B, E, C, J})
    Find_Visible_Cells (H, P = [I/J, J/H2], V = {I, F, B, E, C, J, H})
```

The last line shows that the cell-to-cell visibility $\mathcal{V}$ returned is $\{I, F, B, E, C, J, H\}$.

The recursive nature of *Find_Visible_Cells()* suggests an efficient data structure: the *stab tree* (Figure 8). Each node or vertex of the stab tree corresponds to a cell visible from the source cell (cell I in Fig. 7). Each edge of the stab tree corresponds to a portal stabbed as part of a portal sequence originating on a boundary of

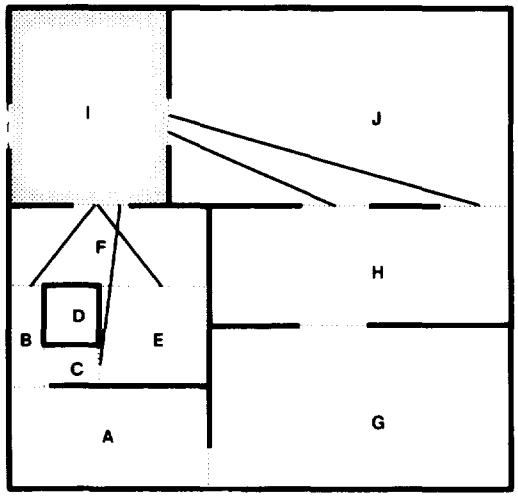


Figure 7: Finding sightlines from I.

the source cell. Note that the stab tree is isomorphic to the call graph of *Find_Visible_Cells()* above, and that leaf cells are included in the stab tree once for each distinct portal sequence reaching them. A stab tree is computed and stored with each leaf cell of the spatial subdivision; the cell-to-cell visibility is explicitly recoverable as the set of stab tree vertices.

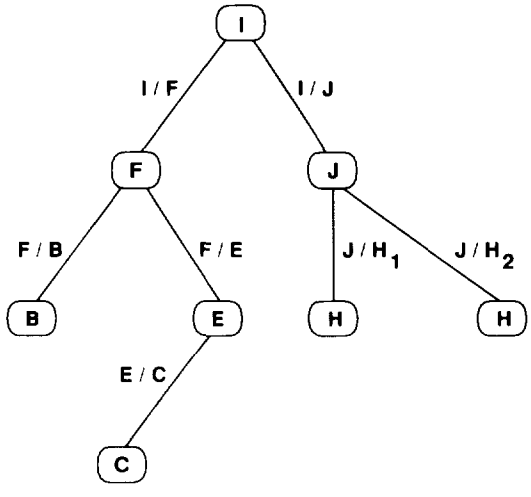


Figure 8: The stab tree rooted at I.

3.4 Algorithmic Complexity

Since linear programs are solvable in linear time, *Find_Visible_Cells* adds or rejects each candidate visible cell in time linear in the length of the portal sequence reaching that cell. Determining a useful upper bound on the total number of such sequences as a function of $|\mathcal{V}|$ seems challenging, as this quantity appears to depend on the spatial subdivision in a complicated way. However, for architectural models, we expect the length of most portal sequences to be a small constant, since most cells will not see more than a constant number of other cells. Were this not so, most of the model would be visible from most vantage points, and visibility preprocessing would be futile.

Our algorithm does not yet fully exploit the coherence and symmetry of the visibility relationship. Visibility is found one cell at a time, and the sightlines so generated are effectively useful only to the source cell. Later visibility computations on other cells do not "reuse" the already computed sightlines, but instead regenerate them from scratch. To see why reusing sightlines is not easily accomplished, consider a general cell with several portals. Many sightlines traverse this cell, each arriving with a different "history" or portal sequence. Upon encountering a cell, it may be more work for a sightline to check every prior-arriving sightline than it is for the new sightline to simply generate the (typically highly constrained) set of sightlines that can reach the cell's neighbors.

The algorithm as stated may require storage quadratic in the number of leaf cells (since, in the worst case, every leaf cell may see every other through many different portal sequences). In practice we expect the storage required to be linear in the number of leaf cells, with a constant close to the average portal sequence length. Nevertheless, we are seeking ways to combine all of the stab trees into a single, suitably annotated adjacency graph.

4 Eye-to-Cell Visibility

The cell-to-cell visibility is an upper bound on the view of an *unconstrained observer* in a particular cell; that is, one able to look simultaneously in all directions from all positions inside the cell. During an interactive walkthrough phase, however, the observer is at a known point and has vision limited to a *view cone* emanating from this point (in two dimensions, the cone can be defined by a view direction and field of view; in three dimensions, by the usual left, right, top, and bottom clip planes). We define the *eye-to-cell* visibility as the set of cells partially or completely visible to an observer with a specified view cone (Figure 9). Clearly the eye-to-cell visibility of any observer is a subset of the cell-to-cell visibility for the cell containing the observer.

4.1 Eye-to-Cell Culling Methods

Let O be the cell containing the observer, C the view cone, $\mathcal{S}$ the stab tree rooted at O, and $\mathcal{V}$ the set of cells visible from O (i.e., $\{O, D, E, F, G, H\}$). We compute the observer's eye-to-cell visibility by *culling* $\mathcal{S}$ and $\mathcal{V}$ against C. We discuss several methods of performing this cull, in order of increasing effectiveness and computational complexity. All but the last method yield an overestimation of the eye-to-cell visibility; that is, they can fail to remove a cell from $\mathcal{V}$ for which no sightline exists in C. The last method computes exact eye-to-cell visibility.

Disjoint cell. The simplest cull removes from $\mathcal{V}$ those cells that are disjoint from C; for example, cells E and F in Figure 9-*a*. This can be done in $O(|\mathcal{V}|)$ time, but does not remove all invisible cells. Cell G in Figure 9-*a* has a non-empty intersection with C, but is not visible; any sightline to it must traverse the cell F, which *is* disjoint from C. More generally, in the cell adjacency graph, the visible cells must form a single *connected component*, each cell of which has a non-empty intersection with C. This connected component must also, of course, contain the cell O.

Connected component. Thus, a more effective cull employs a depth-first search from O in $\mathcal{S}$, subject to the constraint that every cell traversed must intersect the interior of C. This requires time $O(|\mathcal{S}|)$, and removes cell G in Figure 9-*a*. However, it fails to remove G in Figure 9-*b*, even though G is invisible from the

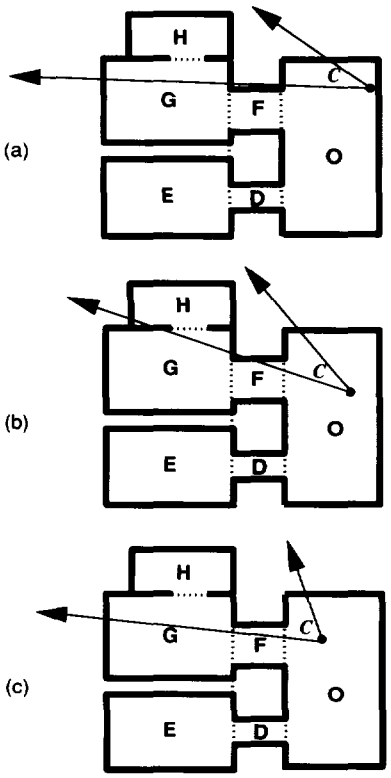


Figure 9: Culling O's stab tree against a view cone C.

observer (because all sightlines in C from the observer to G must traverse some opaque input face).

Incident portals. The culling method can be refined further by searching only through cells reachable via *portals* that intersect C's interior. Figure 9-*c* shows that this is still not sufficient to obtain an accurate list of visible cells; cell H passes this test, but is not visible in C, since no sightline from the observer can stab the three portals necessary to reach H.

Exact eye-to-cell. The important observation is that for a cell to be visible, some portal sequence to that cell must admit a sightline that lies *inside* C and *contains the view position*. Retaining the stab tree $\mathcal{S}$ permits an efficient implementation of this sufficient criterion, since $\mathcal{S}$ stores with O every portal sequence originating at O. Suppose the portal sequence to some cell has length m. As before, this sequence implies $2m$ linear constraints on any stabbing line. To these we add three linear constraints: one demanding that the stabbing line contain the observer's view point, and two demanding that the stabbing *ray* lie inside the two halfspaces whose intersection defines C (in two dimensions). The resulting linear program of $2m + 3$ constraints can be solved in time $O(m)$, i.e., $O(|\mathcal{V}|)$ for each portal sequence.

This final refinement of the culling algorithm computes exact eye-to-cell visibility. Figure 9-*c* shows that the cull removes H from the observer's eye-to-cell visibility since the portal sequence [O/F, F/G, G/H] does not admit a sightline through the view point.

During the walkthrough phase, the visible area (volume, in 3-D) can readily be computed from the stored stab tree. The visible area in any cell is always the intersection of that (convex) cell with one or more (convex) wedges emanating from the observer's position (Figure 10). The stab tree depth-first search starts at the source cell, and propagates outward along the stab tree. Upon passing through a portal, the wedge is either suitably narrowed by the portal's extrema (e.g., portal I/F in Figure 10), or completely eliminated if the wedge is disjoint from the portal (e.g., portal F/B in Figure 10). In this case, the DFS branch terminates, descending no further into the stab tree.

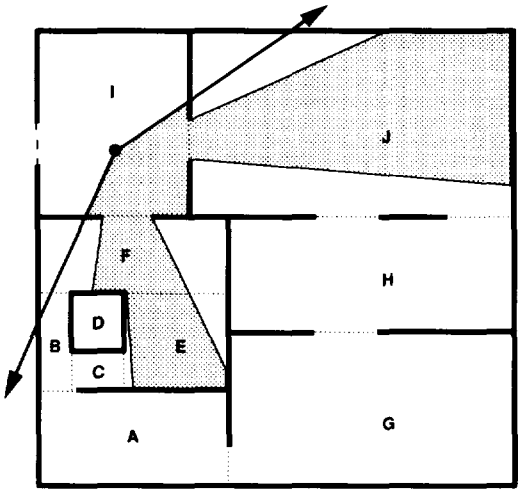


Figure 10: The view cone during the stab tree DFS.

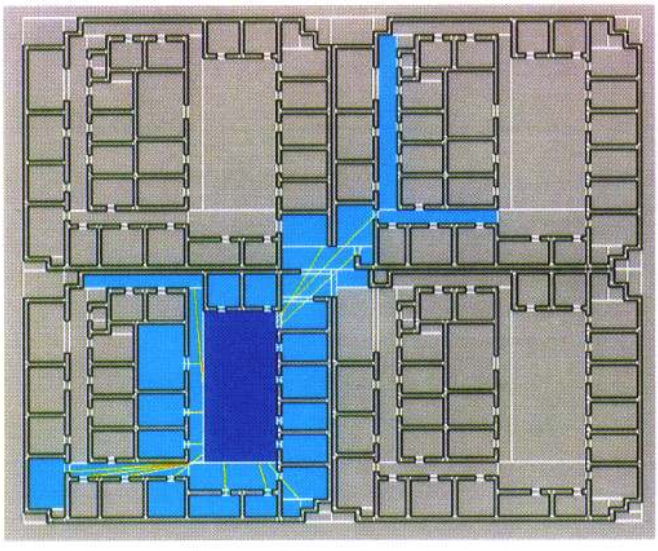
4.2 Frame-to-Frame Coherence

In practice, there is considerable *frame-to-frame* coherence to be exploited in the eye-to-cell visibility computation. During smooth observer motion, the observer's view point will typically spend several frame-times in each cell it encounters. Thus, the stab tree for that cell can be cached in fast memory as long as the observer remains in the cell. Moreover, the cell adjacency graph allows substantial predictive power over the observer's motion. For instance, an observer exiting a known cell must emerge in a neighbor of that cell. An intelligent walkthrough program might prefetch all polygons visible to that cell *before* the observer's arrival, minimizing or eliminating the waiting times associated with typical high-latency mass-storage databases.

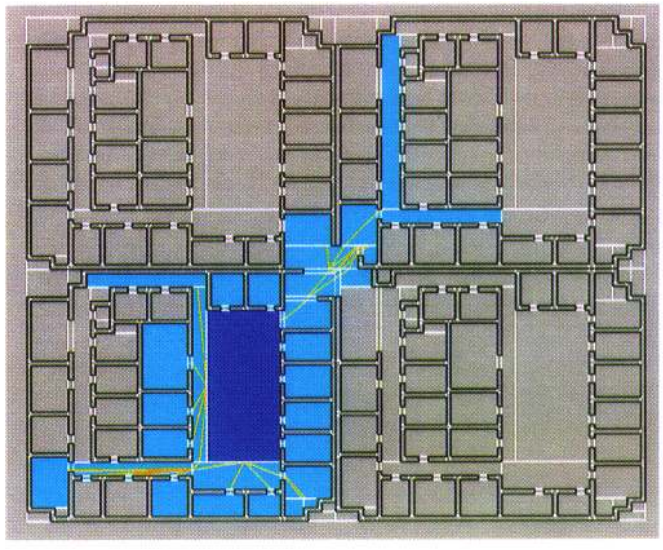
5 Experimental Results

We have implemented the algorithms described for 2-D axial environments, and all but the eye-to-cell computation for 3-D axial environments. The subdivision and visibility computation routines contain roughly five thousand lines of C language, embedded in an interactive visualization program written for a dual-processor, 50-MIP, 10-MFLOPS graphics superworkstation, the Silicon Graphics 320 GTX.

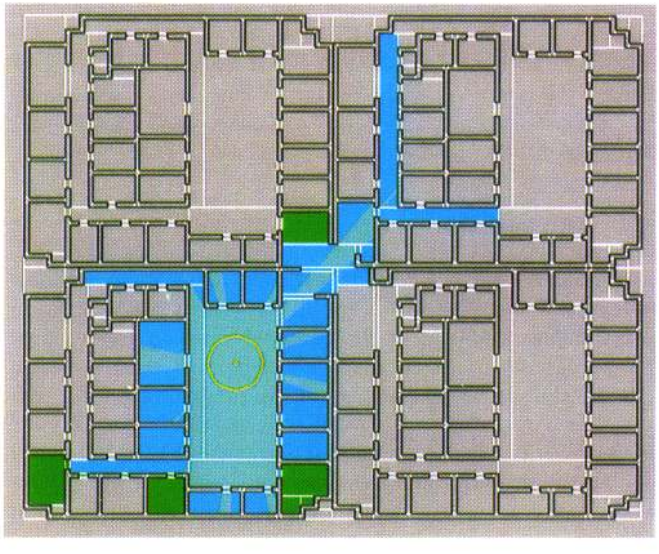
Our test model was a floorplan with 1000 axial faces (Figure 11-*a*). Subdividing the k-D tree to termination with the procedure of Section 2.3 required 15 CPU seconds, allocated 1 Mb of main memory, and produced about 700 leaf cells. Portal enumeration, creation of the cell adjacency graph, and the cell-to-cell visibility computation were then performed for every leaf cell. This required 30 CPU seconds and increased the memory usage to 2 Mb. Roughly 10,500 total stab tree vertices were allocated to store all of the 700 leaf cells' stab trees (Figure 11-*b*). Thus the average stab tree size was about 15.



(a) A source cell (dark blue), its cell-to-cell visibility (light blue), and stabbing lines.



(b) The source cell (dark blue), cell-to-cell visibility (light blue), and stab tree.



(c) An observer with a 360° view cone. The eye-to-cell visibility is shown in blue; the exact visible area is shown in blue-green. The green cells have been dynamically culled.

(d) The same observer, with a 60° view cone. The eye-to-cell visibility is shown in blue; the exact visible area is shown in blue-green. The green cells have been dynamically culled.

Figure 11: An axial model with roughly 1,000 faces (black), subdivided into about 700 spatial cells (white)

We empirically evaluated the efficacy of cell-to-cell visibility pruning and several eye-to-cell culling methods using the above floorplan. We performed 10,000 visibility queries at random locations within the model, with the view direction chosen randomly, and for both 360° and 60° view cones (Figures 11-*c* and 11-*d*). For every generated view cone, visibility was computed with each of the culling methods of Sections 3 and 4. The area of the potentially visible region was averaged over the random trials to produce the figures tabulated below. The quantities shown are generally sums of cell areas and are expressed as percentages of total model area. The last row displays the total area of the view cone's *intersection* with all cells reached by the stab tree DFS (e.g., the shaded areas in Figure 10).

culling method	360° view cone		60° view cone	
	vis. area	*reduction factor*	*vis. area*	*reduction factor*
none (cell-to-cell vis.)	8.1%	10x	8.1%	10x
disjoint cell	8.1%	10x	3.1%	30x
connected component	8.1%	10x	2.4%	40x
incident portals	8.1%	10x	2.2%	40x
exact eye-to-cell	4.9%	20x	1.8%	50x
exact visible area	2.1%	50x	0.3%	300x

6 Extensions and Discussion

We briefly discuss extensions of the visibility computation algorithms to three-dimensional scenes.

6.1 Three-Dimensional Models

Here we assume that all faces are rectangles whose normals and edges are parallel to the x, y, or z axis. Subdivision again proceeds with a k-D tree (and $k = 3$). The face classification and splitting criteria extend directly to three dimensions. Portals are no longer line segments, but are instead rectilinear non-convex regions formed by (rectangular) cell boundaries minus unions of covering faces.

There are at least two ways to accommodate these more general portals. First, given any set of non-convex portals, rectangular *large portals* may be created by computing the axial bounding box of the set. Replacing collections of portals (e.g., all portals through a boundary) with large portals can only increase the computed cell-to-cell visibility estimation, ensuring that it remains a superset of the true visibility.

A second alternative is to decompose each non-rectangular portal into rectangles. This approach should produce smaller potentially visible sets than the one above, since it does not overestimate portal sizes. However, this improved upper bound comes at the cost of increased combinatorial complexity, since many invocations of *Find_Visible_Cells* will be spawned in order to explore the more numerous portals.

In either event, sightlines are found by stabbing oriented rectangle sequences (Figure 12), in analogy to the two-dimensional case. To accomplish this, we have developed and implemented a novel algorithm that determines sightlines through rectangles [13]. Briefly, the algorithm operates in a dual space in which the problem reduces to performing a linear number of convex polygon-polygon intersections, each requiring logarithmic time [7]. The algorithm finds a stabbing line through n oriented, axis-aligned rectangles, or determines that no such stabbing line exists, in $O(n \lg n)$ time.

Assuming a rectangular display for rendering, culling against a three-dimensional view pyramid is a direct extension of the planar

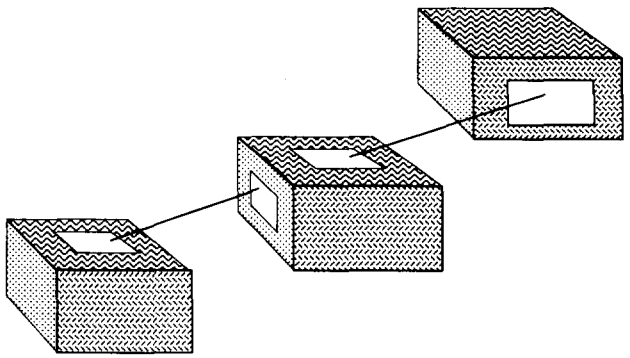


Figure 12: Stabbing a sequence of rectangular portals in 3-D.

culling methods described earlier. When the observer's position is known, each portal edge contributes a linear constraint 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. Thus, computing eye-to-cell visibility in three dimensions again reduces to a linear-time linear programming problem.

Generalizing the visibility computations described here to non-axial scenes appears to pose problems both conceptual and technical in nature. First, suitable techniques must be found for decomposing large collections of general polygons into convex spatial subdivisions, generating an appropriate cell adjacency graph, and enumerating the portals of each subdivision cell. Second, efficient algorithms are needed for stabbing portal sequences comprised of general polygons in three dimensions. We have made some headway against the latter problem by developing a randomized $O(n^2)$ algorithm that stabs sequences of n oriented convex polygons [21].

6.2 Discussion

The methods described here are particularly appropriate for input with somewhat restricted "true" visibility, such as that occurring in many architectural models. However, adversarially chosen input can produce unbalanced spatial subdivision trees under our naive criteria, slowing basic operations on the subdivision. Input with a large number of portals per cell boundary (for example, walls with tens or hundreds of windows) may confound the cell-to-cell visibility algorithm with a combinatorially explosive set of sightlines. Large portals ameliorate this problem, at the possible cost of decreasing the usefulness of the attained (overlarge) visibility estimates.

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). Again, there is an ameliorating factor: when subdividing a leaf cell, 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. Moreover, the portals generated by the subdivision will generally restrict visibility during the walkthrough phase. We are studying the issue of how to subdivide spatial cells as a function of cell-to-cell visibility and cell data density.

Conclusion

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

The cell-based visibility determination relies on an efficient search for sightlines connecting pairs of cells through non-opaque portals. In two dimensions, this search reduces to a linear programming problem. Finding sightlines through portals in three dimensions is somewhat harder. We show that, when relevant portal sequences are retained, determining viewpoint-based visibility in both two and three dimensions also reduces to a linear programming problem.

We present some empirical evidence of rendering speedups for axial two-dimensional environments. The visibility computation can be performed at reasonable preprocessing and storage costs and, for most viewpoints, dramatically reduces the number of polygons that must be processed by the renderer.

Acknowledgments

Our special thanks go to Jim Winget, who has always held an active, supportive role in our research. We gratefully acknowledge the support of Silicon Graphics, Inc., and particularly thank Paul Haeberli for his help in preparing this material for submission and publication. Michael Hohmeyer contributed much valuable insight and an implementation of Raimund Seidel's randomized linear programming algorithm. Finally we thank Mark Segal, Efi Fogel, and the SIGGRAPH referees for their many helpful comments and suggestions.

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. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24(2):41–50, 1990.

[3] Kurt Akeley. The Silicon Graphics 4D/240GTX superworkstation. *IEEE Computer Graphics and Applications*, 9(4):239–246, 1989.

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

[5] 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.

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

[7] David P. Dobkin and Diane L. Souvaine. Detecting the intersection of convex objects in the plane. Technical Report No. 89-9, DIMACS, 1989.

[8] 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, July 1980.

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

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

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

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

[13] Michael E. Hohmeyer and Seth J. Teller. Stabbing isothetic rectangles and boxes in $O(n \lg n)$ time (in preparation).

[14] David Kirk and Douglas Voorhies. The rendering architecture of the DN10000VS. *Computer Graphics (Proc. SIGGRAPH '90)*, 24(4):299–307, August 1990.

[15] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal Computing*, 12:759–776, 1983.

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

[17] 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.

[18] W. H. Plantinga and C. R. Dyer. An algorithm for constructing the aspect graph. In *Proc. IEEE Symp. Foundations of Computer Science*, pages 123–131, 1986.

[19] Raimund Seidel. Linear programming and convex hulls made easy. In *Proc. 6th ACM Symposium on Computational Geometry*, pages 211–215, 1990.

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

[21] Seth J. Teller and Michael E. Hohmeyer. Stabbing oriented convex polygons in randomized $O(n^2)$ time (in preparation).

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