Database Management for Interactive Display of Large Architectural Models

Thomas A. Funkhouser
AT&T Bell Laboratories
Murray Hill, NJ

Abstract

This paper describes algorithms for predictive database management used in The UC Berkeley Building Walkthrough System. The algorithms forecast a range of possible observer viewpoints during upcoming frames and use precomputed cell visibility information to determine a set of objects likely to be visible to the observer in the near future. For each of these objects, detail elision techniques determine which levels of detail must be stored in a memory resident cache. Cache management algorithms determine which objects to load into memory from disk, and which to replace when the cache is full. Using these algorithms, the system is able to maintain real-time frame rates during interactive visualization of large building models with furniture and radiosity illumination.

Key Words: Interactive Visualization, Database Systems, Computer-Aided Architectural Design.

1 Introduction

Today, graphics workstations offer a great potential for real-time display of complex 3D environments. An interactive visualization system can simulate the experience of moving through a three dimensional model by rendering images of the model as seen from a hypothetical observer viewpoint under interactive control by the user. If images are rendered smoothly and quickly enough, the visual illusion of real-time exploration of a virtual environment can be achieved.

Interactive visualization is particularly valuable in computer-aided architectural design. A *building walkthrough system*, which uses three dimensional computer graphics to simulate "walking" through a building, can be used by architects and interior designers to visualize and evaluate architectural designs before a building has been constructed. As a result, visual simulation and verification of an architectural design may be performed early in the design cycle, thereby saving time and money.

Radiosity methods [11] are often used to perform lighting simulations of building interiors. The advantage of radiosity methods for interactive visualization is that they are able to model physically realistic indirect diffuse illumination and shadows, and therefore generate fairly realistic-looking images. Also, the radiosity solution does not depend on a particular observer viewpoint. Therefore, a radiosity computation can be performed for an entire building model during a precomputation phase in which results are stored in a database for use later during interactive visualization. This approach offloads the expensive illumination computations required to capture realistic lighting effects, such as shadows, so that rendering during interactive visualization can produce high-quality images quickly.

Figure 1: Radiosity mesh for furnished office.

One challenge of using radiosity methods for interactive visualization is management of the large amount of data required to describe a radiosity so-

lution. A separate color is stored for each vertex of every polygon in the model, and large polygons are split into many mesh elements in order to capture complex illumination effects along the boundaries of shadows and highlights. Furthermore, although many of the polygons in the original model can be shared via hierarchical instancing, each polygon is illuminated and meshed independently during the radiosity computation, and must be stored separately in the resulting model (see Figure 1). As a result, a model that originally contains millions of possibly shared polygons may contain tens of millions of separate polygons, and may require gigabytes of data after a radiosity computation. This is too much data to store all at once in memory on most graphics workstations.

In order to support a real-time walkthrough of a large building model with radiosity illumination, a visualization system must store only a subset of the model in memory (i.e., the working set) and swap different parts in and out of memory in real-time as the observer navigates through the model interactively. Unfortunately, standard virtual memory systems are not suitable for real-time interactive visualization because: 1) they do not pre-fetch data, and 2) they do not load data asynchronously. In standard virtual memory operation, a page fault occurs when disk-resident data is accessed. At that time, the system executes a synchronous read for the page containing the data, and continues execution only after the page has been loaded. This sequence of operations causes unacceptable delays for real-time immersive walkthrough applications. If many data accesses fault in the same frame (e.g., when the user navigates around a corner into a new corridor), pauses up to several seconds in length may be observed as the virtual memory system loads all the pages into memory.

In order to avoid delays to the perceptible frame rate due to page faults, an interactive walkthrough system must pre-fetch data asynchronously so that the data is resident in memory before it is displayed. Since loading data from secondary storage into memory can take a relatively large amount of time, the system must predict data access patterns several frames in advance. The challenges are to develop an algorithm to compute a small subset of the data that is most likely to be accessed in the near future, and to use this algorithm to manage a cache of resident data in real-time as the user navigates through the environment interactively.

This paper describes the predictive database management algorithms developed for The UC Berkeley Building Walkthrough System. The algorithms forecast a range of possible observer viewpoints during future frames and then use visibility determination and detail elision techniques derived from the display algorithms of the system to predict which polygons can be rendered in upcoming frames. Cache management algorithms determine which objects to load into memory from disk, and which to replace when the cache is full, as the observer moves through the model interactively.

The paper is organized as follows. The next section contains a summary of related work. Section 3 provides an overview of The UC Berkeley Building Walkthrough System. This overview is important in order to understand the data access patterns required for display by the system. The predictive database management algorithms are described in Section 4. Results of experiments with these algorithms using a radiosity model of two floors of Soda Hall are presented in Section 5. The results are followed by a brief discussion and conclusion.

2 Related Work

There has been a considerable amount of work in interactive visualization of 3D models for computer-aided design and vehicle simulation systems [5, 7]. Although some systems support real-time database management, little has been published on this topic since most systems are proprietary. Vehicle simulation systems often use quadtrees to represent terrain models and load patches of terrain within the viewer frustum at appropriate resolutions based on viewing distance [12, 16].

Although there are many similarities between vehicle simulators and building walkthrough systems, there are several important differences. First, building models tend to be more "densely occluded" than terrain environments. This property allows a building walkthrough system to take advantage of visibility determination algorithms that cull not only polygons outside the observer's view, but also ones occluded by other polygons (e.g., walls). Second, the types of navigation supported by vehicle simulators are very different than those in building walkthrough systems. In a vehicle simulator, the observer viewpoint corresponds with the view from the driver's seat of the vehicle, and observer viewpoint navigation is limited to movements possible by the vehicle. In a building walkthrough system, the observer viewpoint corresponds to the view from the eyes of

a human being walking through the building. The observer may step in any direction, or spin around quickly. Therefore, many of the optimizations used by vehicle simulators based on assumptions of observer navigation are not possible in a building walkthrough system.

Commercial products for visualization of architectural models are now readily available [2, 15]. However, to the author's knowledge, none of them employs sophisticated memory management algorithms, and none supports models larger than fit in memory. The UNC Building Walkthrough System [3] provided much inspiration for this work, including ideas on spatial subdivisions and cell-to-cell visibility precomputation [1]. However, it too supports only memory resident models.

3 System Overview

The UC Berkeley Building Walkthrough System simulates an observer moving through a 3D building model under interactive user control [10]. The goal is to render the model as seen from the observer viewpoint in a window on the workstation display at interactive frame rates as the user moves the observer viewpoint through the model.

Prior to execution, an efficient *display database* is constructed for the architectural model [9]. The display database describes the model as a set of *objects*, each of which can be represented at multiple *levels of detail* (LODs) [4]. It also contains a spatial subdivision constructed by partitioning space into *cells* split by the major, axis-aligned polygons of the building model (e.g., walls, ceilings, and floors). For each cell, C, a visibility precomputation is performed that determines which cells (cell-to-cell visibility) and which objects (cell-to-object visibility) are potentially visible to any observer in C [13, 14].

Execution during an interactive walkthrough proceeds as diagrammed in Figure 2. In every frame, the system performs seven operations, each of which can run asynchronously in a separate concurrent process in a two-forked pipeline.

The operations in the upper fork of the pipeline generate images for the user, and thus are very sensitive to throughput and latency. For each observer viewpoint generated by the user interface, the system first executes a visibility determination algorithm to compute a set of potentially visible objects to render (a z-buffer is used later during rendering to resolve visibility priority among potentially visible objects). The set of objects determined to be visible

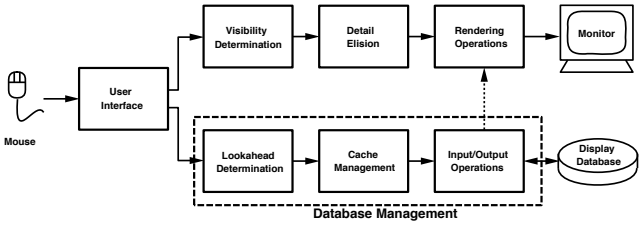


Figure 2: Interactive walkthrough pipeline.

from the observer viewpoint is always a proper subset of the cell-to-object visibility set of the observer's cell (see Figure 3). Next, a detail elision algorithm is used to choose an appropriate level of detail with which to render each potentially visible object. A static screen area threshold (pixels/polygon) is used to bound LODs, and then an optimization algorithm is used to possibly further reduce LODs used for some objects in order to maintain a bounded frame rate [8]. Finally, rendering commands are sent to the graphics workstation to display the potentially visible objects with the chosen levels of detail.

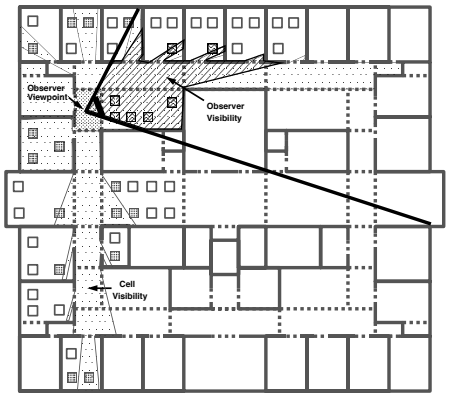


Figure 3: The observer's visibility (hatch) is a proper subset of its cell's visibility (stipple).

The operations in the lower fork of the walkthrough pipeline perform database management. The system uses predictive visibility and detail elision algorithms to determine the set of objects, and a level of detail for each one, to store in a memory resident cache. Then, cache management techniques are used to determine the sets of objects to load from the display database and release from memory during each frame. Finally, database input/output operations (e.g., read, write and release) are used to transfer data between the memory resident cache and display database. These operations are described in detail in the following sections.

4 Database Management

The goal of the database management process of the walkthrough system is to maintain a cache of data in memory so that the display process never faults by trying to access data only available in secondary storage. An ideal memory management algorithm predicts the observer viewpoint in each future frame perfectly. Then it can use the visibility determination and detail elision algorithms described in Section 3 to determine exactly which objects and LODs will be rendered during future frames and pre-fetch them into memory, replacing objects that will not be rendered for the longest time in the future. Unfortunately, since the observer viewpoint is under interactive control by the user and cannot be predicted perfectly, we must consider a range of possible future observer viewpoints in our memory management algorithm.

Observer Lookahead

In order to pre-fetch objects into memory before they are rendered, we must continually predict which objects are likely to become visible to the observer in the upcoming future. Given a particular observer viewpoint in the current frame and constraints on observer movement and rotation enforced by the user interface, we can determine an *observer range* that contains a superset of all observer viewpoints possible during the next N future frames. For example, if the observer is allowed to move and turn in any direction, but is constrained by maximum positional and rotational velocities, v_p and v_r, the upper bound on the observer range during the next N frames is a sphere centered at the observer eye position with radius Nv_p. All possible observer view directions are enclosed in a *range frustum* whose eye position is directly behind the observer, and whose view angle is widened by Nv_r, and which contains the range sphere. Since there is usually coherence in observer motion from frame to frame, the current direction of observer movement can be used to help predict future observer eye positions by weighting the observer range in the direction of travel. Moreover, if the observer is prevented from moving directly through solid walls (a parameter in our user interface), the observer range is further constrained.

Since real-time visibility determination for a finite, non-zero volume of space (the observer range) seems to be too compute intensive for real-time execution, we use precomputed cell-to-cell and cell-to-object visibility information to conservatively predict a superset of the objects potentially visible from the observer range. During each frame, we compute a set of *range cells*, R, that cumulatively contain the observer range by performing a shortest path search of the cell adjacency graph. The search, implemented using Dijkstra's method [6], adds cells to the range set in order of minimum number of frames before the observer can enter the cell. When a new range cell is discovered during the shortest path search, we add each object in its cell-to-object visibility to a *lookahead set* of objects that may potentially be visible to the observer during the next N future frames.

Figure 4 shows an example computation of the lookahead set of objects, assuming the observer cannot walk through walls. Each cell is labeled by the minimum number of frames before objects inside it can become visible to a cell intersecting the observer range. For $N = 4$, cells in the observer range set are highlighted in cross-hatch, and cells containing objects in the observer lookahead set are highlighted in stipple gray.

Figure 4: Object lookahead set computation.

As each object is added to the lookahead set, we mark and claim memory for all LODs for the object that can possibly be rendered during the next N future frames. We use a size threshold for static detail elision, along with precomputed information regarding which objects can be drawn at a given LOD for an observer inside a particular cell, to choose a maximum LOD at which to store each potentially visible object. The effect is that objects near the observer range are stored in memory up to higher LODs than ones further away. Figure 5 shows an example computation of lookahead LODs for objects. Each cell is labeled and shaded by the maximum level of detail any object incident upon it is stored in memory – darker shades of gray represent higher levels of detail.

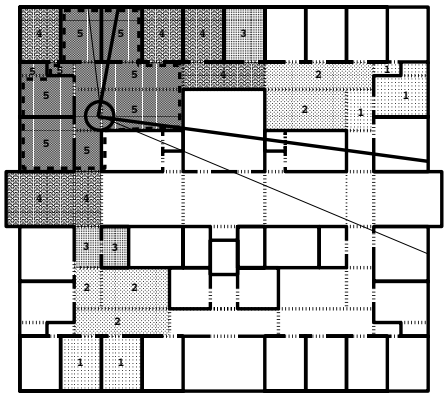


Figure 5: Maximum LODs for lookahead objects.

The shortest path search for range cells and lookahead objects terminates when either: 1) there are no cells remaining that can contain the observer during the next N frames, or 2) all available memory has been claimed (as long as all objects visible from the current observer viewpoint are in the lookahead set). In either case, if the range cull algorithm and lookahead cull algorithm are both *any direction*, the range set is guaranteed to contain the cells that the observer can enter soonest (since cells are added to the range set in order of minimum distance from the current observer position), and the lookahead set is guaranteed to contain objects represented at LODs that can potentially be rendered for an observer viewpoint within a range cell within the next N frames. If the algorithm terminates due to condition (1), the set of lookahead objects is a provable superset of the objects that can possibly be rendered during the next N frames, and fits into available memory. Otherwise, if the algorithm terminates due to condition (2), the set of lookahead objects is certainly a superset of the objects visible from the current observer viewpoint, as well as a good estimate of the objects that are most likely to be rendered in upcoming frames.

Cache Management

After computing the set of lookahead objects, we must determine which objects to load into memory (i.e., the *read set*) and which to remove from memory (i.e., the *release set*) during each frame of an interactive walkthrough. Conceptually, memory resident objects are stored in a fully associative, write-back cache which is the size of available memory (i.e., the size of the physical memory of the workstation minus the amount reserved for the spatial subdivision and precomputed visibility information).

To determine which objects to load into memory during each frame, we first check every object in the lookahead set to determine whether or not it is already represented at the appropriate LODs in the memory resident cache. In principle, we should issue read requests for every lookahead object that is not already in the memory resident cache. However, since a new lookahead set is constructed during every frame, and lookahead sets computed during later frames have more up-to-date predictive power, it is pointless (and even counterproductive) to start loading all such lookahead objects into memory during the current frame, since they may take several frame times to transfer from disk. Instead, during each frame, we load into memory only as many objects as can be read from disk in a single frame time. We construct a *read set* of objects to load from disk by adding lookahead objects in order of LOD (i.e., lowest to highest) and when they can possibly become visible to a range cell (i.e., the order they are added to the lookahead set). Construction of the read set terminates when the cumulative size (in bytes) of the set exceeds the estimated capacity of disk reads during a single frame time (*maximum bytes read per frame*), and all objects visible to the observer in the current frame are in either the memory resident cache or the read set. Read requests are issued for each object in the read set from an asynchronous database input/output process.

As objects from the lookahead set are added to the memory resident cache, other objects originally in the cache might need to be removed to free memory for the new ones. Our object replacement algorithm closely resembles a *least recently used* (LRU) policy. Objects in the memory resident cache are kept ordered by when they can possibly become visible to a range cell. As objects are added to the lookahead set, they are marked and moved to the head of the memory resident cache queue. Objects that are not in the lookahead set maintain their relative ordering in the queue across successive frames. We construct a *release set* of objects to remove each frame by choosing objects from the tail of the memory resident cache queue (i.e., the ones that have least recently been a member of the lookahead set) until enough memory is available for all objects in the read set. Objects in the release set are removed from memory before objects in the read set are loaded so that memory is never overburdened.

Figure 6 shows results of the cache management algorithm for a particular observer path. Each cell

is labeled by the number of frames since objects incident upon it were included in the lookahead set. The shade of each cell indicates whether or not it contains objects in the memory resident cache (stipple gray), read set (left-hatch), or release set (right-hatch).

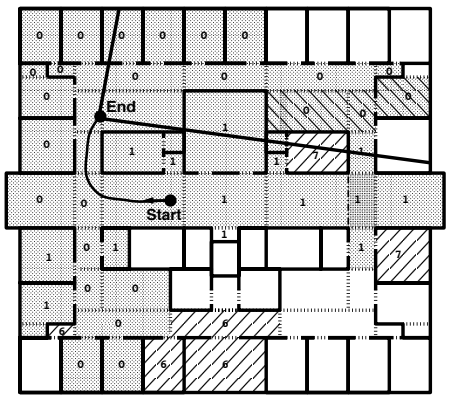


Figure 6: Cells containing resident objects (stipple), read objects (left-hatch), and released objects (right hatch).

Fault Tolerance

During each frame of an interactive walkthrough, an asynchronous database input/output process loads objects in the read set into memory from disk. Meanwhile, the walkthrough system renders objects potentially visible from the current observer viewpoint using LODs chosen by the detail elision algorithm. What happens if the database input/output process is not fast enough to load an object into memory before it is selected for rendering? This situation must be considered since there is no bound on the rate at which new data can become visible to the observer. For instance, the observer can "run" through the building, or turn several corners quickly to view portions of the model not previously visited. In these cases, the rate at which data becomes visible to the observer may be faster than the rate at which data can be loaded from disk.

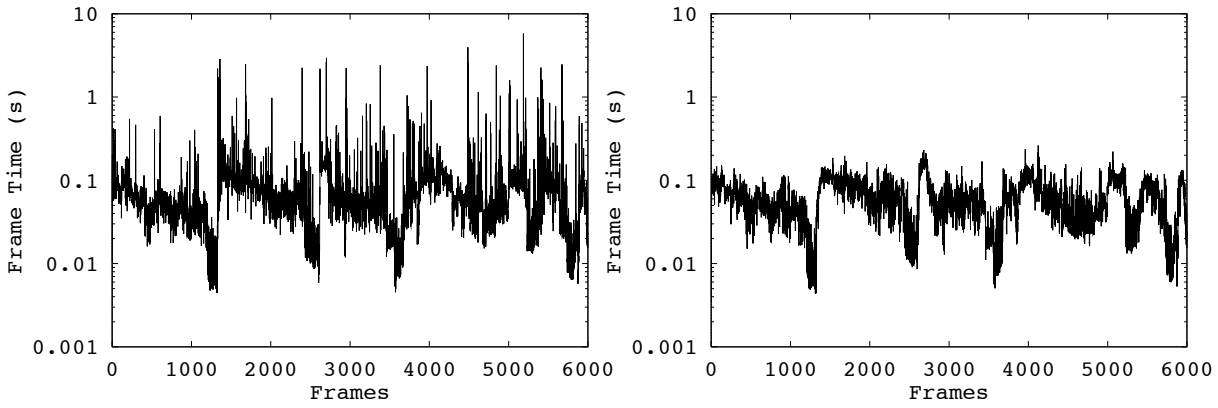
In our first implementation, the walkthrough system stalled when it found that an object to be rendered had not yet been loaded into memory at the appropriate LOD. It simply waited until the appropriate LOD for an object was loaded into memory, and then it continued rendering. Needless to say, this behavior was extremely bothersome. At times, the system would stall for several seconds waiting for a particular object geometry that was rendered for only a few frames.

In our current implementation, the system never waits for an object to be loaded into memory. Instead, if a potentially visible object has not been loaded into memory at the desired LOD, the rendering process simply skips that LOD and renders the object at the next highest LOD that is resident in memory. If the object is not resident in memory at any LOD, the object is skipped completely. Like detail elision during display, we trade image quality for interactivity using this approach. When the asynchronous database input/output process cannot keep up with the rest of the system, some objects may be rendered at lower LODs. Or, if the database input/output process falls behind the rest of the system by several frames, some potentially visible objects may not be rendered at all. Fortunately, since the lookahead algorithm orders objects based on when they are likely to be visible to the observer, and the cache manager loads object geometries in order from lowest LOD to highest LOD, generally only the higher LODs for newly visible objects are skipped.

5 Results

In order to evaluate the effectiveness of the algorithms described in this paper, we collected statistics during real-time execution of The UC Berkeley Building Walkthrough System both with and without predictive database management. The test model was a radiosity solution for the sixth and seventh floors of an academic building. The model comprised approximately sixty rooms containing 15,265 polygons which were split into 382,090 mesh elements by a radiosity computation. Each polygon was stored in the display database with its radiosity mesh as a separate object with only one LOD. The model contained 137MB of data.

Using a Silicon Graphics Power Series 320 workstation with 128MB of memory, two 33MHz R3000 processors, a Reality Engine[1] graphics processor, and a local disk, we collected frame time statistics (i.e., elapsed wall-clock time between successive frames) as the observer navigated along a path through the sixth and seventh floors of Soda Hall. The test path was chosen in order to span a large part of the model and visit the same portion of the model more than once in order to test both the lookahead and caching features of the database management algorithms. The observer velocity was representative of a normal walking pace stroll through the building.



(A) No database management.

(B) Predictive database management.

Figure 7: Frame times during experiments with and without real-time predictive database management.

During test 'A', without database management, the operating system managed page swapping after the entire model was read into virtual memory (and partially swapped back out). During test 'B', with database management, the algorithms described in this paper were used to pre-fetch objects asynchronously up to 16 frames in advance. The application was configured as a four-process pipeline with one process used for visibility determination and detail elision, a second process for rendering, a third process for lookahead determination and cache management computations, and a fourth process for database input/output operations. Logarithmic scale plots of frame times measured during these two tests are shown in Figure 7.

For this model, which was just barely bigger than physical memory (137MB versus 128MB), the system was able to display most frames at about ten frames per second without database management (see Plot 'A'). However, every few seconds, whenever the observer viewed a new portion of the model, the system stalled for up to several seconds while while the virtual memory system serviced page faults synchronously. Such stalls are indicated by spikes in the frame time Plot 'A' in Figure 7 (note the logarithmic scale on the vertical axis). Although the mean frame time was 0.082 seconds, the standard deviation in the frame time was 0.175 seconds, and the longest frame time was 5.78 seconds. Qualitatively, the frequent stalls not only destroyed the illusion of immersive exploration but also caused the user to have difficulty navigating.

Using the database management algorithms described in this paper, the system was able to display all frames at interactive rates without any stalls (see plot 'B'). The mean frame time was 0.063 seconds with a standard deviation of 0.035 seconds and a maximum frame time of 0.263 seconds. Since the system pre-fetched data into memory before it was displayed and didn't wait for database I/O operations synchronously, the user was able to navigate into new portions of the model without delay. The system was able to compute a small subset of *lookahead objects* to store in memory during each frame (22MB on average) and required a small amount of data to be loaded into memory during each frame (28.4KB on average). The lookahead determination and cache management computations took 4ms per frame on average, while the database input/output operations took 28ms per frame.

During the test with database management, there were a few instances in which the system was not able to successfully load all visible objects into memory in time to be displayed as the user navigated quickly around a corner into a new, densely populated portion of the model. This situation is encountered more frequently as the observer velocity is increased. In these cases, the system continued frame generation as those objects were loaded into memory asynchronously and began rendering them as they became available. Although the omission (and subsequent appearance) of some objects in rendered images was disturbing to the user, we feel the impact on navigation and interaction in the virtual building model was far less than the delays incurred using virtual memory.

6 Conclusion

A visualization system must pre-fetch data into memory asynchronously as a user navigates through the model in order to avoid delays during interactive walkthroughs of models larger than memory.

This paper describes the database management algorithms used in The UC Berkeley Building Walkthrough System. A lookahead algorithm computes a set of objects that are likely to become visible to the observer during upcoming frames and determines a maximum level of detail to store in memory for each object. An approximate least-recently-used cache management algorithm is used to determine which objects to load into memory, and which to replace, during each frame of an interactive walkthrough. In cases where more data becomes visible to the observer than can be loaded into memory in real-time, incomplete images are rendered while data is being loaded asynchronously in order to maintain interactivity. Using these algorithms, the system is able to maintain interactive frame rates during walkthroughs of radiosity models larger than memory.

References

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

[2] Bechtel, Inc. *WALKTHRU: 3D Animation and Visualization System.* Promotional literature, 1991.

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

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

[5] Deyo, R. J., J. A. Briggs, and P. Doenges. Getting Graphics in Gear: Graphics and Dynamics in Driving Simulation. *Computer Graphics (Proc. SIGGRAPH '88)*, 24, 4 (July 1988), 317-326.

[6] Dijksta, E.W. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* **1**, 1959, 269-271.

[7] Evans and Sutherland Computer Corporation. *ESIG:4000*, Promotional literature, 1993.

[8] Funkhouser, Thomas A., and Carlo H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. *Computer Graphics (Proc. SIGGRAPH '93)*, (August 1993), 247-254..

[9] Funkhouser, Thomas A. Database and Display Algorithms for Interactive Visualization of Architectural Models. Ph.D. thesis, Computer Science Division (EECS), University of California, Berkeley, 1993. Also available as UC Berkeley technical report UCB/CSD-93-771.

[10] Funkhouser, Thomas A., Seth Teller, Carlo Séquin, and Delnaz Khorramabadi. The UC Berkeley System for Interactive Visualization of Large Architectural Models. *Presence*, 5, 1, January, 1996.

[11] Goral, Cindy M., Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the Interaction of Light Between Diffuse Surfaces. *Computer Graphics (Proc. SIGGRAPH '84)*, 18, 3 (July 1984), 213-222.

[12] Schachter, Bruce J. (Ed.). *Computer Image Generation.* John Wiley and Sons, New York, NY, 1983.

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

[14] Teller, Seth J. *Visibility Computations in Densely Occluded Polyhedral Environments.* Ph.D. thesis, Computer Science Division (EECS), University of California, Berkeley, 1992. Also available as UC Berkeley technical report UCB/CSD-92-708.

[15] *Virtus Walkthrough.* Promotional literature, 1991.

[16] Zyda, Michael J., David R. Pratt, James G. Monahan, and Kalin P. Wilson. NPSNET: Constructing a 3D virtual world. *ACM SIGGRAPH Special Issue on 1992 Symposium on Interactive 3D Graphics*, March, 1992.