[4] Bentley, J.L. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18 (1975), 509-517.

[5] Brooks, Jr., Frederick P. Walkthrough - A Dynamic Graphics System for Simulating Virtual Buildings. *Proceedings of the 1986 Workshop on Interactive 3D Graphics*.

[6] Clark, James H. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, 19, 10 (October 1976), 547-554.

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

[8] Jones, C.B. A New Approach to the 'Hidden Line' Problem. *The Computer Journal*, 14, 3 (August 1971), 232-237.

[9] Khorramabadi, Delnaz. A Walk through the Planned CS Building. Masters Thesis UCB/CSD 91/652, Computer Science Department, U.C. Berkeley, 1991.

[10] Séquin, Carlo H. Introduction to the Berkeley UNIGRAFIX Tools (Version 3.0). Technical Report UCB/CSD 91/606, Computer Science Department, U.C.Berkeley, 1991.

[11] Teller, Seth J., and Séquin, Carlo H. Visibility Preprocessing for Interactive Walkthroughs. *Computer Graphics (Proc. SIGGRAPH '91)*, 25, 4 (August 1991), 61-69.

[12] Zyda, Michael J. Course Notes, Book Number 10, Graphics Video Laboratory, Department of Computer Science, Naval Postgraduate School, Monterey, California, November 1991.

most likely to be visible to the observer in upcoming frames in order of decreasing urgency. One of the two processors of the VGX is used for pre-fetching data concurrently with the rendering of the current frame. The results presented here were gathered from a walk along the test path shown in Figure 13. Since the current floor model is not very large compared to the memory capacity of our machine, we impose an artificial 8MB limit on the amount of object data that can be stored in memory at any one time. As the observer, "walks" along the path, we swap data in and out of memory, never exceeding the 8MB limit. We are still experimenting with techniques to control the interaction between our memory management algorithm and the paging of the operating system. Thus the data below must be regarded as tentative and rather preliminary. More reliable data will be gathered once the fully furnished model of the whole building becomes available.

Figure 14 shows a plot of the number of bytes that must be in memory in order to render the visible parts of the scene (lower curve); superimposed is a plot of the number of bytes our algorithm loads into memory in preparation for possible near-term observer moves. As expected, these amounts of data fluctuate strongly depending on whether the observer is in a relatively simple part of the model with rather confined views, or whether the visible cells stretch out to great depth along several directions. In all, we read 52MB during the 261 frames.

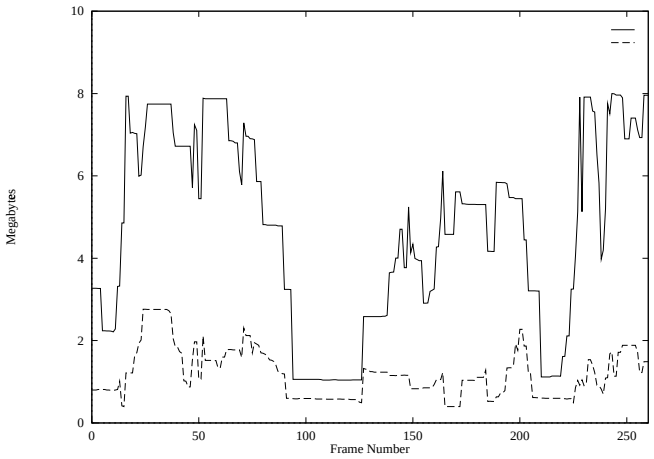

Figure 14: Comparison of the amounts of data fetched from disk (top curve) and actually needed for rendering (bottom curve) while following the walkthrough test path; marked spots correspond to the labels shown in Figure 13.

In general, we are able to pre-fetch objects before they are rendered, and so the observer can move smoothly through the model. However, there are a few cases in which the memory manager is not able to predict which objects are going to become visible to the observer far enough in advance to pre-fetch them, and so the user may have to wait while they are read into memory. As the observer turns a corner in a corridor, the visible set of objects can change dramatically This prompts a request for a large amount of new data to be loaded into memory. For the worst-case corners (labels 'B' and 'C'), the coprocessor is busy for about 8 seconds to prefetch on the order of 2 MB of data that might be used in the near future. However, the amount of data needed immediately for the rendering of the next frame is much smaller; because of parallel processing, resulting observable delays are on the order of a couple of seconds for a worst-case situation in our model. We are developing more sophisticated pre-fetching techniques that use a better prediction of the observer's motion.

7 Conclusion

Our paper describes a system for interactive walkthroughs of very large architectural models. It builds a hierarchical display database containing objects represented at multiple levels of detail during the modeling phase, performs a spatial subdivision and visibility analysis during a precomputation phase, and uses real-time display and memory management algorithms during a walkthrough phase to judiciously select a relevant subset of data for rendering. We have implemented a first version of this system, and tested it in real walkthroughs of a completely furnished model of the sixth floor of the planned Computer Science building at UC Berkeley. Our initial results show that these display and memory management techniques are effective at culling away substantial portions of the model, and make interactive frame rates possible even for very large models.

8 Acknowledgements

We are grateful to Delnaz Khorramabadi for her efforts constructing the building model, and to Paul Haeberli for his help in producing the color plates. Silicon Graphics, Inc., donated a 320 VGX workstation to this project as part of a grant from the Microelectronics Innovation and Computer Research Opportunities (MICRO) program of the State of California.

References

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

[2] Airey, John M., Rohlf, John H., and Brooks, Jr., Frederick P. 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 (1990), 41-50.

[3] *Autocad Reference Manual*, Release 10, Autodesk Inc., 1990.

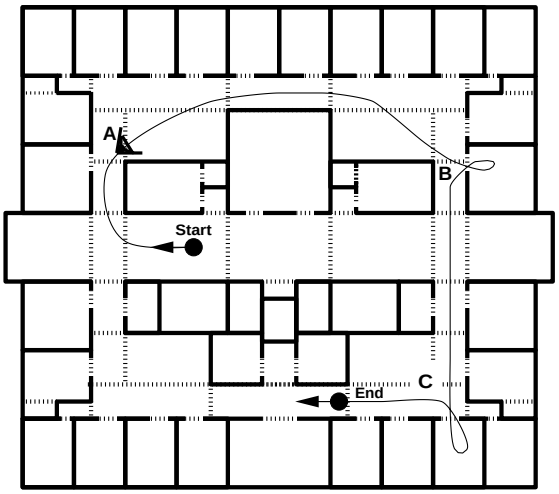

Figure 13: Test path through the building model.

Display Management

As discussed in Section 5.1, we compute the set of potentially visible objects by generating successively smaller supersets, culling away objects invisible to the observer. The sizes of these sets, and the times (in seconds) required to render them are shown for viewpoint 'A' in Table 1 and averaged over the test walkthrough path in Table 2. On average, we are able to cull away 94% of the model and reduce rendering time by a factor of 17 by rendering only objects in the eye-to-object visibility set rather than the entire building model.

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

Table 1: Visibility cull results for viewpoint 'A'.

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

Table 2: Average visibility cull results for test walkthrough.

We further reduce the number of faces rendered at each frame by choosing an appropriate level of detail at which to render each potentially visible object based on its apparent size and speed to the observer. Statistics regarding the number of faces and the time required to render each frame using different pixels-per-face thresholds for viewpoint 'A' and averaged over the test path are shown in Tables 3 and 4, respectively. Usable rendering modes for which little or no degradation in image quality is perceptible ($\geq$ 256 pixels per face), are shown in bold typeface.

Color Plates IV, V and VI show the difference between a static image produced using the highest level of detail for all objects (Plate IV) and one generated with reduced levels of detail for objects with fewer than 256 pixels per face (Plate V). Plate IV has 23,468 faces and took 0.34 seconds to render, whereas Plate V has 7,555 faces and took 0.17 seconds. These images were rendered without interpolated shading or antialiasing in order to accentuate differences – notice the reduced tessellation of the chairs further from the observer. Plate VI shows which level of detail was used for each object in Plate V (a darker shade represents a higher level of detail).

Overall, after computing the set of potentially visible objects and choosing an appropriate level of detail for each object, we are able to cull away an average of 97% of the building model and reduce rendering time by an average factor of 39 in each frame.

Min. Pixels Per Face	# Objs.	# Faces	Draw Time	% of Model
0	**165**	**18,927**	**0.33**	**7.8%**
64	**165**	**11,763**	**0.26**	**4.8%**
128	**165**	**8,861**	**0.22**	**3.6%**
256	**165**	**6,204**	**0.17**	**2.6%**
512	165	3,889	0.13	1.6%
1024	165	2,871	0.12	1.2%

Table 3: Average detail cull results for viewpoint 'A'.

Min. Pixels Per Face	# Objs.	# Faces	Draw Time	% of Model
0	**141**	**13,701**	**0.23**	**5.6%**
64	**141**	**9,700**	**0.18**	**4.0%**
128	**141**	**7,979**	**0.16**	**3.3%**
256	**141**	**6,176**	**0.14**	**2.5%**
512	141	4,745	0.12	2.0%
1024	141	3,427	0.10	1.4%

Table 4: Average detail cull results for test walkthrough.

Memory Management

As described in Section 5.2, the memory manager tries to store in memory the objects incident upon the cells that are

at the lowest level of detail for which the average size of a face is greater than some threshold, and the size of an average face divided by its speed is greater than another threshold. If either of these values is less than the corresponding threshold for all available levels of detail of an object, we render the object at its lowest level of detail.

As the observer moves through the model, an object may be rendered at different levels of detail in successive frames. Rather than abruptly snapping from one level of detail to the next, we blend successive levels of detail using partial transparency. Since the complexity of any level is typically small compared to the one of the next higher higher level (by more than a factor of two), the extra time spent blending the two levels during transition does not constitute an undue overhead, considering the small fraction of objects making a transition at the same time.

5.2 Memory Management

Since the entire model cannot be stored in memory at once, we must choose a subset of objects to store in memory for each frame, and swap objects in and out of memory in real-time as the observer moves through the model. As a minimum, we must store in memory all objects to be rendered in the next frame. However, since it takes a relatively large amount of time to swap data from disk into memory, we must also predict which objects might be rendered in future frames and begin swapping them into memory in advance. Otherwise, frame updates might be delayed, waiting for objects to be read from disk before they can be rendered.

As described in Section 4.2, we group each level of detail for all objects incident upon the same cell contiguously in the display database. To take advantage of the relative efficiency of large IO operations, we always load all objects incident upon the same cell into memory together at the same level of detail. Thus, our memory management algorithm must compute for each frame which cell contents to store in memory at which levels of detail.

In general, we store in memory the contents of the cells containing the objects most likely to be rendered in upcoming frames. Specifically, we determine which cells are most likely to contain the observer in upcoming frames, and store in memory all objects incident upon cells visible from any of these cells. Each time the observer steps across a cell boundary, we traverse the cell adjacency graph, considering cells in order of the minimum amount of time before the cell can possibly contain the observer using a shortest path algorithm. The user interface also enforces some limits on the size of a step or turn that the observer may take in a single frame. For each cell C, visited in the search, we mark and claim memory for the contents of all cells visible from C in the direction of the observer's frustum up to the precomputed maximum level of detail at which any object incident upon the cell might be rendered for an observer in C. Our search terminates when all available memory has been claimed or when we have considered all possible observer viewpoints more than some maximum amount of time in the future. We then read the contents of all newly marked cells into memory, possibly replacing the contents of unmarked cells.

For instance, consider the observer viewpoint shown in Figure 12. Cells are labeled by the minimum amount of time (in seconds) before they can possibly become visible to the observer; and shaded by the level at which their contents are stored in memory – darker shades represent higher levels. The cells surrounded by the thick-dashed line represent the cells visited during the search, i.e. the range of observer positions for which we store visible objects in memory.

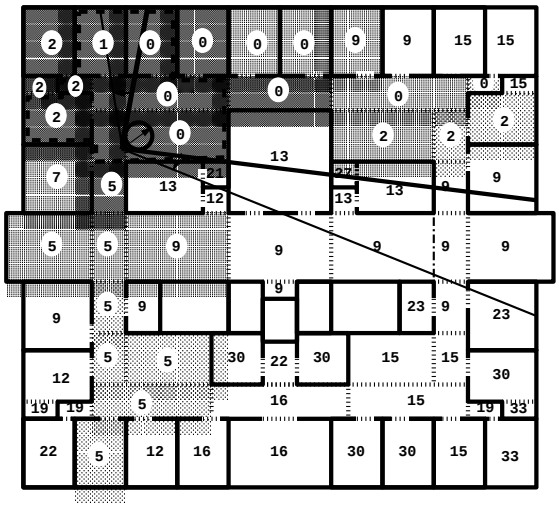

Figure 12: Cells labeled by the number of seconds before they can possibly become visible to the observer, and shaded by level of detail stored in memory (a darker shade represents a higher level of detail). White cells are not loaded into memory.

6 Results and Discussion

In this section we present and analyze test results collected during real interactive walkthroughs performed with our system. During these tests, we logged statistics regarding the performance of our display and memory management algorithms in real time as a user walked through the building model.

We present results for one observer viewpoint used as an example in the previous discussion (marked by an 'A' in Figure 13), as well as for a full sequence of observer viewpoints generated during an actual walkthrough along the path shown in Figure 13). The path is about 300 feet long, and a realistic physical walk along it should take approximately one minute. All tests were performed on a VGX 320 Silicon Graphics workstation with two 33 MHz processors and 64 MB of memory.

Since the observer is at a known point and has vision limited to a *view cone* emanating from this point, we can cull the set of visible objects even further. We define the *eye-to-cell* visibility as the set of all objects incident upon any cell partially or completely visible to the observer (the light stippled regions in Figure 9). Clearly, the eye-to-cell visibility is also a superset of the objects actually visible to the observer. The visible area in any cell is always the intersection of that (convex) cell with one or more (convex) wedges emanating through portals from the eyepoint. To compute the eye-to-cell visibility, we initialize the visible area wedge to the interior of the view cone, and the eye-to-cell visibility to the source cell. Next, we perform a constrained depth-first-search (DFS) of the stab tree, starting at the source cell, and propagating outward. Upon encountering a portal, the wedge is suitably narrowed, and the newly reached cell is added to the eye-to-cell visibility set. If the wedge is disjoint from the portal, the active branch of the DFS is terminated.

Finally, we estimate the *eye-to-object* visibility, a narrower superset of the objects actually visible to the observer, by generating the intersection of the cell-to-object and eye-to-cell sets. For example, consider the observer viewpoint shown in Figure 9. The eye-to-object visibility set (filled squares) contains all objects in the intersection between the cell-to-object (all squares) and eye-to-cell (gray regions) sets. It is a small subset of all objects in the model, but still an over-estimate of the actual visibility of the observer. In Figure 9, only one square lies in a cell visible to the observer and can be seen from some point inside the cell containing the observer, but is not visible from the observer's current viewpoint. Color Plate III depicts the eye-to-object visibility set for this observer viewpoint in three dimensions.

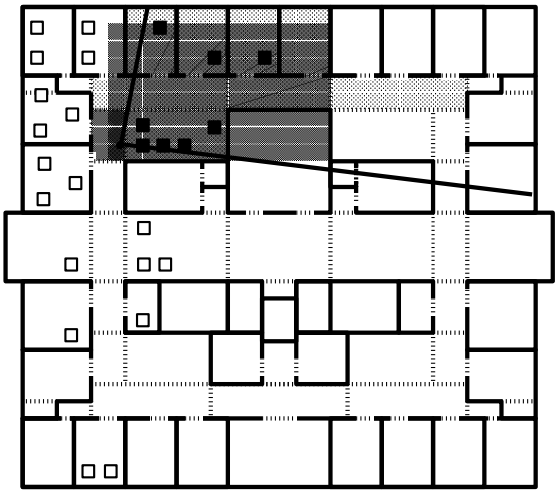

Figure 9: Eye-to-object visibility. Shown are only the potentially visible objects, i.e. the black objects from Figure 5.

Object Hierarchy

After we have culled away portions of the model that are invisible from the observer's viewpoint, we can further reduce the number of faces rendered in each frame by choosing an appropriate level of detail at which to render each visible object. Since the image must ultimately be displayed in pixels, it is useless to render very detailed descriptions of objects that are very small or far away from the observer and which map to just a few pixels on the display (Figure 10). Likewise, it is wasteful to render details in objects that are moving quickly across the screen and which appear blurred or can be seen for only a short amount of time (Figure 11). Instead, we can achieve the same visual effect by rendering simpler representations of these objects, consisting of just a few faces with appropriate colors. This is a technique used by commercial flight simulators, however little has been published on these systems [12].

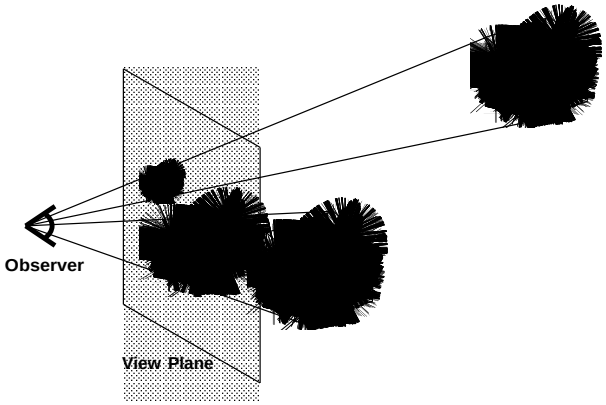

Figure 10: Perceptible detail is related to apparent size.

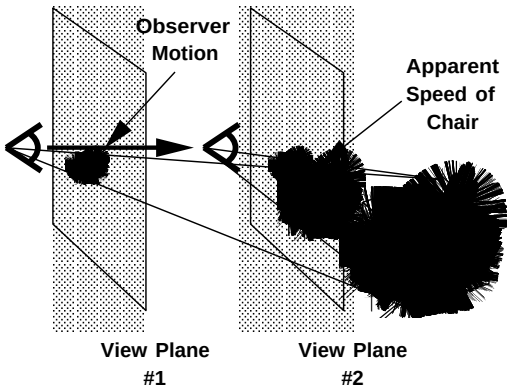

Figure 11: Perceptible detail is related to apparent speed.

Rather than rendering all objects at the highest level of detail in every frame, we choose a level of detail at which to render each object based on its apparent size and speed from the point of view of the observer. For each level of detail, we estimate the size of an average face in pixels, and the speed of an average face in pixels per frame. We render an object

is stored with each segment so that segments can be read and released through multiple segment references quickly and transparently.

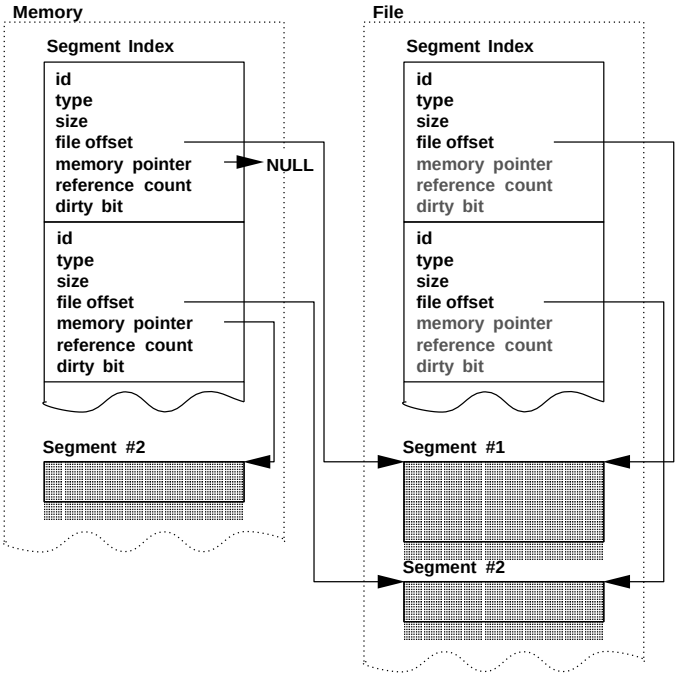

Figure 7: The implementation of display database segments.

4.2 Layout

Since the latency overhead of each read operation is relatively large, we group the segments for all objects incident upon the same cell contiguously in the display database file. This layout allows us to utilize the cell-to-cell visibility information from the precomputation phase to load groups of objects (those likely to become visible at the same time) into memory in a single IO operation. If an object is incident upon more than one cell (i.e. straddles a cell boundary), then we store it redundantly, once for each cell.

Furthermore, we store descriptions of all objects incident upon the same cell at the same level of detail contiguously in the display database, as shown in Figure 8. Within a single cell, the object headers appear first, followed by descriptions of the objects at increasing levels of detail. As a result, all objects incident upon a cell at or up to any level of detail may be read at once in a single read operation during the interactive walkthrough phase.

5 The Walkthrough Phase

During the walkthrough phase, we simulate an observer moving through the architectural model under user control. The goal is to render the model as seen from the observer's viewpoint in a window on the workstation display at interactive frame rates as the user moves the observer's viewpoint through the model.

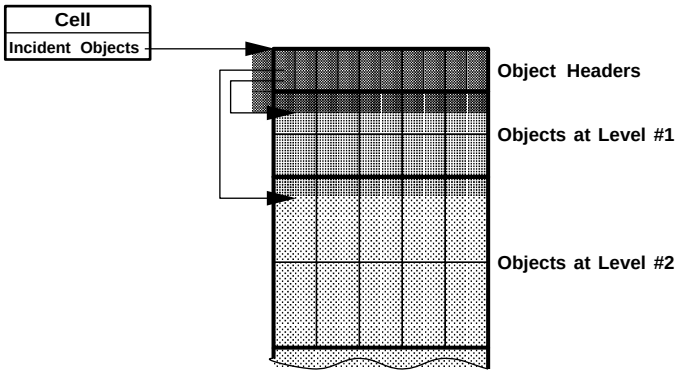

Figure 8: The layout of objects incident upon the same cell in the display database.

The primary problem is that building models are very large and so 1) do not fit into memory, and 2) cannot be rendered completely in an interactive frame time. Thus we must identify a small, but relevant, portion of the model to store in memory and to render in each frame. We use the results of the visibility precomputation along with the object hierarchy of the display database and dynamic culling algorithms to identify which objects are visible to the observer, and choose an appropriate level of detail for each one. We load into memory and render only relevant levels of detail for potentially visible objects.

5.1 Display Management

We use two techniques to reduce the amount of data rendered in each frame: 1) we compute the subset of objects visible to the observer using a real-time visibility analysis based on the results of the precomputation phase, and 2) we choose an appropriate level of detail at which to render each visible object from the object hierarchy constructed during the modeling phase. Using these techniques, we are able to cull away large portions of the model that are irrelevant from the observer's viewpoint, and therefore achieve much shorter refresh times. Moreover, computations are done in parallel with the display of the previous frame and do not increase the effective frame time.

Visibility Analysis

To compute the set of objects to render for a given observer viewpoint, we first identify the cell containing the observer's position and fetch its cell-to-object visibility from the display database. Since the cell-to-object visibility contains all objects visible from any viewpoint in a given cell, it is always a superset of the objects actually visible to a particular observer in that cell. It is typically a small subset of the entire model.

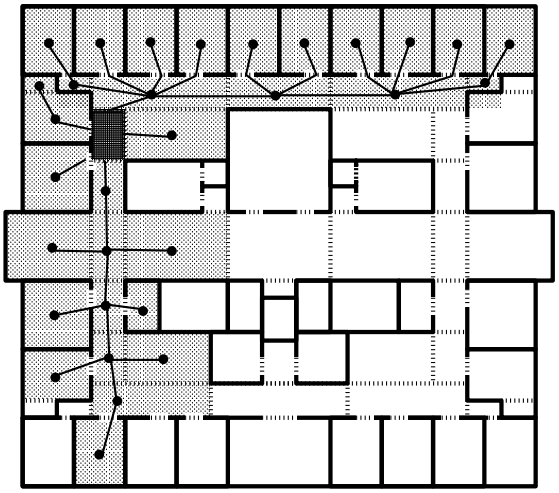

Figure 3: Cell-to-cell visibility and stab tree.

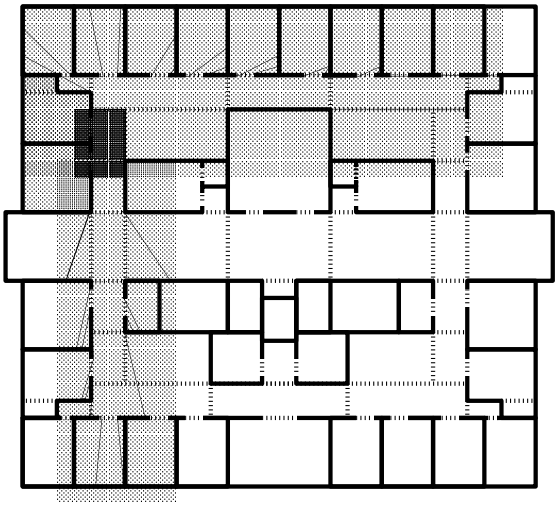

Figure 4: In general, only a fraction of the reached cell is visible to the source.

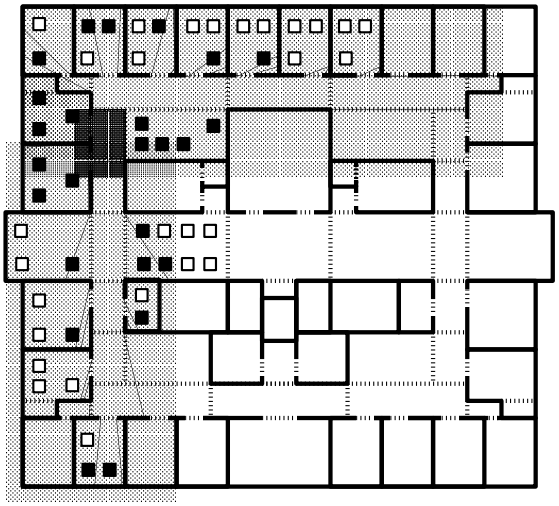

Figure 5: Computing cell-to-object visibility; the filled squares are marked visible.

4 The Display Database

The results of the modeling and precomputation phases are stored in a display database designed specifically to identify and swap relevant objects into memory quickly as the observer moves through the model during the interactive walkthrough phase. The structure of the display database is shown in Figure 6.

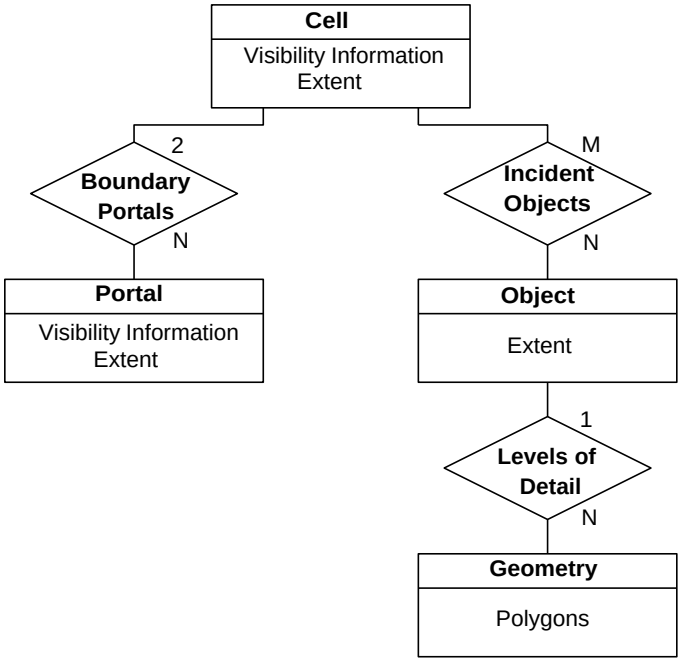

Figure 6: A structural diagram of the display database showing entities (boxes) and relationships (diamonds).

4.1 Segments

All entities (e.g. cells, portals, objects, etc.) are stored in *segments* in the display database. A segment is simply an abstraction for a variable-sized contiguous group of bytes in a display database file that can be read and released as a unit. Each segment is represented by its size, a byte offset into a file, and a pointer into memory, as shown in Figure 7. The arrangement of bytes in a segment is identical in memory and on disk so that only pointers within a segment must be updated when a segment is read (requiring one addition per pointer); there is no need to allocate extra memory or to move or copy bytes. With these properties, segments can be swapped quickly in and out of memory.

All relationships (e.g. adjacent, incident, visible, etc.) are stored in *segment references* in the display database. A segment reference can be represented by either an integer segment ID (if it has not yet been read into memory) or a pointer to a segment's data in memory. At any time, a segment reference may be read (converted from an ID to a pointer) or released (converted from a pointer to an ID). A reference count

The subdivision terminates when all sufficiently large, axial opaque elements in the model are coplanar with an axial boundary plane of at least one subdivision leaf cell.

After subdivision, cell *portals* (i.e., the transparent portions of shared boundaries) are identified and stored with each leaf cell, along with an identifier for the neighboring cell to which the portal leads (Figure 2). Enumerating the portals in this way amounts to constructing an *adjacency graph* over the leaf cells of the subdivision; two leaves (nodes) are adjacent (share an edge) if and only if there is a portal connecting them. All the visibility computations to be described exploit the adjacency graph data structure.

This procedure can be applied quickly. At the cost of performing an initial $O(n \lg n)$ sort, the split dimension and abscissa can be determined in time $O(f)$ at each split, where f is the number of faces stored with the node. We have found that these subdivision criteria yield a tree whose cell structure reflects the "rooms" of our architectural model. For our floor model with 1920 split faces, the subdivision created 1280 cells and 3600 portals in 23 seconds.

3.2 Cell-to-Cell Visibility

Once the spatial subdivision has been constructed, we compute and store *cell-to-cell visibility* for each leaf cell, i.e. the set of cells visible to an observer able to look in all directions from any position within the cell. The cell-to-cell visibility for a cell C contains exactly those cells to which an unobstructed *sightline* leads from C. Such a sightline must be disjoint from any opaque elements and must intersect, or *stab*, a portal in order to pass from one cell to the next (Figure 2). Sightlines connecting cells that are not immediate neighbors must traverse a *portal sequence*, each member of which lies on the boundary of an intervening cell. We have implemented a procedure that finds sightlines through axial portal sequences, or determines that no such sightline exists, in $O(n \lg n)$ time, where n is the number of portals in the sequence [7].

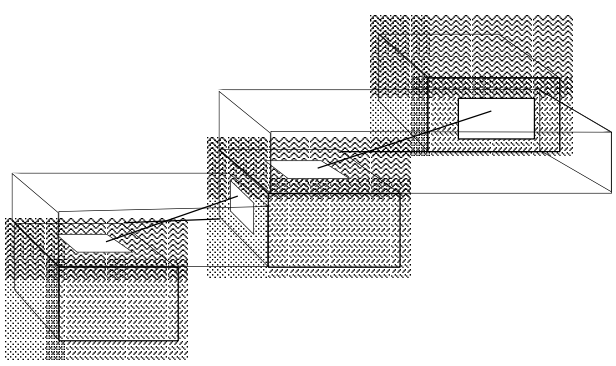

Figure 2: Stabbing an axial portal sequence in three dimensions.

We compute the cell-to-cell visibility by constructing a *stab tree* for each leaf cell C of the subdivision [11] as shown in Figure 3. Each node of the stab tree corresponds to a cell visible from C; each edge of the stab tree corresponds to a portal stabbed as part of a portal sequence originating on a boundary of C. The stab tree is constructed incrementally using a constrained depth-first search on the adjacency graph. As each cell is encountered by the depth first search, it is effectively marked "visible" by its inclusion into the source cell's stab tree. For any source cell C, we say that a cell R is *reached* if R is in C's cell-to-cell visibility set.

3.3 Cell-to-Object Visibility

Cells that are immediate neighbors of the source cell are entirely visible to it, since the eyepoint can be placed on the shared portal. Cells farther away from the source, however, are in general only partially visible to an observer in the source cell. This is due to the fact that, as the length of a portal sequence increases, the collection of lines stabbing the entire sequence typically narrows.

Casting the sightline search as a graph traversal yields a simple method for computing the partially visible portion of each reached cell. First, the traversal *orients* each portal encountered, since the portal is traversed in a known direction. Thus each portal contributes a "lefthand" and a "righthand" constraint to the set of sightlines stabbing the sequence. The result, after stepping through n portals in the plane, is a bowtie-shaped bundle of lines that stabs every portal of the sequence, and which "fans out" beyond the final portal into an infinite wedge. This wedge can then be clipped to the boundary of the reached cell. In our three dimensional models, all portals are axial rectangles, so any portal sequence can generate at most three pairs of bowtie constraints (one from each collection of portal edges parallel to the x, y, and z axes). Color Plate II depicts the clipped polyhedral wedges for a source cell in three dimensions.

We define *cell-to-object* visibility as the set of *objects* that can be seen by an observer constrained to a given source cell C (but, again, free to move anywhere in C and look in any direction). For each reached cell R, we compute a *superset* of C's cell-to-object visibility in R by assembling a set of *halfspaces* bounding the portion of R visible from C. We then store with C those objects in R that are completely or partially inside the assembled halfspaces. One special case exists: all objects in C's neighbor cells are tagged as visible from C without any bowtie computations.

Figure 5 depicts this process in two dimensions, using a simplified floorplan of our three-dimensional test model. The objects found potentially visible from the source (the filled squares in Figure 5) are associated with the source cell and reached cell in a compacted representation of the stab tree. Later, in the interactive walkthrough phase, this object list will be retrieved and culled dynamically based on the observer's position and view direction.

building model from AutoCAD floor plans and elevations, and populate the model with furniture. Next, during the *precomputation phase*, we perform a spatial subdivision and observer-independent lighting and visibility calculations. Finally, during the *walkthrough phase*, we simulate an observer moving through the building model under user control with the mouse, rendering the model as seen from the observer's viewpoint in each frame. The *display database* is the link between these three phases. It stores the complete building model, along with the results of the precomputation phase, for use during the walkthrough phase.

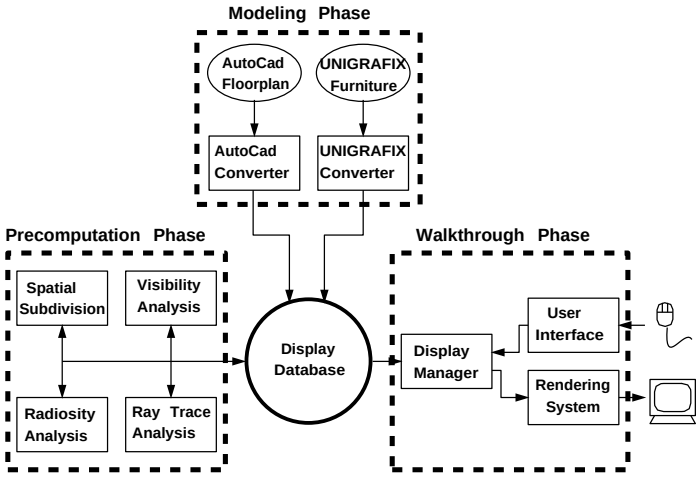

Figure 1: System overview.

2 Modeling Phase

Our walkthrough system requires a detailed 3D model of a building, complete with furniture and realistic material and lighting information.

We first convert the raw $2\frac{1}{2}$D model received from the architects in AutoCAD DXF format [3] into a consistent 3D representation in Berkeley UNIGRAFIX format [10]. Unfortunately, the raw architectural models that we received were not true three-dimensional models and contained non-planar faces, coincident coplanar faces, improper face intersections, and inconsistent face orientations. During conversion, our programs [9] detect and automatically correct many of these anomalies. Any remaining modeling errors are corrected manually using interactive tools.

We then populate the architectural model with stairs, furniture and other objects that a user would expect to find in a typical building. We have generated highly detailed descriptions for several pieces of furniture using interactive modeling programs, and received others from Greg Ward of Lawrence Berkeley Laboratories. We place instances of these objects into the building model using both automatic and interactive placement programs. We have written several programs that automatically place objects into specific types of rooms based on sets of parameters. For instance, the "conference room generator" places a rectangular or elliptical table in the middle of a room, chairs all around it, a blackboard on one wall, a transparency projector on the table, and so on. The "office generator" places a desk against one wall, a chair in front of the desk, some bookshelves against the walls, and so on. Numerous parameters are available for the user to control the size, number and placement of objects with each of these programs. We have also written a program for interactively placing objects into a three-dimensional model. It allows a user to add, delete, or move object instances with real-time visual feedback.

Gradually, we load the walls and furniture of the building model into the walkthrough display database. The display database represents the building model as a set of *objects* (e.g. walls, desks, chairs, telephones, pencils, etc.), each of which can be described at multiple levels of detail [6]. We construct less detailed representations of objects from the highly detailed originals using an interactive design tool that allows a user to simplify 3D objects by deleting and merging vertices and faces. For instance, we construct five representations of a desk: 1) a highly detailed desk with faces subdivided along gradients of radiosity, 2) a slightly less-detailed desk with simple handles and larger faces, 3) an even less-detailed desk without any handles at all, 4) a coarsely detailed desk with only legs and drawers, and 5) a simple box. These object abstraction hierarchies are adjusted interactively so that transitions between levels are barely noticeable as one zooms closer to an object and detail is refined. Levels of detail are chosen dynamically during the interactive walkthrough phase to improve refresh rates and memory utilization.

So far, we have built a completely furnished model of the sixth floor of Soda Hall, the planned computer science building at U.C. Berkeley. This floor model has a total of 2,320 objects, represented at up to five levels of detail, and contains over 400,000 faces, requiring 68MB of storage. Color Plate I shows a top-view of the model.

3 The Precomputation Phase

After the complete building model has been loaded into the display database, we distribute the model into a *spatial subdivision* and perform a *visibility analysis* of the model cells and objects. The resulting information is stored in the display database for use by the display and memory management algorithms during the walkthrough phase.

3.1 Spatial Subdivision

We subdivide the model using a variant of the k-D tree data structure [4]. Splitting planes are introduced along the major opaque elements in the model, namely the walls, door frames, floors, and ceilings (details are given in [11]).

Management of Large Amounts of Data in Interactive Building Walkthroughs

Thomas A. Funkhouser, Carlo H. Séquin and Seth J. Teller
University of California at Berkeley[‡]

Abstract

We describe techniques for managing large amounts of data during an interactive walkthrough of an architectural model. These techniques are based on a spatial subdivision, visibility analysis, and a display database containing objects described at multiple levels of detail. In each frame of the walkthrough, we compute a set of objects to render, i.e. those potentially visible from the observer's viewpoint, and a set of objects to swap into memory, i.e. those that might become visible in the near future. We choose an appropriate level of detail at which to store and to render each object, possibly using very simple representations for objects that appear small to the observer, thereby saving space and time. Using these techniques, we cull away large portions of the model that are irrelevant from the observer's viewpoint, and thereby achieve interactive frame rates.

CR Categories and Subject Descriptors:
[Information Systems]: H.2.8 Database Applications. **[Computer Graphics]:** I.3.5 Computational Geometry and Object Modeling – *geometric algorithms, languages, and systems*; I.3.7 Three-Dimensional Graphics and Realism – *visible line/surface algorithms*.

Additional Key Words and Phrases: architectural simulation, virtual reality.

[‡]Computer Science Department, Berkeley, CA 94720

1 Introduction

Interactive computer programs that simulate the experience of "walking" through a building interior are useful for visualization and evaluation of building models before they are constructed. However, realistic-looking building models with furniture may consist of tens of millions of polygons and require gigabytes of data - far more than today's workstations can render at interactive frame rates or fit into memory simultaneously. In order to achieve interactive walkthroughs of such large building models, a system must store in memory and render only a small portion of the model in each frame; that is, the portion seen by the observer. As the observer "walks" through the model, some parts of the model become visible and others become invisible; some objects appear larger and others appear smaller. The challenge is to identify the relevant portions of the model, swap them into memory and render them at interactive frame rates (at least ten frames per second) as the observer's viewpoint is moved under user control.

Using the design of Soda Hall, a planned computer science building at UC Berkeley, as a test object, we have completed the first version of a system that supports interactive walkthroughs of large, fully furnished building models. Our system builds upon pioneering work by Airey and Brooks [1,2,5] and uses conceptual ideas going back to Jones [8] and Clark [6]. The special features of our system are 1) a hierarchical display database that describes the building model as a set of objects represented at multiple levels of detail; 2) a spatial subdivision and visibility analysis in which the building model is divided into cells, and cell-to-cell and cell-to-object visibility information is computed; 3) a real-time memory management algorithm for swapping objects in and out of memory as the observer moves through the model; and 4) a real-time refresh algorithm for choosing which objects to render at which levels of detail in each frame.

1.1 System Overview

Our system is divided into three distinct phases as shown in Figure 1. First, during the *modeling phase*, we construct the