Chapter 9

Results: Soda Hall Data

9.1 Geometric Data Set

The test case for these algorithms is a complex geometric model of a planned computer science building. The original floorplan and elevation data were obtained in AutoCAD format [Aut90], but have since undergone substantial, and crucial, alteration [Kho91]. Some alterations were relevant to visibility determination, and these are described in what follows. Many were effected as team efforts by members of the Walkthrough research group.

After the elevation data and floorplan were integrated into a collection of 3D wall, floor, ceiling, door, and window polygons, the data was converted into a convenient format. We chose the Berkeley UniGrafix format [SSW83, SS91], since it is simple, human-readable, and has many associated utilities available in the form of textual filters. Moreover, oriented polygonal faces are central UniGrafix definitional primitives.

The articulated (but unpopulated) geometric model was then subjected to a series of automated *filters* and to some interactive editing, to attain a sensible form [Kho91]. Interpenetrating or overlapping polygons were adjusted or removed. Misoriented polygons (e.g., wall polygons whose face normals pointed into the wall interior, not into the room interior) were reversed. Cracks, or small gaps, in the model were filled by subtly adjusting nearby polygons. This filtering task is formidable, perhaps even ill-defined, for general polyhedral environments. However, it seems tractable for architectural environments which, after all, are intended to represent physically realizable structures.

The filtering task was a practical necessity, rather than a theoretical one, since omitting it would only have increased the running time of the visibility algorithms, without compromising their correctness. In practice, we observed that leaving cracks uncorrected noticeably increased the time spent in preprocessing, since each uncorrected crack (i.e., portal) became a conduit for many dynamic and online visibility tests that otherwise would not have occurred. In the sense of algorithmic correctness, the filtering operation is independent of any abstract notions of visibility determination. We have assumed, throughout, the availability of filtered input to the visibility computation module, and we perform no

systematic analysis of the possible time and storage penalties incurred by violations of the filtering goals.

9.1.1 Test Models and Test Walkthrough Path

We extracted two test cases from the geometric model. The "partial" model comprised only the sixth and seventh floors, and the roof, of the building. The "full" model was all five completed floors and the roof; its geometric data required almost three times as much storage as that of the partial model. We constructed the two models in order to analyze the effect of tripling the model data size on visibility storage requirements and computation times.

Our goal in instrumenting the algorithms was to probe their effectiveness under realistic workloads, rather than random or systematic queries. Consequently, we gathered data about the effectiveness of the visibility algorithms by recording a representative "test path" defined during an actual interactive walkthrough session. The test path wanders through the sixth and seventh floors of the model, through the atrium, into and out of several offices, out and in a window, out over the terraces, and through the open-air regions among the columns (Figure 9.2). The path spends considerable time in free-space surrounding the building. These regions challenge the more discriminating culling operations, since almost all portal sequences there admit eye-based sightlines. Consequently, the eye-based DFS spends considerable time traversing empty cells.

9.2 Implementation

9.2.1 Programming Methodology

The visibility algorithms described in this thesis have been implemented in the C language on a Silicon Graphics 340 GTX superworkstation. The implementation mirrors the progression of this thesis; the abstract operations were first implemented for 2D occluders, for axial 3D occluders, and finally for general 3D occluders.

These algorithms exist as part of a large collection of programs that support interactive graphical visualization and manipulation of geometric data. The development of each algorithm was facilitated by a "visual programming" framework in which visual, selectable instantiations were defined for each geometric data type, in order not only to display the behavior of the data, but to manipulate it directly, for example by use of an interactive input device. This implementation methodology had some surprising and beneficial aspects, and some noteworthy implications.

First, the notion of *displaying* geometric data objects as they are manipulated gives powerful clues as to the correct or incorrect behavior of associated algorithms. We found that if the computed solution "looked wrong," i.e., contradicted intuition, it generally *was* wrong, or at least merited further examination. Second, *manipulating* a geometric algorithm's input with an interactive device is a powerful method for exploring the space of possible inputs. This coverage of, for example, input degeneracies and boundary cases facilitates the development of extremely robust algorithms, because instances of exceptional input are effortlessly generated as test cases. Third, the easy availability of a graphics infrastructure for prototyping algorithms removed much of the onus of implementing complex geometric

Figure 9.1: The top two floors and roof of the geometric model. The test path (grey) snakes in and out of the building.

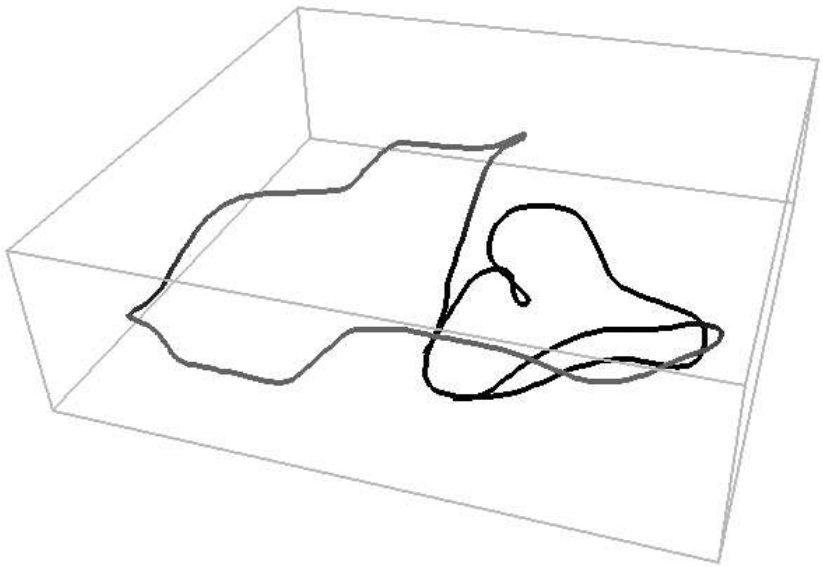


Figure 9.2: The test path, grey-scaled from low z-values (black) to high z-values (white).

algorithms.

9.2.2 Free-Space

In practice, our system encountered a difficulty that we termed the "free-space" problem. Define a *free-space* cell as one incident on no major occluders or detail objects; i.e., an empty cell all of whose boundaries are completely transparent. Two factors caused clumps of free-space cells to arise while subdividing the building model: first, non-local splitting caused by the early k-D tree subdivision planes; second, subdivision of large cells and bad-aspect cells (for example, large open areas such the balconies and atrium, or long corridors) to reduce their visibility to and from the remainder of the model.

The free-space problem arises because of conflicting demands on the subdivision method. First, it should exploit the presence of occluders wherever possible (e.g., by splitting along them). Cell boundaries should therefore be obscured by occluders. Second, the subdivision method should produce cells that have usefully small cell-to-cell visibilities, thus highly constraining the generalized observer, even when no occluders are present. This forces some splitting to occur in regions of little "nearby" occlusion.

One important future aspect of this research is the development of improved spatial subdivision techniques for both axial and general three-dimensional data. In the interim, however, we rely on ad-hoc techniques for handling problematic local cell configurations such as clusters of free-space cells.

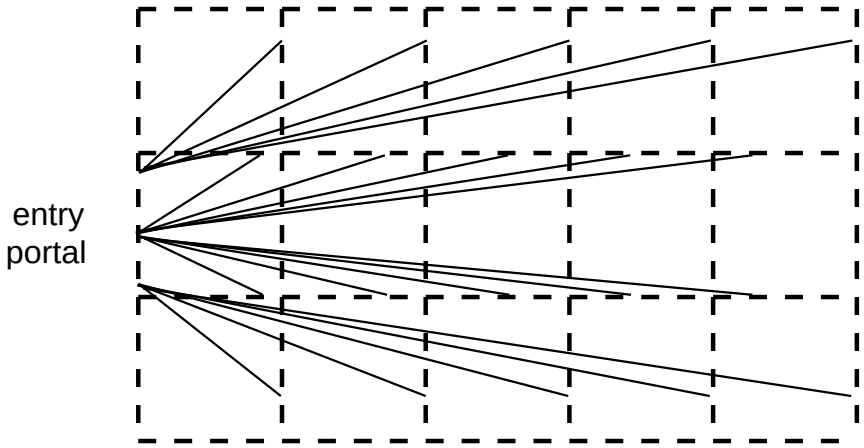


Figure 9.3: Naive search through free-space causes a combinatorial explosion. Dashed lines are portals; some of the sightlines computed are shown.

Clusters of free-space cells, treated naively, slow both the precomputation and query visibility phases. During precomputation, a sightline search encountering the cluster explores all combinatorial paths through it (Figure 9.3), expending time and storage to compute essentially no useful information (since no objects are populated in free-space cells, no cell-to-object visibility determinations can be made). During the query phase, the eye-to-cell search expends computational effort to find eye-based

sightlines, and manipulates dynamic stacks of halfspace constraints that need never be applied to discover eye-to-cell visibility. This search is wasteful in that it discovers no visible objects to be drawn; it is necessary, however, because otherwise the traversal algorithms would not be able to proceed beyond free-space cells to more typical cells containing visible objects. Consequently, we have developed a method which searches through a free-space cluster, and emerges from its perimeter, in time comparable to that required for searching a single cell.

Handling Free-Space Cells

We developed a method for handling free-space cells which is a straightforward generalization of the cell and adjacency graph, and of the notion of traversing such a graph. Moreover, in the case of sparsely occluded environments such as terrains, our method has the desirable feature that it degenerates to a purely spatial cull (cf. [GBW90]) – probably as well as one can do in such an environment without a fully general object-space hidden-surface algorithm.

We generalize the notion of a spatial cell and define a *metacell* as any connected cluster of *constituent* cells. The metacell is generally non-convex, and has a portal set containing all constituent cell portals that lead to a non-constituent cell. Note that this definition prevents a metacell portal from leading into itself or another metacell, a desirable property that simplifies the abstract operations defined over metacells. Point-locating into any constituent cell is equivalent to point-locating into the associated metacell.

The geometric notion of portals is unchanged. However, without cell convexity, it is no longer true that portals in general are mutually visible through their common cell interior (Figure 9.4). Consequently we associate, with each portal *entering* the cell, a list of *portal crossover* constraints which, given the metacell and an entry portal, catalogues those portals through which the metacell may be exited. The crossover constraints can be statically computed by finding all pairs of portals A and B that admit a stabbing line in the interior of the metacell, and can be done with the standard cell-to-cell DFS, constrained to traverse only constituent cell portals. Although this sub-problem may be combinatorially expensive, the advantage of constructing the portal crossover lists is that it need be done only *once* per metacell (e.g., when the cell is first encountered during visibility precomputation), rather than once each time any of the constituent cells is encountered, as in the unmodified scheme. Finally, note that the size of the portal-crossover table can be $O(p^2)$, where p is the number of constituent cell portals in the metacell.

Figure 9.4, depicts a sightline search that enters a metacell via portal A, but that is unable to leave the metacell via portals C or K since, to do so, the sightline would have to reverse itself (i.e., the portal pairs [A,C] and [A,K] do not admit sightlines). The stabbing line search further rules out exit via portals D or J because no sightline entirely within the metacell exists to either of them from A. Thus the portal crossover constraints for portal A comprise the remaining metacell portals, the set B, E, F, G, H, I, L.

Once the notion of a conforming subdivision has been generalized to include metacells, we need only generalize the abstract operation to be performed upon encountering a metacell in the precomputation and query phases. Recall that, in the precomputation phase, the sightline and cell-to-region DFS arrives at a cell with a set of halfspaces encoding the portal constraints encountered so far. When encountering a metacell, rather than recursing and searching through the constituent cells, we merely subject the

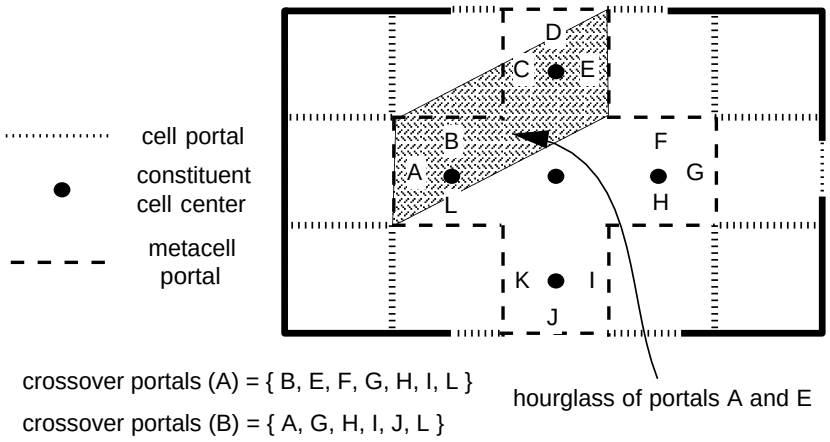


Figure 9.4: A metacell (bold dashed outline), with portal crossover constraints attached to each entry portal.

constituent cell *extents* to the cell-to-object computation as if they were objects, and continue the search using the portal crossover list, as a function of the entry portal (Figure 9.5). (The constituent cells, by definition, contain no occluders or objects to be found visible, and consequently can be ignored for the purposes of computing cell-to-cell or cell-to-object visibility.) Note that without the portal crossover constraints, this algorithm could recurse indefinitely by entering and leaving the same metacell in a single portal sequence.

Similarly, during the query phase, when the eye-to-cell and eye-to-region DFS arrives at a metacell, it subjects the constituent cells to the active halfspace constraints. Note that reporting the incident cells visible, and subjecting their contents to the eye-to-object test, will generally be unnecessary since the cells have no associated occluders or detail objects. Moreover, the portal crossover list can be used to recurse, through visible portals only, out of the metacell to surrounding cells. For example, in Figure 9.5, the constituent cell A is not visible to the observer, and the crossover list portal K should not be traversed.

Note that the portal crossover constraints serve not only to generalize the notion of cells to include non-convexity, but might also be useful in determining the effect of incident detail objects on inter-portal visibility within a given source cell (Figure 9.6). We have not fully explored this possibility.

9.3 Utility Metrics

Several *metrics* serve to evaluate the effectiveness of the visibility preprocessing and on-line culling scheme. These are empirical metrics, in that they describe quantities measured by instrumenting algorithms to record their storage needs, culling and drawing time, output size, visibility overestimation, and depth complexity, in units to be described below.

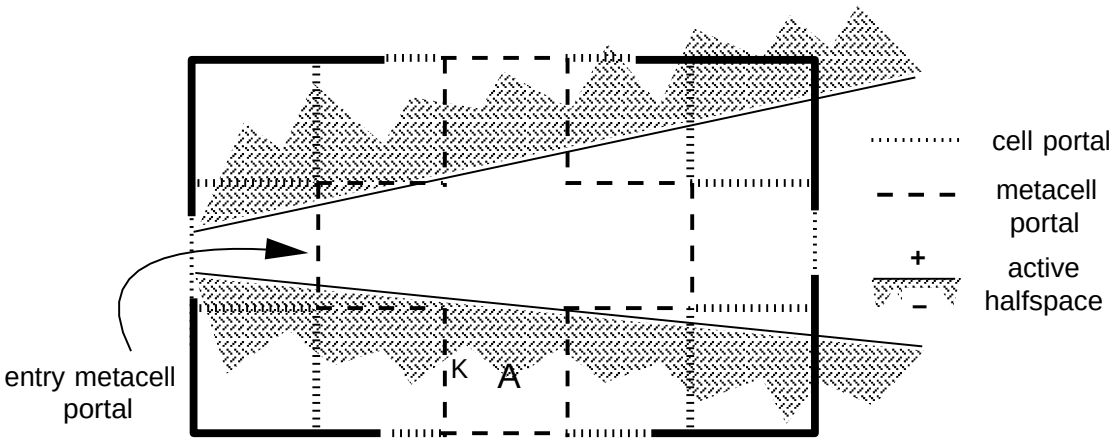


Figure 9.5: Encountering a metacell during a constrained DFS.

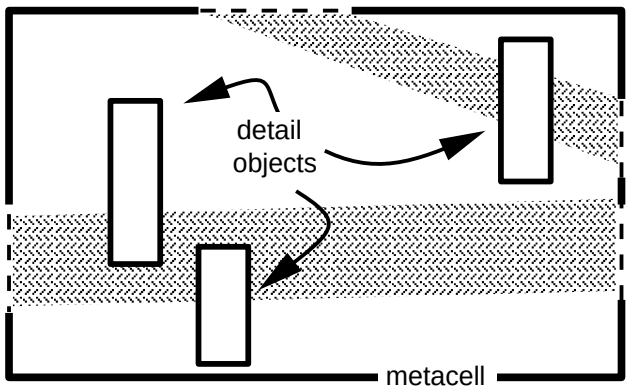


Figure 9.6: Detail objects can prohibit inter-portal visibility (constituent cells not shown).

The *storage overhead* metric quantifies the storage overhead incurred in computing and storing visibility information for a geometric model, expressed both in absolute terms and as a percentage of the storage required to store the model elements (the model "geometry" [FST92]) alone. We do not formulate an analogous *time overhead* metric, since it is difficult if not impossible to objectively weigh the value of off-line CPU resources expended in light of the fact the on-line simulation may be run indefinitely long for any given populated model with precomputed visibility.

The *culling time* and *drawing time* metrics quantify the time, in milliseconds, required for the culling operation (i.e., producing a collection of potentially visible objects), and the drawing operation (i.e., rendering the collection), as a function of the culling mode. The *frame time speedup* quantifies how much the visibility computation accelerated the rate of delivery of successive images to the framebuffer, compared to rendering using only the graphics hardware and no on-line culling. A frame was *cull-bound* if culling took longer than drawing, and *draw-bound* if drawing took longer than culling; the frame-time speedup is defined as the base (unculled) drawing time divided by the maximum of the culling and drawing times. (In practice, things are more complicated, since the culling and drawing processes may run in parallel and, depending on transport delay, may be almost entirely decoupled [Jon92, FST92].)

For all timing measurements, we simulated a machine with infinite memory (or, equivalently, a perfect database intermediary that incurred no swapping penalty) by "touching" potentially visible objects to bring them in to memory, then performing the culling or drawing operations in a timing loop. We used a hardware clock with 16 μ-second resolution to time operations typically requiring tens of milliseconds. All timing measurements reflect "wallclock time"; that is, we did not instrument CPU usage, but actually measured time-to-completion of the culling and drawing operations as if using a high-accuracy stopwatch. The machine was otherwise quiescent during the timing runs.

The *output size* metric quantifies the efficacy of the preprocessing and on-line culling algorithms in upper-bounding the potentially visible set for generalized and actual observers. This metric can be expressed in units of visible volume, visible objects, or visible polygons. None of these choices is entirely satisfactory. Visible volume units do not account for the fact that the model space is not uniformly populated with detail objects. Visible object units do not account for the differing times required to render different objects. Finally, visible polygon metrics can not encompass the existence of level-of-detail (LOD) algorithms that strive to render far-away and fast-moving objects with fewer polygons than close-by objects presumably visible in greater detail. No level of detail polygon-reduction algorithms [FST92] were used; we worked only in units of objects as defined by the modeling system, and rendered all objects at a constant level of detail (the highest available).

Although LOD algorithms do draw *fewer* polygons for far-away objects, the polygons drawn are typically *larger*, on average. Large polygons can cause the graphics hardware we used to become "fill-limited" [FK91], i.e., exhibit bottlenecks during scan conversion and pixel writes. The result is that, although LOD modeling might reduce the number of polygons drawn by a factor of five or more, the drawing time typically decreases by only a factor of two to three [FST92]. On the other hand, eschewing LOD analysis can result in wasted detail rendering, needlessly high depth-complexity, and *transform-limited* drawing, i.e., a bottleneck at the top of the graphics pipeline. We have tried to minimize these confounding factors by rendering at a constant level of detail throughout, and by counting rendering in units of objects, assuming the time to render any given object is roughly constant.

The *object survival* and *object overestimation* metrics quantify how much of the graphics hardware

capacity is "wasted" by rendering invisible objects (i.e., by classifying invisible objects as potentially visible). We say that an object *survives* in a rendered image if some pixel in the image is painted by some polygon of the object and this pixel is never obscured by a closer polygon. Clearly a culling operation with perfect knowledge would render *only* surviving objects; our culling algorithms generally overestimate by rendering non-surviving objects. One measure of how well the culling performs is the percentage by which object visibility is *overestimated*; i.e., the ratio of the number of objects that are rendered to the number that actually survive in the framebuffer.

The *depth complexity* metric quantifies the reduction in pixel depth complexity, or the number of times each pixel is "painted", due to on-line culling (a depth-buffer test counts as a painting operation). A depth complexity of 1.0 can occur only in an environment containing only a single convex object (with backfaces removed), or when using an "ideal" visibility algorithm with perfect knowledge of the entities and entity fragments visible to the observer (e.g., a screen-space scan-line renderer that writes each pixel exactly once). We ignore the fact that each pixel is painted exactly once while clearing the framebuffer at the start of each frame. This fixes the theoretically optimal depth complexity at one rather than at two (the former choice is more intuitive).

In practice, the lowest-achievable depth complexity is substantively higher than one, since the whole point of depth-buffering is to obviate fragmentation of polygons for rendering; consequently if polygons have to be either drawn, or not drawn, they will overlap substantially in any given (interesting) scene and even exact visibility algorithms (those that render only surviving entities) must produce depth complexities greater than one. Finally, we note that the computed depth complexity may sometimes be less than 1.0, since some scenes (for example, when looking outside of the building) do not paint the entire window.

Analogously, we justify our use of object-based utility metrics, rather than metrics based on polygons, on the fact that the abstract visibility operations don't "know" about polygons – but only about objects. The culling algorithms make binary decisions about objects, determining only whether to draw them or not. The definition of what constitutes an object is more properly left to the modeling agent than to the visibility scheme.

We quantify depth complexity discretely using the graphics hardware. The rendering engine can be configured to count, in hardware, the number of times each screen pixel is touched during rendering. In Silicon Graphics terminology, depth-buffering is disabled, the framebuffer is cleared to zero, the color of each polygon is set to the integer 1, and the blending function is set to *add* each rendered pixel value to the existing framebuffer value. The final values of each pixel are then summed and divided by the total number of pixels (in our case, 800,000 pixels in a window 1,000 pixels wide by 800 high).

Each of the six metrics – objects drawn, survived, and overestimated, culling and drawing times, and depth complexity – were tabulated (average, minimum, and maximum, where appropriate) for each of seven interesting culling modes. In the following, the "observer cell" is that cell containing the actual observer. The seven culling modes were (Figure 9.7):

1. **No Culling (NC)**: all cells and all objects were drawn (not shown in Figure 9.7).
2. **Spatial Culling (SC)**: cells and objects incident on the view frustum were drawn. We implemented the method of [GBW90] on a conforming subdivision, but did not parallelize the cull as did those researchers.

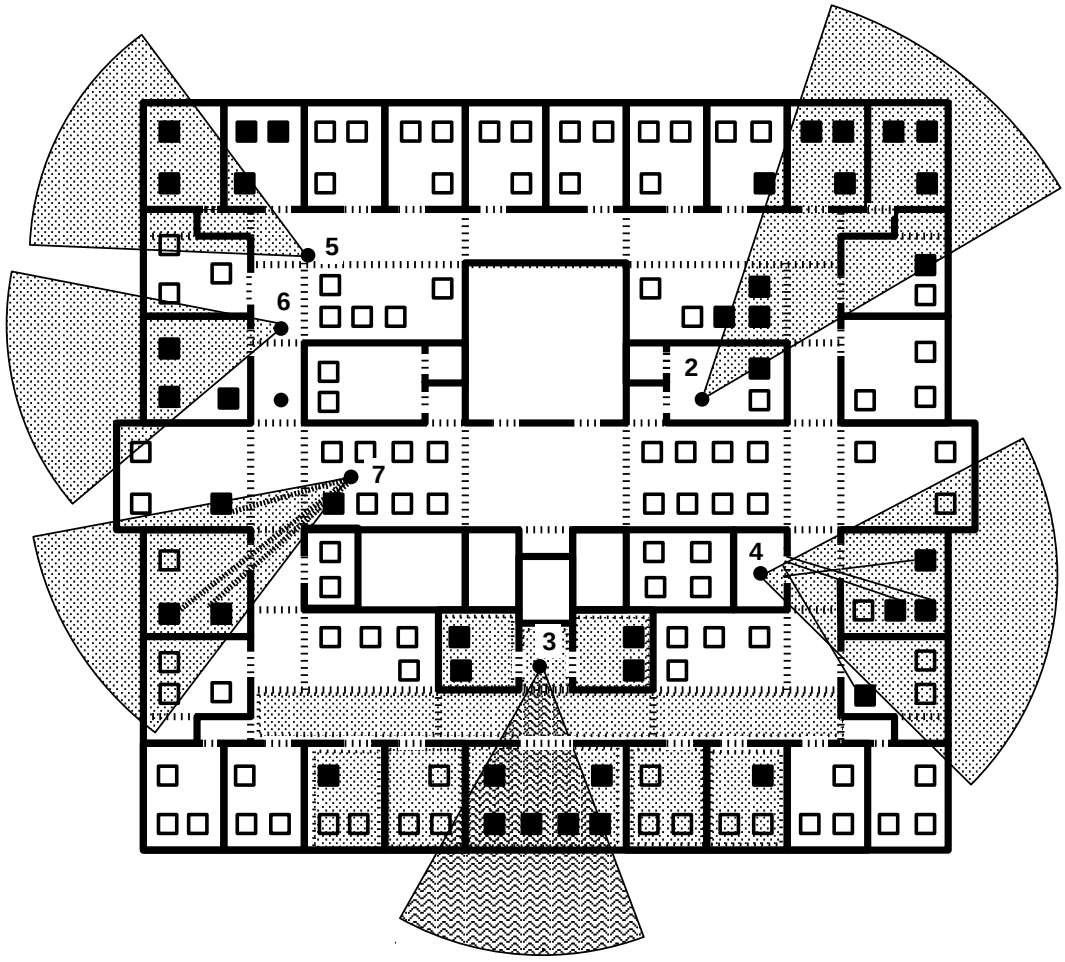


Figure 9.7: The instrumented culling modes (NC mode not shown).

3. **Cell-to-Cell, Cell-to-Object (CtC,CtO)**: all objects potentially visible from the observer cell were drawn. This metric quantifies how well we can do with no on-line computation.

4. **Frustum-to-Cell, Frustum-to-Object (FtC,FtO)**: all objects incident on the view frustum, and visible from the source cell, were drawn. This mode intersects the observer cell's generalized visibility with the actual (i.e., instantaneous) view frustum.

5. **Eye-to-Cell, Cell-to-Object (EtC,CtO)**: all objects visible from the source cell, and in a cell reached by the eye-to-cell traversal, were drawn. This mode performs sightline computations for cells, and no computation for objects.

6. **Eye-to-Cell, Frustum-to-Object (EtC,FtO)**: all objects visible from the source cell, in a cell reached by the eye-to-cell traversal, and incident on the view frustum, were drawn. This mode performs sightline computations for cells, and frustum-incidence for objects.

7. **Eye-to-Cell, Eye-to-Object (EtC,EtO)**: all objects admitting a sightline from the eye were drawn. This is the "exact" on-line cull described in §5.4.4, in that it performs sightline computations for both cells and objects. It is as discriminating as is possible without knowledge of the detail objects' constituent polygons.

Some of the metrics were not tabulated for all culling modes. For example, the object survival metric should be constant regardless of culling mode. We verified that this was the case in practice, and so tabulate it only for the most discriminating culling mode. The culling time for some methods – No Culling and Cell-to-Object – is theoretically zero (in practice, considerable time is required to make the visible sets available to the rendering process [FST92])). The pixel depth complexities for methods differing only by a frustum incidence operation – No Culling vs. Spatial Culling; CtC,CtO vs. FtC,FtO; and EtC,CtO vs EtC,FtO; – are verifiably identical. These simplifications are evident in the tables to follow.

9.4 Experimental Results

9.4.1 Test Walkthrough Path

We sampled every eighth element of a 4,000-frustum recorded walkthrough path through the building model to produce a 500-frustum path (the top two floors and roof of the model, and the test path, are shown in Figure 9.1). (The statistics instrumentations absorbed tens of hours of CPU time, so the unsampled path would have used unreasonable machine resources.) The field of view was constant at 90 degrees wide and 77.3 degrees high (the latter figure was determined by the 5-to-4 aspect ratio of the test window), subtending roughly 15 percent of the observer's view sphere.

We instrumented the culling algorithms to run on both the partial and full models. By constraining the test path to the top portion of the building we were able to utilize it in both models, and compare the resulting storage and time metrics from each run.

9.4.2 Storage Overhead and Precomputation Times

Partial Model

The partial geometric model consisted of the sixth and seventh floors, and roof. In total, this model comprised 4,598 objects and 238,861 polygons (at the highest level of detail), of which 3,871 (about 1.6%), were classified by the modeling system as occluders. Of these, 325 occluders (about 8.4%) were not axial rectangles and were ignored during the spatial subdivision rounds. Thus, more than ninety percent of the occluder-sized polygons in the geometric model supplied suitable axial splitting planes to the spatial subdivision rounds (this is consistent with the observations of an earlier visibility researcher who found that roughly ninety-five percent of the candidate splitting polygons in his test model were axial [Air92]).

Selecting candidate occluders and subdividing the k-D tree to termination required 55 seconds. Once the subdivision directives were stored, regenerating the k-D tree into 2,163 leaf cells, and enumerating the 11,500 portals of the conforming spatial subdivision, required 6 and 8 seconds, respectively.

Storing the occluders and conforming spatial subdivision required about 3 megabytes (the details of the database format are given in [FST92]). Inserting the detail objects at all levels of detail took about one hour, and increased the size of the model to 62.5 megabytes. Finally, computing visibility information for the model took almost five hours, and increased the model size by about 5.2 megabytes to a total size of 67.7 megabytes. (More than one and half million portal sequences were attempted, with average length of 9.07 portals. The average length of a *failed* portal sequence, i.e., one that did not admit a sightline, was 9.94 portals. The successful and failed portal sequence lengths do not differ by exactly 1.0 because many branches of the DFS terminated not due to impassable portals but to the fact that many cells had only one entrance and no exit, other than the entry portal, which was known *a priori* to be impassable. The reaching portal sequence therefore counted as a success, with no incremental test and failure.) Thus, the storage overhead incurred by visibility computation was less than ten percent of the size of the polygon data in the geometric model (less than fifteen percent if the conforming subdivision is treated as visibility data).

Full Model

The entire geometric model consisted of the third through seventh floors, and roof. In total, this model comprised 13,191 objects and 712,499 polygons (at the highest level of detail), of which 9,699 (about 1.4%), were classified by the modeling system as occluders. Of these, 550 (about 5.7%) were not axial rectangles and were ignored during the spatial subdivision rounds. As in the partial model, more than ninety percent of the occluder-sized polygons in the geometric model supplied suitable axial candidate splitting planes.

Selecting candidate occluders and subdividing the k-D tree to termination required 4 minutes. Regenerating the k-D tree into 5,762 leaf cells, and enumerating the 34,916 portals of the conforming spatial subdivision, required 26 and 21 seconds, respectively,

Storing the occluders and conforming spatial subdivision required about 8.4 megabytes. Inserting the detail objects at all levels of detail took about four hours, and increased the size of the model to 175.3 megabytes. Finally, computing visibility information for the model leaf cells took almost nineteen

hours[1], and increased the model size by about 13.7 megabytes, to a total size of 189.0 megabytes. (We did not compile portal sequence length statistics for this data set.) Thus, the storage overhead incurred by visibility computation was less than 8% of the size of the polygon data in the geometric model (less than 15% if the conforming subdivision is treated as visibility data).

9.4.3 Tabulated Utility Metrics

Tables 9.1 through 9.10 summarize the data for the partial and full models (the tables are arranged in pairs). The most important figures of merit are shown in bold.

Averaged Object and Pixel Depth Metrics

Culling Mode	# Objects Drawn	% Objects Drawn	# Objects Survived	Object Overestimation (%)	Discrete Depth Complexity
None	**4598.0**	100.0%	**40.4**	**11000%**	3.31
Spatial	988.2	21.5%	"	2400%	3.31
CtC,CtO	385.8	8.4%	"	850%	1.78
FtC,FtO	100.0	2.2%	"	150%	1.78
EtC,CtO	87.2	1.9%	"	120%	1.56
EtC,FtO	65.6	1.4%	"	60%	1.56
EtC,EtO	**49.7**	**1.1%**	"	**23%**	1.44

Table 9.1: Object and pixel metrics for the partial model.

The disparity between object overestimation and pixel depth complexity is surprising in both models (Tables 9.1 and 9.2). While the number of objects drawn varied by a factor of more than one hundred, the depth complexity varied by only a factor of about three. Depth complexity grows slowly with the number of objects since, as objects get farther away, they paint fewer screen pixels. The difference between the object counting and depth complexity metrics arises from the fact that the culling operations are best at culling remote objects, but these typically paint relatively few screen pixels. In contrast, nearby objects paint relatively large screen areas but do not interact in the culling operation; i.e., they cannot prevent each other from being rendered, as can occluders.

The number of objects passed by each culling mode, on average, drops monotonically with increasing cull complexity. Note that static (i.e., cell-to-object) alone reduces the amount of visible data to 8.4% of the model, on average; subjecting this generalized visibility to a frustum-incidence operation reduces this amount by almost three-quarters, to 2.2% of the model (in a roughly spherically symmetric model, we expect frustum-incidence to reduce the generalized visibility by the fraction of the unit sphere excluded by the view frustum – in our case, about 85 percent).

[1] Static visibility computations were truncated after thirty minutes for free-space cells.

Culling Mode	# Objects Drawn	% Objects Drawn	# Objects Survived	Object Overestimation (%)	Discrete Depth Complexity
None	13191	100 %	**40.4**	32600 %	4.93
Spatial	2433.5	18.5 %	"	5900 %	4.93
CtC,CtO	418.8	3.2 %	"	940 %	1.84
FtC,FtO	105.4	0.80%	"	160 %	1.84
EtC,CtO	90.8	0.69%	"	124 %	1.58
EtC,FtO	66.7	0.51%	"	65 %	1.58
EtC,EtO	49.5	0.38%	"	**22.5%**	1.46

Table 9.2: Object and pixel metrics for the full model.

Finally, note that the number of surviving objects is identical in the partial and full models (viz. Table 9.1, Table 9.2). This is expected, since object survival should be independent of the spatial subdivision or (superset) culling algorithm used.

Average Object Counts and On-Line Operation Times

Culling Mode	# Objects Passed	% Objects Culled	Cull Time (msec)	Draw Time (msec)	Frame Time Speedup Factor
None	4598	0.0%	n.a.	2805.2	1.0
Spatial	988.2	78.5%	73.1	655.0	4.3
CtC,CtO	385.8	91.6%	n.a.	252.3	11.1
FtC,FtO	100.0	97.8%	6.4	96.4	29.1
EtC,CtO	87.2	98.1%	25.6	90.9	30.9
EtC,FtO	65.6	**98.6%**	**26.1**	**72.9**	38.5
EtC,EtO	49.7	**98.9%**	**75.8**	**55.0**	37.0

Table 9.3: Culling and drawing time metrics for the partial model.

Note that, for either model, the two most discriminating modes exhibit culling and drawing times that bracket an intermediate frame rate (in this case, about 50 milliseconds). In the partial model, **EtC,FtO** culling requires only 26 milliseconds but produces a potentially visible object set requiring 73 milliseconds to draw (Table 9.3). The exact **EtC,EtO** cull takes three times as long to complete, but produces a potentially visible set 25 percent smaller (50 vs. 66 objects), reducing the corresponding

Culling Mode	# Objects Passed	% Objects Culled	Cull Time (msec)	Draw Time (msec)	Frame Time Speedup Factor
None	13191	0 %	n.a.	8900	1.0
Spatial	2433.5	81.6%	192.4	2050	4.3
CtC,CtO	418.8	96.8%	n.a.	316.7	28.1
FtC,FtO	105.4	99.2%	7.3	103.9	85.7
EtC,CtO	90.8	99.3%	33.2	96.9	91.8
EtC,FtO	66.7	99.5%	**33.5**	76.5	**116.3**
EtC,EtO	49.5	99.6%	**86.8**	56.9	**102.5**

Table 9.4: Culling and drawing time metrics for the full model.

rendering time by about a quarter to 55 milliseconds (57 milliseconds in the full model[2]). In practice, parallelizing the culling and drawing processes decreases the dependence of frame time on worst-case cull time, but at the cost of increased transport delay.

Average, Minimum and Maximum Object Counts

The purely spatial cull exhibited a wide variance in the number of objects passed for rendering: from one-fifth of one percent of the model to almost seventy-five percent of it (Table 9.5). In practice, the real worst-case frame time for this culling mode is much worse than the six-tenths of a second we quote using the infinite-memory machine model. Since at worst seventy-five percent of the model data (about 45 megabytes) must be read from disk and issued to the graphics pipeline, even (optimistically) assuming a one megabyte per second disk bandwidth, drawing the data returned by the spatial cull would take about forty-five seconds. Drawing all objects in the partial model (i.e., performing no culling) would take a full minute *per frame*, even under this optimistic bandwidth assumption; drawing the full model in its entirety would require three minutes.

Static culling (**CtC,CtO**) exhibited worst-case maximum visibility of about a fourth of the partial model, or a tenth of the full model. All of the occlusion-based dynamic culling modes did much better. The least discriminating occlusion-based cull mode **FtC,FtO**, categorized no more than 10% (3.5%, in the full model) of the model objects visible. As the complexity of the dynamic cull mode increases, the maximum number of objects drawn decreases monotonically, as expected.

Average, Minimum and Maximum Cull Times

The minimum cull time roughly the same for each culling mode. In the case of purely spatial culling, this is coincidentally due to the fact that the test path leaves the building and the actual observer looks

[2] Small discrepancies in times between the partial and full models arise from timer aliasing and different caching effectiveness while simulating the two data sets.

Culling Mode	Avg. # (%) Objects Passed	Min # (%) Objects Passed	Max # (%) Objects Passed
None	4598 (100 %)	4598 (100 %)	4598 (100 %)
Spatial	988.2 (21.5%)	9 (0.19 %)	3439 (74.8 %)
CtC,CtO	385.8 (8.4%)	88 (1.9 %)	1130 (24.6 %)
FtC,FtO	100.0 (2.2%)	6 (0.13 %)	455 (9.9 %)
EtC,CtO	87.2 (1.9%)	8 (0.17 %)	298 (6.5 %)
EtC,FtO	65.6 (1.4%)	5 (0.11 %)	218 (4.7 %)
EtC,EtO	49.7 (**1.1**%)	1 (0.022 %)	177 (3.8 %)

Table 9.5: Average, minimum and maximum object metrics for the partial model.

Culling Mode	Avg. # (%) Objects Passed	Min # (%) Objects Passed	Max # (%) Objects Passed
None	13191 (100 %)	13191 (100 %)	13191 (100 %)
Spatial	2433.5 (18.5 %)	66 (0.5 %)	8278 (62.8%)
CtC,CtO	418.8 (3.2 %)	131 (1.0 %)	1257 (9.5%)
FtC,FtO	105.4 (0.8 %)	6 (0.05 %)	462 (3.5%)
EtC,CtO	90.8 (0.69%)	17 (0.1 %)	313 (2.4%)
EtC,FtO	66.7 (0.51%)	5 (0.04 %)	219 (1.7%)
EtC,EtO	40.1 (0.30%)	1 (0.007%)	136 (1.0%)

Table 9.6: Average, minimum and maximum object metrics for the full model.

outside of the model; an observer keeping to the core of the building would experience consistently higher spatial cull times (and consistently lower sightline-based cull times due to the rarity of free-space cells in the core regions).

Culling Mode	Average Cull Time (msec)	Minimum Cull Time (msec)	Maximum Cull Time (msec)
None	n.a.	n.a.	n.a.
Spatial	73.1	4.1	225.4
CtC,CtO	n.a.	n.a.	n.a.
FtC,FtO	6.4	1.6	21.2
EtC,CtO	25.6	2.7	159.0
EtC,FtO	26.1	2.8	157.8
EtC,EtO	75.8	2.8	**499.7**

Table 9.7: Average, minimum and maximum culling times for the partial model.

Culling Mode	Average Cull Time (msec)	Minimum Cull Time (msec)	Maximum Cull Time (msec)
None	n.a.	n.a.	n.a.
Spatial	192.4	14.0	555.6
CtC,CtO	n.a.	n.a.	n.a.
FtC,FtO	9.1	2.1	46.7
EtC,CtO	31.0	3.5	429.5
EtC,FtO	30.6	2.7	318.8
EtC,EtO	86.6	2.8	669.2

Table 9.8: Average, minimum and maximum culling times for the full model.

Maximum culling times (Table 9.7) behave differently than object counts as a function of culling mode. The maximum cull time no longer drops monotonically as culling effectiveness increases, but is minimum at **FtC,FtO** (21 milliseconds, in the partial model) and climbs again as the sightline-based cull is applied first to cells, then the frustum-based cull to objects, then finally the sightline-based cull to both cells and objects. The sightline-based cull exhibited a worst-case computation time of half a second in the partial model, five times slower than the desired 10 Hz frame rate, which occurred in a complex free-space region outside the model. For the full model, the worst-case culling time increased by about 170 milliseconds (34%).

Average, Minimum and Maximum Draw Times

The average and maximum drawing times drop monotonically with increasingly expensive culling modes for both the partial and full model, as expected.

Culling Mode	Average Draw Time (msec)	Minimum Draw Time (msec)	Maximum Draw Time (msec)
None	2805.2	2732.2	3339.0
Spatial	655.0	18.1	2069.5
CtC,CtO	252.3	37.6	957.3
FtC,FtO	96.4	13.5	509.8
EtC,CtO	90.9	13.0	330.3
EtC,FtO	72.9	13.3	247.7
EtC,EtO	55.0	7.9	157.6

Table 9.9: Average, minimum and maximum drawing times for the partial model.

Culling Mode	Average Draw Time (msec)	Minimum Draw Time (msec)	Maximum Draw Time (msec)
None	9146.4	9042.3	9485.1
Spatial	2052.3	89.1	6253.0
CtC,CtO	316.7	81.7	1138.7
FtC,FtO	103.9	18.7	513.3
EtC,CtO	96.9	14.4	373.4
EtC,FtO	76.5	14.3	266.1
EtC,EtO	56.9	8.1	159.1

Table 9.10: Average, minimum and maximum drawing times for the full model.

When no culling is employed, the draw time is fairly consistent at just under three seconds for the partial model, and just over nine seconds for the full model. Some variation occurs because, although all of the model data is drawn each frame, the graphics hardware expends differing amounts of time to dispense with polygons that are, for example, outside the view frustum, partially clipped, backfacing, or obscured by nearer polygons, and these types of polygons occur in different relative numbers as the actual observer moves through the model.

If the infinite-memory model is abandoned, drawing either the full model or the entities passed by the purely spatial cull would require more than two minutes.

For both models, the spatial culling mode exhibited widely varying draw times (Tables 9.9 and 9.10), spending as much as two to six seconds (on the idealized infinite-memory machine). The dynamic culling modes exhibited draw times in a tighter range, most notably the **EtC,EtO**, which in the worst case passed objects that required about 160 milliseconds to draw in either model. That is, the worst-case drawing time increased by about one-and-a-half milliseconds (1.0 %) under a tripling of the model complexity.

9.4.4 Museum Park

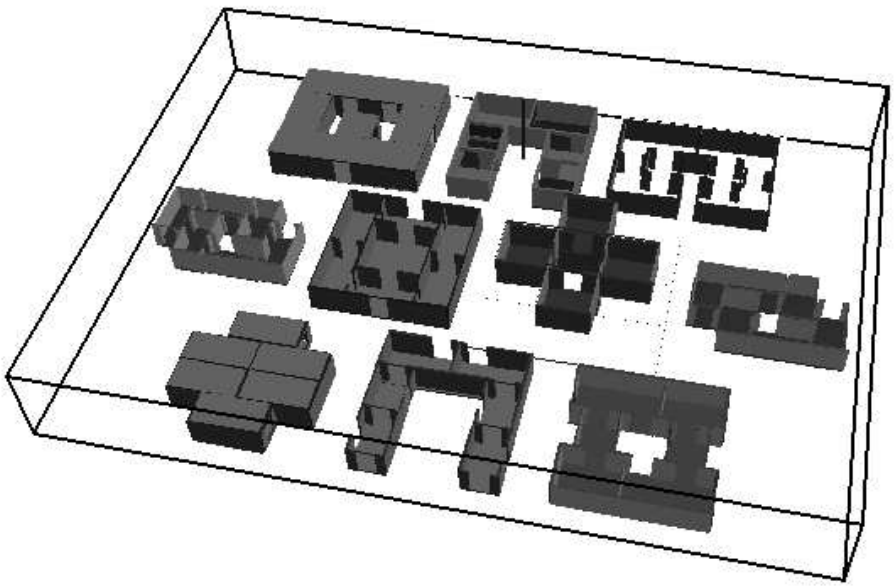
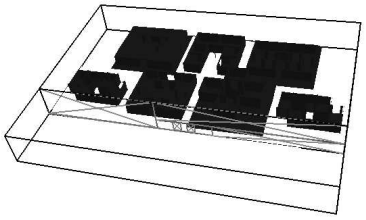


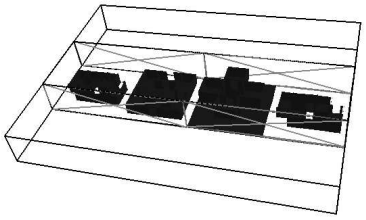
Figure 9.8: The ten museums in museum park.

We briefly tested the spatial subdivision algorithm, which had been devised for building interiors, on a data set that we call "museum park." Ten one- and two-story structures with similar footprints were assembled in a loose array (Figure 9.8), but the individual buildings were not aligned with one another.

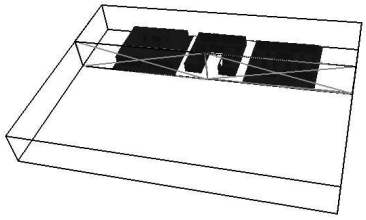
The minimum-cleaving subdivision criterion produced a fairly good spatial subdivision of the museum park. Major splitting planes between the rows of museums were found early, as was the ground-plane on which the museums rested (Frames 1 through 3 of Figure 9.9). After the rows of museums had been separated, the minimum-cleaving criterion separated each individual museum within rows (the delineation of one such museum is shown in Frames 4 through 8). Finally, the interior of each museum was subdivided as usual (Frames 9 through 12). The splitting process took less than 30 seconds. The final subdivision (Figure 9.10) has substantial free-space splitting, but very few superfluous partition planes inside the structures themselves.



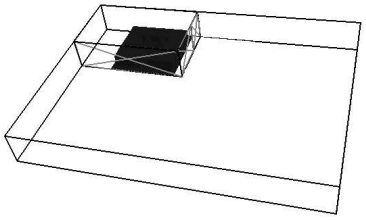
Frame 1



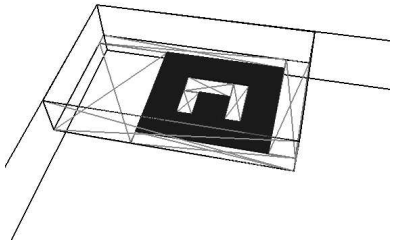
Frame 2



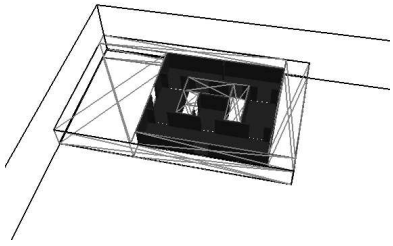
Frame 3



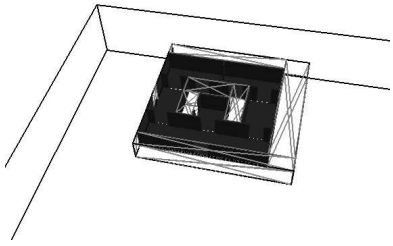
Frame 4



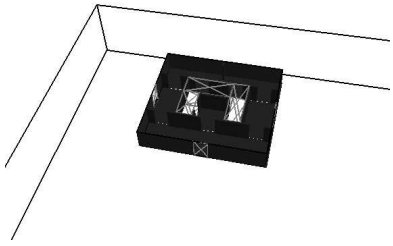
Frame 5



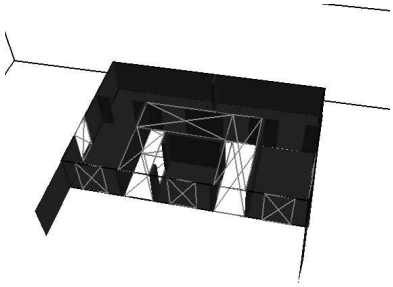
Frame 7



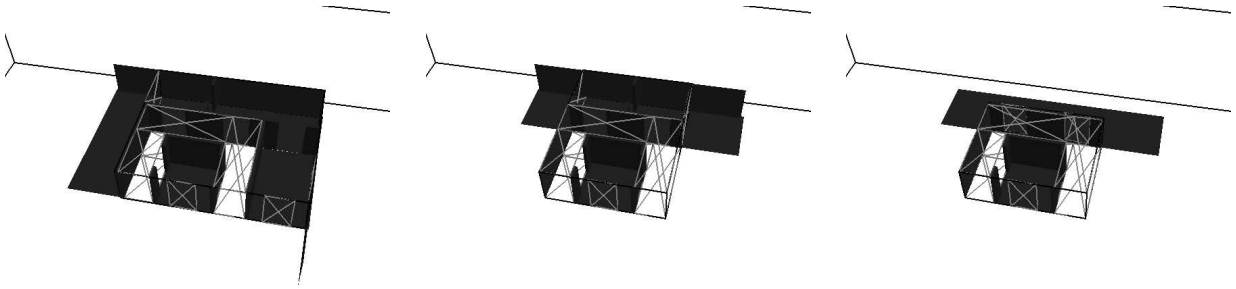
Frame 7



Frame 8



Frame 9



Frame 10 Frame 11 Frame 12

Figure 9.9: Snapshots of the museum park spatial subdivision.

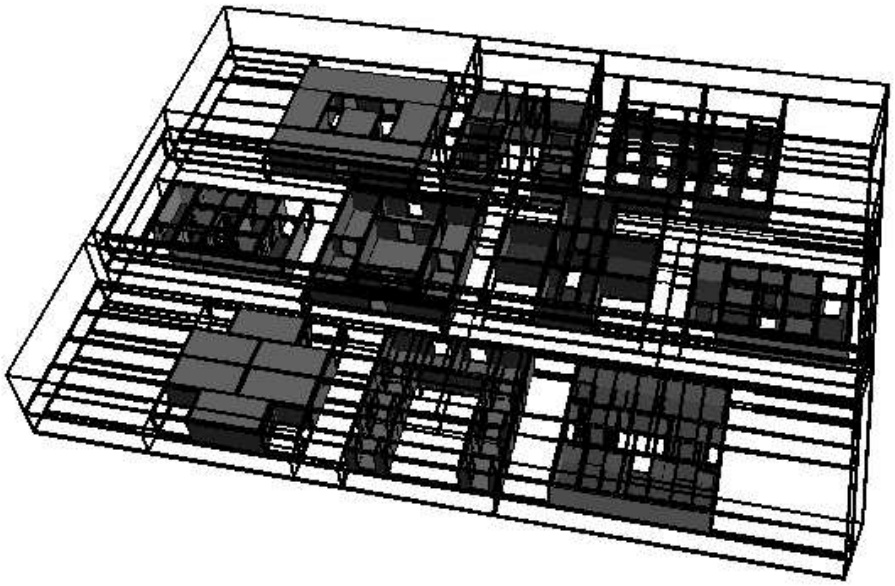


Figure 9.10: The final subdivision of museum park.

9.4.5 General Polyhedral Data

Although the implementations of the conforming subdivision for generally oriented occluders are still in somewhat embryonic stage, we compared subdivision construction and portal enumeration times for identical data. Unfortunately, we have no access to non-axial data sets of complexity comparable to the building model. Consequently, we timed the general BSP-tree based subdivision scheme on the partial geometric model, rotated about two axes to cause all occluders to become non-axial (although they remained rectangular).

Selecting candidate occluders and subdividing the k-D tree to termination required 15 minutes. Once the subdivision directives were stored, regenerating the k-D tree into 1,963 leaf cells, and enumerating the 9,465 portals of the conforming spatial subdivision, required 284 and 475 seconds, respectively. Storing the occluders and conforming spatial subdivision required about 12 megabytes of memory. We did not compute cell-to-cell and cell-to-object visibility, as the implementation is still somewhat nascent. We can observe that the storage requirements and computation times grew at different rates. The storage required by the conforming subdivision was four times larger than that for the equivalent axial data set. On the other hand, the conforming subdivision construction times was fifty times longer than that for the equivalent axial data set. The considerable computation increase is partially due to substantial "sanity-checking" code that ensured the correctness of the incremental spatial subdivision after each split. Finally, we observe that both the number of cells and number of portals generated are actually slightly *smaller* for the rotated data set.

9.5 Summary and Reflections

One benefit of implementing the subdivision and visibility schemes described in this thesis was the body of practical intuition and engineering experience that resulted. In the following, we briefly describe the non-theoretical, but nonetheless extremely important, issues that arose during the implementation process.

9.5.1 ADTs, Invariants, and Witnesses

First, use of *ADTs* or abstract data types [AHU83], and data and program *invariants* such as Unix C-language "assertions" [Bel79], facilitates the connection between the logical conception and the implementation of an algorithm. An abstract data type encapsulates the definition of, and all operations defined upon, any data object. For example, an ADT for convex polygons might store an ordered list of vertices forming a simple convex planar contour, with an associated normal, plane equation, bounding region, centroid, etc., and provide the operations of creation, deletion, intersection, rendering, etc. This encapsulation hides the details of defining, manipulating, storing, and releasing the data object from conceptually higher-level algorithms that manipulate such data objects as primitive operands.

The C-language `assert()` construct ([Bel79], §3X) provides a very simple and powerful mechanism for gaining confidence in implemented algorithms. The functional statement `assert(`*predicate*`)` can be embedded anywhere in the program text, where the *predicate* is simply any boolean expression that can be computed at that point in the program. When the assertion is encountered, the boolean expression

is evaluated (this itself may cause function evaluation and other assertions to occur), and if it is found to be false, an error message is generated, an image of the memory state of the running program is stored as a "core" file, and program execution is terminated. For optimization, or when the algorithm is considered iron-clad, all assertions can be deactivated by a recompilation. Even in this deactivated state they serve as logical sidebars to the program's text. We have augmented the `assert()` statement to execute a run-time definable action before terminating execution, so that, for example, troublesome interactively-generated input can be saved for later study in isolation.

Another useful notion is that of a *data invariant* which can be *enforced* when an algorithm receives input. For example, a program that operates only on convex polygons might simply discard any non-convex polygons in its input (and generate a warning message), ensuring that any later computation will operate only on convex polygons.

Finally, a *witness* is a data object that evinces the outcome of a geometric query. This outcome can be a success or failure; for example, a witness to the intersection of a line and a plane is a point lying on both flats (if there is no intersection, the witness might be a polygon edge with the requisite sidedness with respect to the query line, as in Chapter 6). Witnesses can be thought of as short geometric proofs that particular geometric configurations do or do not have specific properties. During implementation, we often relied on witnesses during visual inspection of algorithmic behavior for assurance that our algorithms were functioning correctly. Indeed, the *absence* of a witness was often a powerful short proof of the misbehavior of an algorithm.

9.5.2 Input Filtering

We have found it useful to enforce data invariants on the geometric model for the walkthrough system, since its occluders and objects are derived from a real-world modeling system. Since the architectural model should represent a physically realizable space, it is intuitively clear that the model should have "inhabitable" portions (e.g., common areas, corridors) and "uninhabitable" portions (e.g., the insides of walls, crawl-spaces beneath floors). Moreover, both portions should consistently have non-zero thickness, so that, for example, a single occluder can not serve as a wall to two rooms on opposite sides of the occluder. Ignoring passable elements such as doors and windows, the inhabitable space of the building model should form a single connected component and have a manifold (although perhaps high-genus) boundary.

The manifold nature of the model can be checked by inspecting the *orientation* and *connectivity* of occluders. Every occluder can be oriented according to the ordering of its vertices. The occluder's front face (i.e., that face on which the vertices appear counter-clockwise) should abut an inhabitable portion of the model, and its back face an uninhabitable portion. The occluder's connectivity is inspected through its shared edges with other occluders; these edges should be oppositely directed, and every edge should be shared by exactly two occluders.

Models satisfying these data invariants prove easier to process both theoretically and in an engineering sense. In particular, it should not be possible to move from an inhabitable region to an uninhabitable one (i.e., inside the wall) without traversing some occluder. The advantage of enforcing this invariant is that the spatial subdivision and adjacency graph construction then "find" the inhabitable portions of the model and store them as connected components. All subsequent visibility computations, defined as

traversals of this graph, will then be guaranteed to operate only within inhabitable regions of the model. In other words, this invariant assures that no portals exist between inhabitable and uninhabitable model regions.

The "thickness" of the model has also proven extremely important to the efficiency of the visibility algorithms, in practice: it prevents an observer looking down a long corridor from seeing into far-away offices. Typically, the observer's lines of sight cannot propagate "around" the door frames into the offices (cf., for example, Figure 1.4 in the Introduction).

Polygon orientation proved useful in another sense. The fastest stabbing algorithm for an unordered sequence of general, unoriented polygons having e edges total, requires $O(e^4\alpha(e))$ time [MO88]. When the polygons (i.e., portals) of the sequence are *oriented* to admit stabbing lines in only one direction, the complexity of the stabbing algorithm can be reduced to $O(e^2)$ time (cf. §8.3.1).

9.5.3 Spatial Subdivision

In the presence of general occluders, subdividing three-dimensional space well is an ill-defined and difficult task. Since even the relatively well-posed problem of deciding whether a polyhedron is tetrahedralizable is NP-complete [RS89], it seems plausible that finding "optimal" general subdivisions may be as hard or harder. In the near term, therefore, the most one can practically hope for is "good" subdivisions that exhibit subquadratic size and near-logarithmic depth for practically occurring environments, and which satisfy our occlusion requirements acceptably well.

We have generated useable subdivisions for general occluders using BSP trees, although they are both slower to construct and search, and more expensive to store, than k-D trees in practice. We have seen that BSP trees may cost as much as $O(n^3)$ time and $O(n^2)$ storage to construct in the worst case [PY90]. Things are somewhat easier when the occluders are axial. It has been shown, for example, that BSP trees over n axial 3D occluders can be constructed with size $O(n^{\frac{3}{2}})$ in optimal $O(n^{\frac{3}{2}})$ time [PY89]. We have not yet implemented this algorithm and so do not know if the subdivisions so generated have reasonable complexity constants, and behave well when subjected to combinatorial visibility algorithms such as those described here.

Another issue concerning subdivisions is that of symmetry between source and reached cells. Our framework uses the same set of cells both as visibility sources (i.e., bounds on a generalized observer) and as reached cells (i.e., bounds on potential visibility of the generalized observer). Perhaps the notions should be decoupled. Thus, a cell could be more highly constrained when acting as a source, but less constrained when traversed by the search algorithms, to reduce the amount of effort spent searching through portals. The concepts of free-space and metacells are essentially ad hoc modifications to the subdivision implementation (but not, substantively, to the abstraction) so that it may operate more asymmetrically in this sense.

The abstraction of a conforming spatial subdivision allows the *annotation* of the cell adjacency graph with *transition* visibility information. Since the cell-to-cell and cell-to-object visibility sets change qualitatively only when a portal is traversed, one might attach to each portal the changes in these visibility sets experienced by a generalized observer crossing the plane of the portal. Such difference lists, of course, would be antisymmetric and functions of the sense in which the portal was traversed (i.e., objects added while traversing the portal from cell A to cell B would be subtracted when traversing

it from cell B to cell A). This seems like a worthwhile generalization of the conforming spatial subdivision data structure, but it has not yet been implemented.

The base issue is not one of adjacency graph annotation, since this can be viewed as a compacting postprocess to follow the visibility computations. More importantly, the issue of reusing the visibility information, while it is being computed, is itself a challenging theoretical issue (cf. §10.2). This remains an active area of investigation.

9.5.4 Scaling Effects

We set out to measure the effects of increasing model complexity on three metrics: the storage overhead incurred by precomputing visiblity information; the time required to precompute visibility information; and the cull time and speedup factor during on-line culling, for a fixed path.

We found that tripling the complexity of the geometric model increased the amount of computed visibility information by a factor of roughly two and two thirds. Since the spatial subdivisions were not identical for the two data sets, it is difficult to compare the storage factors with a high degree of confidence. However, we find it encouraging that the increase in visibility storage requirements was about the same as that for geometric data storage.

Visibility computation time scaled less well; tripling the model complexity caused a fourfold increase in visibility precomputation times. This increase had two causes. First, the relative amount of free space in the full model increased, causing greater numbers of expensive cell-to-cell and cell-to-object determinations from empty cells and through clusters of empty cells. Second, the working set and memory requirements of the full model visibility precomputation were greater for the (180 megabyte) full model, and in some cases taxed the (64 megabyte) physical memory capacity of our hardware platform. Excessive swapping activity may have substantially lengthened the observed "wall clock" precomputation times for certain cells with very high visibility.

Finally, the average on-line culling time worsened only slightly when the model tripled in size, from 75.8 to 86.6 milliseconds for an increase of about 14%. During most of the interior portions of the walkthrough path, the culling times were roughly equivalent for the two models. When in the free-space of the full model, however, the cull time increased substantially over the cull time in the same region of the partial model. For example, the longest measured cull time was 500 milliseconds in the partial model, and almost 670 milliseconds in the full model; a difference of 34%.

From these results we can garner two insights. First, we are optimistic that visibility storage needs will indeed scale linearly with geometric data size, for visual models of similar visual complexity. Second, it is clear from the increases in precomputation and on-line cull time that good spatial subdivisions and minimization of problems due to free-space cells are cells must be more thoroughly investigated.

Chapter 10

Discussion

10.1 Spectrum of Applicability

We have classified as "densely occluded" those geometric models for which interior visibility is very limited. Such models occur at one end of a spectrum, which at its other extreme includes "sparsely occluded" models, such as relatively open terrain. The visibility algorithms proposed here would not perform well for sparsely occluded models, because of the predominance of free-space cells (cf. §9.2.2). The cell-to-cell computations would explore (and find sightlines through) all portal sequences, and would find very few restrictions on cell-to-cell, cell-to-region, or cell-to-object visibility.

The spectrum of geometric models implies a corresponding spectrum of visibility, and shadow algorithms. Classical rendering algorithms make the basic assumption that the universe (i.e., geometric model) is fully illuminated, unless it can be established that light is *prevented* from reaching a particular region by an occluder. This is probably the right approach when space is sparsely occluded, for example when the model represents predominantly open terrain, or is a complex localized environment with little gross occlusion, such as a complicated ray-traced scene. Visibility preprocessing makes little sense for this class of model, since computing, storing, retrieving, and reusing coarse visibility information might be computationally harder than simply deriving fine-grain visibility (illumination) information as a function of the instantaneous observer (light source) position.

Light propagation algorithms, such as those presented in this thesis, make the opposite assumption: that *nothing* is visible (illuminated) unless a path can be found to it from the observer (light source). This turns out to be an effective assumption for certain common kinds of scenes, like architectural models, and leads to dramatic reductions in rendering times. Computing this visibility information requires a few CPU hours on models with complexity corresponding to a design cycle of weeks or months. Storing the visibility information required less than ten percent of the space required for the geometric model data.

Spatial subdivision and hierarchical representation are promising techniques for management of complexity, a pressing problem for many engineering applications such as visual simulation systems.

These techniques exploit the *coherence* inherent in spatial data. This coherence is spatial, since nearby entities can be stored near each other, and temporal, since nearby entities are typically referenced (e.g., rendered) near each other in time. Coherence can also be exploited in a host of other ways, for example, while editing or performing global illumination calculations on the model.

The ability to predict visibility-related data demands is a continuing part of an engineering effort to build a working walkthrough system that simulates geometric models several times the size of physical memory [FST92]. We conclude that the tradeoff between expending precomputation and storage, in order to gain predictive power and reduce on-line rendering time, is a worthwhile one for architectural models. Finally, we are optimistic that the techniques presented here will scale to larger environments, since visually complexity seems to grow less than linearly with overall model size. We observed average portal sequence lengths to be roughly constant for a one-story, three-story, and eight-story building model. We expect also that the on-line algorithms will straightforwardly generalize to multiple processors, since they are essentially "read-only" graph traversals.

10.2 Algorithmic Complexity

We have presented the complexity of the stabbing and visible-volume algorithms only in terms of the length of the portal sequence to which they are applied. The question remains as to the length of an average portal sequence, and how many such feasible portal sequences can be expected to emanate from a particular cell, in either two or three dimensions.

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 axial portal sequence reaching that cell. Determining a useful upper bound on the total number of such sequences as a function of the total number of cells visible to a given source cell seems challenging, as this quantity appears to depend on the spatial subdivision in a complicated way. However, for architectural models, we have observed the length of most portal sequences to be a small constant (about ten; cf. §9.4). That is, most cells see only a constant number of other cells (and this constant is small enough to have practical implications). Were this not so, most of the model would be visible from most vantage points, and visibility preprocessing would be futile. The consequence of the small stabbing length is that visibility storage requirements should be only linear in the number of subdivision cells, since each cell will have only a constant amount of associated information.

The incremental stabbing and antipenumbra algorithms do not fully exploit the coherence and symmetry of the visibility relationship. Visibility is found one cell at a time, and the sightlines and visible regions so generated are meaningful only for the source cell. Later visibility computations on other cells do not "reuse" previously 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. It is difficult to see how the constraints due to a reaching portal sequence of arbitrary length may be manipulated efficiently.

For example, a common efficiency technique is to perform algorithmic operations *incrementally*, by considering the input data in turn, and maintaining a correct solution (in our case, a stabbing line and description of the cell-to-region visibility) as each input element (e.g., portal) is encountered. The advantage of incrementality is that we generally do not need to examine and construct the entire portal sequence each time a new portal is encountered.

On the other hand, when used as part of a DFS, incrementality has some important disadvantages. First, we must be able to "undo" the insertion of a portal as efficiently as we can add a portal, since we ascend (i.e., backtrack on) the call stack as often as we descend it. For stabbing lines, this is easy; we merely maintain a stack of (constant size) stabbing lines (constant size) linear constraints and "cut it back" whenever the DFS backs out of a cell. But cell-to-region volumes present a problem; their complexity is linear in the length of the active portal sequence, and their detailed structure (i.e., polyhedral face graph) may involve every edge of the active sequence. Thus, "backing out" of a general portal sequence involves general insertion and deletion operations on convex polyhedra, or (equivalently, but even less efficiently), maintaining a copy of the polyhedron generated for each cell, and discarding it upon ascension of the call stack.

The second disadvantage of incrementality is that the DFS nature of the constrained graph traversal search forces portals to be added in the order in which they are encountered; this destroys the random nature of the linear programs we use, and without randomness the expected time of [Sei90b], for example, increases to $O(n^2)$ from $O(n)$ time. In practice, we do use incrementality and exploit the fact that, for axial portals, the cell-to-region polyhedra have constant complexity.

The other complexity issue, apart from incrementality, is re-use of portal sequences. Suppose some source cell DFS reaches a cell through a long portal sequence, and finds no further portals to explore. This portal sequence should, in principle, be useable by the neighbors of the source cell for their visibility computations and, when those sequences fail, by their neighbors, until all cells have had visibility computed. Analogously to the incrementality case, we are effectively faced with problem of removing portals from the *beginning*, rather than the end, of the portal sequence, and appropriately propagating all resulting changes in cell-to-region visibility through the resulting sequence. It is difficult to see how to do this robustly and efficiently. Consequently, we are still seeking satisfactory solutions to the portal incrementality and sequence-reuse problems.

Although the visibility computations are not optimal, their results may be compacted by annotating the adjacency graph with visibility information. We call this compacted representation *portal transition data*, since it exploits the fact that qualitative changes in cell-to-cell cell-to-object visibility can only occur at cell boundaries, i.e., along portals. Cell-to-cell and cell-to-object visibility can be encoded as a pair of lists attached to each portal. Each list describes the *changes* in the associated visibility set encountered by a generalized observer traversing the portal. This consists of a set of cells no longer visible from the attained cell (i.e., those cells to be subtracted), and a set of cells newly visible from the attained cell (i.e., those cells to be added).

10.2.1 Time and Storage Complexities

We summarize the complexity results of the subdivision and visibility algorithms described in the preceding text. Foremost, all of our complexity results are highly dependent on the quality of the spatial

subdivision produced for the given input, and indeed the extent to which visibility is curtailed inside the environment model. Since our algorithms are conceptually independent of the type of spatial subdivision employed, we have chosen to express algorithm running times as functions of the length of the portal sequences over which the algorithms are run. We reason that the average length of a portal sequence is an inherent attribute of the geometric model, and that the relative goodness or badness of the spatial subdivision should not affect this average length by more than a constant factor. In practice, moreover, for most architectural environments to which these algorithms will be applied, the average stabbable portal sequence length is itself a small constant, perhaps ten or twenty.

There are four critical algorithms whose running times and storage complexities are of theoretical and practical interest here. First, a conforming spatial subdivision must be constructed. We express the construction times and sizes as functions of f, the number of occluders. Next, stabbing, antipenumbra, and on-line queries must be performed. We consider the latter algorithm running times as functions of the portal sequence length n, total edge complexity e, and antipenumbra boundary complexity b (in 3D). First, how efficiently can a *stabbing line* be found through a portal sequence? Second, how efficiently can the *antipenumbra* be cast through a portal sequence, and an object bounding box be examined for inclusion in the antipenumbra? Third, how efficiently can an eye-centered sightline be found through a portal sequence, and an object bounding box be examined for admission of an eye-centered sightline?

The complexity results are presented in the Table 10.1, organized by the three occluder input classes (2D, axial 3D, and general 3D), and by the algorithm of interest. The quantity f is the number of input occluders; $O(b)$ is the complexity of the antipenumbral boundary, which is at present known only to be upper-bounded by $O(e^2)$.

10.3 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. The point-location query will often produce the same result as its previous invocation, and this result (i.e., an identifier for the previously found cell) can be cached. The stab tree for that cell can also be cached 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 through a portal must emerge in a neighbor of that cell (our implementation disallows "walking through walls" by prohibiting incremental paths that penetrate occluders). A well-designed walkthrough system 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 [FST92].

10.4 Scaling to Larger Models

In practice, we have observed that these subdivision and visibility methods scale well with increasing model size. Our test model appeared to have a fairly constant visual complexity, even when increased

Occluder Class ⇒ Algorithm ⇓	General 2D	Axial 3D	General 3D
Spatial Subdivision	$O(f \lg f)$ time $O(f)$ space	$O(f^{\frac{3}{2}})$ time $O(f^{\frac{3}{2}})$ space in [PY89]	$O(f^3)$ time $O(f^2)$ space in [PY90]
Static Portal Stabbing	$O(n)$	$O(n \lg n)$ $O(n)$ in [Ame92, Meg91]	$O(e^2)$
Static Antipenumbra	$O(n)$	$O(n^2)$ polyhedral bounds in $O(n \lg n)$	$O(e^2)$
Static Object Incidence (per object)	$O(1)$	$O(1)$	$O(b)$ $O(e)$ linearized
On-line Portal Stabbing (per portal)	$O(1)$	$O(1)$	$O(e)$
On-line Object Incidence (per object)	$O(1)$	$O(1)$	$O(e)$

Table 10.1: Summary of algorithm complexities for operations described in this thesis, as functions of: f, the number of occluders; n, the length of an active portal sequence; e, the total number of edges in a 3D portal sequence; and b, which is $O(e^2)$, the worst-case complexity of the 3D antipenumbral boundary.

from a single floor to seven floors. The achieved rendering speedup therefore improved almost linearly as the size of the model increased. Indeed, one could imagine replicating the building model vertically and horizontally several times, without significantly increasing the number of occluders and detail objects visible to an interior observer. This is consistent with the intuitive notion that the internal visual complexity of many architectural environments "tops out" locally, rather than globally. That is, the visual complexity of most architectural models tends to be 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. Even if this fact could be determined visually (say, by looking out of a window), the remainder of the model would not be visible in its entirety.

Theoretical complexity measures can be reasonably good indicators of how well a given algorithm will scale. However, one must be wary of algorithmic constants, since they are hard to express exactly for non-trivial algorithms. Moreover, any implementation of an algorithm will be, at some level, highly machine-dependent. In our case, the walkthrough system depends on such quantities as computation, memory, bus, and disk bandwidth, and on such hard-to-quantify graphics characteristics as transformation, clipping, and fill rates [DL90]. Since there is no "typical" rendering load, one must often gather empirical data (as we have done) as one measure of the utility of an implemented algorithm.

Scaling must also be regarded in the context of a practical difficulty, the problem of subdivision in "free space." Our algorithms are posited on the assumption that many occluders exist, and that they

are good candidates for splitting space. However, current techniques for subdividing three-dimensional space have "non-local" effects. At early stages of splitting, some occluder must be chosen to split along. Choosing any such occluder may result in the introduction of a subdivision surface (i.e., a splitting plane) in a region of no occlusion, elsewhere in the model. This subdivision surface then later contributes to a large number of cell boundaries, none of which have any coaffine occluders. Consequently, the graph traversal algorithms expend computational resources to traverse these open portals. In §9.2.2, we described a modification to the data structures, precomputation, and query algorithms that overcomes local free-space cell clusters. However, global free-space problems are sure to arise when applying these techniques to, say, an office complex, in which many densely-occluded clusters populate an otherwise sparse space. In these instances, we propose an initial splitting round that attempts merely to separate clusters, rather than to find gross occlusion. This approach has been shown to improve the storage behavior of BSP trees in such environments [Tor90]. We found that the minimum-cleaving splitting criterion, originally formulated for building interiors (§5.2.1), efficiently found cluster separators in our "museum park" test dataset (§9.4.4).

10.5 Future Directions

Practical visibility determinations in three dimensions is by no means a closed subject. Fruitful avenues of investigation abound. Here, we briefly catalogue some areas which seem ripe for further attention.

10.5.1 Practical Spatial Subdivisions

Foremost is the question of efficient and effective spatial subdivision techniques in three dimensions. Managing complex spatial and general data is an important open problem. At present there exist no general, practical schemes for partitioning data in three dimensions in a local fashion (much as the constrained Delaunay triangulation and Voronoi diagram partition point and segment data in the plane [Sei90a]). Indeed, one has some reason to be pessimistic on this front, since even the task of determining whether or not a polyhedron can be tetrahedralized has been shown to be NP-hard [RS89]. Nevertheless, algorithms that do spatially subdivide real environments continue to emerge (e.g., [CP89, DBS91, MW91, SP91]). Although convexity of cells is a powerful and useful invariant to rely upon, perhaps this may be shown to be unnecessary as more powerful subdivision abstractions emerge (e.g., [Rap91]). Eventually, these schemes will have to handle more general environments as well, including curved surfaces, procedurally generated objects and occluders, and time-dependent entities such as moving objects and multiple visual simulation participants. We are optimistic about the use of cell-based visibility algorithms in large and ever-more variegated architectural models.

10.5.2 High-Order Visibility Effects

In the specific realm of visibility, there are many other interesting open questions. For example, how might the visibility computation influence the subdivision algorithm in order to increase subdivision in the regions where visibility is changing most rapidly? Our scheme approximates "zeroth order" visibility

effects by splitting on occluders, "first-order" effects by splitting along portal boundaries (i.e., at occluder edges), and partially recognizes "second-order" and "third-order" effects by identifying antipenumbral boundaries arising from two-way and three-way interactions among occluder (portal) edges. A more complete investigation of these multiple-order visibility effects, as well as effects due to the interaction among objects and occluders, is warranted. Visibility is an inherently *directional* phenomenon, and this too may be incorporated into the preprocessing phase, although angular variables are generally more difficult to manipulate than spatial variables.

The methods described here are particularly appropriate for densely occluded environments, such as many architectural models. However, if the model has many "holes", many distinct portals will be produced along cell boundaries, confounding the cell-to-cell and cell-to-object computations with a combinatorially explosive set of sightlines and halfspaces. This problem can be ameliorated by coalescing portals to allow at most one per cell boundary. Since portals must be convex, this generally results in an aggregate portal bigger than the union of its components, and larger (i.e., less useful) upper bounds on visibility. It would be useful to strike an optimal balance between these competing effects.

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 associated 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 motion of the child generalized observers 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. Another future direction of this research is the potential of adaptive cell and object subdivision, based on the results of the cell-based (cell, region and object) visibility computations.

10.5.3 Occluders and Objects

We have made an arbitrary distinction between major occluders and detail objects, since conceptually they have different occlusive properties and because, in practice, they were easily distinguishable in our data. However, detail objects and occluders clearly form a continuum, and at some density of objects, sparsity of occluders, or general homogeneity of both, they become indistinguishable. Another worthy area of investigation is the automated classification of model entities as occluders or objects (and the concomitant investigation of visibility effects arising therefrom).

10.5.4 Mirroring and Translucency

Real environments include translucent and reflective surfaces, which themselves affect visibility in complicated ways. Both surface types can be incorporated fairly easily into the visibility framework we describe. Translucent surfaces can act as occluders or portals, depending on the incident angle of light encountering them, their condition (i.e., cleanliness), and the difference in light levels on opposite sides

of the glass (the latter is a perceptual effect that may be exploited, for example, to preclude rendering dark surroundings as viewed from inside a bright house).

Mirrors, provided they are accompanied by explicit subdivision boundaries, can be treated as portals that impose a coordinate transform on all visibility searches encountering the mirror. This transform simply reflects the observer, and all active portals, about the plane of the mirror, and continues searching. Note that the portal "handedness" must also be reversed.

10.5.5 Visibility Algorithm Efficiency

One basic thrust of this thesis is the hierarchical chunking of data, and the establishment of visibility links between chunks at different levels of the hierarchy. Practically, the workhorse of this visibility search has been a stabbing-line or light-propagation algorithm that, for various abstractions of occluders and spatial subdivisions, finds straight-line paths or light bundles connecting chunks of the model. For real data, our implementation required six 20-MIP CPU-hours to determine complete cell-to-cell and cell-to-object visibility links for a realistic architectural model. More efficient algorithms, and faster implementations, would allow another dimension of interaction with the model (for example by altering an occluder, or opening or closing a door, and recomputing the affected visibility relationships). These effects can be simulated currently via a "conditional occluder" approach that embeds the conditional nature of a small number of occluders into the cell-based visibility sets. This approach, however, requires foreknowledge of all occluders and objects that may change position over time, and is not viable as a general technique.

10.5.6 Coherence and Parallelism

Both static and dynamic visibility computations are highly coherent. This coherence makes them attractive candidates for parallelization, since 1) visually isolated model regions can have no data interaction in the static phase, and can therefore be isolated in memory, and 2) the eye-based queries amount to read-only tree traversals in the on-line phase, and are straightforwardly parallelizable.

10.6 Other Applications

The subdivision-based visibility techniques described in this thesis can also be applied in other domains. Some of these are briefly discussed.

10.6.1 Global Illumination and Shadow Computations

Existing global illumination algorithms treat visibility as a difficulty to be overcome (typically) by sampling [CG85, HSA91]. However, these computations can be *driven* by visibility determinations in the sense that only visually interacting entities should be allowed to exchange energy directly. Thus, for example, the $O(n^2)$ "for each face, for every other face" construct in [HSA91] could be replaced by

"for each face, for every other visible face"; which will typically be sub-quadratic in occluded environments. Moreover, visually isolated subregions of the model can be solved with independent radiosity computations, to first order.

The antipenumbra computation algorithm of §8.3.2 can be slightly modified to produce all of the VE and EEE critical surfaces induced by a given portal sequence. These critical surfaces encode strong and weak visibility information about source and receiver polygons, and the first- and second-derivative illumination discontinuities that arise due to interactions between two or three occluders. Polygon meshing algorithms that wish to capture these discontinuities as data objects (see, e.g., [BRW89, CF90, Hec91, SLD92]) should find the antipenumbra characterization useful.

10.6.2 Geometric Queries

The subdivision and portal abstractions might be of use to computational geometers, in answering queries about the entities intersected by a given line segment or volume, or about collision-free paths between pairs of points. Similarly, ray-tracing queries typically use sampling to determine the fraction of an area light source visible to a point; analytic visibility-based methods could be used instead.

Chapter 11

Conclusions

We have shown that practical visibility preprocessing and on-line culling is achievable, both theoretically and as a functioning implementation, for an important class of densely-occluded environments. The visibility scheme first subdivides a geometric model into spatial cells, introducing cell boundaries wherever major occluders are present in the model. Next, a coarse visibility determination links cells, and objects within them, that are mutually visible through sightlines, or incident on conservative light bundles.

This coarse visibility determination constitutes a per-cell potentially visible set or PVS of objects as an upper bound on the set of objects visible to any actual observer in a given cell. Thus, a simulated actual observer can be tracked through the spatial cells, while the coarse visibility information stored with each cell is retrieved, and subjected to on-line culling operations. The observer's instantaneous view position and field of view are used to reduce the set of objects potentially visible from *anywhere* in the cell, to those potentially visible from the observer's eyepoint. The resulting subset, typically a small fraction of the model data, is then issued to graphics hardware for rendering and discretized hidden-surface removal.

These subdivision and visibility determination techniques have been implemented for general polyhedral geometric models in three dimensions. Using a furnished model of a planned computer science building as test data, we have achieved static culling rates of over 90% and dynamic culling rates of more than 99%, on average, decreasing the average refresh rates of a simulated walk through the model by a factor of about one hundred. The subdivision and visibility precomputation stages required several hours of compute-time for a five-story building, a reasonable figure given the design time cycle for a project of this size. The results of the visibility computation incurred less than 10% storage overhead (15 megabytes) with respect to the amount of storage required by the geometric model itself (180 megabytes).

Together, these techniques demonstrate a successful application of computational geometric and computer graphics techniques to the engineering problem of visually simulating a very large geometric model. Due to the hierarchical and local nature of the partitioning and constrained graph traversal computations involved, these techniques should scale effectively to even more complex models. Finally, the visibility computations should have other applications in areas such as global illumination, ray-

tracing, and shadow computations.

Bibliography

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[Air90] 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.

[Air92] John Airey. Personal communication, 1992.

[AK87] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics (Proc. SIGGRAPH '87)*, 21(4):55––63, 1987.

[Ake89] Kurt Akeley. The Silicon Graphics 4D/240GTX superworkstation. *IEEE Computer Graphics and Applications*, 9(4):71––83, 1989.

[Ame92] Nina Amenta. Finding a line traversal of axial objects in three dimensions. In *Proc.* 3^{rd} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 66––71, 1992.

[ARB90] 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.

[Arv88] James Arvo. Ray tracing with meta-hierarchies. *SIGGRAPH '88 Course Notes (Introduction to Ray Tracing)*, 1988.

[Aut90] AutoDesk, Inc. Autocad reference manual release 10, 1990.

[AW87] D. Avis and R. Wenger. Algorithms for line traversals in space. In *Proc.* 3^{rd} *Annual Symposium on Computational Geometry*, pages 300––307, 1987.

[Bau72] B. Baumgardt. Winged-edge polyhedron representation. Technical Report No. CS-320, Stanford Artificial Intelligence Report, Computer Science Department, 1972.

[Bel79] Bell Telephone Laboratories, Incorporated. *UNIX User's Manual*, 1979.

[Bel92] Gavin Bell. Personal communication, August 1992.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509––517, 1975.

[BRW89] Daniel R. Baum, Holly E. Rushmeier, and James M. Winget. Improving radiosity solutions through the use of analytically determined form factors. *Computer Graphics (Proc. SIGGRAPH '89)*, 23(3):325––334, 1989.

[CF90] A.T. Campbell III and Donald S. Fussell. Adaptive mesh generation for global diffuse illumination. *Computer Graphics (Proc. SIGGRAPH '90)*, 24(4):155––164, 1990.

[CF92] Norman Chin and Steven Feiner. Fast object-precision shadow generation for area light sources using BSP trees. In *Proc. 1992 Symposium on Interactive 3D Graphics*, pages 21––30, 1992.

[CG85] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics (Proc. SIGGRAPH '85)*, 19(3):31––40, 1985.

[CHG+88] K. Clarkson, H.Edelsbrunner, L. Guibas, M. Sharir, and E. Welzl. Combinatorial complexity bounds for arrangements of curves and surfaces. *Proc. 29^{th} Annual IEEE Symposium on Foundations of Computer Science*, pages 568––579, 1988.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *CACM*, 19(10):547––554, 1976.

[Com90] Computer Science 270: Combinatorial Optimization. Class notes. NP-completeness of minimal parallelepipedization, 1990.

[CP89] Bernard Chazelle and Leonidas Palios. Triangulating a nonconvex polytope. In *Proc. 5^{th} Annual ACM Symposium on Computational Geometry*, pages 393––400, 1989.

[CS86] R. Cole and M. Sharir. Visibility problems for polyhedral terrains. Technical Report 92, NYU Courant Institute of Mathematical Sciences, Computer Science Division, 1986.

[CS89] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry II. *Discrete Computational Geometry*, pages 387––421, 1989.

[DBS91] Tamal Dey, Chanderjit Bajaj, and Kokichi Sugihara. On good triangulations in three dimensions. In *Proceedings of 1991 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 431––441, 1991.

[DFP+86] Leila DeFloriani, Bianca Falcidieno, C. Pienovi, D. Allen, and George Nagy. A visibility-based model for terrain features. In *Proc. Int. Symp. on Spatial Data Handling*, July 1986.

[DG87] J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *CACM*, 30(5):403––407, 1987.

[DL90] J. Craig Dunwoody and Mark A. Linton. Tracing interactive 3D graphics programs. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24(2):155––163, 1990.

[DS89] David P. Dobkin and Diane L. Souvaine. Detecting the intersection of convex objects in the plane. Technical Report No. 89-9, DIMACS, 1989.

[dV91] René de Vogelaere. Personal communication, November 1991.

[Ede85] H. Edelsbrunner. Finding transversals for sets of simple geometric figures. *Theoretical Computer Science*, 35:55––69, 1985.

[EGS86] Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. Optimal point location in monotone subdivisions. *SIAM Journal of Computing*, 15:317––340, 1986.

[FI85] Akira Fujimoto and Kansei Iwata. Accelerated ray tracing. *Computer Graphics: Visual Technology and Art (Proc. Computer Graphics Tokyo '85)*, pages 41––65, 1985.

[FK91] Sharon Fischler and George Kong. *Graphics Tuning, Section 20 of the Power Series Performance Guide, Ed. Veeleen Roufchaie.* Silicon Graphics Computer Systems, 2011 N. Shoreline Boulevard, Mountain View, CA 94039-7311, February 1991.

[FKN80] 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.

[FST92] Thomas A. Funkhouser, Carlo H. Séquin, and Seth Teller. Management of large amounts of data in interactive building walkthroughs. In *Proc. 1992 Workshop on Interactive 3D Graphics*, pages 11––20, 1992.

[FvD82] J.D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1982.

[GBW90] 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.

[GCS91] Ziv Gigus, John Canny, and Raimund Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):542––551, 1991.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability.* Freeman, 1979.

[Gla84] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15––22, 1984.

[GM90] Ziv Gigus and Jitendra Malik. Computing the aspect graph for line drawings of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):113––122, 1990.

[Grü67] Branko Grünbaum. *Convex Polytopes.* Wiley-Interscience, New York, 1967.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74––123, 1985.

[GV89] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[Han84] Patrick Hanrahan. A homogeneous geometry calculator. Technical Report 3-D No. 7, Computer Graphics Laboratory, New York Institute of Technology, 1984.

[Hec91] Paul S. Heckbert. *Simulating Global Illumination Using Adaptive Meshing.* PhD thesis, Computer Sciences Department, UC Berkeley, June 1991.

[Her87] John E. Hershberger. *Efficient Algorithms for Shortest Path and Visibility Problems.* PhD thesis, Stanford University, 1987.

[HSA91] Patrick Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics (Proc. SIGGRAPH '91)*, 25(4):197––206, 1991.

[HT92] Michael E. Hohmeyer and Seth Teller. Stabbing isothetic rectangles and boxes in $O(n \lg n)$ time. *Computational Geometry Theory and Applications*, 4:201––207, December 1992.

[Jon71] C.B. Jones. A new approach to the 'hidden line' problem. *The Computer Journal*, 14(3):232––237, 1971.

[Jon92] Michael T. Jones. A high performance visual simulation toolkit. Technical report, Silicon Graphics Computer Systems, Mountain View CA, 94043, March 1992.

[Kho91] Delnaz Khorramabadi. A walk through the planned CS building. Technical Report UCB/CSD 91/652, Computer Science Department, UC Berkeley, 1991.

[Kir83] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal of Computing*, 12:28––35, 1983.

[KLZ85] M. Katchalski, T. Lewis, and J. Zaks. Geometric permutations for convex sets. *Discrete Mathematics*, 54:271––284, 1985.

[Kv79] J.J. Koenderink and A.J. van Doorn. The internal representation of solid shape with respect to vision. *Biol. Cybern.*, 32:211––216, 1979.

[KV90] David Kirk and Douglas Voorhies. The rendering architecture of the DN10000VS. *Computer Graphics (Proc. SIGGRAPH '90)*, 24(4):299––307, 1990.

[Meg83] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal Computing*, 12:759––776, 1983.

[Meg91] N. Megiddo. Stabbing isothetic boxes in deterministic linear time. *Personal communication to Nina Amenta*, 1991.

[Mit92] Don Mitchell. On the quadratic behavior of BSP trees for real-world data. *Personal Communication*, 1992.

[MO88] M. McKenna and J. O'Rourke. Arrangements of lines in 3-space: A data structure with applications. In *Proc.* 4^{th} *Annual Symposium on Computational Geometry*, pages 371––380, 1988.

[MW91] Doug Moore and Joe Warren. Bounded aspect ratio triangulation of smooth solids. In *Proceedings of 1991 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 455––464, 1991.

[NN83] Tomoyuki Nishita and Eihachiro Nakamae. Half-tone representation of 3-D objects illuminated by area sources or polyhedron sources. In *Proc. IEEE COMPSAC, 1983*, pages 237––242, 1983.

[NN85] Tomoyuki Nishita and Eihachiro Nakamae. Continuous-tone representation of three-dimensional objects taking account of shadows and interreflection. *Computer Graphics (Proc. SIGGRAPH '85)*, 19(3):23––30, 1985.

[O'R87] Joseph O'Rourke. *Art Gallery Theorems and Algorithms.* Oxford University Press, 1987.

[PD90] W.H. Plantinga and C.R. Dyer. Visibility, occlusion, and the aspect graph. *Int. J. Computer Vision*, 5(2):137––160, 1990.

[Pel90a] Marco Pellegrini. Stabbing and ray-shooting in 3-dimensional space. Technical Report 540; Robotics Report No. 230, NYU Courant Institute of Mathematical Sciences, Computer Science Division, 1990.

[Pel90b] Marco Pellegrini. Stabbing and ray-shooting in 3-dimensional space. In *Proc.* 6^{th} *ACM Symposium on Computational Geometry*, pages 177––186, 1990.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, 1985.

[PW88] Ken Perlin and Xue-Dong Wang. An efficient approximation for penumbra shadow. Technical Report 346, NYU Courant Institute of Mathematical Sciences, Computer Science Division, 1988.

[PW89] R. Pollack and R. Wenger. Necessary and sufficient conditions for hyperplane traversals. In *Proc.* 5^{th} *Annual Symposium on Computational Geometry*, pages 152––155, 1989.

[PY89] 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.

[PY90] 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.

[Rap91] Ari Rappoport. The n-dimensional extended convex differences tree (ECDT) for representing polyhedra. In *Proceedings of 1991 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 139––147, 1991.

[RS89] Jim Ruppert and Raimund Seidel. On the difficulty of tetrahedralizing 3-dimensional non-convex polyhedra. In *Proc.* 5^{th} *Annual ACM Symposium on Computational Geometry*, pages 380––392, 1989.

[SBGS69] R. A. Schumacker, B. Brand, M Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL TR-69-14, U.S. Air Force Human Resources Laboratory, 1969.

[Sei90a] Raimund Seidel. Constrained Delaunay triangulations and Voronoi diagrams with obstacles. *Extended Abstract*, 1990.

[Sei90b] Raimund Seidel. Linear programming and convex hulls made easy. In *Proc.* 6^{th} *ACM Symposium on Computational Geometry*, pages 211––215, 1990.

[Sha87] Micha Sharir. The shortest watchtower and related problems for polyhedral terrains. Technical Report 334, NYU Courant Institute of Mathematical Sciences, Computer Science Division, 1987.

[SLD92] David Salesin, Dani Lischinski, and Tony DeRose. Reconstructing illumination functions with selected discontinuities. In *Proc.* 3^{rd} *Eurographics Workshop on Rendering*, pages 99–112, May 1992.

[Som59] D.M.Y. Sommerville. *Analytical Geometry of Three Dimensions*. Cambridge University Press, 1959.

[SP91] Nickolas Sapidis and Renato Perruchio. Domain Delaunay tetrahedrization of arbitrarily shaped curved polyhedra defined in a solid modeling system. In *Proceedings of 1991 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 465––480, 1991.

[SS90] Sack and Suri. An optimal algorithm for detecting weak visibility of a polygon. *IEEE Transactions on Computers*, 0(0):0––0, 1990.

[SS91] Carlo H. Séquin and Kevin P. Smith. Introduction to the Berkeley UNIGRAFIX tools (Version 3.0). Technical Report UCB/CSD 91/606, Computer Science Department, UC Berkeley, 1991.

[SSS74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1––55, 1974.

[SSW83] Carlo H. Séquin, Mark Segal, and Paul R. Wensley. UniGrafix 2.0 user manual and tutorial. Technical Report UCB/CSD 83/161, Computer Science Department, UC Berkeley, 1983.

[ST86] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *CACM*, 29(7):669––679, 1986.

[Sto89] Jorge Stolfi. Primitives for computational geometry. Technical Report 36, DEC SRC, 1989.

[TH92] Seth Teller and Michael E. Hohmeyer. Stabbing oriented convex polygons in randomized $O(n^2)$ time. Technical Report UCB/CSD 91/669, Computer Science Department, UC Berkeley, 1992.

[Tor90] Enric Torres. Optimization of the binary space partition algorithm (bsp) for the visualization of dynamic scenes. In *Proc. 1^{st} Eurographics Workshop on Rendering*, pages 507––518, 1990.

[WA77] Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. *Computer Graphics (Proc. SIGGRAPH '77)*, 11(2):214––222, 1977.

[WMB92] Allan Wilks, Allen McIntosh, and Richard A. Becker. The data dual. *In preparation*, 1992.