The UC Berkeley System for Interactive Visualization of Large Architectural Models

Thomas Funkhouser, Seth Teller,
Carlo Séquin, and Delnaz Khorramabadi

June 3, 1996

Abstract

Realistic-looking architectural models with furniture may consist of millions of polygons and require gigabytes of data – far more than today's workstations can render at interactive frame rates or store in physical memory.

We have developed data structures and algorithms for identifying a small portion of a large model to load into memory and render during each frame of an interactive walkthrough. Our algorithms rely upon an efficient display database that represents a building model as a set of objects, each of which can be described at multiple levels of detail, and contains an index of spatial cells with precomputed cell-to-cell and cell-to-object visibility information. As the observer moves through the model interactively, a real-time visibility algorithm traces sightline beams through transparent cell boundaries to determine a small set of objects potentially visible to the observer. An optimization algorithm dynamically selects a level of detail and rendering algorithm with which to display each potentially visible object in order to meet a user-specified target frame time. Throughout, memory management algorithms predict observer motion and prefetch objects from disk that may become visible during imminent frames.

This paper describes an interactive building walkthrough system that uses these data structures and algorithms to maintain interactive frame rates during visualization of very large models. So far, the implementation supports models whose major

occluding surfaces are axis-aligned rectangles (e.g., typical buildings). This system is able to maintain over twenty frames per second with little noticeable detail elision during interactive walkthroughs of a building model containing over one million polygons.

1 Introduction

Traditionally, two-dimensional floor plans of buildings and elevations or perspective projections have been the basic communication media between architects and their clients. Particularly with respect to the interior of buildings, architects rely on the client's imagination to visualize a proposed building from its architectural plan views, assuming that clients are familiar with architectural symbols, and have the training and experience to construct three-dimensional images from two-dimensional plan views.

Today, graphics workstations offer a great potential for real-time simulation of movement through complex environments, particularly large buildings. A computer-based, interactive building walkthrough system can simulate the visual experience of moving through a three dimensional model of a building on the screen of a computer workstation by displaying rendered 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 illusion of real-time exploration of the proposed building can be achieved.

A building walkthrough system might be useful for architects, interior designers, and clients to visualize and evaluate architectural designs before a building has been constructed. For example, an architect may be able to detect errors, such as material interferences; test lighting conditions, evaluating them at different times of day or during different seasons; and check the view out the windows of particular offices. An interior designer might experiment with several furniture arrangements, color schemes, and lighting arrangements in a computer simulation before purchases are made. Most importantly, a building walkthrough system provides a means by which architects, interior designers and clients can communicate their ideas to one another. In particular, it is extremely difficult for a typical client, who has

commissioned a building, but has not been trained in visualization of three dimensional spaces, to understand what the inside of a building might actually look like by viewing blueprints or cardboard models. An interactive, three dimensional building walkthrough system allows an architect to show a client a proposed architectural design, and elicit real-time feedback as the client "walks" through the interior of the building interactively. As a result, faults in the architectural plan may be found earlier in the design cycle, thereby saving time and money.

We have completed the first version of a system that supports interactive visualization of large architectural models with axis-aligned walls, ceilings, and floors using as a test case the design of Soda Hall, the new computer science building at the University of California at Berkeley. The system includes tools for conversion of architectural plans into three dimensional models, and database and display algorithms for interactive visualization of large architectural models. This paper describes the goals of the system, the challenges encountered during implementation of the system, and algorithmic approaches we have taken to generate interactive frame rates during walkthroughs of very complex architectural models.

1.1 Previous Work

1.1.1 Vehicle Simulators

Most work in interactive visualization has been done on vehicle simulators. Several sophisticated commercial vehicle simulators have been built over the last thirty years, including many which contain algorithms for field-of-view culling, detail elision, and real-time management of very large databases [11, 31, 32, 43]. However, since most are commercial systems, very little has been published on this work.

Although there are many similarities between vehicle simulators and building walkthrough systems, there are several important differences. First, the types of environments encountered in vehicle simulators are quite different from building interiors. Typical vehicle simulator models contain terrain data augmented with plants, buildings, roads, etc. In these

models, space tends to be "sparsely occluded" – i.e., there are few observer viewpoints for which a significant portion of the model is occluded by other parts of the model. In contrast, building models typically contain walls, ceilings, and floors which partition space into rooms. These models tend to be "densely occluded" – i.e., a large portion of the model is occluded by some polygon for observer viewpoints in the interior of the building. Therefore, visibility determination algorithms that cull not only polygons outside the observer's view, but also ones occluded by other polygons (e.g., walls) may be better suited for visualization of building models than for vehicle simulator models.

Second, the types of navigation supported by vehicle simulators are very different than those used 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. During normal execution, the observer does not generally move sideways, or change direction suddenly. As a result, there is a large amount of coherence in the observer position (and hence the visible portion of the model) from frame to frame, and it is relatively easy to predict future observer viewpoints from the current observer viewpoint and direction of travel. In addition, since the observer rarely travels close to detailed model features (e.g., aircraft are typically several thousand feet up in the air, and cars are typically on roads), realistic-looking detail can be achieved using texture maps applied to relatively few, distant polygons. In contrast, 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, spin around quickly, or look very closely at any feature of the model. Therefore, many of the optimizations used by vehicle simulators based on assumptions of observer navigation are not possible in a building walkthrough system.

Finally, the performance and hardware constraints for vehicle simulators are very different than for building walkthrough systems. Since inaccurate vehicle simulation during training may cause serious accidents during operation later, vehicle simulation systems must maintain

strict frame rates in order to approximate vehicle operation as realistically as possible (e.g., exactly thirty frames per second). To do this, they typically enforce restrictions on model complexity, and use special-purpose display hardware costing millions of dollars. In contrast, the frame rate requirements of building walkthrough systems are not as strict – nobody will be killed if the simulation is inaccurate. Although uniform frame rates are desirable in a building walkthrough system, they are not essential. Frame rates must be only fast enough and uniform enough for a user to intergrate impressions derived from sequential images to derive a feeling of the building interior space. We aim to support near-uniform, bounded frame rates, while using standard, off-the-shelf hardware, and allowing visualization of arbitrarily complex models.

1.1.2 Mechanical CAD Systems

There also has been a considerable amount of work in interactive visualization of three dimensional models in mechanical computer-aided design (CAD) systems. Mechanical CAD models can be quite complex, containing tens of millions of polygons (e.g., a car engine, spacecraft assembly, or an airplane), and thus may require sophisticated real-time display algorithms for interactive visualization.

There are differences between CAD applications and building walkthrough systems. First, visual realism is generally less important in mechanical CAD applications than in building walkthrough systems. In most CAD applications, objects are represented symbolically. For instance, parts in a complex assembly may be displayed with attributes (e.g., color) representing semantic characteristics (e.g., function, interference, connectivity, etc.), and meta-data may be included in the display (e.g., the path through which a part moves). Visual verification of a CAD model is based mainly on positional and semantic characteristics rather than appearance. In contrast, surface color and illumination characteristics are usually very important in architectural design. Thus, realistic-looking images generated using physically-based lighting simulations are required for lighting design verification.

Second, mechanical CAD systems generally simulate an observer looking at the model "through a window" from the outside. The user typically uses a *Scene in Hand* metaphor [41] to manipulate the object by means of translation, scaling and rotation. In contrast, building walkthrough systems simulate an observer moving through the interior of the model, the *Flying Vehicle Control* metaphor [41]. These different metaphors for observer navigation may imply different approaches to observer viewpoint prediction, visibility determination, and detail elision.

1.1.3 Architectural CAD Systems

Commercial products for visualization of architectural models have recently become available. However, many of these systems do not allow a user to control the simulated observer viewpoint interactively. Instead, travel along a predetermined, fixed path is simulated by displaying a sequence of precomputed images. These systems can generate very realistic-looking walkthroughs (since images are rendered off-line), and are well-suited for presentation of a completed design. However, they do not support interactive visualization or design.

Currently available commercial products that do allow interactive, real-time navigation generally support only small buildings models (e.g., less than one hundred thousand polygons), displayed with simple rendering algorithms (e.g., wire-frame or flat shading) [5, 39]. These commercial systems generally make little use of sophisticated precomputation, visibility determination, or detail elision, and require that the entire model be resident in memory.

Research on increasing frame rates during interactive visualization of architectural models has been under way for over twenty years [24]. Pioneering work in spatial subdivision and visibility precomputation has been done at the University of North Carolina at Chapel Hill [1, 2, 8]. Airey developed algorithms for partitioning architectural models into *cells*, and precomputing a *potentially visible set* of polygons (PVS) for each cell. Cell visibility was determined by tracing ray samples through transparent portions of cell boundaries to find polygons visible from particular viewpoints within the cell. The disadvantage of this

approach is that computation is stochastic, and thus can under-estimate true cell visibility and requires a large amount of computation.

Recently, other algorithms have been described for culling occluded polygons during interactive visualization. The hierarchical z-buffer algorithm [20] uses a pyramid of z-buffers to determine the cells of an octree (and the enclosed polygons) that are potentially visible for a particular viewpoint. This algorithm may be effective with appropriate hardware acceleration, but cannot be applied to determine visibility from a volume of space (which is useful for visibility precomputation and for predictive memory management) and considers every octree-cell inside the observer view frustum for each frame (which may be infeasible for very large models – e.g., a city).

1.2 Goals

1.2.1 Model Size

A primary goal of our work is to support computer-aided visualization of very large, detailed three dimensional models. For accurate evaluation of a building design using visual simulation, it is important that the building model contain a large amount of detail, including representative light fixtures and furniture modeled with accurate materials and textures.

Our test case is a model of Soda Hall, a seven floor academic building containing more than one hundred faculty and student offices, twelve computer laboratories, and six class rooms (Figure 1). Almost every room in the model contains a functionally complete set of furniture (Figure 2). For instance, each office has at least one desk, a few chairs, a bookshelf, a plant, etc. On each desk, there is at least one book, a desk lamp, and a few pencils (Figures 3 and 4). Furthermore, most pieces of furniture are modeled with a large amount of detail. For example, each pencil has an explicitly modeled graphite point, each door has a shiny brass handle, and each book has a separate binder.

The model is described completely by planar polygons. Curved surfaces, such as those found in desk lamp shades and the seat cushions of chairs, are approximated by many flat

polygons. Hundreds, and sometimes thousands, of polygons are required to describe the most detailed representations of plants and complex pieces of furniture. In all, the model of Soda Hall contains 1,418,807 polygons, of which only 31,625 represent the walls, ceilings, and floors of the building, while the remainder represent its "contents."

1.2.2 Image Quality

A second goal of our work is to use *radiosity* illumination simulation methods [19, 26] to generate realistic-looking images with indirect diffuse reflections and shadows (Figures 5 and 6). Radiosity methods, based on models of radiative transfer methods in thermaengineering, consider every polygon a potential emitter or reflector of radiance (or luminance). Conceptually, for every pair of polygons, A and B, a form factor is computed which measures the fraction of the energy leaving polygon A that arrives at polygon B. This approach yields a set of simultaneous linear equations which can be solved to obtain the radiance for each polygon.

The advantage of radiosity methods for interactive visualization is that a global radiosity solution can be precomputed. The solution includes only the diffuse component of reflection and does not depend on a particular observer viewpoint (Figure 7). 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 off-loads 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.

One difficulty associated with radiosity methods is that a tremendous amount of data may be required to describe the radiosity solution. A separate color is stored for each vertex of each polygon in the model; and large polygons are split into many smaller ones (where the gradient of radiosity is high) in order to capture complex illumination effects, such as highlights and shadow boundaries (Figure 8). Furthermore, although many of the polygons

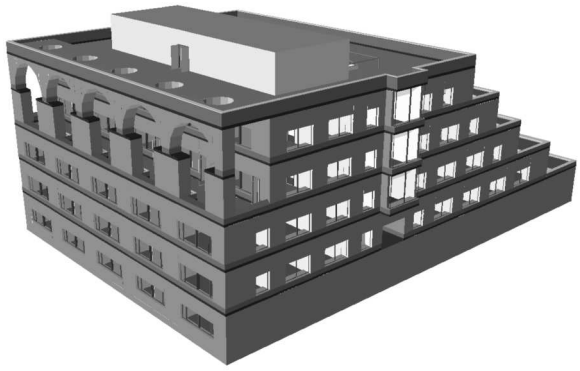


Figure 1: Exterior view of Soda Hall.

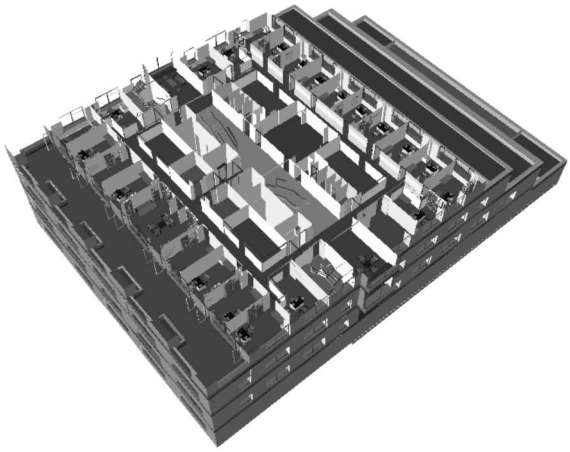


Figure 2: Exterior view of Soda Hall "cut open" by a horizontal plane at the sixth floor.

Figure 3: Typical office with furniture.

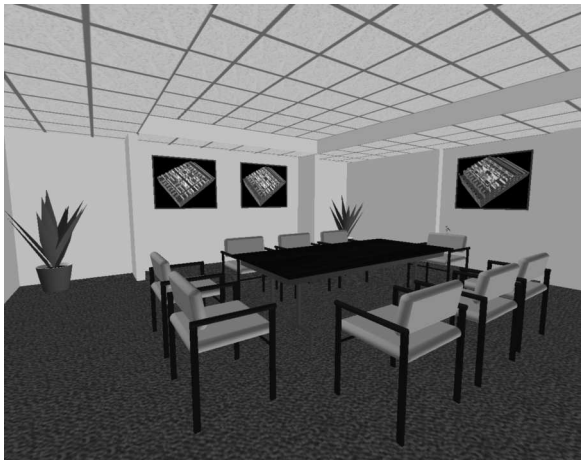


Figure 4: Board room with furniture.

in the original model can be shared via hierarchical instancing, each polygon is meshed and illuminated independently during the radiosity computation, and must be stored separately in the resulting model. As a result, a model that originally contains millions of possibly shared polygons can generate a radiosity solution with tens of millions of separate polygons requiring gigabytes of data.

We aim to support visualization of large radiosity solutions in our building walkthrough system. However, techniques for computing such solutions have just recently been developed [38] and are not addressed in this paper.

1.2.3 Performance

Another critical goal of our system is to provide performance that is adequate to maintain the real-time feel of interactive visualization. If frame rates (i.e. the number of images displayed per second) are too slow or too variable, the illusion of being present in a virtual environment is likely to be diminished significantly.

It is not only important that the rate at which images appear on the screen be as fast as possible, but also as uniform as possible. For instance, a walkthrough sequence in which nine out of ten images are on the screen for 1/100th of a second and the tenth is on the screen for 9/10ths of a second most likely would not be as satisfying as one in which each image is on the screen for exactly 1/10th of a second, even though the average frame rates are nearly identical. After initial experimentation with interactive walkthroughs, we have chosen a target frame rate of at least ten frames per second.

Short response times (i.e. the time required for the system to react to user input) is also important during interactive visualization. If there are delays in system response, a user may become disoriented or have difficulty navigating in a virtual environment. Even worse, users often complain of feeling sick after using virtual reality systems with especially poor response times. We would like to keep the response time of our visualization system under 1/2 of a second.

Figure 5: Radiosity rendering of hallway.

Figure 6: Radiosity rendering of office.

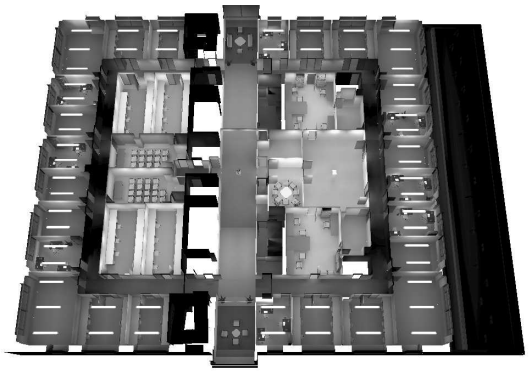


Figure 7: Radiosity computation and results are independent of observer viewpoint.

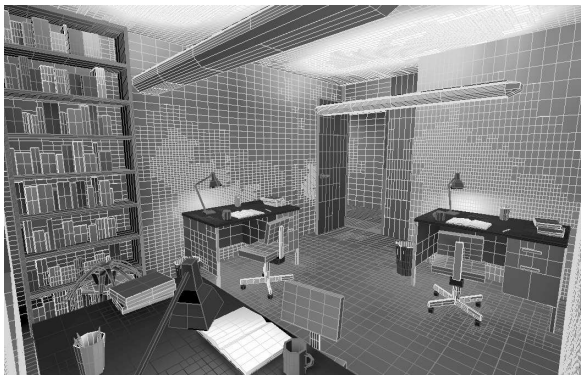


Figure 8: Polygonal mesh generated for the office during radiosity computation.

1.2.4 Hardware

Finally, we aim to support interactive visualization of large, detailed building models using commercially available, off-the-shelf computer systems – rather than building special-purpose hardware. For our display algorithms, we assume a graphics subsystem (either hardware or software) that is able to perform the basic steps required for rendering three dimensional polygons – i.e., modeling transformations, viewing transformations, clipping, projection, rasterization, and hidden surface removal. For our most recent tests, we have used a Silicon Graphics Power series 320 workstation with reality engine graphics. It can draw approximately 50-100K Gouraud shaded, texture-mapped polygons per second.

1.3 Problem Statement

Large, furnished building models are far too complex to be rendered with realistic-looking images at interactive frame rates on currently available hardware. After radiosity analysis, a building model may contain 10^7 independent polygons and require 10^9 bytes of data. However, currently available graphics workstations can render only 10^4 polygons in a tenth of a second, and store only 10^8 bytes of data. Therefore, realistic building models are 10^3 times too large to be rendered at interactive frame rates, and 10^1 times too large to fit into memory.

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 while others become invisible; some objects appear larger and others appear smaller. The challenge is to identify the relevant portion of the model, pre-fetch it into memory, and render images at interactive frame rates as the observer viewpoint is moved under user control.

Our basic approach is to use an efficient display database that describes a building model as a set of objects, each of which is represented at multiple levels of detail, and contains an

index of spatial cells with precomputed visibility information. This display database is used by adaptive display algorithms to compute the set of objects potentially visible from each observer viewpoint. In each frame, an appropriate level of detail is selected for each object during rendering in order to maintain an interactive frame rate. Real-time memory management algorithms are used to predict observer motion and pre-fetch objects from disk that may become visible during upcoming future frames. Using these techniques, we are able to determine a small portion of the model to store in memory and render during each frame of a building walkthrough.

1.4 System Organization

Our building walkthrough system is divided into three distinct computational phases as shown in Figure 9. First, during the *modeling phase*, we construct the 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 indexing, and compute observer-independent lighting and visibility information. Finally, during the *walkthrough phase*, we simulate an observer moving through the building model and render the model as seen from the observer 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.

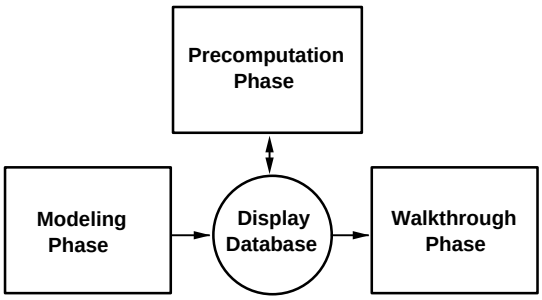


Figure 9: System overview.

Since details of many aspects of this system have been published in previous papers [15, 16, 17, 25, 35, 36, 37], this paper focuses on the overall system design. The paper is organized

as follows. Sections 2 through 4 describe the three phases of the system, respectively. Special attention is paid to how the data structures and algorithms used in each phase are integrated into the entire system. Section 5 presents results of interactive tests using the walkthrough system. Finally, Section 6 contains a summary and conclusions.

2 Modeling

2.1 Model Loading

A primary concern of any interactive visualization system is to obtain a complete and geometrically consistent electronic model. In the case of Soda Hall, a set of files describing the major structural elements of the building (i.e., walls, ceilings, and floors) were prepared by architects using AutoCAD [4]. We implemented a converter that extracts the geometrical and surface attribute information embedded in AutoCAD DXF files [4] and translates them into the Berkeley UNIGRAFIX format [33] a format suitable for 3D object modeling, and then loads them into a display database, as shown in Figure 10.

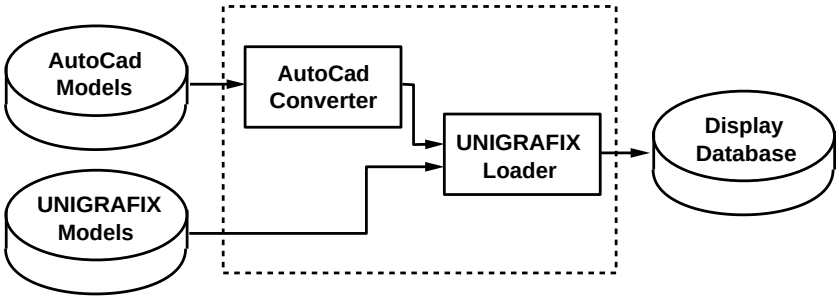


Figure 10: Loading operations of the modeling phase.

Unfortunately, the raw architectural models were not true three-dimensional models. They were originally created for the generation of architectural blueprints such as plan views and crude three dimensional views of buildings. Drafting packages used to prepare them were not written with anticipation that their products might be used in interactive walkthrough systems. Consequently, the initial version of the model that we received from the architects had many problems. Stairs and many other objects were missing; building components

were not modeled as closed objects; windows were only line drawings on the walls; polygons were drawn with no consistent orientation; many co-planar polygons coincided, and many polygons were non-planar or intersected without sharing edges. Therefore, an element-by-element conversion of the architectural database to the UNIGRAFIX format would not have produced a viable model. We used automated programs to detect and correct most of these anomalies [25] and then manually corrected the remaining modeling errors with interactive tools.

Furniture, stairs, and other objects that a user would expect to find in a typical building have been modeled in a variety of ways. Stairs, window frames, and doors were created by Khorramabadi using AutoCAD [4]. Models for many types of furniture (e.g., chairs, desks, and coffee cups) were created with interactive modeling programs by Ward [40]. Other types of furniture (e.g., bookshelves, plants, door handles, and lights) were created by procedural object generators developed by students at UC Berkeley.

Instances of these objects were placed into the building model using both automatic and interactive placement programs. Students in a graduate course on geometric modeling wrote programs that place objects into specific types of rooms automatically based on sets of parameters. For instance, the "conference room generator" places a rectangular or elliptical table in the middle of a room, chairs around the table, 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 to the user for control of object size, number, and placement. Alternatively, we use interactive placement programs, such as AutoCAD, `ugitools` [25] (an interactive UNIGRAFIX tool), or `wkedit` [9] (an interactive walkthrough editor) to generate object instances. These programs allow a user to add, delete, copy, or move object instances interactively with real-time visual feedback.

2.2 Model Representation

The walkthrough display database represents the model as a set of *objects*, each of which can be described at more than one level of detail (LOD). For example, a chair may be described by three different representations, as shown in Figure 11: 1) a highly detailed chair containing hundreds of polygons to approximate the curved surfaces of the cushions and rounded edges of the arms and legs, 2) a slightly less-detailed chair with simpler polygonal approximations for the cushions, arms, and legs, and 3) a coarsely detailed chair with just a simple box for each cushion, arm, and leg. Simpler representations for objects are used during the interactive walkthrough phase to improve refresh rates and memory utilization.

889 Polygons 241 Polygons 44 Polygons

Figure 11: Three LODs for a chair.

In general, if there is more than one instance of the same object type (e.g., if the same type of chair appears in many positions throughout the building), all instances share the same *object definition*, which stores the *geometries* (i.e., vertices, polygons, materials, and textures) describing the object at each LOD. Each instance of the object may specify a 4x4 transformation matrix which is to be applied to the object definition, and a material which is to be applied to polygons that do not already have material attributes. An example hierarchy of object instances and definitions is shown in Figure 12. The model shown contains four instances of a chair whose definition has three LODs, and two instances of a table whose definition has only one LOD.

Although models are most often stored as a hierarchy with shared object definitions, the display database also allows an object instance to store a specific, separate geometry for any LOD. Geometries stored specifically with an object instance override the geometries for the corresponding LODs stored with the object definition. Other LODs (i.e., ones not explicitly stored with an object instance) are inherited from the object definition. This feature is important for the storage of radiosity information, since different instances of the same object definition are likely to be meshed and illuminated differently after a radiosity computation. An example object hierarchy containing an object instance with its own geometries is shown in Figure 13. Chair instance #1 inherits its lowest LOD from the object definition, whereas its medium and high LODs are specified explicitly.

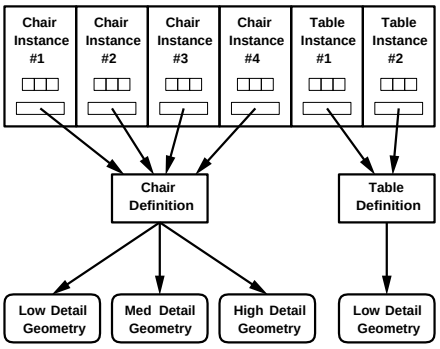


Figure 12: Object instances can share geometries stored in an object definition.

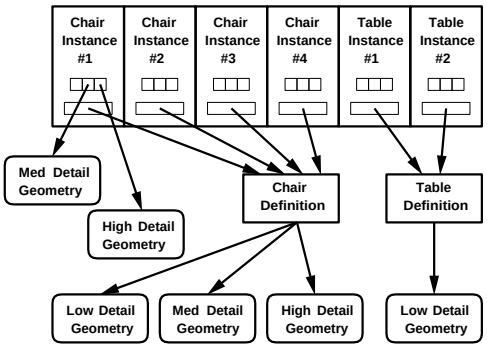


Figure 13: An object instance can store its own geometries.

Objects that move over time are represented by a simple extension to this hierarchy using a technique derived from `ugbump` [27]. The 4x4 transformation of any object instance can be represented by a sequence of strings representing translate, rotate and scale transformations as functions of a variable, t. For instance, the string "-rz \$10 $*$ t\$" means rotate the object around the 'z' axis by 10 degrees every second. Objects are animated as the strings representing their transformation matrices are re-evaluated with a new value of t, which is incremented by the elapsed frame time during the walkthrough.

2.3 Object Abstraction

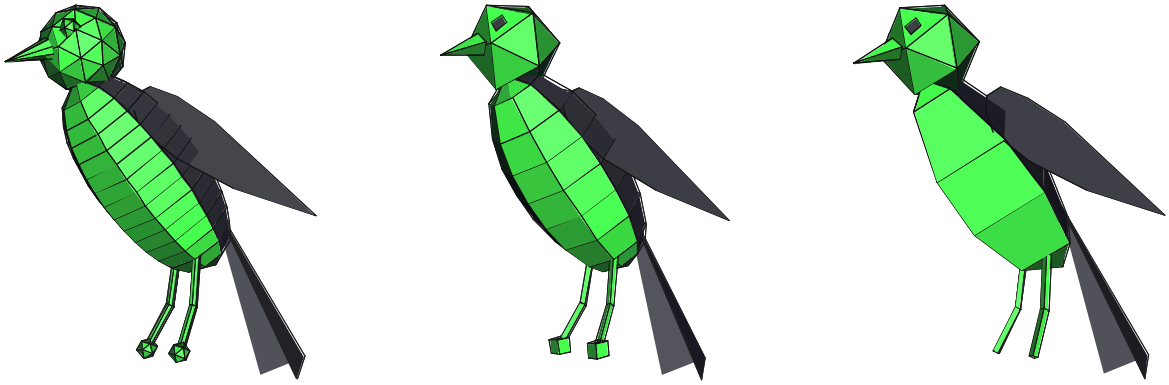
Although we are currently developing automatic tools for object abstraction, so far we have used a variety of ad hoc techniques to create multiple LODs for each object definition. For objects created by procedural generation programs, it is usually possible to extend the programs to produce not only a very detailed model of the object, but also simpler representations as appropriate. For instance, the program used to generate the door handle shown in Figure 14 is parameterized to output segments with a user-specified number of sides.

6-sided prisms. 4-sided prisms.

Figure 14: Different door handle representations constructed using a parameterized generator.

For objects described as a CSG hierarchy of high-level shapes (e.g., boxes, spheres, cones, cylinders, etc.), it is possible to create simpler representations by a combination of: 1) choosing simpler representations for some shapes and 2) removing some shapes. Using these techniques, more than one polygonal representation must be constructed for only a few standard shapes, rather than many complex objects. Example results of this abstraction technique is shown in Figure 15.

For other types of objects, many of which are described originally in a flat, polygonal format containing no information about how they were generated or whether there is a hierarchy of parts, we have constructed less detailed representations from highly detailed originals using an interactive UNIGRAFIX editor, called *animator* [14, 34]. This 3D polyhedron editor has features aimed specifically at reducing the complexity of 3D polyhedral models, including merging vertices, collapsing edges, collapsing faces, etc, while maintaining the topological consistency of the polyhedral object.



336 Polygons 134 Polygons 70 Polygons

Figure 15: Three LODs for a canary.

We have attempted to maintain guidelines regarding construction of LODs in our model of Soda Hall. For example, we fill LODs from lowest to highest for each object. We also aim to generate geometries with no more than 100 polygons in the lowest LOD, and at least double the number of polygons in each successively higher LOD. For each object, appropriate LODs are evaluated and adjusted so that transitions between LODs are barely noticeable as one zooms closer to an object and detail is refined.

2.4 Results

Using the techniques described in this section, we built a three dimensional polygonal model of Soda Hall, complete with furniture, textured materials, and multiple LODs for all furnishings. In all, the model contains 2,217,792 polygons, of which 1,418,807 represent objects at their highest LOD. Including the walls, ceilings, and floors of the building, the model contains 14,478 object instances of 8,037 unique object descriptions. It contains 129 unique pieces of furniture, 406 unique materials, and 58 unique textures. The display database for this model requires 21.5MB of storage if object instances reference shared object definitions, and 349.5MB of storage if all object instances are flattened (i.e., a separate copy of the object definition is stored for each instance). Overall statistics regarding the number of objects described at each LOD, and the cumulative number of polygons used to represent them are shown in Table 1.

Level of Detail	Number of Objects	Polygons			
		Number of Polygons	Minimum Per Object	Mean Per Object	Maximum Per Object
Low	14,478	240,149	1	16.59	2,598
MedLow	3,954	256,895	16	64.97	810
Medium	2,497	476,957	29	191.01	2,707
MedHigh	949	728,978	97	768.15	4,211
High	437	514,813	157	1,178.06	1,977
All	14,478	1,418,807	1	98.00	4,211

Table 1: Multi-resolution modeling statistics for Soda Hall.

3 Precomputation

During the precomputation phase, we perform a set of calculations on the building model that do not depend on a specific observer viewpoint, and thus can be done off-line, before a user begins an interactive building walkthrough. The idea is to precompute complex spatial, visibility, and lighting relationships, and store the results in the display database. Then, during the walkthrough phase, the precomputed relationships can be fetched from the database rather than computed in real-time. By taking this approach, we trade increased space for reduced real-time computation, accelerating frame rates during the walkthrough phase.

The steps of the precomputation phase are shown in Figure 16. We first perform a *spatial subdivision* in which the building model is partitioned into roughly room-sized *cells*, and a cell adjacency graph and an index of objects spatially incident upon each cell is constructed. We then perform a *visibility precomputation* in which sets of cells and objects visible from each cell are computed. Finally, a radiosity computation can be performed. The results of the precomputation phase are stored in the display database for use during the walkthrough phase.

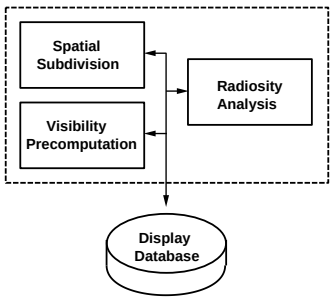


Figure 16: Functional steps of the precomputation phase.

3.1 Spatial Subdivision

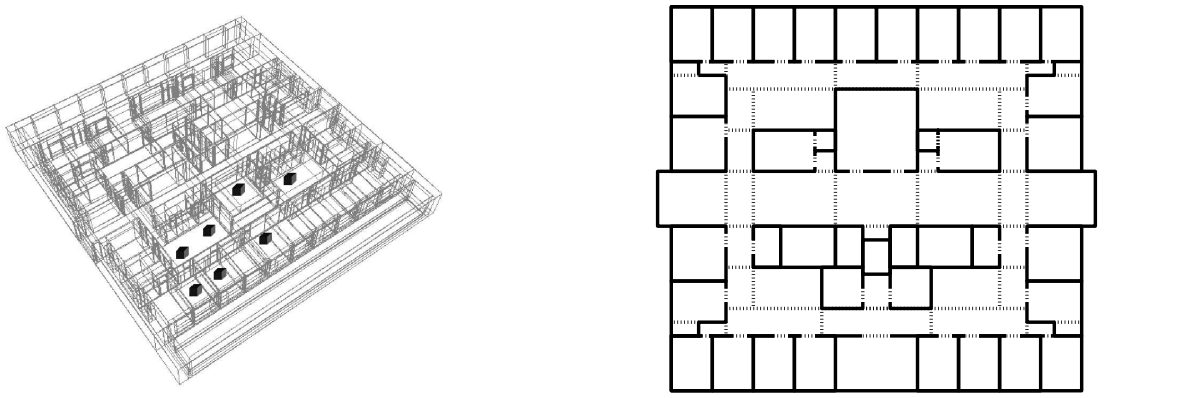
We partition the model into a *spatial subdivision* of *cells* using a variant of the k-D tree data structure [6]. Splitting planes are introduced along the major, axis-aligned, opaque surfaces of the model (i.e., the walls, floors, and ceilings of the building). See [37] for details.

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. Enumerating the portals in this way amounts to constructing an *adjacency graph* over the leaf cells of the spatial subdivision – two cells (nodes) are adjacent (share an edge) if and only if there is a portal connecting them.

Sets of objects partially and completely inside cell boundaries are also constructed and stored with each cell. The space occupied by each object in the model is classified with respect to each cell in the spatial subdivision by a traversal of the k-D tree. At each node of the tree traversal, the bounding box of the object is compared to the bounding box of the node's children. If the intersection has zero volume, the traversal of that branch is terminated. Otherwise, if the node is an interior node, the traversal is applied recursively to the node's children. If the node is a leaf node, the object is classified as either completely or partially inside the cell, and added to the cell's list of incident objects.

The current implementation of the building walkthrough system supports spatial subdivisions containing only axis-aligned, rectangular cell boundaries and portals (i.e., cells are axial three dimensional boxes, and portals are axial two dimensional rectangles). Such a spatial

subdivision for the sixth floor of Soda Hall is shown in three dimensions in Figure 17a – cell boundaries are shown as gray outlines. Figure 17b shows a two dimensional schematic representation of the subdivision, in which opaque cell boundaries are represented by thick, black lines and portals are represented by dashed lines. During spatial subdivision, we precompute and store in the display database for each cell: 1) the portals on its boundaries, 2) the cells sharing its boundaries, and 3) the objects completely or partially inside its boundaries. See [35, 37] for more details.



a) Actual three dimensional subdivision. b) Two dimensional schematic representation.

Figure 17: Spatial subdivision of the sixth floor of Soda Hall. The image on the left shows the actual three dimensional subdivision, while the image on the right shows a two dimensional schematic representation.

3.2 Visibility Precomputation

Once the spatial subdivision has been constructed, we perform a *visibility precomputation* in which the portion of the model visible from each cell is determined. We define a cell's *visibility* to be the region of space visible to a *generalized observer* (i.e., one that is able to look in any direction and move to any position within the cell). The precomputed cell visibility is stored in the display database and used to aid real-time visibility determination and database management algorithms during the walkthrough phase.

We observe that a cell's visibility is the region of space to which an unobstructed *sightline* can lead from some point inside the cell. Such a sightline must be disjoint from any opaque cell boundaries, so it must intersect, or *stab*, a portal in order to pass from one cell to the next

(Figure 18). 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. Therefore, a cell's visibility is the union of all points in space that can possibly be reached by a sightline that originates inside the cell, and intersects only portals at cell boundaries along the way.

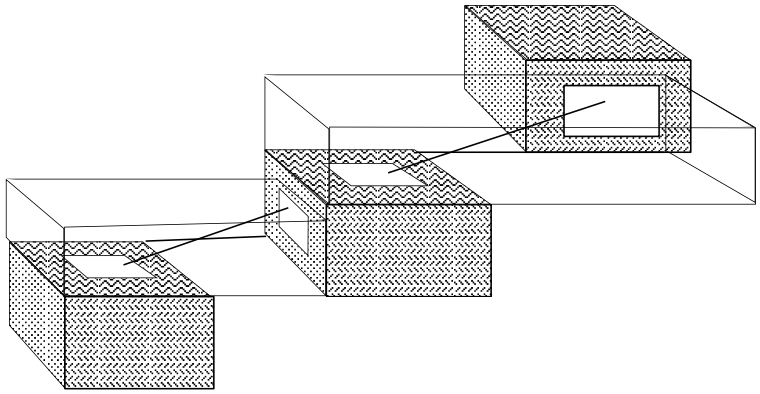


Figure 18: A sight-line stabbing a portal sequence.

These observations suggest that the visibility for a source cell, C, can be computed using a depth-first search of the cell adjacency graph. Cells that are immediate neighbors of C are always entirely visible to it, since cells are convex and all points in the cell can be reached by some sightline stabbing the adjoining portal. Each step into a cell, R, farther away from C, adds another portal to the sequence of portals through which a sightline must pass in order for the cell to be visible to C. If we determine that the portal sequence does not admit a sightline, then R is determined to be *unreachable* along the path. Otherwise, we call R a *reached cell*, and recurse, stepping into cells neighboring R.

The visible region of cells farther away from C typically narrows as the length of the portal sequence increases. After stepping through n portals, the visible region is a bowtie-shaped bundle of lines that stab every portal of the sequence, and which "fans out" beyond the final portal into an infinite wedge (Figure 19). During each iteration of the depth-first search, the visible region of the reached cell is determined by clipping the infinite wedge to the reached cell's boundary. The visibility of the source cell is the union of the visible regions of cells reached during the search (Figure 20).

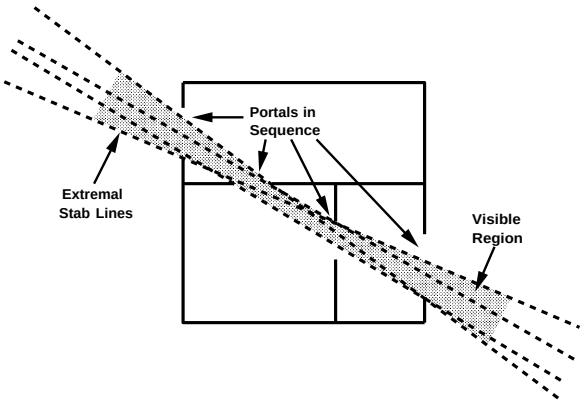


Figure 19: Bowtie-shaped region containing sightlines stabbing a portal sequence (stipple gray). Opaque boundaries are shown in solid black. Extremal sightlines are shown as dashed lines.

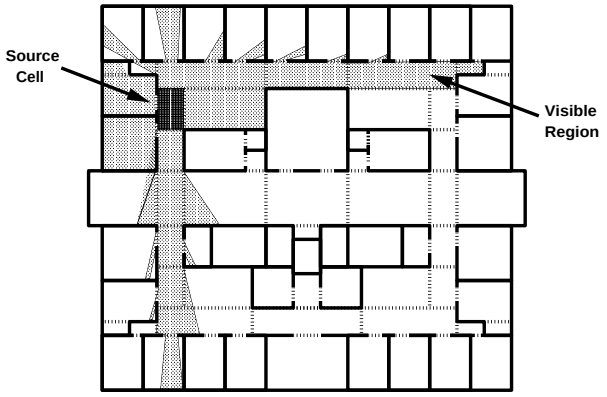
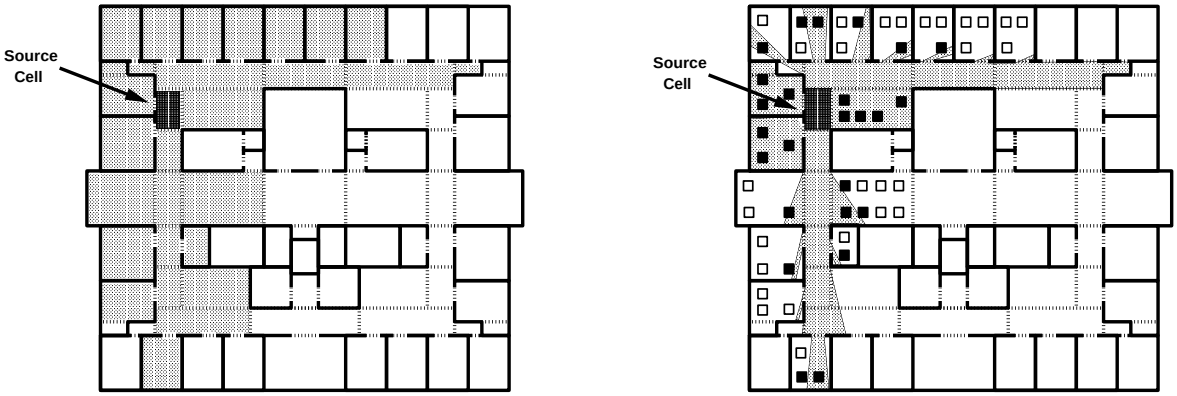


Figure 20: Cell visibility (stipple gray) for source cell (dark gray).

In axial three dimensional models, all portals are axial rectangles, so any portal sequence can generate at most three pairs of bowtie constraints (one from each of two portal edges parallel to the x, y, and z axes). Hohmeyer and Teller have implemented a procedure to find sightlines through axial portal sequences, or determine that no such sightline exists, in $O(n \log n)$ time, where n is the number of portals in the sequence [21]. Amenta has proposed an $O(n)$ solution for this problem [3], although it has not yet been implemented.

During the depth-first search for the source cell C, we construct its *cell-to-cell* and *cell-to-object* visibilities – i.e., the sets of cells and objects, respectively, that are potentially visible from C. The cell-to-cell visibility for C is the set of cells reached during the depth-first search originating from C. The *cell-to-object* visibility is the set of objects whose bounding boxes are incident upon some region visible from C. In Figure 21, schematic diagrams show the cell-to-cell visibility (stipple gray) and cell-to-object visibility (solid black squares) for a particular source cell (dark gray) in two dimensions. Figure 22 shows the cell visibility for a source cell in three dimensions. The visible region is the volume of space enclosed in the polyhedral beams emanating from the source cell; the cell-to-cell visibility is the set of cells outlined; and the cell-to-object visibility (not shown) includes all objects incident upon the brown polyhedral beams.



a) Cell-to-cell visibility. b) Cell-to-object visibility.

Figure 21: Two dimensional schematic diagram of a) cell-to-cell visibility (stipple gray), and b) cell-to-object visibility (shown as solid black squares) for a particular source cell (dark gray).

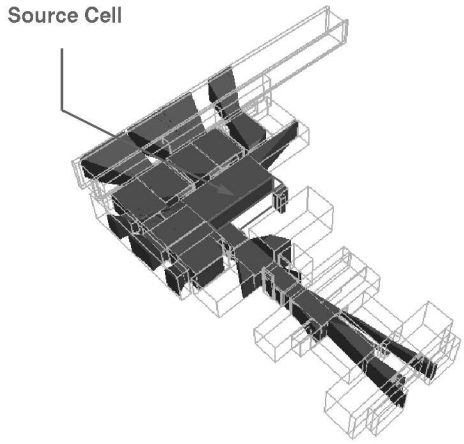


Figure 22: Cell visibility in three dimensions. Visible region (beams) and cell-to-cell visibility (outlined) are shown for one source cell.

The *cell-to-cell* and *cell-to-object* visibility sets are stored in the display database for each source cell, C, in the form of a *stab list*. A stab list contains an entry for each reached cell, R, consisting of: 1) a reference to R, 2) a set of *halfspaces* bounding the portion of R visible from C, and 3) a set of objects in R visible from C, i.e., that are completely or partially inside the assembled halfspaces. One special case exists: if R is a neighbor of C, all objects are tagged as visible from C without any halfspace or object set computations. During the interactive walkthrough phase, the stab list for the observer's cell is retrieved from the display database and culled dynamically based on the observer's position and view direction to help determine the set of objects to load into memory and to render during each frame (Section 4.2.1).

3.3 Results

Mean and maximum precomputation statistics for cells in the spatial subdivision of Soda Hall are shown in Table 2. In all, the spatial subdivision contains 5,060 leaf cells, of which 1,889 are possibly inhabitable by an observer – the other 3,171 cells occupy dead space inside the walls and ceilings. Except for a few large cells that have many portals and objects incident upon them (e.g., the cells that span the entire length of the building just outside the windows), the spatial subdivision classifies the object distribution and visibility properties of the building model fairly well.

Computing the spatial subdivision took 4 minutes and 36 seconds on a SGI 320 using one 33MHz MIPS R3000. 6.9MB are required to store the cells, portals, and lists of objects incident upon each cell. The visibility precomputation took 3 hours and 31 minutes on the same machine and requires 9.4MB of storage for the stab lists.

4 Interactive Walkthrough

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 viewpoint

Statistic	Mean	Maximum
# Portals	3.10	63
# Objects Completely Inside	4.16	86
# Objects Partially Inside	6.04	195
# Cells Visible	65.28	652
# Objects Visible	263.10	3830

Table 2: Mean and maximum statistics for cells in the spatial subdivision of Soda Hall.

in a window on the workstation display at interactive frame rates as the user moves the observer viewpoint through the model.

We use the object hierarchy, spatial subdivision, and results of the visibility precomputation stored in the display database (summarized in Figure 23), along with real-time display and database management algorithms to achieve interactive frame rates.

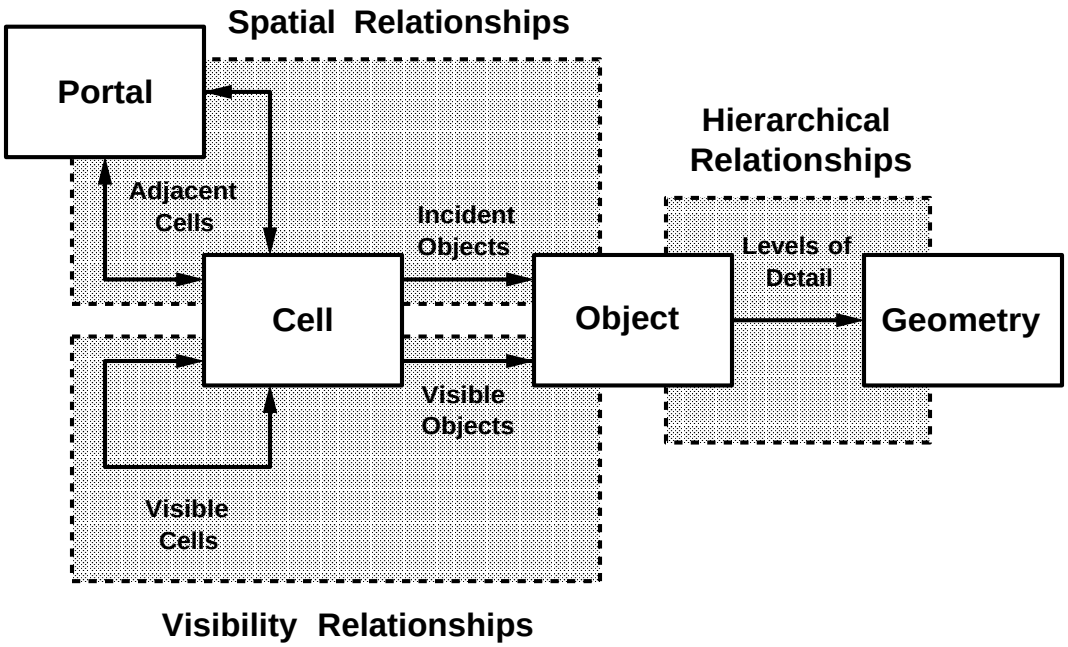


Figure 23: Organization of data in the display database.

Execution during the walkthrough phase proceeds as diagrammed in Figure 24. In every frame, the system performs seven operations, each of which can run asynchronously in a separate concurrent process in a two-forked pipeline:

- **User Interface:** Interact with the user to generate a sequence of observer viewpoints.
- **Visibility Determination:** Compute the set of objects potentially visible from the current observer viewpoint.
- **Detail Elision:** Choose a level of detail and rendering algorithm for each potentially visible object.
- **Rendering Operations:** Render potentially visible objects with the chosen level of detail and rendering algorithm.
- **Lookahead Determination:** Compute the set of objects to store in physical memory, i.e., the ones that might be rendered in upcoming frames.
- **Cache Management:** Determine which objects must be added to or removed from the memory resident cache.
- **Input/Output Operations:** Load and update objects in the display database.

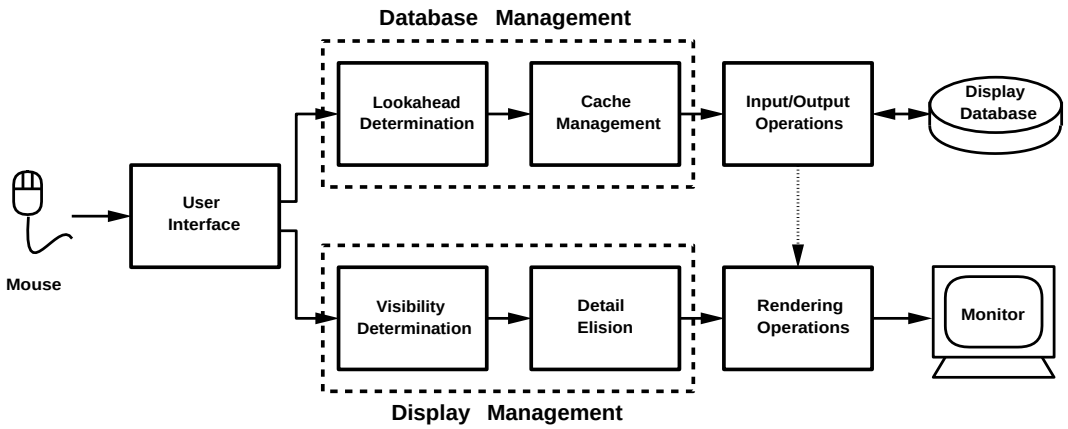


Figure 24: Functional operations of the walkthrough phase.

The operations in the lower fork of the walkthrough pipeline address the *display management* problem. For each observer viewpoint generated by the user interface, the system performs a visibility determination to compute a set of potentially visible objects. Next, detail elision algorithms are used to choose an appropriate level of detail and rendering

algorithm for each potentially visible object. Finally, rendering commands are sent to the graphics workstation to display the potentially visible objects with the chosen levels of detail and rendering algorithms.

The operations in the upper fork of the walkthrough pipeline address the *memory management* problem. The system uses 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.

4.1 User Interface

During every frame of an interactive walkthrough, the system first determines the simulated observer viewpoint. We represent an observer viewpoint by a *view frustum*, which is specified by an *"eye" position*, a *view direction*, *azimuthal* and *altitudinal half angles*, and an *up vector* (Figure 25). This representation is mapped to a perspective viewing transformation [13] for display purposes by setting the center of projection to the observer eye position, the view reference point to the center of the window, and the view plane normal and up vectors to the corresponding frustum parameters (Figure 26). The ratio of the tangents of the azimuthal and altitudinal half angles is kept equal to the aspect ratio of the viewport on the graphics workstation display in order to avoid distortion due to anisotropic scaling.

The walkthrough system used for demonstrations has a simple user interface to control the simulated observer viewpoint based on mouse and keyboard input devices. When the user holds down a mouse button, the simulated observer moves along the current view direction and turns toward the direction indicated by the mouse cursor. For instance, to move straight forward, the user must hold down the left mouse button, while keeping the mouse cursor near the middle of the screen. To turn left, the user holds down a mouse button and moves

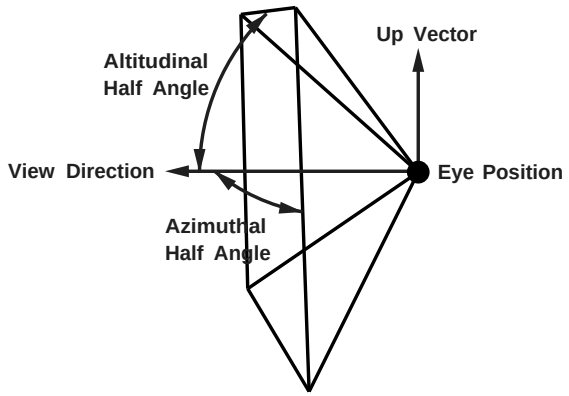


Figure 25: View frustum variables.

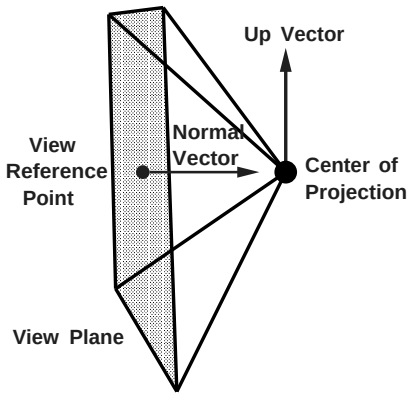


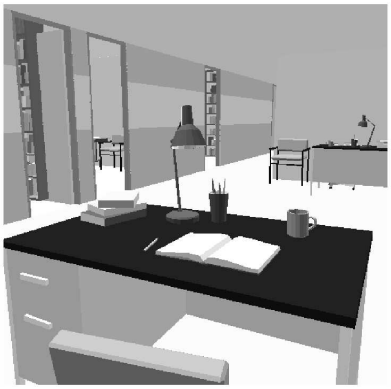
Figure 26: Mapping view frustum to perspective projection.

the mouse cursor to the left part of the screen. By pressing different mouse buttons the user may move forward, backwards, or spin in place.

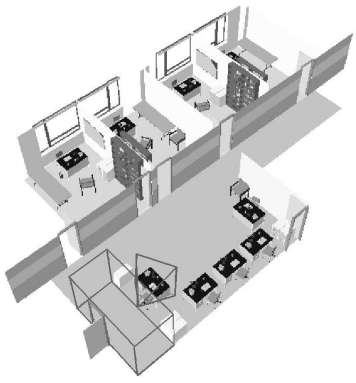
The walkthrough program also supports a variety of panels with toggles, buttons, and sliders to control various program options and display modes for visualization of our algorithms (Figure 28). For instance, the program allows a user to view the building model from either an "interior view" in which the model is rendered from the simulated observer viewpoint (the default), or from a "bird's eye view" in which the model is rendered as if the user were looking down from the sky (Figure 27). In either case, the user can choose to see outlines of cells and objects in various visibility sets, shade objects based on the LOD chosen for display, and/or draw outlines of cells and objects cached in memory. We have found these visualization controls to be essential for debugging and verifying the behavior of complex three dimensional geometric algorithms.

4.2 Display Management

Three dimensional models of large, furnished buildings contain too many polygons to be rendered at interactive frame rates (e.g., ten frames per second) on currently available hardware. We use two techniques to compute a small, but most relevant, portion of the model to render in each frame: 1) we determine the set of objects visible to the observer, and 2) we choose a level of detail and rendering algorithm with which to render each visible object in



a) Interior View



b) Bird's Eye View

Figure 27: Visualization program supports two views.

order to generate the "best" image possible within a user-specified target frame time. Using these techniques, we are able to cull away large portions of the model that are irrelevant from the observer viewpoint, and achieve faster, more uniform frame times than would be possible otherwise.

4.2.1 Visibility Determination

The procedure to compute the portion of the model visible to an observer during the walkthrough phase is similar to the one used to compute cell visibility during the precomputation phase (see Section 3.2). The difference is that we can compute the visibility for the actual observer viewpoint during the walkthrough phase, whereas we computed visibility for each cell (i.e., a generalized observer free to look in any direction and move to any position within the cell) during the precomputation phase.

Given an observer view frustum, F, we first identify the cell, C, containing the observer position and initialize the *visible region* to be the wedge which is the intersection of the volumes enclosed by F and C. Next, we perform a constrained depth-first traversal of the cell adjacency graph, starting at C and propagating outward through portals to neighboring cells. The region of space visible to a view frustum through a sequence of axial rectangular portals can be well-approximated by a polyhedral wedge with at most ten sides (one for each of four planes bounding the view frustum, plus one for each of six planes derived from

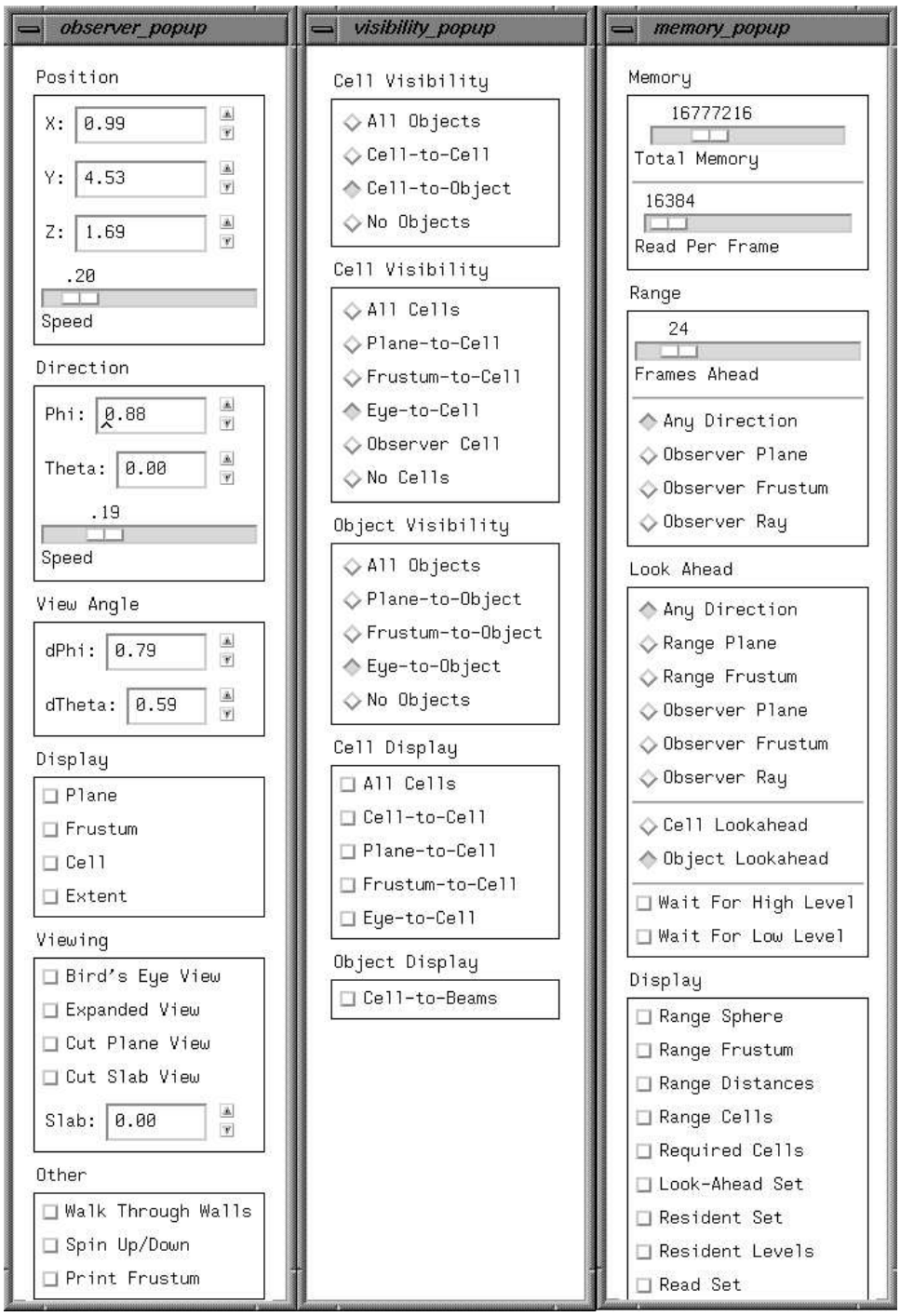


Figure 28: Panels used for controlling parameters for a) observer navigation, b) visibility determination, and c) memory management.

portal edges parallel to the x, y, and z axes). For each step through a portal, P, into a reached cell, R, we update the wedge visible through the current portal sequence, W, by intersection with a wedge bounded by planes through the observer eyepoint and edges of the portal (Figure 29). If the resulting intersection is empty (i.e., W is disjoint from P, so the new portal sequence does not admit any sightline passing through the observer eye position), that branch of the depth-first search is terminated. Otherwise, the region of R visible to the observer (i.e., the intersection of R and W) is included in the visible region, and the search recurses into neighboring cells. See [35, 37] for more details.

A schematic diagram of the visible region computation is shown in two dimensions in Figure 30. The observer view frustum is represented in two dimensions by a wedge outlined by thick black lines, and the visible region is shown in stipple gray.

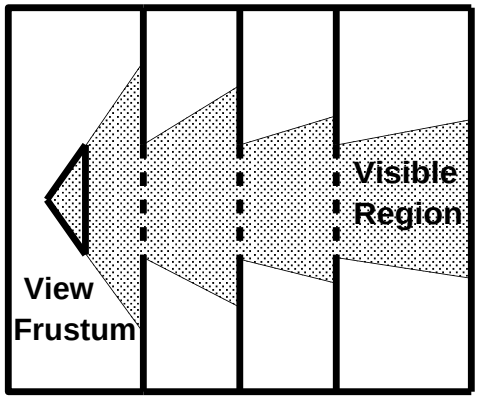


Figure 29: Visible region is a wedge that typically narrows as it traverses through more portals.

During the depth-first search, we construct the *eye-to-cell* and *eye-to-object* visibilities for the observer view frustum – i.e., the sets of cells and objects, respectively, that are reached by some sightline that contains the observer view position, lies within the observer view frustum, and does not pierce any opaque cell boundaries. Construction of these sets can be accelerated by taking advantage of the precomputed visibility information stored in the display database since only cells and objects in the precomputed visibility sets of the cell containing the observer must be checked for a feasible sightline in real-time. Note, however, that the real-time visibility computation does not depend on the availability of precomputed

visibility information. All objects incident upon cells reached during the real-time visibility search can be checked for a feasible sightline with only slightly extra computation (less than 20%) if a visibility precomputation has not been performed.

Figure 31 shows a schematic representation of the eye-to-cell visibility and eye-to-object visibility for a particular observer frustum in two dimensions. Cells in the eye-to-cell visibility are shown in gray stipple. Objects in the eye-to-object visibility are represented by filled squares; whereas those that are in the observer cell's cell-to-object visibility, but not in the observer view frustum's eye-to-object visibility, are shown as hollow squares. Example of eye-to-cell visibility and eye-to-object visibility in three dimensions are shown in Figures 32 and 33, respectively.

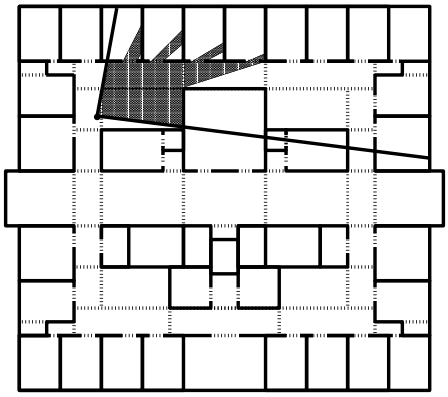


Figure 30: Visible region for view frustum in two dimensions.

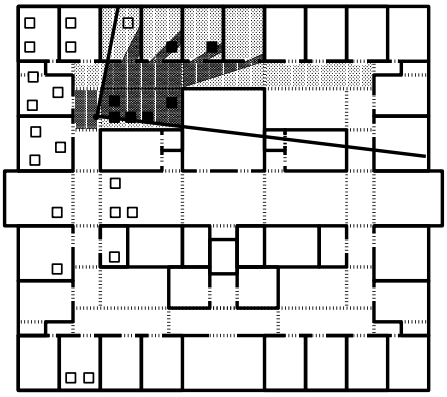


Figure 31: Objects in the eye-to-object visibility are shown as by solid squares.

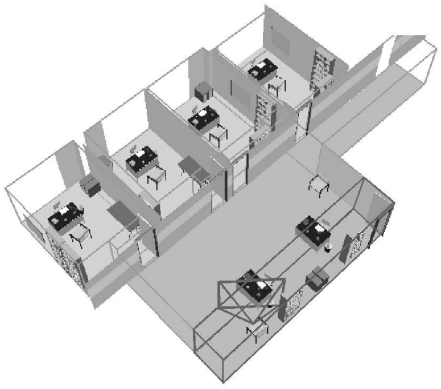


Figure 32: All objects incident upon cells in the eye-to-cell visibility.

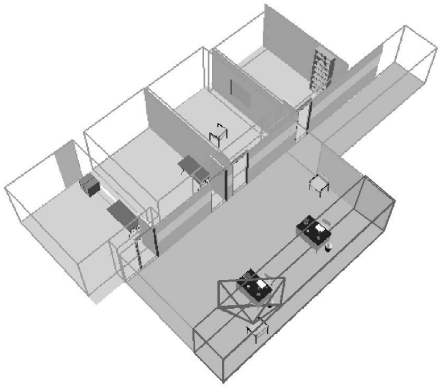


Figure 33: Objects in the eye-to-object visibility.

The eye-to-object visibility for most observer viewpoints is a small subset of all objects in the model, but still a superset of the objects actually visible to the observer. Therefore, we can greatly reduce rendering times if we render only objects in the eye-to-object visibility during each frame of an interactive walkthrough.

4.2.2 Detail Elision

Visibility determination is very effective at culling away a large portion of the model that is invisible to the observer, thereby accelerating frame rates considerably. However, the complexity of the model visible to the observer can still be quite large and highly variable. Tens of thousands of polygons might be simultaneously visible from some observer viewpoints, whereas just a few can be seen from others. For example, in our model of Soda Hall, there are some viewpoints from which an observer can see more than eighty thousand polygons simultaneously, and there are other viewpoints from which only one polygon is visible. Clearly, visibility processing alone is not sufficient to guarantee a uniform, interactive frame rate.

To reduce the number of polygons rendered in each frame, an interactive visualization system can use *detail elision*. If a model can be described by a hierarchical structure of *objects*, each of which is represented at multiple *levels of detail* (LODs) (see Section 2.2), simpler representations of an object can be used to improve frame rates and memory utilization during interactive visualization. This technique was first described by Clark [10], and has been used by numerous commercial visualization systems [32]. If different representations for the same object have similar appearances and are blended smoothly, using transparency blending or three dimensional interpolation, transitions between levels of detail are barely noticeable during visualization.

Previously described detail elision techniques [7, 15, 28, 29, 32, 42], use size or distance heuristics (often with feedback control) to choose a LOD for each object rendered. Simpler representations are used for objects that are small or far away from the observer and thus map to very few pixels on the workstation screen. Although size and distance heuristics for

LOD selection improve frame rates in many cases, they do not generally produce a *uniform* or *bounded* frame rate, even if the threshold is adjusted dynamically with feedback control. During visualization of very large and discontinuous virtual environments, scene complexity can vary radically between successive frames. For instance, in a building walkthrough, the observer may turn around a corner into a large atrium, or step from an open corridor into a small, enclosed office. In these situations, the number and complexity of the objects visible to the observer may change suddenly. Thus, the size threshold chosen based on the time required to render previous frames is inappropriate, and can result in very poor performance until the system reacts. Overshoot and oscillation can occur as the feedback control system attempts to adjust the size threshold to achieve the target frame rate. These effects can be quite disturbing to a user of an immersive visualization system.

Our approach is to use a predictive algorithm to bound the rendering time during each frame. We have developed benefit and cost heuristics and a constrained optimization algorithm to choose an appropriate LOD and rendering algorithm for each potentially visible object in order to generate the "best" image possible within a user-specified target frame time. This approach bounds the frame rate, independent of the complexity of the scene visible to the observer, and thus can generate smoother, more uniform, frame rates during navigation through complex scenes.

The algorithm works as follows. We define an *object tuple*, (O, L, R), to be an instance of object O, rendered at level of detail L, with rendering algorithm R. We define two heuristics for object tuples: $Cost(O, L, R)$ and $Benefit(O, L, R)$. The *Cost* heuristic estimates the time required to render an object tuple, and the *Benefit* heuristic estimates the "contribution to model perception" of a rendered object tuple. We then choose a set of object tuples to render each frame, S, by solving the following constrained optimization problem:

$$\begin{aligned} &\text{Maximize :} \\ &\qquad \sum_S Benefit(O, L, R) \\ &\text{Subject to :} \end{aligned} \tag{1}$$

$$\sum_S Cost(O, L, R) \leq TargetFrameTime$$

This constrained optimization approach captures the essence of image generation with real-time constraints: "do as well as possible in a given amount of time." As such, it can be applied to a wide variety of problems that require images to be displayed in a fixed amount of time, including adaptive ray tracing (i.e., given a fixed number of rays, cast those that contribute most to the image), and adaptive radiosity (i.e., given a fixed number of form-factor computations, compute those that contribute most to the solution). If levels of detail representing "no polygons at all" are allowed, this approach handles cases where the target frame time is not long enough to render all potentially visible objects even at the lowest level of detail. In such cases, only the most "important" objects are rendered so that the frame time constraint is not violated. Using this approach, it is possible to generate images in a short, fixed amount of time, rather than waiting much longer for images of the highest quality attainable.

Of course, *Cost* and *Benefit* heuristics for a specific object tuple cannot be predicted with perfect accuracy, and may depend on other object tuples rendered in the same image. A perfect *Cost* heuristic may depend on the model and features of the graphics workstation, the state of the graphics system, the state of the operating system, and the state of other programs running on the machine. A perfect *Benefit* heuristic would consider occlusion and color of other object tuples, human perception, and human understanding. We cannot hope to quantify all of these complex factors in heuristics that can be computed efficiently. However, using several simplifying assumptions, we have developed approximate *Cost* and *Benefit* heuristics that are both efficient to compute and accurate enough to be useful.

Our *Cost* heuristic estimates rendering time as a linear combination of the number of primitives drawn and the number of pixels filled. We have found that this simple heuristic is able to predict rendering times within 10% at the 95% confidence level on an SGI VGX 320 workstation.

Our $Benefit(O, L, R)$ heuristic estimates the "contribution to model perception" of an

object tuple by a linear combination of quantitative and qualitative factors, including the object size, rendering accuracy, object semantics, position in the image, motion blur, and hysteresis. Greater *Benefit* is assigned to object tuples that are larger (i.e., cover more pixels in the image), more realistic-looking (i.e., rendered with higher levels of detail, or better rendering algorithms), more important (i.e., semantically, or closer to the center of the screen), and more apt to blend with other images in a sequence (i.e., hysteresis).

We use the *Cost* and *Benefit* heuristics to choose a set of object tuples to render each frame by solving the constrained optimization in Equation 1. Unfortunately, this constrained optimization problem is NP-complete. It is the Continuous Multiple Choice Knapsack Problem [18, 22], a version of the well-known Knapsack Problem in which elements are partitioned into candidate sets, and at most one element from each candidate set may be placed in the knapsack at once. We have implemented a simple, greedy approximation algorithm for this problem that selects object tuples with the highest $Benefit/Cost$ [23, 30]. Our implementation finds an approximate solution that is at least half as good as the optimal solution in $O(n \log n)$ for n potentially visible objects. However, since the initial guess for the LOD and rendering algorithm for each object is generated from the previous frame, and there is often a large amount of coherence from frame to frame, the algorithm completes in just a few iterations on average. Moreover, computations are done in parallel with the display of the previous frame on a separate processor in a pipelined architecture and thus do not increase the effective frame rate as long as the time required for computation is not greater than the time required for display. See [16, 17] for more details.

4.3 Memory Management

Realistic-looking three dimensional models may be much larger than can fit into main memory. Thus, an interactive walkthrough system must swap portions of the model in and out of memory in real-time as the observer navigates through the model. This is especially difficult since it takes a large amount of time to load data from disk into memory. We must

not only store in memory the portion of the model that is visible from the current observer viewpoint, but also we must pre-fetch portions of the model that might become visible in future frames. Consequently, we have implemented a predictive memory management algorithm that forecasts a range of possible observer viewpoints during the next N frames and uses precomputed cell-to-cell and cell-to-object visibility information fetched from the display database to determine a *lookahead set* of objects (i.e., a set of objects that are likely to be visible to the observer during the next N frames). We choose a level of detail at which to store each lookahead object in a *memory resident cache*. Simple 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.

4.3.1 Pre-Fetch Algorithm

An ideal memory management algorithm predicts the observer viewpoint for each future frame perfectly. It can then use the visibility determination and detail elision algorithms described in Sections 4.2.1 and 4.2.2 to determine exactly which objects and LODs will be rendered during future frames and pre-fetch them into memory, replacing ones 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.

In order to pre-fetch objects into memory before they are rendered, we must determine in advance which objects are likely to become visible to the observer. In each frame of an interactive walkthrough, we compute a set of *range cells*, R, that are likely to contain the observer eye position during the next N frames by performing a shortest path search of the cell adjacency graph. The search, implemented using Dijkstra's method [12], adds cells to the range set ordered by the minimum number of frames before the observer can enter the cell. Maximum positional and rotational velocities and constraints preventing observers from walking directly through solid walls are used to determine the number of frames required

for the observer to enter a cell. Figure 34 shows an example shortest path search result. Each cell is labeled by the minimum number of frames before it can contain the observer, assuming the observer is constrained to the maximum positional and rotational velocities, and cannot walk through walls. For $N = 4$, range cells are highlighted in cross-hatch.

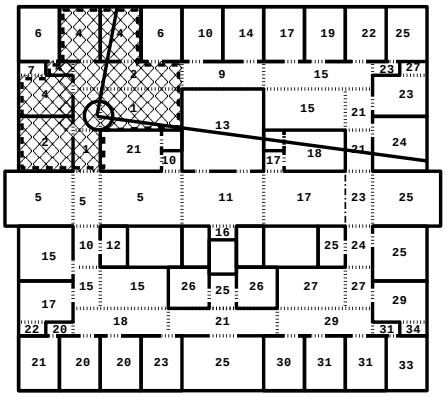


Figure 34: The observer range cells (cross-hatch) contain all observer view positions possible during the upcoming $N = 4$ frames. Each cell is labeled by the number of frames before the observer can be resident in it.

We use precomputed cell-to-cell and cell-to-object visibility information for the range cells fetched from the display database to compute a set of *lookahead objects* that are likely to be visible to the observer during the next N frames. When a new range cell is discovered during the shortest path search, we add all potentially visible objects incident upon the cell to the lookahead set. Figure 35 shows an example computation of the lookahead set of objects. Each cell is labeled by the minimum number of frames before it can become visible to a cell in the observer range set. For $N = 4$, cells in the range set are highlighted again in cross-hatch, and cells containing objects in the lookahead set are highlighted in stipple gray.

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 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, as shown in Figure

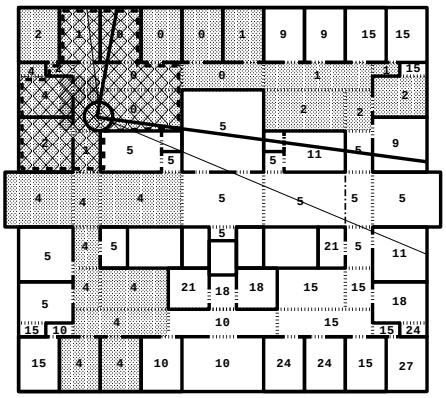


Figure 35: The lookahead cells (stipple gray) contain all objects that can be visible to the observer during the upcoming N frames. Each cell is labeled by the number of frames before it can become visible to the observer.

36.

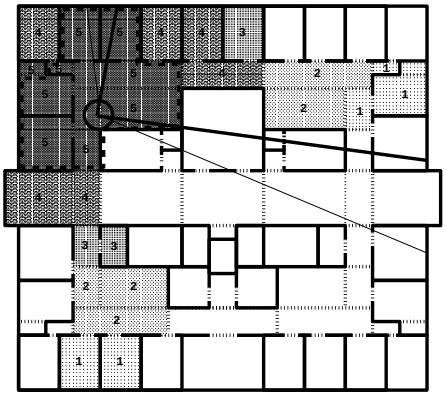


Figure 36: Lookahead objects are stored in memory only up to the LOD at which they can possibly be rendered during the next N frames. Each cell is labeled and shaded by the maximum level of detail any object incident upon it is stored in memory – darker shades of gray represent higher levels of detail.

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, 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. See [17] for more details.

4.3.2 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 accurate 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 37 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 (SE/NW-hatch), or release set (SW/NE-hatch).

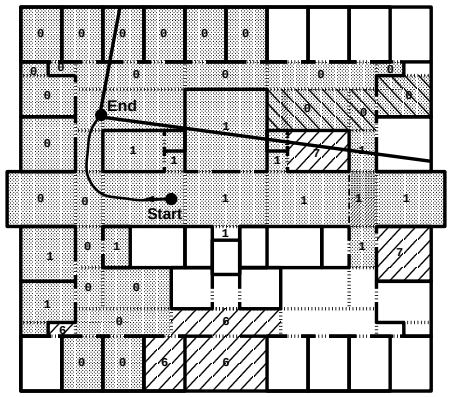


Figure 37: Cache management algorithm results. Each cell is labeled by the number of frames since objects incident upon it were included in the lookahead set. Shading for each cell indicates whether or not it contains objects in the memory resident cache (stipple gray), read set (left-hatch), and resident set (right-hatch).

4.3.3 Asynchronous Pre-Fetch

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 sys-

tem concurrently 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 new data becomes visible to the observer may be greater 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 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

To evaluate the effectiveness of the display and memory management algorithms described in the previous section, we ran a series of tests using a complete building walkthrough application, logging statistics as a user "walked" through the model of Soda Hall.

We ran two experiments: one that tested only display management algorithms (i.e., visibility determination and detail elision), and another that also tested memory management algorithms (i.e., pre-fetching objects into memory in real-time). Both experiments were performed on a Silicon Graphics Power series 320 workstation with reality engine graphics, two 33MHz MIPS R3000 processors, 128MB of memory, and a 16μs timer. In the display management experiment, the application was configured as a two-process pipeline with one processor used for user interface, visibility determination and detail elision computations, and a second processor used for rendering. In the memory management experiment, the application was configured as a four-process pipeline using the same processes as the display management experiment, but also with memory management computations in a separate third process, and database input/output operations in a separate fourth process.

In both experiments, we used the observer path shown in Figure 38. It contains over 9,000 observer viewpoints, and visits the top four floors of Soda Hall.

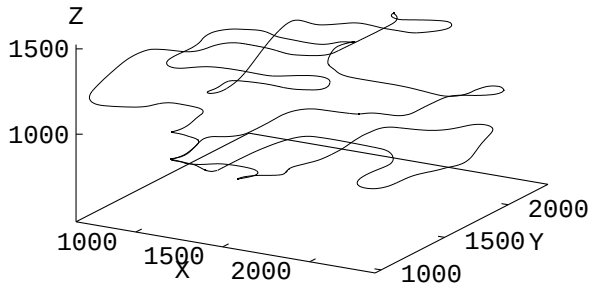


Figure 38: Test observer path through the top four floors of Soda Hall.

5.1 Display Management

In the first experiment, we tested the combined effectiveness of the display management algorithms described in this section, independent of memory management. We used a hierarchical representation of the Soda Hall model with shared object definitions requiring only 21.5MB of storage so that the entire model could be resident in memory for the duration of the experiment. We ran three tests using the following combinations of real-time visibility determination and detail elision algorithms:

- **Entire Model:** Renders every object in the model at its highest level of detail – i.e., without any visibility determination or detail elision.
- **Visibility:** Renders each object in the eye-to-object visibility of observer viewpoint at its highest level of detail.
- **Detail Elision:** Render each object in the eye-to-object visibility of observer viewpoint with the level of detail chosen by the optimization detail elision algorithm to bound the frame rate at a minimum of fifteen frames per second.

During each test, we measured the compute time (i.e., the real-time visibility determination and detail elision compute times), the rendering time, and the frame time (i.e., total time between successive frames), as well as the numbers of cells, objects, and polygons rendered in each frame. Mean and maximum statistics for all observer viewpoints along the test walkthrough path are shown for each combination of visibility determination and detail elision algorithms in Tables 3–4. Figure 39 contains a plot of the frame time for each observer viewpoint along the test path during the test using both visibility and detail elision.

During the *Entire Model* test, in which we rendered every object at the highest level of detail, the graphics engine processed all 1,418,807 polygons during every frame and the frame time was 15.413 seconds (0.06 frames per second!) on average.

During the *Visibility* test, in which we used the eye-to-object real-time visibility determination algorithm without detail elision, we rendered 5,002 polygons (0.35% of the model) on

Cull Method	Compute Time (s)		Render Time (s)		Frame Time (s)	
	Mean	Max	Mean	Max	Mean	Max
Entire Model	–	–	15.413	17.426	15.413	17.426
Visibility	0.018	0.107	0.061	0.557	0.067	0.562
Detail Elision	0.033	0.199	0.032	0.083	0.047	0.199

Table 3: Mean and maximum compute time, rendering time, and frame time statistics collected during display management tests.

Cull Method	# Cells		# Objects		# Polygons		% of Model	
	Mean	Max	Mean	Max	Mean	Max	Mean	Max
Entire Model	5,060		14,478		1,418,807		100%	
Visibility	19	70	105	469	5,002	45,828	0.35%	3.23%
Detail Elision	19	70	104	445	2,534	6,912	0.18%	0.49%

Table 4: Mean and maximum numbers of cells, objects, and polygons rendered during display management tests.

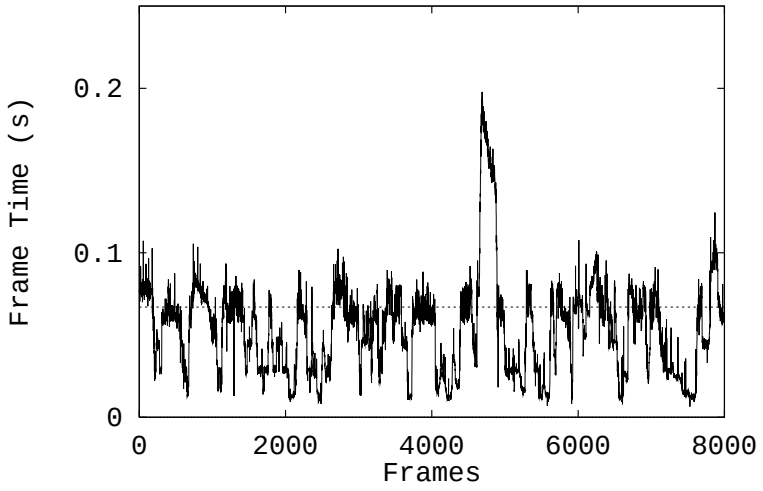


Figure 39: Frame time for each observer viewpoint along the entire test observer path during the test using both visibility determination and detail elision.

average, and the mean frame time was 0.067 seconds (14.9 frames per second). However, the maximum frame time was still quite large using only visibility determination because there are some viewpoints along the test path at which the scene visible to the observer is very complex (469 objects containing 45,828 polygons at the highest level of detail). At these viewpoints, the rendering time increased to 0.557 seconds (1.8 frames per second).

During the *Detail Elision* test, in which we used both the eye-to-object visibility algorithm and the optimization detail elision algorithm to choose an appropriate LOD for each potentially visible object dynamically, we rendered only 2,534 polygons (0.18% of the model) and generated an image every 0.047 seconds (21.3 frames per second) on average. The maximum rendering time was 0.083 seconds, and the the maximum response time was 0.274 seconds – short enough for the system to maintain a highly interactive feel. A plot of rendering times measured during the test with detail elision are shown in Figure 41. Rendering time is bounded by the detail elision algorithm and the observer step size is adjusted by the previous frame time resulting in a smooth, interactive walkthrough with constant velocity. Although these bounded rendering times were achieved by rendering simpler representations for some objects during some frames, the visual effects of detail elision were barely noticeable (see Figure 40 for an example).

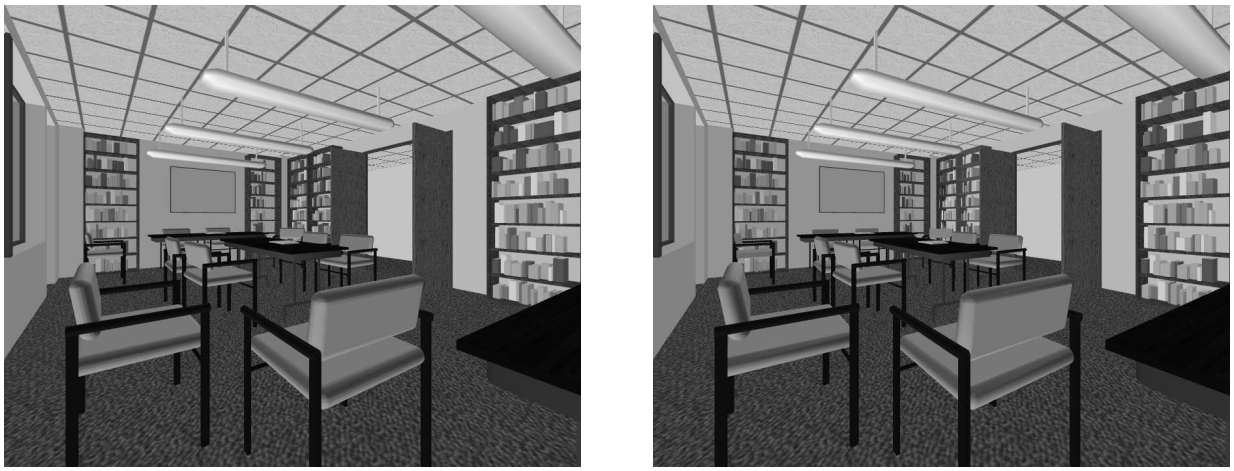


Figure 40: Images of library generated using (left) no detail elision (19,821 polygons), and (right) using the *Optimization* detail elision algorithm with a target frame time of 0.05 seconds (3,568 polygons).

There is one peak in the frame time during the detail elision test (near frame 4700 in Figure 39). This peak occurred because the computation time was larger than the rendering time during this portion of the walkthrough test. Figure 42 shows the measured compute time, and rendering time for each frame along the portion of the walkthrough path for which computation was most expensive. Since the walkthrough system was configured as a two process concurrent pipeline in this test, with visibility determination and detail elision computations performed in one process and rendering in another, the effective frame time was closely correlated with the maximum of the compute time and the rendering time. The process performing visibility and detail elision computations slowed to 0.199 seconds per frame in the worst case, and caused the effective frame rate to slow to 5 frames per second for a short period, even though the rendering time was still less than 0.8 seconds. However, note that the frame times would have been far greater if either the visibility determination or detail elision computation were not performed. There is currently no adaptive control over the time required for computation in our system. In a future version, perhaps computations can be optimized, executed with more parallelism, or be made adaptive so that they never exceed the target frame time.

5.2 Memory Management

In the second experiment, we tested the cumulative effectiveness of the display and memory management algorithms described in the previous sections. We used the same display management algorithms as were used in the test labeled *Detail Elision* in the previous test. However, in this experiment, we used the flattened model of Soda Hall (349.5MB of storage) and the predictive memory management algorithm described in Section 4.3 to swap objects in and out of memory in real-time as the observer moved along the test walkthrough path. To test the algorithms to their fullest, we artificially limited the size of the memory resident cache to 16MB.

In addition to the statistics measured during the display management experiment, we

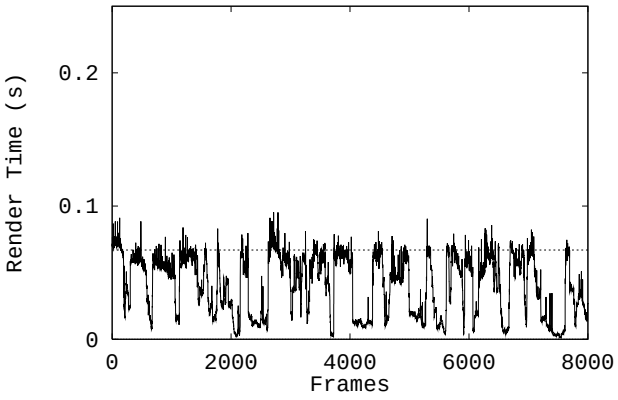


Figure 41: Rendering times during *Detail Elision* test for each observer viewpoint along the test observer path.

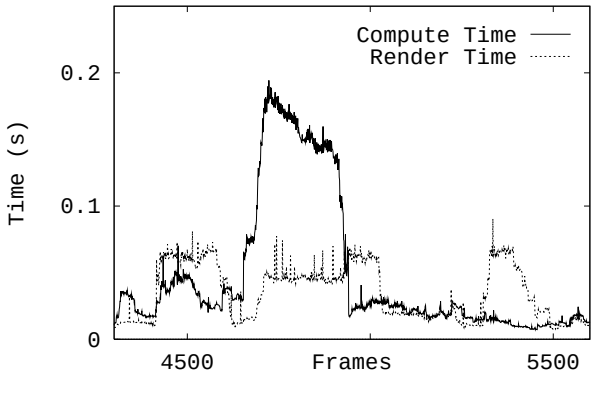


Figure 42: Compute time and rendering time during *Detail Elision* test for each observer viewpoint along the portion of test observer path where computation was most expensive.

gathered statistics regarding the the swap time (i.e., the time required to load objects into memory from disk), and the sizes of resident, lookahead, rendered, and read sets. Mean and maximum statistics for all observer viewpoints along the test walkthrough path are shown for tests with and without memory management in Table 5. Table 6 contains mean and maximum statistics regarding the lookahead sets collected during the memory management test. Figure 43 contains a plot of the frame time for each observer viewpoint along the test path during the test using memory management.

The entire flattened model of Soda Hall requires 349.5MB of storage. However, using the predictive memory management algorithm during an interactive walkthrough with 16MB of memory resident cache, we were able to maintain a frame rate of 21.3 frames per second on average, the same average frame rate achieved during the test on an entirely memory resident model described in the previous section. The addition of memory management processing added to the frame time variance only slightly.

Of the 349.5MB of data in the entire model, only 382KB of data was required to describe the polygons rendered during each frame on average. The lookahead algorithm was able

Mem Mgmt	Compute Time (s)		Render Time (s)		Swap Time (s)		Frame Time (s)	
	Mean	Max	Mean	Max	Mean	Max	Mean	Max
No	0.033	0.199	0.032	0.083	–	–	0.047	0.199
Yes	0.040	0.244	0.034	0.124	0.020	0.898	0.047	0.244

Table 5: Mean and maximum compute time, rendering time, swap time, and frame time statistics collected during tests with and without memory management.

Object Set	# Objects		# Polygons		# MBytes		% of Model	
	Mean	Max	Mean	Max	Mean	Max	Mean	Max
Entire Model	14,478		1,418,807		349.5		100%	
Resident	986	1,698	97,081	128,102	14.517	18.102	4.15%	5.18%
Lookahead	628	3,209	66,297	362,831	9.916	54.409	2.84%	15.6%
Rendered	104	445	2,534	6,912	0.382	0.993	0.11%	0.28%
Read	2	432	266	6,501	0.024	0.795	0.01%	0.23%

Table 6: Mean and maximum set statistics collected during memory management tests.

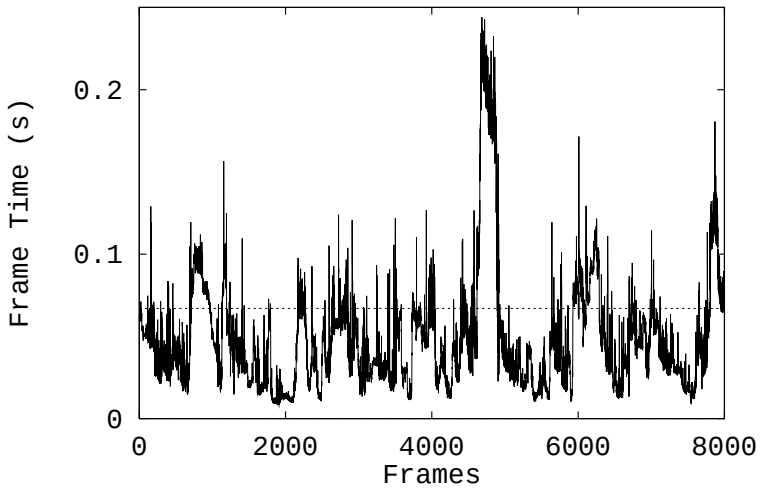


Figure 43: Frame time for each observer viewpoint along the entire test observer path during the test with memory management.

to determine a relatively small set of object descriptions to store in memory (9.916MB on average) by predicting which object descriptions were most likely to be rendered during upcoming frames. The lookahead computation required very little extra time (0.002 seconds per frame on average), and executed in a separate asynchronous process, and thus had very little impact on the effective frame time.

During each frame on average, 24KB of data describing objects new to the lookahead set was read from disk into memory, requiring 0.020 seconds per frame. Although the read time was highly variable (the maximum read time was 0.898 seconds), read operations were performed in an asynchronous database input/output process; so large read times did not affect the frame time directly.

In order to maintain an interactive frame rate, the rendering process did not wait for object descriptions to be loaded into memory during the memory management test. Instead, it simply skipped object LODs that were not yet memory resident, and rendered the object at the next highest LOD that was resident in memory. During this test, 1.43 LODs were skipped on average, and as many as 35 LODs were skipped during some frames (e.g., when the observer entered a region of the model with many complex objects). Although this effect was noticeable during one short portion of the walkthrough path, we found it to be less disturbing than the alternative – i.e., requiring the rendering process to stall while waiting for the appropriate LOD for an object to be read into memory before it was rendered.

6 Conclusion

This paper describes a system for interactive walkthroughs of very large architectural models. Our test case is a description of Soda Hall, the CS building at the University of California at Berkeley. The original model, which was generated by architects using AutoCAD, was converted to a three dimensional format (UNIGRAFIX) using algorithms that detect and correct the many modeling errors automatically. A considerable amount of direct user interaction with manual tools was needed to convert the model into a format suitable for three

dimensional computation and rendering.

Data structures and algorithms described in this paper helped maintain interactive frame rates during walkthroughs of such large architectural models. We built a hierarchical display database containing objects represented at multiple levels of detail during a modeling phase, performed a spatial subdivision and visibility analysis during a precomputation phase, and used real-time culling, display and memory management algorithms during a walkthrough phase to select a relevant subset of the model for rendering and storing in memory.

We have found visibility determination to be extremely useful for generating fast frame rates during visualization of complex models. By partitioning the model into a spatial subdivision of cells whose boundaries coincide with the major axis-aligned, occluding polygons, and by performing visibility precomputations to determine a superset of objects visible from each cell, we characterize the portion of the model visible from different regions of space during an off-line computation. We use this precomputed spatial and visibility information in the real-time visibility determination algorithms to determine the relevant subset of the model for each observer viewpoint during an interactive walkthrough.

However, visibility determination alone is not sufficient to generate fast, uniform frame rates during visualization of complex models. There may be observer viewpoints at which far too many polygons are simultaneously visible to render at interactive frame rates. Therefore, during each frame of an interactive walkthrough, we execute an *Optimization* detail elision algorithm that may choose a reduced level of detail or a simpler rendering algorithm with which to display each potentially visible object in order to achieve a user-specified target frame time.

We have found it possible to swap objects in and out of memory as a simulated observer walks through a model much larger than physical memory. We use an asynchronous prefetch algorithm to load objects that are likely to become visible to the observer up to the maximum level of detail at which they can possibly be rendered during upcoming frames. An approximate least-recently used algorithm determines which objects to replace in the

memory resident cache as new objects are added.

We have implemented a first version of a complete architectural visualization system and tested it in real walkthroughs of a furnished model of Soda Hall containing over 1.4 million polygons. Our initial results show that the display and memory management techniques are effective at culling away a substantial portion of the model, and make interactive frame rates of twenty frames per second possible even for very large models.

7 Acknowledgements

We are grateful to Thurman Brown, Rick Bukowski, Laura Downs, Priscilla Shih Maryann Simmons, and Ajay Sreekanth for their efforts constructing the building model. We also thank Silicon Graphics, Inc. for allowing us to use equipment, and donating a VGX 320 workstation to this project as part of a grant from the Microelectronics Innovation and Computer Research Opportunities (MICRO 1991) 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., 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.

[3] Amenta, Nina. Finding a Line Traversal of Axial Objects in Three Dimensions. *Proc. 3^{rd} Annual ACM-SIAM Symposium on Discrete Algorithms*, 1992, 66-71.

[4] *AutoCAD Reference Manual*, Release 10, Autodesk Inc., 1990.

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

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

[7] Blake, Edwin H. A Metric for Computing Adaptive Detail in Animated Scenes using Object-Oriented Programming. *Eurographics '87.* G. Marechal (Ed.), Elsivier Science Publishers, B.V. (North-Holland), 1987.

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

[9] Brown, Thurman A. *Interactive Object Displacement in Building Walkthrough Models.* Master's Thesis, Computer Science Division (EECS), University of California, Berkeley, 1992.

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

[11] 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.

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

[13] Foley, J.D., A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice.* 2nd ed., Addison-Wesley, Reading, MA, 1990.

[14] Funkhouser, Thomas A. *An Interactive UNIGRAFIX Editor.* Unpublished. May, 1991.

[15] Funkhouser, Thomas A., Carlo H. Séquin, and Seth J. Teller. Management of Large Amounts of Data in Interactive Building Walkthroughs. *ACM SIGGRAPH Special Issue on 1992 Symposium on Interactive 3D Graphics*, March, 1992, 11-20.

[16] 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..

[17] 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.

[18] Garey, Michael R. and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

[19] 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.

[20] Greene, Ned, Michael Kass, and Gavin Miller. Hierarchical Z-Buffer Visibility. *Computer Graphics (Proc. SIGGRAPH '93)*, (August 1993), 231-238.

[21] Hohmeyer, Michael E., and Seth J. Teller. *Stabbing Isothetic Rectangles and Boxes in $O(n \lg n)$ Time.* Technical Report UCB/CSD 91/634, Computer Science Department, U.C. Berkeley, 1991.

[22] Ibaraki, T., T. Hasegawa, K. Teranaka, J. Iwase. The Multiple Choice Knapsack Problem. *J. Oper. Res. Soc. Japan* 21, 1978, 59-94.

[23] Ibarra, O. H. and C. E. Kim. Fast Approximate Algorithms for the Knapsack and Sum of Subset Problems. *J. Assoc. Comput. Mach.* 22, 1975, 463-468.

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

[25] Khorramabadi, Delnaz. *A Walk through the Planned CS Building.* Master's Thesis, Computer Science Division (EECS), University of California, Berkeley, 1991. Also available as UC Berkeley technical report UCB/CSD 91/652.

[26] Nishita, T., and E. Nakamae. Half-Tone Representation of 3D Objects Illuminated by Area Sources or Polyhedron Sources. *Computer Graphics (Proc. SIGGRAPH '85)*, 19, 3 (July 1985), 23-30.

[27] Oakland, Steven Anders. *BUMP, A Motion Description and Animation Package.* Technical Report UCB/CSD 87/370, Computer Science Department, U.C. Berkeley, 1987.

[28] Rossignac, J. and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. *IFIP TC 5.WG 5.10 II Conference on Geometric Modeling in Computer Graphics*, Genova, Italy, 1993. Also available as IBM Research Report RC 17697, Yorktown Heights, NY 10598.

[29] Rubin, S. M. The representation and display of scenes with a wide range of detail. *Computer Graphics and Image Processing.* 19 (1982), 291-298.

[30] Sahni, S. Approximate Algorithms for the 0/1 Knapsack Problem. *J. Assoc. Comput. Mach.* 22, 1975, 115-124.

[31] Schachter, Bruce J. Computer Image Generation for Flight Simulation. *IEEE Computer Graphics and Applications.* 1, 5 (1981), 29-68.

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

[33] 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.

[34] Smith, Kevin, P. *Interactive Modeling Tool.* Unpublished. September, 1990.

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

[36] Teller, Seth J. Computing the Antiumbra Cast by an Area Light Source. *Computer Graphics (Proc. SIGGRAPH '92)*, 26, 2 (August 1992), 139-148.

[37] 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.

[38] Teller, Seth J., Celeste Fowler, Thomas Funkhouser, and Pat Hanrahan Partitioning and Ordering Large Radiosity Computations. To appear in *Computer Graphics (Proc. SIGGRAPH '94)*, July, 1994.

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

[40] Ward, Greg. Lawrence Berkeley Laboratories. Personal Communication, 1993.

[41] Ware, Colin, and Steven Osborne. Exploration and Virtual Camera Control in Virtual Three Dimensional Environments. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24, 2 (1990), 171-176.

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

[43] 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.