Database and Display Algorithms for Interactive Visualization of Architectural Models

by

Thomas Allen Funkhouser

B.S. (Stanford University) 1983
M.S. (University of California at Los Angeles) 1989

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science
in the
GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Carlo H. Séquin , Chair
Professor Lawrence Rowe
Professor Jean Pierre Protzen

1993

The dissertation of Thomas Allen Funkhouser is approved:

Chair Date

Date

Date

University of California at Berkeley

1993

Database and Display Algorithms for

Interactive Visualization of Architectural Models

Copyright ©1993

by

Thomas Allen Funkhouser

Abstract

Database and Display Algorithms for Interactive Visualization of Architectural Models

by

Thomas Allen Funkhouser

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Carlo H. Séquin , Chair

This thesis describes a system for interactive walkthroughs of large, fully furnished architectural models.

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 fit into memory simultaneously. In order to achieve interactive walkthroughs of such large building models, a system must store in memory and render only a small portion of the model in each frame; that is, the portion seen by the observer. As the observer "walks" through the model, some parts of the model become visible and others become invisible; some objects appear larger and others appear smaller. The challenge is to identify the relevant portions of the model, swap them into memory and render them at interactive frame rates as the observer's viewpoint is moved under user control.

We have developed data structures and algorithms for identifying a small portion of a large model to render and store in memory 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. The database also contains an index of spatial cells with precomputed cell-to-cell and cell-to-object visibility information.

As the observer moves through the model during an interactive walkthrough, visibility information for the cell containing the observer is fetched from the display database and used by dynamic culling algorithms to identify a small superset of objects potentially visible to the observer. An optimization algorithm is used to select 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. Meanwhile, memory management algorithms are used to predict

observer motion and to pre-fetch objects from disk that may become visible during upcoming future frames.

We have implemented an interactive building walkthrough system using these data structures and algorithms. During tests, this system is able to maintain over ten frames per second with little noticeable detail elision during interactive walkthroughs of a building model containing over one million polygons.

Carlo H. Séquin
Thesis Chair

To Dad, whose love inspired this work.

Contents

Table of Contents **iv**

List of Figures **vi**

List of Tables **xi**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Goals 2
- 1.3 Problem Statement 7
- 1.4 Basic Approach 7
- 1.5 Organization 8

2 Previous Work **10**
- 2.1 Vehicle Simulators 10
- 2.2 Computer-Aided Design Systems 11
- 2.3 Building Walkthrough Systems 12

3 System Organization **14**
- 3.1 Modeling Phase 15
  - 3.1.1 Model Loading 15
  - 3.1.2 Model Representation 16
  - 3.1.3 Object Abstraction 18
  - 3.1.4 Results 21
  - 3.1.5 Discussion 23
- 3.2 Precomputation Phase 24
  - 3.2.1 Spatial Subdivision 24
  - 3.2.2 Visibility Precomputation 25
  - 3.2.3 Results 30
  - 3.2.4 Discussion 31
- 3.3 Walkthrough Phase 31
  - 3.3.1 User Interface 33
  - 3.3.2 Display Management 39
  - 3.3.3 Memory Management 39

4 Display Management **41**

4.1 Visibility Determination . . . 41
4.1.1 Observer Viewpoint . . . 42
4.1.2 Observer Visibility . . . 42
4.1.3 Results . . . 44
4.1.4 Discussion . . . 53
4.2 Detail Elision . . . 55
4.2.1 Levels of Detail . . . 56
4.2.2 Adaptive Detail Elision . . . 57
4.2.3 Optimization Detail Elision . . . 58
4.2.4 Results . . . 70
4.2.5 Discussion . . . 76

5 Memory Management **82**

5.1 Database System . . . 82
5.1.1 Segments . . . 85
5.1.2 Data Types . . . 86
5.1.3 Operations . . . 89
5.1.4 Implementation . . . 91
5.2 Predictive Memory Management . . . 93
5.2.1 Observer Range . . . 94
5.2.2 Object Lookahead . . . 94
5.2.3 Cache Management . . . 99
5.2.4 Fault Tolerance . . . 100
5.2.5 Results . . . 101
5.2.6 Discussion . . . 117

6 Concurrent Processing **119**

6.1 Pipelining . . . 120
6.2 Results . . . 121
6.3 Discussion . . . 129

7 Results **132**

7.1 Display Management . . . 133
7.2 Memory Management . . . 138

8 Conclusion **142**

8.1 Summary and Reflections . . . 142
8.2 Comparison to Other Approaches . . . 143
8.3 Final Observation . . . 144

Bibliography **146**

List of Figures

1.1 Exterior view of Soda Hall. 3
1.2 Exterior view of Soda Hall "cut open" by a horizontal plane at the sixth floor. 3
1.3 Typical office with furniture and textures. 3
1.4 Board room with furniture and textures. 3
1.5 Radiosity rendering of hallway. 5
1.6 Radiosity rendering from different viewpoint. 5
1.7 Radiosity computation is independent of observer viewpoint. 5
1.8 Polygonal mesh generated during radiosity computation. 5
1.9 A typical sequence of operations performed during rendering of three dimensional polygons. 7

3.1 System overview. 14
3.2 Loading operations of the modeling phase. 15
3.3 Three LODs for a chair. 16
3.4 Object instances can share geometries stored in an object definition. 17
3.5 An object instance can store its own geometries. 17
3.6 Different representations for a doorhandle constructed using a parameterized generator program. 18
3.7 Three LODs for a canary. 19
3.8 Deleting polygons. 19
3.9 Deleting edges. 20
3.10 Deleting the vertices with hexagonal and pentagonal symmetry. 20
3.11 Collapsing the pentagons. 20
3.12 Collapsing the edges shared by hexagons. 21
3.13 Three LODs for a desk lamp. 22
3.14 A hierarchical object representation with multiple LODs. 23
3.15 Functional steps of the precomputation phase. 24
3.16 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. 26
3.17 A sight-line stabbing a portal sequence. 26

3.18 Bowtie-shaped region containing sightlines stabbing a portal sequence (shown in stipple gray). Opaque boundaries are shown in solid black. Extremal sightlines are shown as dashed lines. 27
3.19 Cell visibility (shown in stipple gray) for source cell (shown in dark gray). . 27
3.20 Two dimensional schematic diagram of a) cell-to-cell visibility (shown in stipple gray), and b) cell-to-object visibility (shown as solid black squares) for a particular source cell (shown in dark gray). 28
3.21 Cell visibility in three dimensions. Visible region (beams) and cell-to-cell visibility (outlined) are shown for one source cell. 29
3.22 Stab list data structure. 29
3.23 Organization of data in the display database. 32
3.24 Functional operations of the walkthrough phase. 33
3.25 Panels used for controlling parameters for a) observer navigation, b) visibility determination, and c) memory management. 35
3.26 Panel used for controlling detail elision parameters. 36
3.27 Visualization program supports two views. 37
3.28 Panel containing controls for rendering parameters. 38

4.1 View frustum variables. 42
4.2 Mapping view frustum to perspective projection. 42
4.3 Visible region is a wedge that typically narrows as it traverses through more portals. 43
4.4 Visible region for view frustum in two dimensions. 43
4.5 Cells in the eye-to-cell visibility are shown in stipple gray. 44
4.6 Objects in the eye-to-object visibility are shown as by solid squares. 44
4.7 All objects incident upon cells in the eye-to-cell visibility. 45
4.8 Objects in the eye-to-object visibility. 45
4.9 Test path through the sixth floor of Soda Hall. 46
4.10 Frame time for each observer viewpoint along test walkthrough path. Note different scales along the "Frame Time" axis. 50
4.11 Compute time, draw time, and frame time for each observer viewpoint along the test walkthrough path using (CTO, ETC, ETO) visibility determination. 52
4.12 Two possible spatial subdivisions for the same model. 54
4.13 Trade-offs in spatial subdivision granularity. 54
4.14 Two-stage model of the rendering pipeline. 60
4.15 Cost model coefficients can be determined empirically. The plots show actual flat-shaded rendering times for the chair shown in Figure 3.3 using different numbers of polygons (on the left), and different sizes (on the right). 62
4.16 Comparison of actual and estimated rendering times of frames during an interactive building walkthrough. 62
4.17 Objects that appear larger "contribute" more to the image. 63
4.18 Plots of pixel intensity across a sample scan-line of images generated using different representations and rendering algorithms for a simple cylinder. . . 64

4.19 Images depicting the relative benefit of objects with the *Focus* factor of the *Benefit* heuristic set to a) 1.0 and b) 0.1. Darker shades of gray represent higher values for the *Benefit* heuristic. 67
4.20 Pseudocode for the constrained optimization algorithm. 69
4.21 Test observer path through an auditorium on the third floor of Soda Hall. . 71
4.22 Plots of frame time for every observer viewpoint along test observer path using a) no detail elision, b) static algorithm, c) feedback algorithm, and d) optimization algorithm. Note: the "Frame Time" axis in plot (a) is five-times larger than the others. 72
4.23 Images depicting the LODs selected for each object at the observer viewpoints marked 'A' using the *Static* and *Optimization* algorithms. Darker shades of gray represent higher LODs. 74
4.24 Images depicting the LODs selected for each object at the observer viewpoints marked 'C' using the *Feedback* and *Optimization* algorithms. Darker shades of gray represent higher LODs. 75
4.25 Images for observer viewpoint 'A' generated using a) no detail elision (72,570 polygons), and b) the *Optimization* algorithm with a 0.10 second target frame time (5,300 polygons). 76
4.26 Image of library generated using no detail elision (19,821 polygons). 77
4.27 Images of library generated using the *Optimization* detail elision algorithm with a target frame time of 0.10 seconds (8,882 polygons). LODs chosen for objects in (a) are shown in (b) – darker shades of gray represent higher LODs. Pixel-by-pixel differences from Figure 4.26 are shown in (c) – brighter colors represent greater difference. 78
4.28 Images of library generated using the *Optimization* detail elision algorithm with a target frame time of 0.05 seconds (3,568 polygons). LODs chosen for objects in (a) are shown in (b) – darker shades of gray represent higher LODs. Pixel-by-pixel differences from Figure 4.26 are shown in (c) – brighter colors represent greater difference. 79
4.29 Images of library generated using the *Optimization* detail elision algorithm with a target frame time of 0.02 seconds (1,161 polygons). LODs chosen for objects in (a) are shown in (b) – all objects are rendered using the lowest LOD. Pixel-by-pixel differences from Figure 4.26 are shown in (c) – brighter colors represent greater difference. 80

5.1 References to addresses internal to other segments are not allowed. 87
5.2 The building walkthrough data structures partitioned into segments. Segment boundaries are represented by thick, dashed lines. Inter-segment references are depicted by stipple gray squares labeled by the type of segment referenced (e.g., 'C' = Cell, 'O' = Object, 'G' = Geometry, 'M' = Material, 'T' = Texture, 'I' = Image, and 'L' = List). Intra-segment references are depicted by hollow squares. 88
5.3 Layout of database file. Mandatory inter-segment references are shown as solid arrows, while possible application-defined inter-segment references are shown as dashed-arrows. 91

5.4 The observer range contains all observer view positions (inside sphere) and view directions (inside range frustum) possible during the upcoming N frames. 95
5.5 Observer range is reduced if the user interface prevents traversal through solid walls. 95
5.6 The observer range cells (shown in 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. 97
5.7 The lookahead cells (shown in 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. . . . 97
5.8 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. 98
5.9 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). 100
5.10 Test path through the fifth floor of Soda Hall. 103
5.11 Size of the lookahead set (in MB) for various lookahead depths. 106
5.12 Read times and frame times using cell granularity looking 16 frames in advance. 109
5.13 Number of LODs skipped using cell granularity and object granularity looking 1 frame in advance. Frames in which the observer crosses a cell boundary are marked by a dash on the "Frames" axis. 110
5.14 Visibility from a box bounding the observer range. 118

6.1 Pipeline stage implementation. 120
6.2 Test path through the seventh floor of Soda Hall. 122
6.3 Frame time during each frame along a portion of the test walkthrough path for representative 1-, 2-, 3-, and 4-stage pipelines. 125
6.4 Response time during each frame along a portion of the test walkthrough path for representative 1-, 2-, 3-, and 4-stage pipelines. 125
6.5 Throughput is determined by the slowest stage in a concurrent pipeline. . . 126
6.6 Time for the *Visibility Determination* operation is larger when other operations are executing on other processors. 126
6.7 Frame time during each frame of test walkthrough path for various queue depths in *Rendering* stage. 130
6.8 Response time during each frame of test walkthrough path for various queue depths in *Rendering* stage. 130

7.1 Test observer path through the top four floors of Soda Hall. 133
7.2 Frame time for each observer viewpoint along the entire test observer path during the detail elision test. 134

7.3 Rendering times using cell-to-object, eye-to-object, and detail elision cull methods for each observer viewpoint along complex portion of test observer path. 137
7.4 Frame time, visibility time, detail elision time, and draw time using detail elision cull method for each observer viewpoint along complex portion of test observer path. 137
7.5 Frame time for each observer viewpoint along the entire test observer path during the test with memory management. 140

List of Tables

3.1 Multi-resolution modeling statistics for Soda Hall. 22
3.2 Mean and maximum statistics for cells in the spatial subdivision of Soda Hall. 31
3.3 Statistics gathered during each frame of an interactive building walkthrough. 40

4.1 Mean and maximum set statistics collected during tests with various combinations of precomputation and real-time visibility determination algorithms. 48
4.2 Mean and maximum timing statistics collected during tests with various combinations of precomputation and real-time visibility determination algorithms. 49
4.3 Minimum, mean, and maximum timing statistics collected during tests with various detail elision algorithms (in seconds). 71

5.1 Mean and maximum set statistics collected during tests with various lookahead depths. 105
5.2 Mean and maximum timing statistics collected during tests with various lookahead depths. 105
5.3 Mean and maximum set statistics collected during tests with various lookahead granularities. 108
5.4 Mean and maximum timing statistics collected during tests with various lookahead granularities. 108
5.5 Mean and maximum numbers of range cells during tests with various combinations of range and lookahead cull methods. 111
5.6 Mean and maximum numbers of lookahead objects during tests with various combinations of range and lookahead cull methods. 112
5.7 Mean and maximum numbers of LODs skipped during tests with various combinations of range and lookahead cull methods. 113
5.8 Mean and maximum compute times during tests with various combinations of range and lookahead cull methods. All times are in seconds. 114
5.9 Mean and maximum read times during tests with various combinations of range and lookahead cull methods. All times are in seconds. 115
5.10 Mean and maximum frame times during tests with various combinations of range and lookahead cull methods. All times are in seconds. 116

6.1 Mean and maximum compute time statistics collected during tests with various pipeline stage partitions. 124

6.2 Mean and maximum frame time and response time statistics collected during tests with various pipeline stage partitions. 124
6.3 Mean and maximum compute time statistics collected during tests with various *Rendering* stage queue depths. 128
6.4 Mean and maximum frame time and response time statistics collected during tests with various *Rendering* stage queue depths. 128

7.1 Mean and maximum frame time and response time statistics collected during display management tests. 135
7.2 Mean and maximum compute time and rendering time statistics collected during display management tests. 135
7.3 Mean and maximum numbers of cells, objects, and polygons rendered during display management tests. 135
7.4 Mean and maximum frame time, response time, and numbers of LODs skipped statistics collected during tests with and without memory management. . . 139
7.5 Mean and maximum visibility determination, detail elision, rendering, lookahead, and read time statistics collected during tests with and without memory management. All times are in seconds. 139
7.6 Mean and maximum set statistics collected during memory management tests.139

Acknowledgements

I have been incredibly lucky to have Professor Carlo Séquin as my thesis advisor. His warmth, enthusiasm, and clever insights have been inspirational. I will never forget Carlo, and I hope that his influence never wears off.

Several other professors have contributed to this thesis. Randy Katz, Dave Patterson, and the members of the RAID group got me started thinking about visualization of very large databases. Larry Rowe provided helpful hints about object oriented databases and user interfaces. Professor Jean Pierre Protzen provided insights on building walkthroughs from an architect's perspective.

This thesis would not exist in its current form without the help of the many other graduate students who have been part of the UC Berkeley Building Walkthrough Project. Special thanks are due to Seth Teller who provided code for spatial subdivisions, visibility algorithms, and many other support libraries. Delnaz Khorramabadi created the original three dimensional model of Soda Hall. Thurman Brown, Laura Downs, Dan Rice, Priscilla Shih, Maryann Simmons, Ajay Sreekanth, and Greg Ward generated models of furniture and plants for the building interior. Graham Beasley, Mark Halstead, Henry Moreton, Seth Teller, and Greg Ward provided images for textures to make the model look more realistic.

It's been very pleasant to be a graduate student at UC Berkeley. Seth Teller, Henry Moreton, Mike Hohmeyer, Paul Heckbert, Ziv Gigus, and Nina Amenta provided early lessons in computer graphics and a friendly academic environment. Terry Lessard-Smith, Bob Miller, Liza Gabato, and Kathryn Crabtree were always there when I needed help wading through the UC Berkeley bureaucracy.

Silicon Graphics, Inc. has been very generous, allowing me 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.

Finally, I'd like to thank Marty and my family for having faith in me.

Chapter 1

Introduction

1.1 Motivation

Interactive computer graphics systems for visualization of realistic-looking, three dimensional models are useful for evaluation, design and training in virtual environments, such as those found in architectural and mechanical CAD, vehicle simulation, and virtual reality. These systems simulate the visual experience of moving through a three dimensional model 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 a virtual environment can be achieved.

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

1.2 Goals

Model Size

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. If the visualization system supports interactive manipulation of such models, an architect or interior designer could try out different furniture arrangements and color schemes, or test whether a particular room is large enough to hold furniture for a particular number of people. Also, a client could experiment with different types of furniture to see which type fits best into a proposed architectural design.

A primary goal of our work is to support computer-aided visualization of very large, detailed three dimensional models. Our test case is a model of Soda Hall, the future computer science building at the University of California, Berkeley. Soda Hall is a seven floor academic building containing more than one hundred faculty and student offices, twelve computer laboratories, and six class rooms (see Figure 1.1). Almost every room in the model contains a functionally complete set of furniture (see Figure 1.2). 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 (see Figures 1.3 and 1.4). 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."

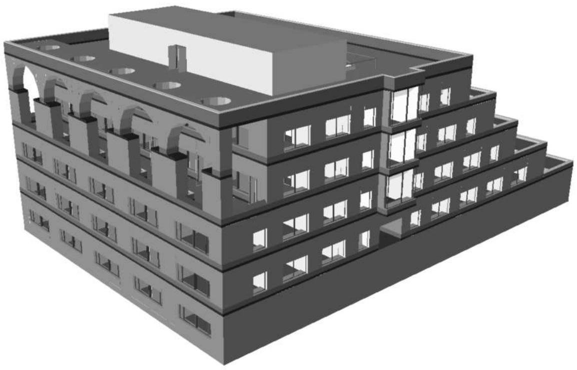


Figure 1.1: Exterior view of Soda Hall.

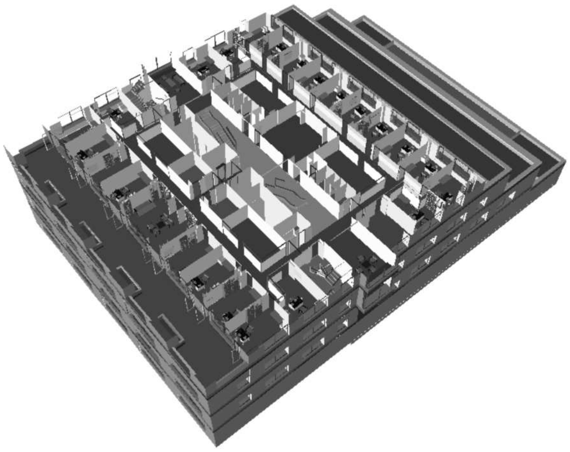


Figure 1.2: Exterior view of Soda Hall “cut open” by a horizontal plane at the sixth floor.

Figure 1.3: Typical office with furniture and textures.

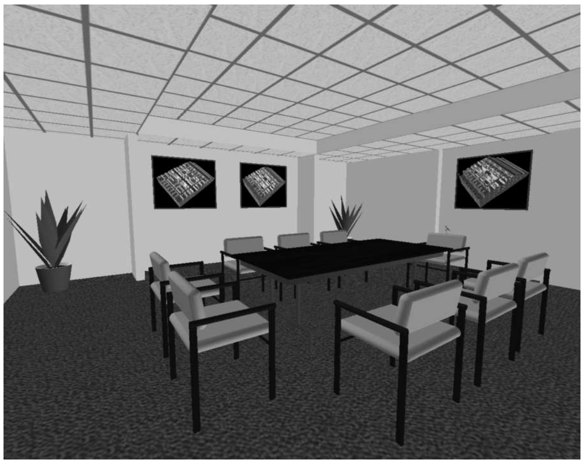


Figure 1.4: Board room with furniture and textures.

Image Quality

If a visualization system includes rendering algorithms based on realistic illumination simulations, an architect can evaluate a building's lighting characteristics before it has been constructed. For instance, an architect might experiment with different wall, window or lamp placements while viewing images containing appropriate reflections and shadows; or an interior designer might try different materials or color schemes, evaluating them at different times of day, or during different seasons.

A second goal of our work is to use *radiosity* illumination simulation methods to generate realistic-looking images with indirect diffuse reflections and shadows (See Figures 1.5 and 1.6). Radiosity methods, based on models of radiation emission and reflection in thermal-engineering, 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 equations which are solved to obtain the radiance for each polygon. See [7, 15, 28, 29, 30, 40] for more information.

The advantage of radiosity methods for interactive visualization is that a global radiosity solution can be precomputed, i.e. the solution does not depend on a particular observer viewpoint (see Figure 1.7). 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 is required to describe a 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 (along gradients of radiosity) in order to capture complex illumination effects along the boundaries of shadows and highlights (see Figure 1.8). Furthermore, although many of the polygons in the original model can be shared via hierarchical instances, each polygon is illuminated and meshed 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 may contain tens of millions of separate polygons and may require gigabytes of data after a radiosity computation.

Figure 1.5: Radiosity rendering of hallway.

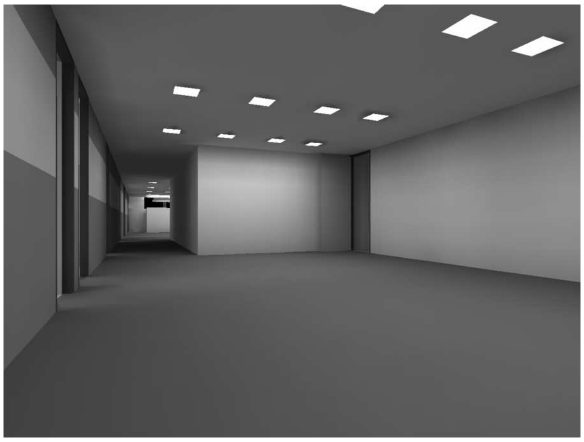


Figure 1.6: Radiosity rendering from different viewpoint.

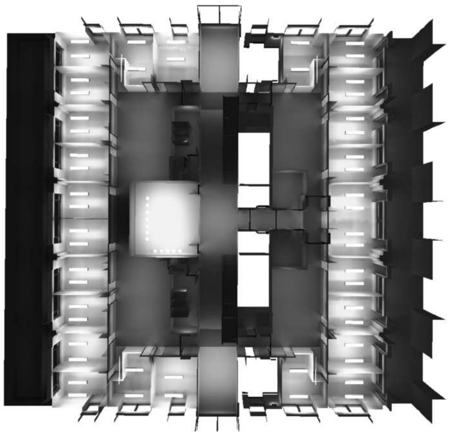


Figure 1.7: Radiosity computation is independent of observer viewpoint.

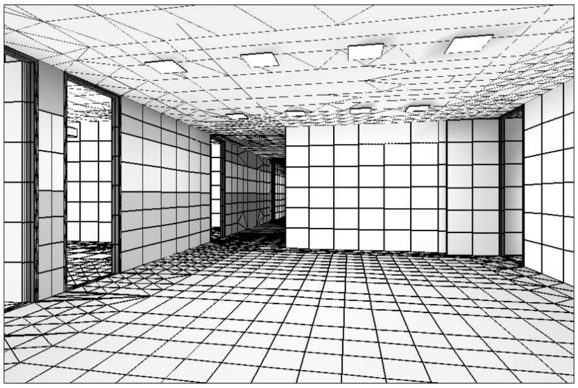


Figure 1.8: Polygonal mesh generated during radiosity computation.

Performance

Another extremely important 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. 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.

Hardware

Finally, we aim to support interactive visualization of large, detailed building models using commonly 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. A typical sequence of rendering operations is shown in Figure 1.9 [22] (assuming radiosity, Gouraud shading, and z-buffer hidden surface removal). This thesis addresses only the first two operations of this sequence: database traversal and trivial accept/reject. We assume that the other five operations are performed by the graphics subsystem, so they are not discussed in this thesis.

Thus far, we have been using a Silicon Graphics 320 VGX with 64MB of memory which

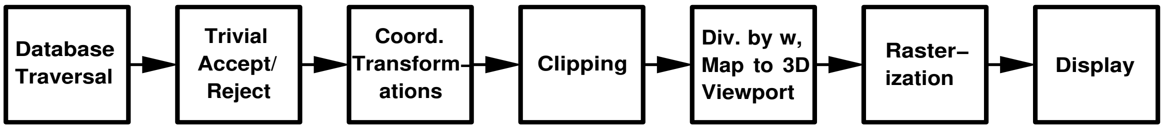


Figure 1.9: A typical sequence of operations performed during rendering of three dimensional polygons.

can draw approximately 50-100K Gouraud shaded polygons per second. In the near future, we hope to use a Silicon Graphics Reality Engine which can render polygons at similar rates, but also with texture mapping and antialiasing [1]. These computer systems currently cost between \$50K and \$100K.

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, load it into memory, and render images at interactive frame rates as the observer viewpoint is moved under user control.

1.4 Basic Approach

We have developed a system that supports interactive walkthroughs of large, fully furnished building models. 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.

1.5 Organization

The thesis is organized as follows. Chapter 2 contains a review of previous work in interactive visualization algorithms. Special emphasis is paid to research on display of very large three dimensional databases.

Chapter 3 contains a description of the organization of our building walkthrough system. It describes the functional components of the system, and the information stored in the display database which links these components.

Chapter 4 contains a description of the real-time display algorithms. It describes the dynamic visibility algorithm used to compute a set of objects seen by a simulated observer. For each observer viewpoint, the algorithm traces visibility beams through the transparent portions of the boundaries of a precomputed spatial subdivision to determine which volume of space is visible. Bounding boxes of objects are checked for intersection with visibility beams to construct a potentially visible set of objects to display. An adaptive detail elision algorithm is described in which a level of detail and rendering algorithm is chosen for the display of each potentially visible object in order to achieve a user-specified target frame time. The algorithm uses a constrained optimization to generate the best possible image within the user-specified target frame time.

Chapter 5 contains a description of the real-time memory management system. We have implemented a database system that supports efficient access to large, dynamic, persistent data structures via asynchronous application-defined functions. We use this database system, along with lookahead algorithms that predict future observer viewpoints, to determine which objects are likely to be rendered during upcoming frames, and to prefetch relevant portions of the model into memory before they are rendered.

Chapter 6 contains a description of the architecture used for concurrent processing in our building walkthrough system. We use a coarse-grained concurrent pipeline in which separate stages execute asynchronously in separate processes on a shared memory multi-processor workstation.

Chapter 7 contains overall results for interactive walkthroughs using the display and memory management algorithms described in this thesis. We find that interactive visualization of a model containing over 1.4 million polygons is possible on currently available hardware.

Chapter 8 contains a summary and conclusion.

Chapter 2

Previous Work

2.1 Vehicle Simulators

Most work in interactive visualization has been done on vehicle simulators. Numerous, sophisticated commercial vehicle simulators have been built over the last thirty years, including many which contain algorithms for visibility-based culling, detail elision, and real-time management of very large databases [20, 47, 48, 60]. However, since most are commercial systems, very little has been published on this vast quantity of 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 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 sixty 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. In contrast to commercial vehicle simulators, we aim to support only near-uniform, bounded frame rates, however for use with standard, off-the-shelf hardware, and allowing visualization of arbitrarily complex models.

2.2 Computer-Aided Design Systems

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

There are several differences between most CAD applications and building walkthrough systems. First, visual realism is generally less important in mechanical and medical 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, lighting and coloring 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 and medical CAD systems generally simulate an observer looking at the model "through a window" from the outside. The *Environment in Hand* metaphor [58] is used to support observer "navigation" by means of translation, scaling and rotation. In contrast, building walkthrough systems simulate an observer moving through the interior of the model. These different metaphors for observer navigation may imply different approaches to observer viewpoint prediction, visibility determination, and detail elision.

Finally, mechanical engineering CAD systems are typically used during design as well as visualization and verification. In such systems, the user can modify the model during interactive visualization. For instance, the designer of a mechanical assembly may experiment with various placements for a particular part. In contrast, current applications of building walkthroughs are aimed at visualization and verification of relatively static models. Therefore, it may be possible to precompute visualization results in these systems.

2.3 Building Walkthrough 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) [8, 56]. 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 [34]. Pioneering work in spatial subdivision and visibility precomputation has been done at the University of North Carolina at Chapel Hill [2, 3, 11]. 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 that can be seen from any observer viewpoint within the cell. The disadvantage of this approach is that computation is stocastic, and thus can under-estimate true cell visibility and requires a large amount of computation.

Improved spatial subdivision and visibility precomputation techniques for building walkthroughs were developed by Teller [53, 55] at UC Berkeley. His visibility precomputation algorithm is deterministic, never under-estimates true cell visibility, and runs efficiently. Teller also developed real-time visibility determination algorithms that compute a set of cells potentially visible from a particular observer viewpoint. His spatial subdivision, visibility precomputation, and real-time visibility determination algorithms are used in conjunction with object-based visibility algorithms we developed jointly with Teller in our building walkthrough system. Summaries of relevant data structures and algorithms appear in Chapters 3.2 and 4.1 of this thesis.

Chapter 3

System Organization

Our building walkthrough system is divided into three distinct phases as shown in Figure 3.1. 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 observer-independent lighting and visibility calculations. Finally, during the *walkthrough phase*, we simulate an observer moving through the building model under user control with the mouse, rendering the model as seen from the observer 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.

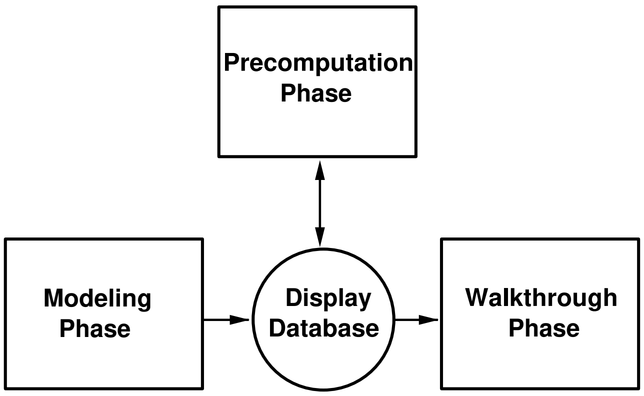


Figure 3.1: System overview.

3.1 Modeling Phase

3.1.1 Model Loading

Our interactive building walkthrough system requires a 3D polygonal model stored in a display database. Currently, we load models into the display database from UC Berkeley UNIGRAFIX format files [49]. Models described in other formats (e.g., AutoCAD DXF) are first converted to UNIGRAFIX before being loaded into our display database, as shown in Figure 3.2.

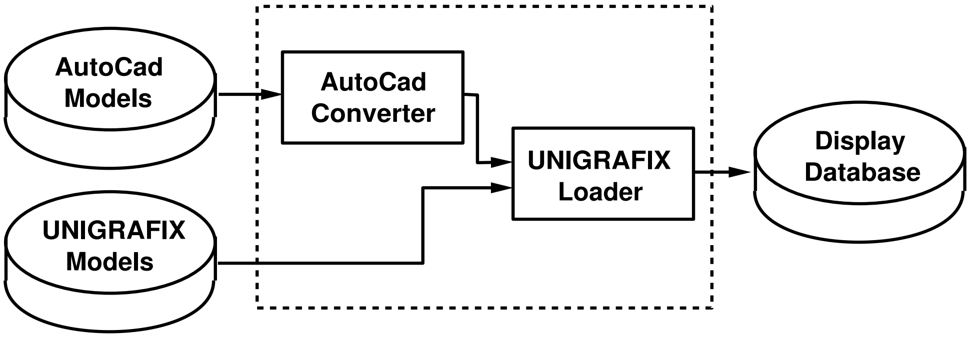


Figure 3.2: Loading operations of the modeling phase.

In the case of Soda Hall, floorplans and elevations for the major structure elements of the building (i.e., walls, ceilings, and floors) were received from the architects in AutoCAD DXF format [5]. We converted these $2\frac{1}{2}$D models into a consistent 3D representation in UC Berkeley UNIGRAFIX format. Unfortunately, the raw architectural models were not true three dimensional models – they contained nonplanar polygons, coincident coplanar polygons, improper polygon intersections, and inconsistent polygon orientations. We used automated programs to detect and correct many of these anomalies [36] and then corrected any remaining modeling errors manually using 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. Models for many types of furniture (e.g., chairs, desks, and coffee cups) were created with interactive modeling programs by Ward [57]. 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 it, a blackboard on one wall, a transparency projector on the table, and so on. The "office generator" places a desk against one wall, a chair in front of the desk, some bookshelves against the walls, and so on. Numerous parameters are available for the user to control the size, number, and placement of objects. Alternatively, we use interactive placement programs, such as AutoCAD, `ugitools` [36] (an interactive UNIGRAFIX tool), or `wkedit` [12] (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.

3.1.2 Model Representation

The walkthrough display database represents the model as a set of *objects*, each of which can be described at multiple levels of detail (LODs) [18]. For example, a chair may be described by three different representations, as shown in Figure 3.3: 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. 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 3.3: Three LODs for a chair.

In general, if there is more than one instance of the same object type (e.g., the same type of chair may appear in many positions throughout the building), they all share the same *object definition*, which stores the *geometries* (i.e., vertices, polygons, materials, and textures) that describe 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 one specified. An example hierarchy of object instances and definitions is shown in Figure 3.4. 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 object instances to store a specific 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 3.5. For chair instance #1, the lowest LOD is inherited from the object definition, whereas the medium and high LODs are specified explicitly.

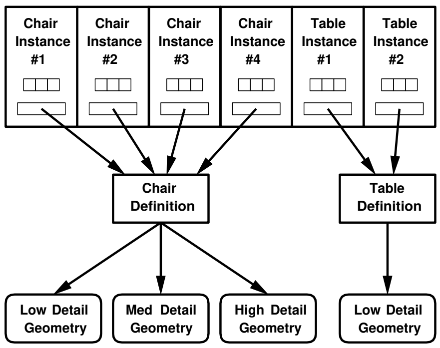


Figure 3.4: Object instances can share geometries stored in an object definition.

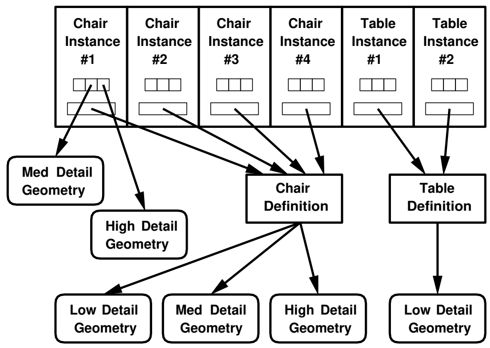


Figure 3.5: An object instance can store its own geometries.

Objects that move over time are represented by a simple extension to this hierarchy using a technique derived from `ugbump` [41]. The 4x4 transformation of any object instance can be represented by a sequence of strings representing translate, rotate and scale transformations that depend on 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.

3.1.3 Object Abstraction

We currently use the following techniques to create multiple LODs for each object definition:

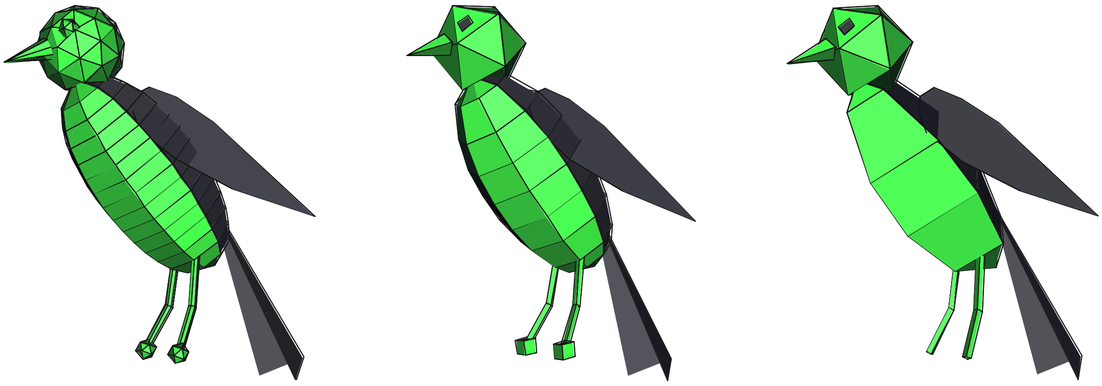
- **Procedural modeling:**
  For objects created by procedural generation programs, it is usually possible to extend the programs to produce not only a very detailed model of the object, but also simpler representations as appropriate. For instance, the program used to generate the door handle shown in Figure 3.6 is parameterized to output segments with a user-specified number of sides.

6-sided prisms. 4-sided prisms.

Figure 3.6: Different representations for a doorhandle constructed using a parameterized generator program.

- **CSG modeling:**
  For objects described as a 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. An example of this abstraction technique is shown in Figure 3.7.

- **Interactive Modeling:**
  Unfortunately, many of the objects we use in our model of Soda Hall are described



336 Polygons 134 Polygons 70 Polygons

Figure 3.7: Three LODs for a canary.

originally in a flat, polygonal format – containing no information about how they were generated, or whether there is a hierarchy of parts. We are currently working on automatic tools for object abstraction. Until now, we have constructed less detailed representations from highly detailed originals using an interactive UNIGRAFIX editor, called *animator* [23, 51], which has editing features aimed specifically at reducing the complexity of 3D polyhedral models. A few of these features are:

– **Deleting Polygons:**
  The *Delete Polygon* command removes a polygon, leaving a hole (see Figure 3.8). Vertices attached to the polygon are not deleted.

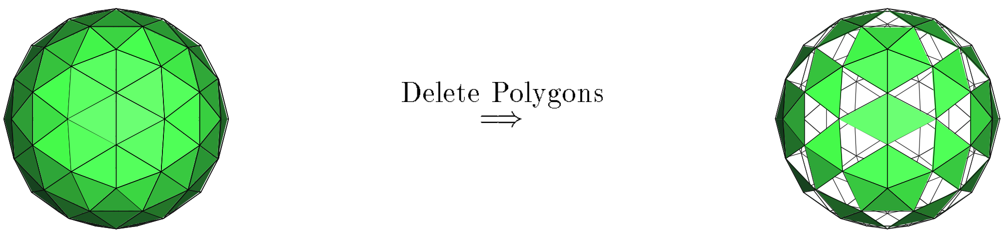


Figure 3.8: Deleting polygons.

– **Deleting Edges:**
  The *Delete Edge* command removes an edge, connecting the two polygons attached to it (see Figure 3.9). This operation may create new polygons that are not planar.

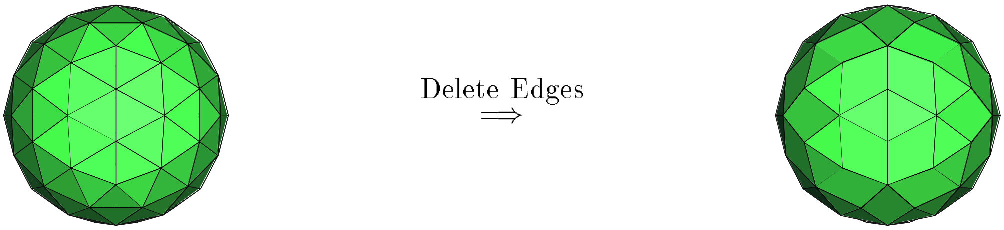


Figure 3.9: Deleting edges.

– **Deleting Vertices:**

The *Delete Vertex* command replaces a vertex, and all polygons attached to it, by a single polygon (see Figure 3.10). It deletes the vertex, and then creates a polygon (which may not be planar) containing all vertices previously connected to the vertex by an edge.

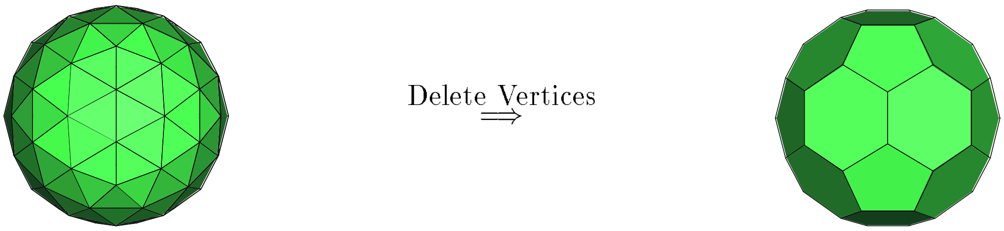


Figure 3.10: Deleting the vertices with hexagonal and pentagonal symmetry.

– **Collapsing Polygons:**

The *Collapse Polygon* command replaces a polygon by a vertex (see Figure 3.11). A new vertex is created at the centroid of the polygon, and then all vertices of the polygon are merged into this new vertex.

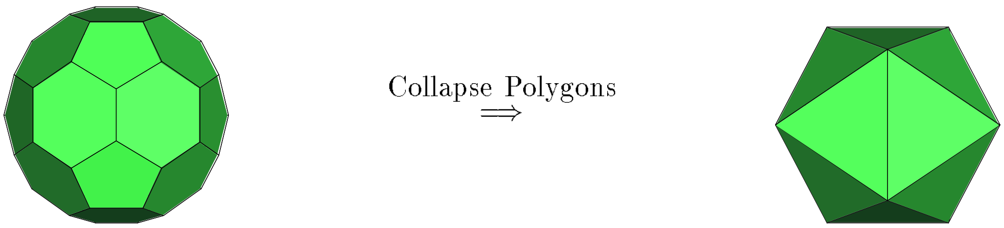


Figure 3.11: Collapsing the pentagons.

– **Collapsing Edges:**

The *Collapse Edge* command replaces an edge by a vertex (see Figure 3.12). A new vertex is created at the midpoint of the edge, and then the two vertices at the endpoints of the edge are merged into this new vertex.

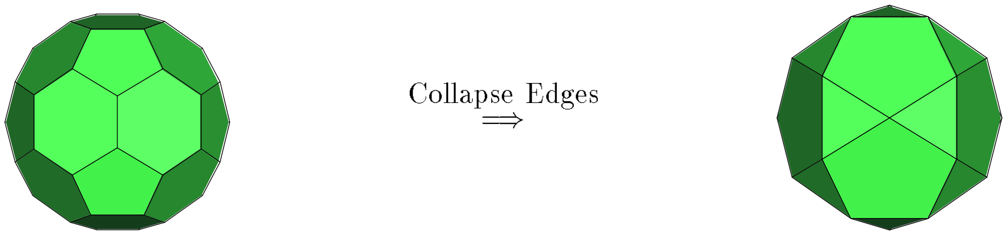


Figure 3.12: Collapsing the edges shared by hexagons.

3.1.4 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. In all, the model contains 2,217,792 polygons, of which 1,418,807 represent objects at the 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 3.1.

We have generated multiple LODs for numerous complex object definitions. An example is the desk lamp shown in Figure 3.13. 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.

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. However, we clearly have more multi-resolution modeling work to do – there is at least one object that has 2,598 polygons describing its lowest (and only) LOD (a sculpture of a Klein bottle)!

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
Lowest	14,478	240,149	1	16.59	2,598
Highest	14,478	1,418,807	1	98.00	4,211
All	14,478	2,217,792	1	153.18	4,211

Table 3.1: Multi-resolution modeling statistics for Soda Hall.

597 Polygons 98 Polygons 28 Polygons

Figure 3.13: Three LODs for a desk lamp.

3.1.5 Discussion

In order to keep our display and memory management algorithms as simple as possible, we implemented a one-level modeling hierarchy in the first version of our building walkthrough system (i.e., geometries cannot instance other geometries). In retrospect, it would have been more interesting, and possibly more efficient, to support geometries with arbitrary hierarchy. A possible data structure for storing multiple LODs in a multi-level hierarchy is shown in Figure 3.14 – each node in the hierarchy stores multiple abstract representations for the visual appearance of its children with reduced detail. For example, a bookshelf may have a very simple representation with only a few polygons or one large texture mapped polygons representing several books. This data structure would be very useful in situations in which there are several orders of magnitude difference in resolution between the finest and coarsest details of the model. Then, abstract representations for higher nodes in the tree can be rendered when the observer is far away and cannot see small details [18].

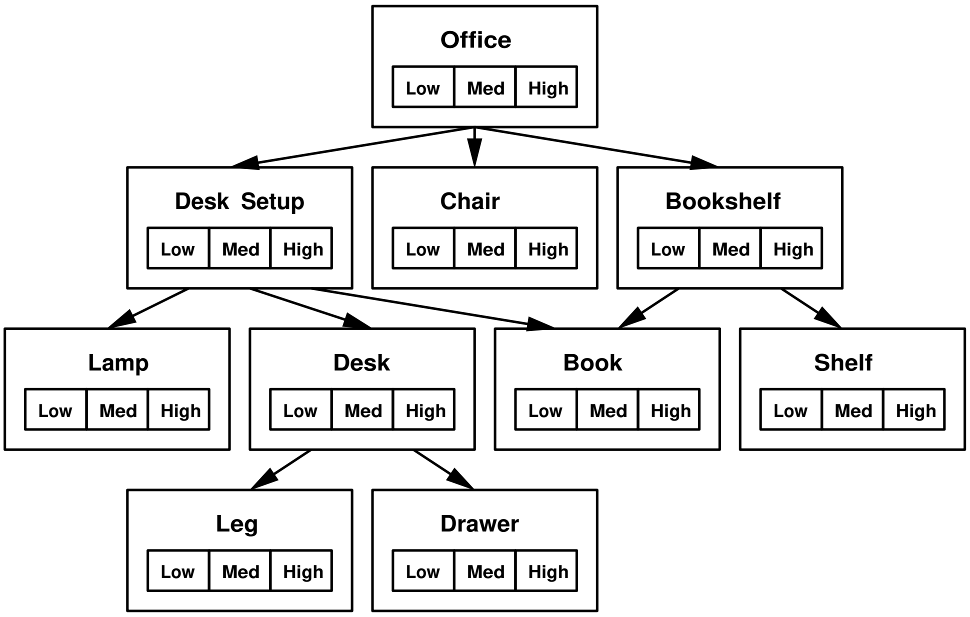


Figure 3.14: A hierarchical object representation with multiple LODs.

3.2 Precomputation Phase

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 space for time to accelerate frame rates during the walkthrough phase.

The steps of the precomputation phase are shown in Figure 3.15. 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 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. The results of the spatial subdivision and visibility precomputation are stored in the display database for use during the walkthrough phase. Viewpoint-independent calculations for ray-tracing and radiosity can also be done during the precomputation phase, but they have not yet been implemented.

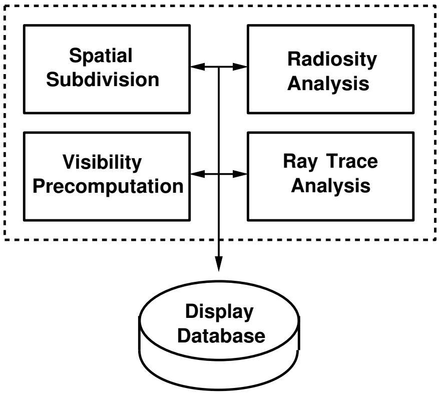


Figure 3.15: Functional steps of the precomputation phase.

3.2.1 Spatial Subdivision

We partition the model into a *spatial subdivision* using a variant of the *k-D* tree data structure [9]. Splitting planes are introduced along the major opaque elements in the model (i.e., the walls, floors, and ceilings of the building). See [55] for details.

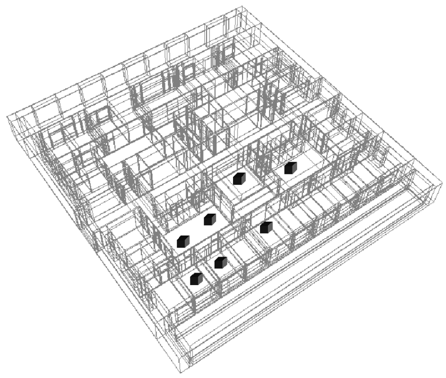
After subdivision, cell *portals* (i.e., the transparent portions of shared boundaries) are identified and stored with each leaf cell, along with an identifier for the neighboring cell to which the portal leads. Enumerating the portals in this way amounts to constructing an *adjacency graph* over the 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 cells under the node. 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.

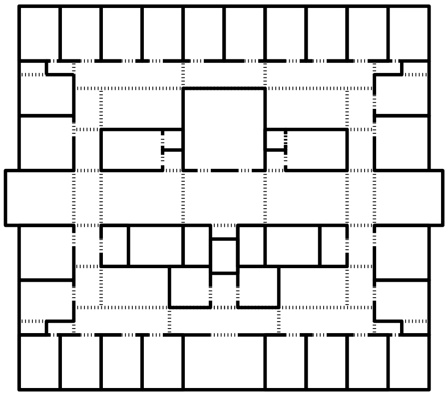
The current implementation of the building walkthrough system supports spatial subdivisions containing only axis-aligned, rectangular cell boundaries and portals (i.e., cells are axial three dimensional boxes, and portals are axial two dimensional rectangles). Such a spatial subdivision for the sixth floor of Soda Hall is shown in three dimensions in Figure 3.16a – cell boundaries are shown as gray outlines. Figure 3.16b shows a two dimensional schematic representation of the subdivison, 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.

3.2.2 Visibility Precomputation

Once the spatial subdivision has been constructed, we perform a *visibility precomputation* in which we determine the portion of the model visible from each cell of the spatial subdivision. 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.



a) Actual three dimensional subdivision.



b) Two dimensional schematic representation.

Figure 3.16: 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.

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 (see Figure 3.17). 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.

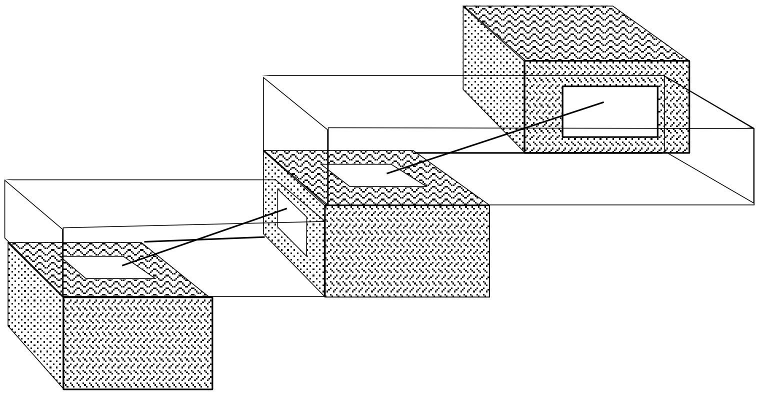


Figure 3.17: A sight-line stabbing a portal sequence.

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 all points in space 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 (see Figure 3.18). 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. In all, the visibility of the source cell is the union of the visible regions of cells reached during the search (see Figure 3.19).

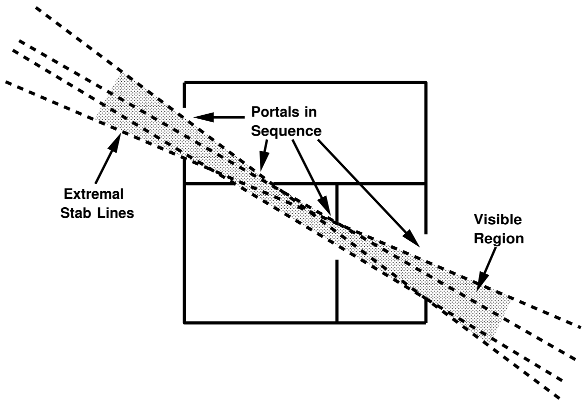


Figure 3.18: Bowtie-shaped region containing sightlines stabbing a portal sequence (shown in stipple gray). Opaque boundaries are shown in solid black. Extremal sightlines are shown as dashed lines.

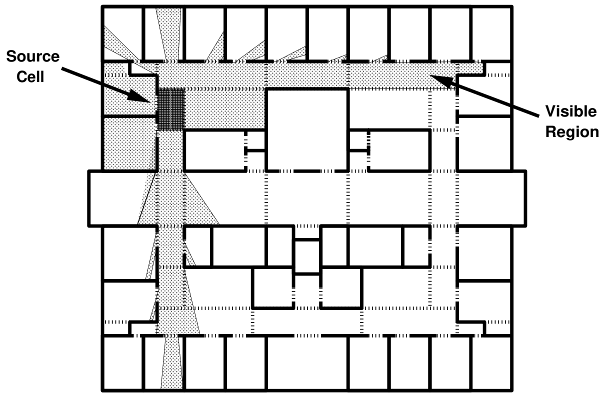
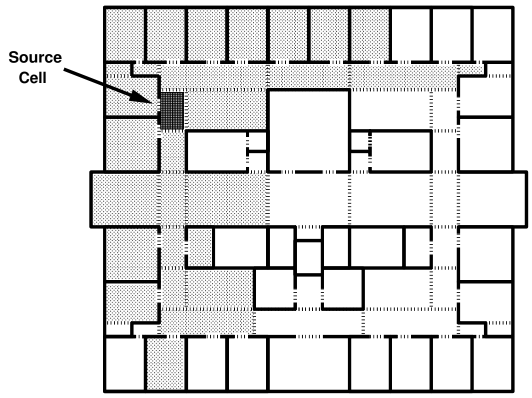


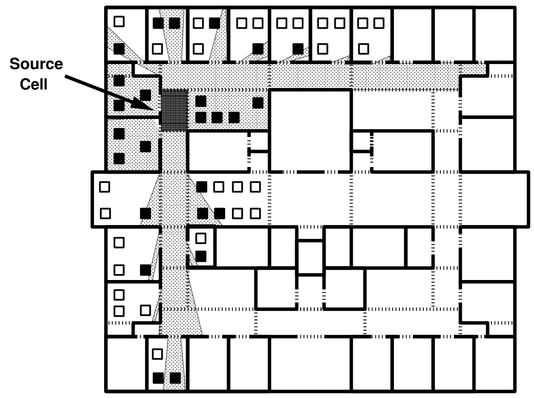
Figure 3.19: Cell visibility (shown in stipple gray) for source cell (shown in dark gray).

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 [31]. Amenta has proposed an $O(n)$ solution for this problem [4], although it has not yet been implemented.

We construct the *cell-to-cell* and *cell-to-object* visibility for the source cell during the depth-first search. We define the *cell-to-cell* visibility for cell C to be the set of cells possibly visible from C, i.e., the cells reached during the depth-first search originating from C (see Figure 3.20a). We define the *cell-to-object* visibility to be the set of objects possibly visible from cell C, i.e., the objects whose bounding boxes are incident upon some region visible from C. In Figure 3.20b, the objects found potentially visible from the source are shown as solid black squares. Figure 3.21 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 3.20: Two dimensional schematic diagram of a) cell-to-cell visibility (shown in stipple gray), and b) cell-to-object visibility (shown as solid black squares) for a particular source cell (shown in dark gray).

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 *superset* of C's cell visibility in R, constructed by assembling 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., ones 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. Figure 3.22 shows a schematic

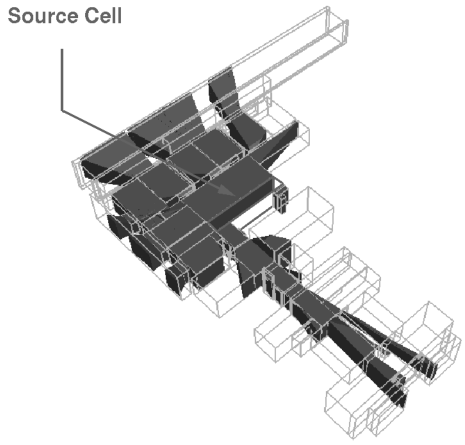


Figure 3.21: Cell visibility in three dimensions. Visible region (beams) and cell-to-cell visibility (outlined) are shown for one source cell.

diagram of the stab list data structure. 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 determine the set of objects visible to the observer.

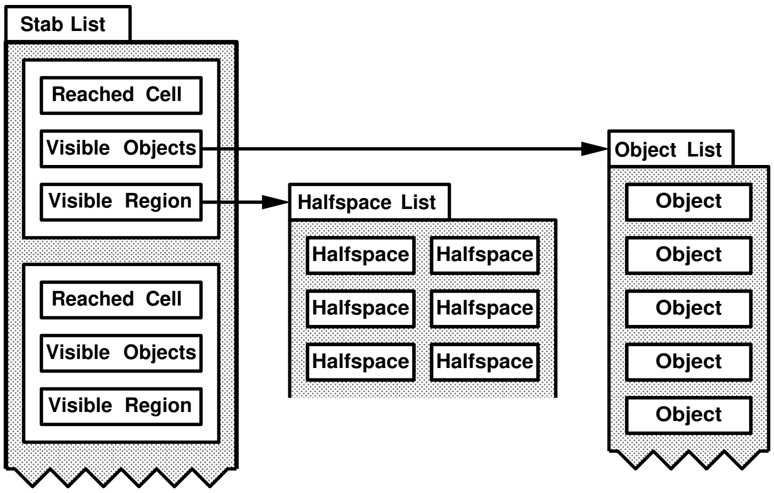


Figure 3.22: Stab list data structure.

When the bounding box of an object in the interior of a cell is moved, we must update the cell-to-object visibility of all cells from which that object was previously visible, or is newly visible. Conceptually, this update could be performed using an algorithm similar to the one used in the cell-to-cell visibility computation – we could find the set of cells visible to an object extent (rather than a cell extent) before the object has moved, and after it has moved, and then update the cell-to-object visibilities for those cells accordingly. However, such an algorithm has not yet been implemented. Instead, we store the set of half spaces that bound the visible region of the reached cell in every stab list entry. By checking for an intersection between the appropriate set of half-spaces and the extent of an object, we can quickly determine whether the object should be added or removed from a cell's cell-to-object visibility set when an object moves. As a practical optimization, objects that move regularly but only over limited distances (e.g., animated sculptures) are defined with bounding boxes that enclose the entire range of motion so that precomputed visibility information does not have to be updated during each frame. Further experimentation is required to determine the time and space trade-offs between different approaches to updating precomputed object visibility information. See [55] for a discussion of updating precomputed visibility information as opaque objects contributing to cell boundaries move (i.e., thereby affecting the opacity of cell boundaries and precomputed cell-to-cell visibility information).

3.2.3 Results

Mean and maximum precomputation statistics for cells in the spatial subdivision of Soda Hall are shown in Table 3.2. 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, and 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 and requires 9.4MB of storage for the stab lists.

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

3.2.4 Discussion

The current implementation of the building walkthrough system supports spatial subdivisions containing only axis-aligned, rectangular cell boundaries and portals. Clearly, this restriction severely limits the types of environments with which our walkthrough system can be effective – i.e., only models with rectangular, axis-aligned walls, ceilings and floors. There are many other types of interesting virtual environments whose major occluders are not axis-aligned, including cars, airplanes, boats, terrain, and many other types of architectural models.

In order to support interactive visualization of these other types of models, our system must use enhanced spatial subdivision and visibility algorithms that allow cell boundaries and portals with arbitrary three dimensional geometry. Teller has implemented such algorithms for models constructed from planar polygons with arbitrary alignment in three dimensions [54, 55]. The spatial subdivisions generated using these algorithms tend to require more storage than axial ones, and the visibility algorithms have higher computational complexity. A study of the impact of using more general spatial subdivisions and visibility algorithms on the space and performance characteristics of our building walkthrough system is a topic for further study.

3.3 Walkthrough Phase

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

The primary problem during the walkthrough phase is that building models are very large, and so 1) they cannot be rendered completely in an interactive frame time (i.e., the *display management* problem), and 2) they do not fit into memory all at once (i.e., the *memory management* problem). Thus, we must identify a small, but relevant, portion of the model to render and store in memory during each frame. We use the object hierarchy, spatial subdivision, and results of the visibility precomputation stored in the display database (summarized in Figure 3.23), along with real-time display and database management algorithms to compute this relevant portion of the model for each observer viewpoint.

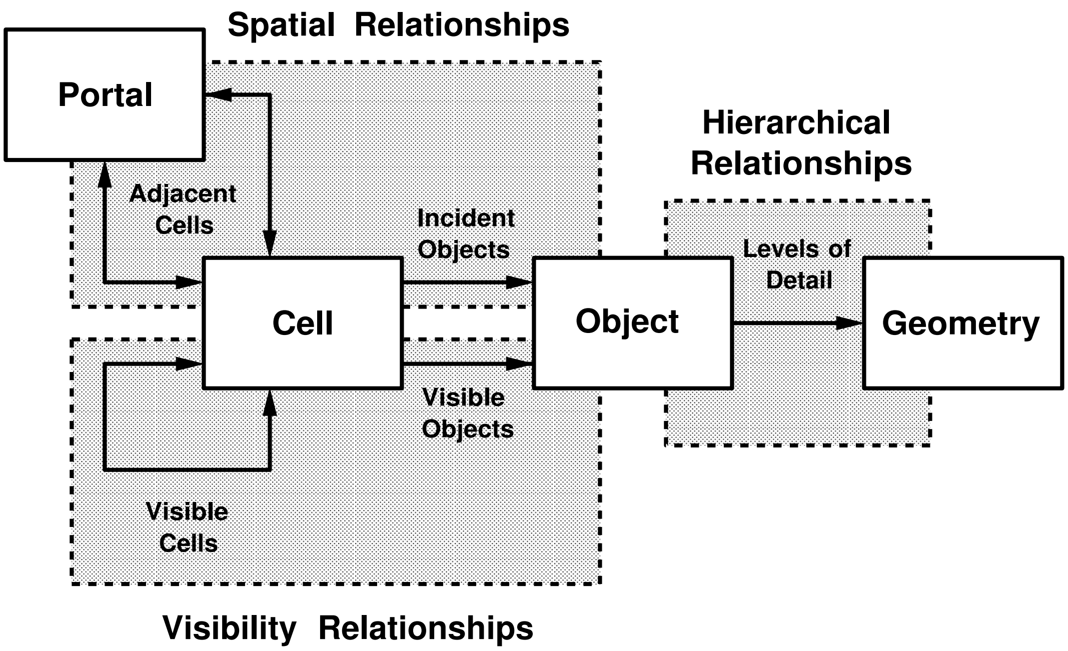


Figure 3.23: Organization of data in the display database.

Execution during the walkthrough phase proceeds as diagrammed in Figure 3.24. 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 an observer viewpoint.
- **Visibility Determination:** Compute the set of objects potentially visible from the observer viewpoint.
- **Detail Elision:** Choose a level of detail and rendering algorithm for each potentially visible object.

- **Rendering Operations:** Render potentially visible objects with a chosen level of detail and a rendering algorithm on the monitor of the workstation.
- **Lookahead Determination:** Compute the set of objects to store in 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.

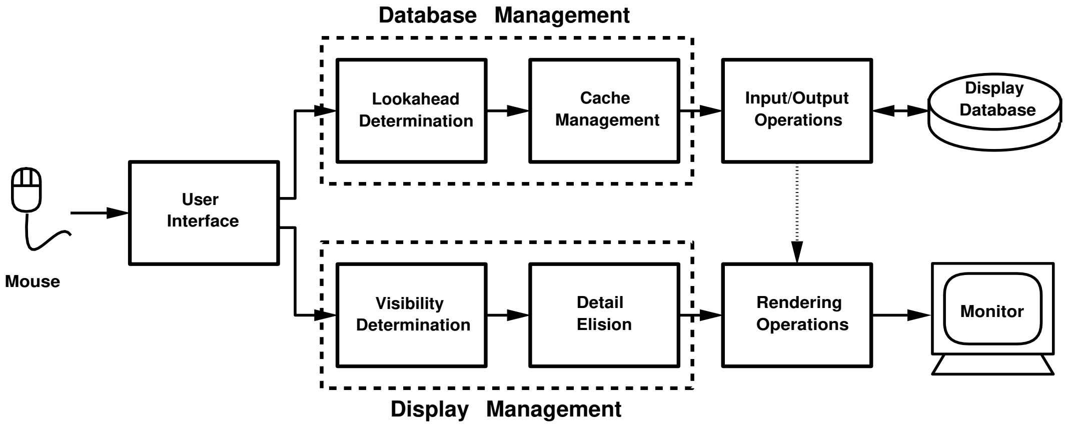


Figure 3.24: Functional operations of the walkthrough phase.

3.3.1 User Interface

During every frame of an interactive walkthrough, the system first determines the simulated observer position. The walkthrough system used for demonstrations has a simple user interface to control the simulated observer based on mouse and keyboard input. In general, the user presses one of three mouse buttons to move through the building:

Input	Action
Hold down left mouse button	Move forward while turning
Hold down right mouse button	Move backward while turning
Hold down middle mouse button	Spin without moving

While a mouse button is down, the observer moves along the current view direction, and turns to look in the direction pointed to by the mouse cursor. Therefore, 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 (or right), the user holds down a mouse button and moves the mouse cursor to the left (or right) part of the screen. In general, the observer always moves toward or away from the portion of the building pointed to by the mouse cursor.

To make navigation simple, our system allows the observer to turn only left or right, by default – i.e. turning up or down is disabled. However, if the user holds down the control key, the observer can turn up (if the mouse cursor is near the top part of the screen) or down (if the mouse cursor is near the bottom part of the screen), as well as left and right. When the user releases the control key, the program adjusts the observer back to horizontal viewing gradually.

The user can also press the arrow and keypad arrow keys to move through the model one step at a time. The four arrow keys, labeled '←', '↓', '→', and '↑', turn the observer viewpoint left, down, right, and up, respectively. The six arrow keys on the keypad, labeled '2', '4', '6', '8', '5', and '/' move the observer viewpoint down, left, right, up, forward, and back, respectively. In all, the relevant keyboard keys perform the following actions:

Input	Action
Left arrow	Spin left
Right arrow	Spin right
Up arrow	Spin up
Down arrow	Spin down
Keypad 4	Move left
Keypad 6	Move right
Keypad 8	Move up
Keypad 2	Move down
Keypad 5	Move forward
Keypad /	Move back
Control	Allow up/down turning with mouse
ESC	Exit the program

The walkthrough program also supports a variety of panels with toggles, buttons, and sliders to control various program options (see Figures 3.25–3.26). We have also included an ASCII text file interface that allows the user to specify default parameter settings at program startup. Using these controls, we can specify and experiment with many possible parameter combinations relatively easily.

In order to evaluate and understand the behavior of our walkthrough algorithms for any given combination of parameter settings, we have included code in our building walkthrough

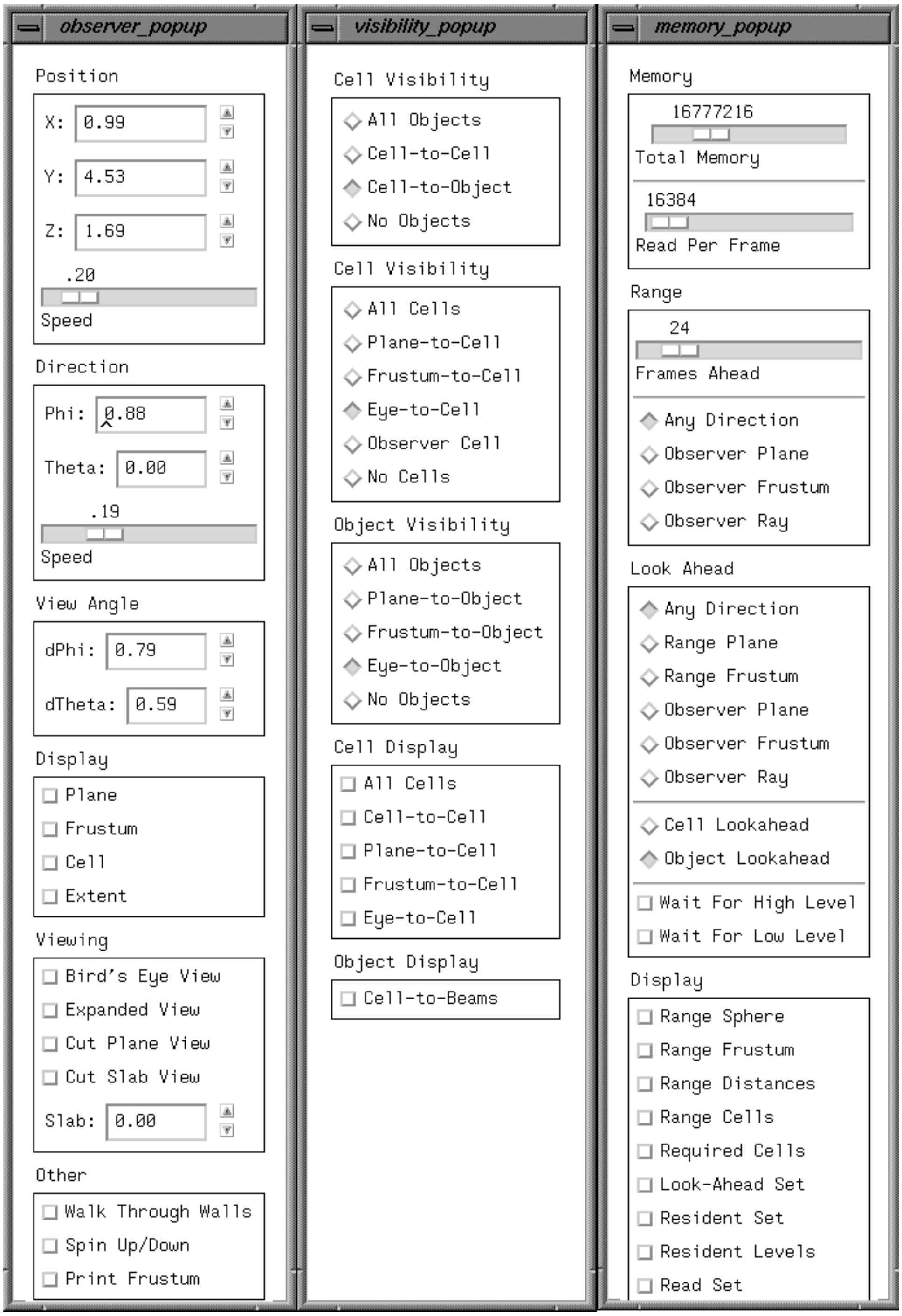


Figure 3.25: Panels used for controlling parameters for a) observer navigation, b) visibility determination, and c) memory management.

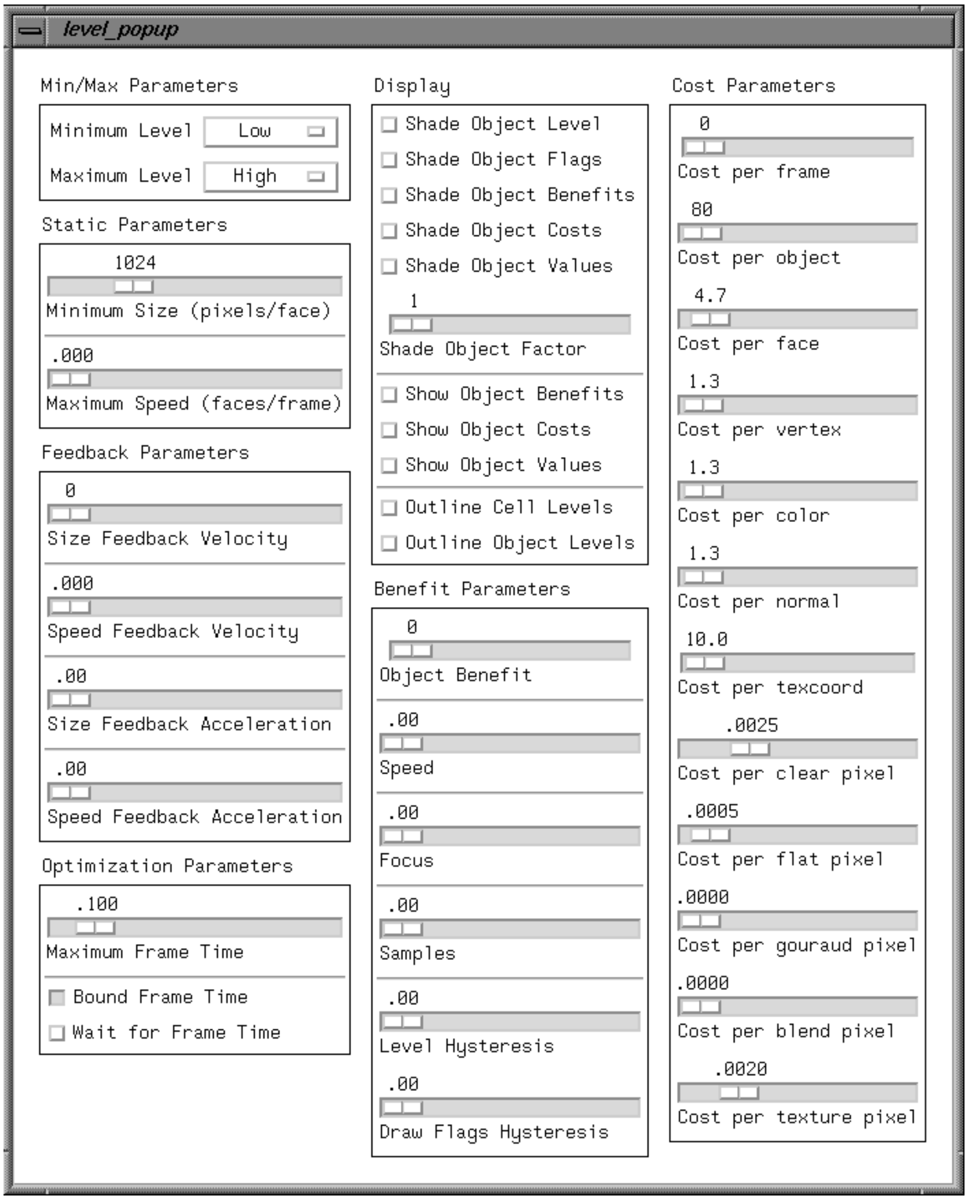
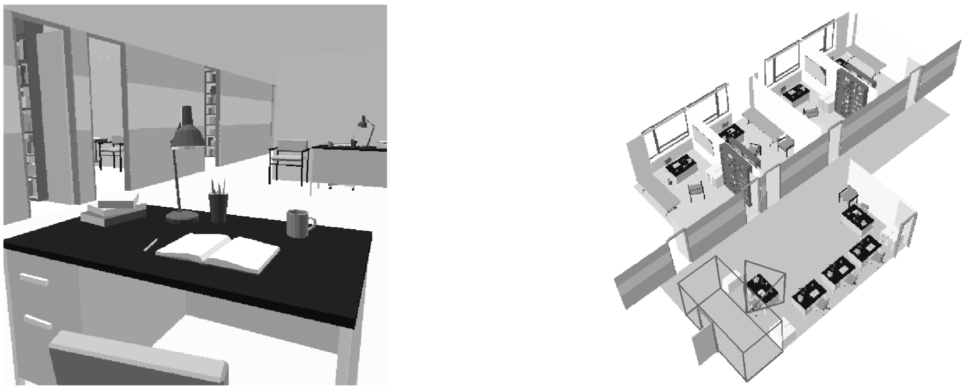


Figure 3.26: Panel used for controlling detail elision parameters.

application for “visualization of the algorithms”. 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 (see Figure 3.27). In either case, we perform exactly the same algorithms, and render exactly the same set of polygons. However, in bird’s eye view, we also include line drawing of the simulated observer view frustum in the image so that the user can visualize where the observer viewpoint is and which polygons are rendered from that observer viewpoint during each frame.



a) Interior View   b) Bird’s Eye View

Figure 3.27: Visualization program supports two views.

The panels that control parameter settings in our building walkthrough program also contain toggles that control features for visualization of the behavior and results of the walkthrough algorithms. For instance, the program can draw outlines of all cells and objects in any visibility set, shade objects based on their benefit, cost, or LOD chosen for display, and/or draw outlines of cells and objects in any memory lookahead set. The program also contains a panel that controls rendering attributes, such as whether edges are drawn on polygons, whether polygon and vertex normals are drawn, whether objects are flat-shaded or Gouraud-shaded, whether texture is applied to object surfaces, and various other lighting characteristics (see Figure 3.28). We have found these visualization controls to be essential for debugging and understanding the behavior of complex three dimensional geometric algorithms. For instance, they were used to generate Figures 3.16, 3.21, 4.7, 4.8, and 4.19.

In order to compare various algorithms and parameter settings quantitatively, we have included features for capturing and repeating a particular observer path, and for gathering

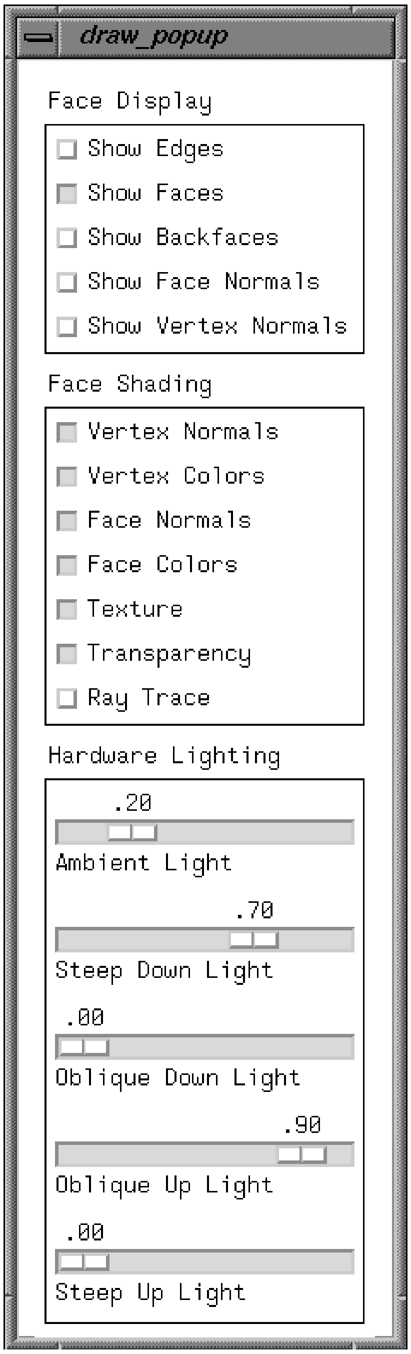


Figure 3.28: Panel containing controls for rendering parameters.

a large set of statistics in our building walkthrough program. Currently, the system can measure the quantities shown in Table 3.3 during every frame of an interactive walkthrough. Using these statistics, we can compare the performance of different tests frame-by-frame. We have found that typical cumulative statistics (e.g., mean, median, standard deviation, minimum, and maximum) are useful, but not complete enough to perform an in-depth analysis of differences between the performance of various tests. For instance, plots in Figures 4.10, 4.11, 4.22, 5.11, 5.12, 5.13, 7.3, and 7.4 show frame-by-frame statistics that are used to understand interesting performance features.

3.3.2 Display Management

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 on the monitor. These display management operations are described in detail in Chapter 4.

3.3.3 Memory Management

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. These memory management operations are discussed in Chapter 5.

Statistic	Description
Frame Time	Time between end-of-rendering of successive frames
Response Time	Time between start-of-frame and end-of-rendering
Visibility Time	Compute time required for visibility determination
Detail elision Time	Compute time required for detail elision
Rendering Time	Time required for rendering polygons
Lookahead Time	Compute time required for lookahead determination
Read Time	Time required for database input/output operations
Rendered Cells	# cells determined potentially visible
Rendered Objects	# objects determined potentially visible
Rendered Polygons	# polygons rendered
Rendered MBytes	# bytes required to describe rendered polygons
Range cells	# cells in range set
Lookahead Cells	# cells containing objects in lookahead set
Lookahead Objects	# objects in lookahead set
Lookahead Polygons	# polygons describing objects in lookahead set
Lookahead MBytes	# bytes describing objects in lookahead set
Resident Cells	# cells containing objects in memory resident cache
Resident Objects	# objects in memory resident cache
Resident Polygons	# polygons describing objects in memory resident cache
Resident MBytes	# bytes describing objects in memory resident cache
Read Cells	# cells containing objects in read set
Read Objects	# objects in read set
Read Polygons	# polygons describing objects in read set
Read MBytes	# bytes describing objects in read set
Release Cells	# cells containing objects in release set
Release Objects	# objects in release set
Release Polygons	# polygons describing objects in release set
Release MBytes	# bytes describing objects in release set
Objects Skipped	# visible objects skipped because not yet in memory
LODs Skipped	# LODs of visible objects skipped because not yet in memory
Wait Time	Time spent waiting for visible object to be read into memory

Table 3.3: Statistics gathered during each frame of an interactive building walkthrough.

Chapter 4

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 using a real-time visibility algorithm based on the spatial subdivision and visibility precomputation results of the precomputation phase and 2) we choose a level of detail and rendering algorithm with which to render each visible object in 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.

4.1 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 3.2.2). 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.

4.1.1 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* (see Figure 4.1). This representation is mapped to a perspective viewing transformation 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 (see Figure 4.2). The ratio of the tangents of the azimuthal and altitudinal half angles is always kept equal to the aspect ratio of the viewport on the graphics workstation display in order to avoid distortion due to anisotropic scaling.

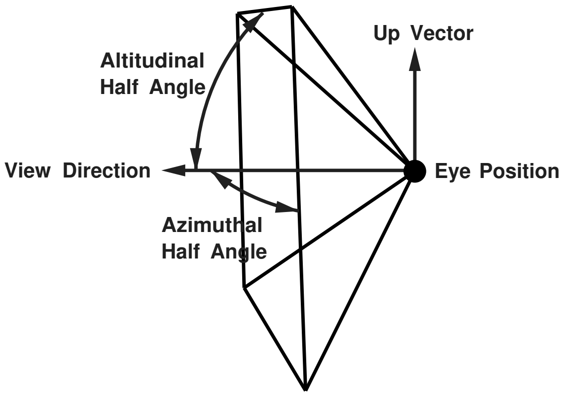


Figure 4.1: View frustum variables.

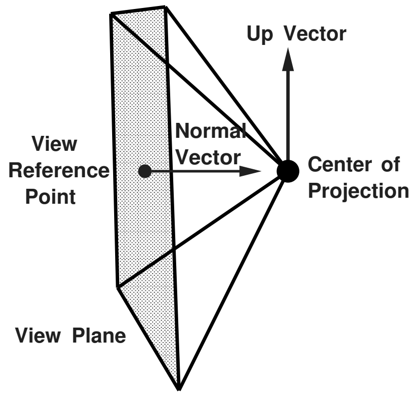


Figure 4.2: Mapping view frustum to perspective projection.

4.1.2 Observer Visibility

Given an observer view frustum, F, we can determine the region visible to it using a procedure similar to the one described in Section 3.2.2. 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 (i.e., only ones in the cell-to-cell visibility of the observer's cell). The region of space visible to a view frustum through a sequence of axial rectangular portals is always a 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 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 (see Figure 4.3). 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 view frustum (i.e., the intersection of R and W) is included in the visible region, and the search recurses into neighboring cells.

A schematic diagram of the visible region computation is shown in two dimensions in Figure 4.4. 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.

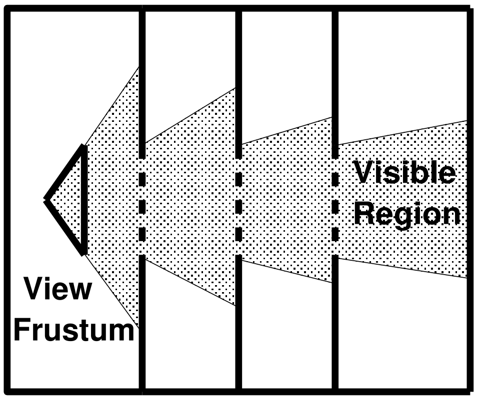


Figure 4.3: Visible region is a wedge that typically narrows as it traverses through more portals.

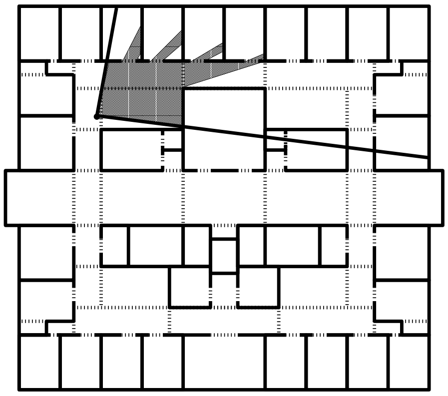


Figure 4.4: Visible region for view frustum in two dimensions.

We construct the *eye-to-cell* and *eye-to-object* visibility for the observer view frustum during the depth-first search. We define the *eye-to-cell* visibility to be the set of cells visible by some sightline that contains the observer view position, that lies within the observer view frustum, and that does not pierce any opaque cell boundaries (see Figure 4.5). That is, the eye-to-cell visibility is exactly the set of cells reached during the depth-first search described in the previous paragraph.

Similarly, we define the *eye-to-object* visibility to be the set of objects whose bounding box is visible by some sightline that contains the observer view position, that lies within the observer view frustum, and that does not pierce any opaque cell boundaries. The eye-to-object visibility is the set of objects incident upon the observer view frustum's visible region. During computation of the eye-to-object visibility using the depth-first search, we check only objects in the observer cell's cell-to-object visibility for a suitable sightline.

Figure 4.6 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 4.7 and 4.8, respectively.

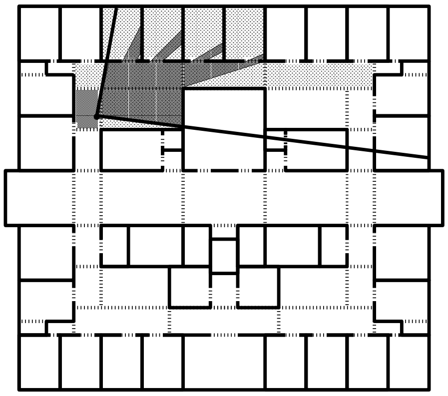


Figure 4.5: Cells in the eye-to-cell visibility are shown in stipple gray.

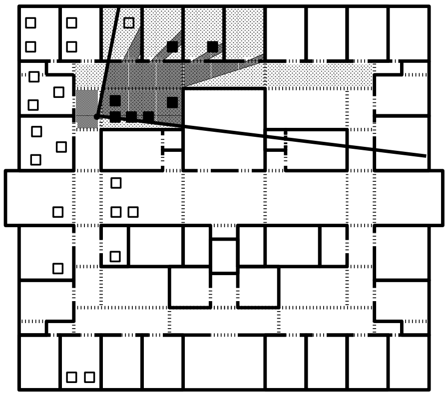


Figure 4.6: Objects in the eye-to-object visibility are shown as by solid squares.

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 accelerate frame rates, if we render only objects in the eye-to-object visibility during each frame of an interactive walkthrough.

4.1.3 Results

To evaluate the effectiveness of the precomputation and real-time visibility determination algorithms, we ran a series of tests using our building walkthrough application. During these tests, we logged statistics for combinations of the following visibility precomputations and real-time visibility determination algorithms as a user walked through the model of Soda Hall:

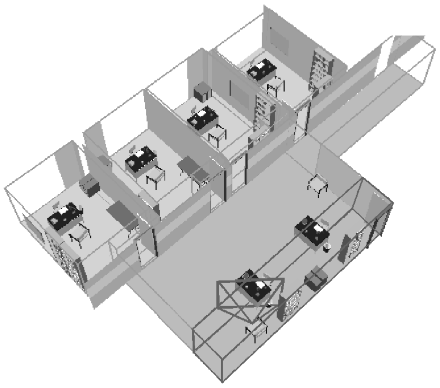


Figure 4.7: All objects incident upon cells in the eye-to-cell visibility.

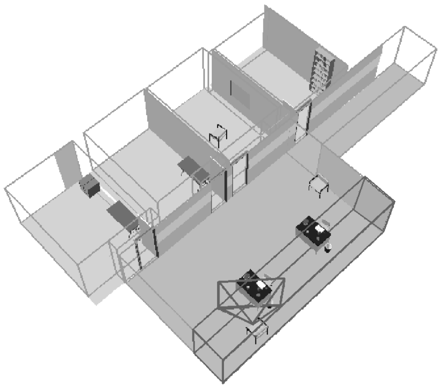


Figure 4.8: Objects in the eye-to-object visibility.

Visibility Precomputations:

- **None:** All objects.
- **Cell-to-Cell (CTC):** All objects incident upon a cell in the cell-to-cell visibility of the observer's cell.
- **Cell-to-Object (CTO):** All objects in the cell-to-object visibility of the observer's cell.

Real-time Cell Visibility Determination:

- **None:** All objects.
- **Frustum-to-Cell (FTC):** All objects incident upon a cell inside the observer view frustum.
- **Eye-to-Cell (ETC):** All objects incident upon a cell in the eye-to-cell visibility of the observer view frustum.

Real-time Object Visibility Determination:

- **None:** All objects.
- **Frustum-to-Object (FTO):** All objects inside the observer view frustum.

- **Eye-to-Object (ETO):** All objects in the eye-to-object visibility of the observer view frustum.

The set of objects rendered in each test is the intersection of the sets generated during the visibility precomputation and real-time cell and object visibility determinations. As a practical optimization, objects consisting of just one polygon (e.g., walls, ceilings, and floors) are trivially accepted during real-time object visibility determination. Since each precomputed, cell visibility, and object visibility set is a superset the objects actually visible to the observer, intersections of these sets is a superset as well. So an image generated by rendering only objects in any intersection of these these sets appears the same as an image generated by rendering all objects in the entire model.

Our notation for referring to the set of objects rendered in each test is a 3-tuple of abbreviated names for the three visibility determination algorithms used (*precomputation*, *cell visibility*, *object visibility*). For example, the notation for the set of objects determined to be visible by the combination of cell-to-object, eye-to-cell, and eye-to-object visibility algorithms is (CTO, ETC, ETO).

In each test, we used the sample observer path through the sixth floor of Soda Hall shown in Figure 4.9. This test path was chosen because it represents typical behavior of real users of a building walkthrough system. Note the observer viewpoints marked 'A', where the observer is viewing a relatively open area, and 'B' and 'C', where the observer is in enclosed offices. These observer viewpoints are marked in graphs and referenced during the following discussion.

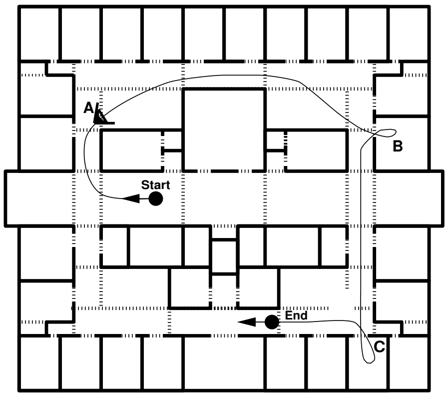


Figure 4.9: Test path through the sixth floor of Soda Hall.

During each test, we measured compute time (i.e., the time required to execute the real-time observer visibility algorithm and to construct a list of objects to be rendered), rendering time (i.e., the time required to render objects), and frame time (i.e., the total time between successive frames), as wells as the numbers of cells, objects, and polygons rendered during each frame. All tests were performed on a Silicon Graphics VGX 320 workstation with two 33MHz MIPS R3000 processors, 64MB of memory, and a $16\mu s$ timer. The application was configured as a two-process pipeline with one process used for visibility computations, while the other was used for rendering. In order to eliminate the effects of memory management in these tests, we used a model of Soda Hall with shared object definitions (21.5MB) that fits in memory entirely.

Mean and maximum statistics for all observer viewpoints along the test walkthrough path are shown for each combination of visibility precomputation and real-time cell visibility and object visibility algorithms in Tables 4.1 and 4.2. Plots of the frame time for every observer viewpoint along the test walkthrough path are shown for (None, None, None), (None, FTC, FTO), (CTO, None, None), and (CTO, ETC, ETO) in Figure 4.10.

No Visibility Determination

If we perform no visibility precomputation and no real-time visibility determination, and render all objects in the entire model during each frame (i.e., (None, None, None)), the frame rate is a near-constant 1/17.9 frames per second – i.e., 1 frame every 17.9 seconds (see Figure 4.10a). The model of Soda Hall has 1,418,807 million polygons. It is too large to be rendered in its entirety at interactive frame rates.

Visibility Precomputation

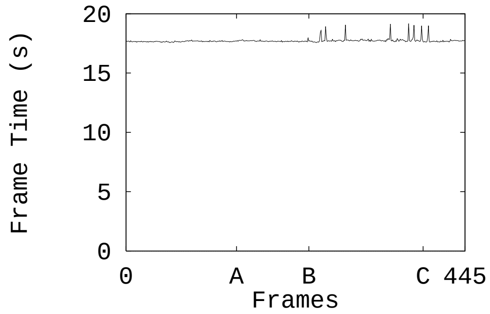
If we use the results of the cell-to-object visibility precomputation described in Section 3.2.2, but do no real-time visibility determination (e.g., (CTO, None, None)), frame rates are generally faster (see Figure 4.10c). The number of polygons required to describe objects in the cell-to-object visibility set for the observer cells along the test walkthrough path is only 3.28% of the entire model on average, and it takes only an average of 0.645 seconds to render them (1.6 frames per second). Furthermore, since the precomputed cell-to-object visibility set can be fetched from the display database and is relatively small (at most 752 objects consisting of 74,071 polygons), little real-time computation is required to identify

Precomp Cull	Cell Cull	Object Cull	# Cells Mean	# Cells Max	# Objects Mean	# Objects Max	# Polygons Mean	# Polygons Max
None	None	None	5,060	5,060	14,478	14,478	1,418,807	1,418,807
		FTO	5,060	5,060	8,949	11,520	228,236	811,469
	FTC	None	851	3122	3,115	9,337	319,359	947,203
		FTO	851	3122	2,624	8,757	221,537	806,850
	ETC	None	19	54	274	657	24,775	70,357
		FTO	19	54	239	562	18,596	62,034
		ETO	19	54	185	426	7,899	19,809
CTC	None	None	117	216	1,309	2,108	102,320	134,818
		FTO	117	216	914	1,567	30,151	70,921
	FTC	None	40	120	534	1,181	44,706	85,940
		FTO	40	120	443	905	29,635	70,608
	ETC	None	19	54	272	657	24,359	70,357
		FTO	19	54	237	562	18,216	62,034
		ETO	19	54	185	426	7,898	19,809
CTO	None	None	117	216	526	752	46,510	74,071
		FTO	117	216	366	566	15,264	47,394
	FTC	None	40	120	228	419	22,426	57,439
		FTO	40	120	186	357	15,048	47,263
	ETC	None	19	54	154	340	16,870	41,895
		FTO	19	54	130	278	12,038	34,989
		ETO	19	54	110	208	7,823	19,768

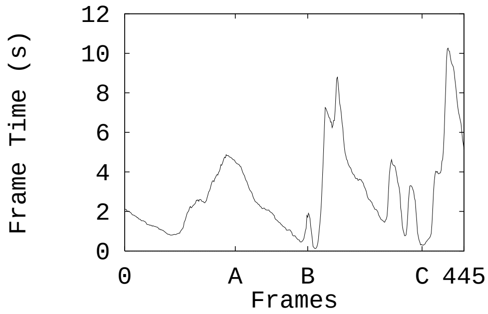
Table 4.1: Mean and maximum set statistics collected during tests with various combinations of precomputation and real-time visibility determination algorithms.

Precomp Cull	Cell Cull	Object Cull	Compute Time (s)		Render Time (s)		Frame Time (s)	
			Mean	Max	Mean	Max	Mean	Max
None	None	None	0.290	0.357	17.937	19.388	17.937	19.388
		FTO	0.300	0.368	3.260	10.379	3.260	10.379
	FTC	None	0.137	0.329	4.199	12.183	4.199	12.183
		FTO	0.150	0.377	2.965	10.371	2.965	10.371
	ETC	None	0.014	0.045	0.358	0.925	0.360	0.925
		FTO	0.015	0.050	0.272	0.839	0.276	0.843
		ETO	0.078	0.209	0.130	0.301	0.141	0.311
CTC	None	None	0.032	0.062	1.410	1.814	1.410	1.814
		FTO	0.033	0.069	0.454	0.979	0.454	0.979
	FTC	None	0.015	0.037	0.634	1.320	0.634	1.320
		FTO	0.017	0.050	0.428	0.967	0.428	0.967
	ETC	None	0.016	0.042	0.354	0.932	0.356	0.942
		FTO	0.017	0.050	0.268	0.841	0.272	0.844
		ETO	0.080	0.225	0.131	0.308	0.142	0.319
CTO	None	None	0.013	0.030	0.645	0.970	0.645	0.977
		FTO	0.014	0.033	0.241	0.637	0.244	0.638
	FTC	None	0.008	0.023	0.329	0.769	0.333	0.774
		FTO	0.008	0.031	0.229	0.627	0.235	0.633
	ETC	None	0.012	0.040	0.250	0.621	0.255	0.629
		FTO	0.012	0.049	0.183	0.531	0.189	0.530
		ETO	0.050	0.128	0.127	0.301	0.133	0.311

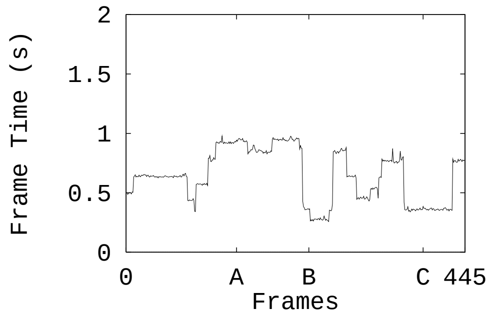
Table 4.2: Mean and maximum timing statistics collected during tests with various combinations of precomputation and real-time visibility determination algorithms.



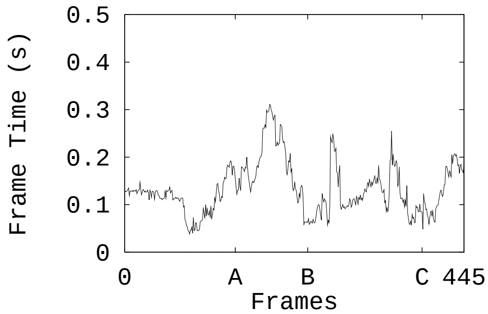
a) (None, None, None)



b) (None, FTC, FTO)



c) (CTO, None, None)



d) (CTO, ETC, ETO)

Figure 4.10: Frame time for each observer viewpoint along test walkthrough path. Note different scales along the “Frame Time” axis.

visible objects and load them onto a list to be rendered.

Real-Time Frustum Cull

If we perform no visibility precomputation, and instead use only a simple real-time cull which rejects objects outside the observer view frustum (i.e., (None, FTC, FTO)), we render 221,537 polygons during each frame on average. Although this simple technique culls away 84.4% of the model on average, frame rates are not nearly interactive (1 frame every 2.965 seconds). Furthermore, the effectiveness of the frustum cull is highly variable (see Figure 4.10b). Few objects are visible from observer viewpoints looking outward from positions near the perimeter of the building (e.g., the observer viewpoint marked 'B'), whereas many objects are visible from other viewpoints. In the worst case along this test walkthrough path, there are 8,757 objects (806,850 polygons) inside the observer view frustum, and the frame time slows to 10.371 seconds, when the observer is looking towards the center of the building from a corner (e.g., the observer viewpoint marked 'C').

If we consider only objects in the precomputed cell-to-object visibility set of the observer cell, and apply a real-time frustum cull to each one during every frame of an interactive walkthrough (e.g., (CTO, ETO, FTO)), we cull away almost 99% of the model and generate frames in 0.235 seconds on average (4.3 frames per second). The real-time compute time using this combination of visibility algorithms is relatively small (0.008 seconds), even smaller than the tests in which no real-time visibility determination is performed (0.013 seconds). This is because the measured compute time is affected by both the complexity of the real-time visibility determination algorithm and the size of the resulting visibility set. In this case, the real-time visibility determination algorithm is relatively simple (i.e., check each object in the cell-to-object visibility of the observer cell to see if it is inside the observer view frustum), and the resulting visibility set is relatively small (i.e., the time required for constructing the list of objects to render is small). Since a large portion of the model is eliminated from consideration during the visibility precomputation, a very small subset of the model can be selected for rendering with little real-time computation.

Real-Time Sight-Line Cull

The fastest frame rates are achieved in our tests using the eye-to-cell and eye-to-object real-time visibility determination algorithms (see Figure 4.10d). For instance, using these algorithms with the cell-to-object visibility precomputation (i.e., (CTO, ETC, ETO)), we are able to cull away almost 99.5% of the building model, and render frames in 0.133 seconds on average (7.5 frames per second).

However, the eye-to-cell and eye-to-object visibility algorithms require the most compute time per frame (0.050 seconds on average). For each cell reached during the eye-to-cell adjacency graph traversal (see Section 4.1), we must check for a feasible sightline from the observer eye position to the bounding box of each object that is both inside the reached cell and in the cell-to-object visibility set of the observer cell. Fortunately, since real-time visibility determination is done on a separate processor in parallel with rendering of the previous frame in a pipelined architecture, compute time does not influence the effective frame rate unless it is the rate-limiting step. Figure 4.11 shows the compute time, draw time, and effective frame time for (CTO, ETC, ETO) in each frame along the test walkthrough path. The effective frame time is correlated with the maximum of the compute time and draw time. Since the compute time is almost always less than the draw time in this test, it contributes little to the effective frame time (this is not true if we use detail elision to further decrease rendering time). A more in-depth discussion of concurrent processing in our walkthrough appears in Chapter 6.

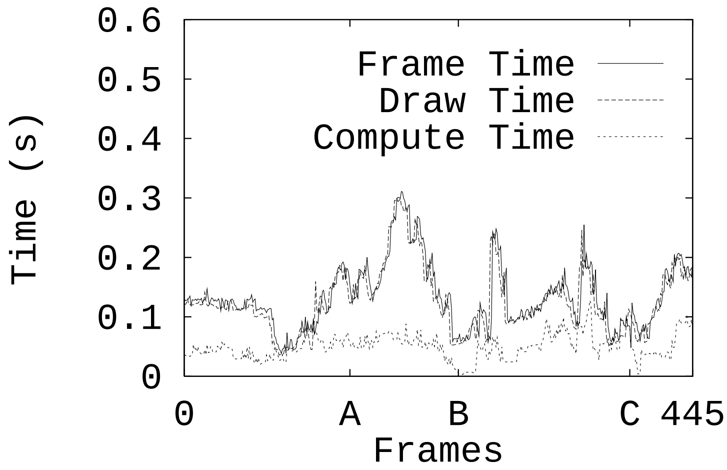


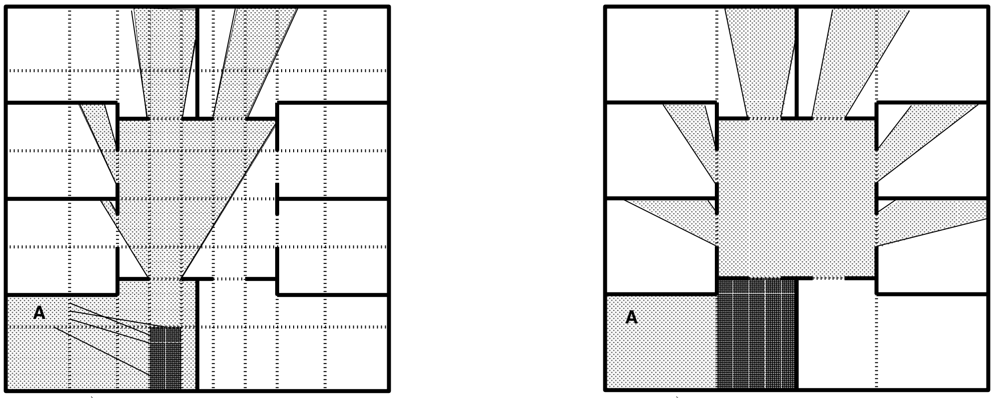
Figure 4.11: Compute time, draw time, and frame time for each observer viewpoint along the test walkthrough path using (CTO, ETC, ETO) visibility determination.

4.1.4 Discussion

The success of our precomputation and real-time visibility algorithms depends greatly on the *granularity* (i.e., the number of cells) of the spatial subdivision. In our approach, the same granularity is used for both "source" cells and "reached" cells, and only opaque polygons that lie on cell boundaries are considered occluders by the visibility algorithms. So, unless the spatial subdivision contains a "split" along a particular polygon, neither precomputation nor real-time visibility algorithms benefit from its occlusion. This factor suggests a very fine spatial subdivision with splits along almost every polygon in the model. Finer spatial subdivisions (i.e., ones with smaller cells) also allow the visibility precomputation to determine a smaller potentially visible subset of the model for each source cell. For instance, the cell-to-region visibility (shown in stipple gray) for the source cell drawn in black in Figure 4.12a is smaller than the cell-to-region visibility for the larger source cell shown in Figure 4.12b.

On the other hand, a finer spatial subdivision requires more data to be stored in the display database. In addition, since our visibility determination algorithms perform a depth-first search of the cell adjacency graph, checking for a feasible sight-line through each portal sequence encountered during the search, a finer spatial subdivision may cause portal sequences to be longer, more cells to be searched, and each cell to be reached by more possible paths through the cell adjacency graph during visibility determination. For instance, the region marked 'A' can be reached by 4 non-degenerate paths (shown by thin black lines) through the spatial subdivision shown in Figure 4.12a (average portal sequence length = 4); whereas the same region can be reached by only 1 path in the spatial subdivision shown in Figure 4.12b (average portal sequence length = 1). In general, there is a combinatorial explosion in the number of paths to be checked during visibility determination due to adding cells to the spatial subdivision whose boundaries do not have "sufficient" occlusion.

The trade-offs in choosing an appropriate spatial subdivision granularity and visibility determination algorithm are summarized in Figure 4.13 which contains graphs of spatial subdivision granularity versus: a) cumulative opaque area on cell boundaries, b) CTO visibility precomputation time, c) cumulative storage requirements for CTO visibility, d) percentage of the model in the CTO visibility for average cell, e) (CTO, FTC, FTO) real-time visibility compute time, and f) (CTO, ETC, ETO) real-time visibility compute time for hypothetical spatial subdivisions. Estimated granularity for the spatial subdivisions



a) Fine granularity. b) Coarse granularity.

Figure 4.12: Two possible spatial subdivisions for the same model.

shown in Figure 4.12a and 4.12b are marked on these graph by 'A' and 'B', respectively.

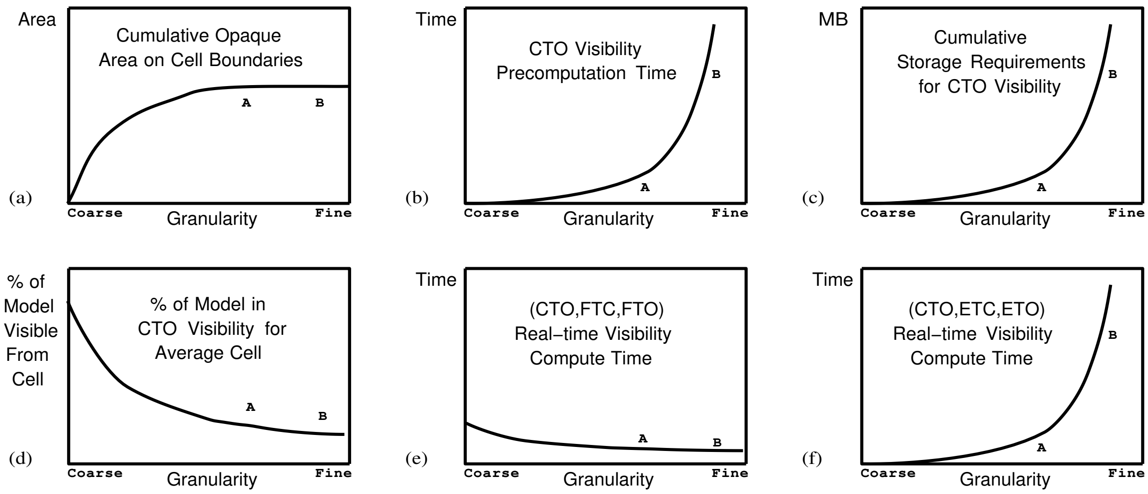


Figure 4.13: Trade-offs in spatial subdivision granularity.

It seems that there are two "local optima" in the search for an appropriate combination of spatial subdivision granularity and visibility determination algorithm. First, a very fine spatial subdivision might be used with a cell-to-object visibility precomputation and a simple frustum-based real-time visibility determination algorithm (i.e., (CTO, FTC, FTO)). Visibility precomputation might even be extended to classify visibility for particular view direction ranges. Using this approach, precomputation time and storage requirements are quite high, but real-time visibility compute time is very small. Since the cell-to-object visibility precomputation for a fine spatial subdivision culls away a large portion of the model and is performed off-line, a close approximation to the true observer visibility can be generated with very little real-time computation.

Second, a rather coarse subdivision might be used with a cell-to-object visibility precomputation and an eye-to-object real-time visibility determination algorithm (i.e., (CTO, ETC, ETO)). Using this approach, the precomputation time and storage requirements are much smaller, but the real-time visibility compute time is larger. If the spatial subdivision is fine enough to capture the occlusion in the model, yet coarse enough that there are not too many different possible portal sequences to check during visibility determination (i.e., to the right of the "knee" near point 'A' in Figure 4.13b), then a very close approximation to the true observer visibility can be generated with very little visibility precomputation and storage, yet reasonable compute time.

In our work, we have chosen to pursue the latter approach – we use a relatively coarse spatial subdivision (i.e., with cells roughly corresponding to rooms of the building) and the eye-to-object real-time visibility determination algorithm. This choice is made because we expect the storage requirements for a very fine spatial subdivision to be prohibitive in our current implementation. Perhaps other types of visibility precomputation results (e.g., ones that store incremental visibility results with portals) might be more storage efficient. Or perhaps non-uniform spatial subdivisions (i.e., ones in which the granularity of "source" cells differs from the granularity of "searched" cells) might be more appropriate. Our current spatial subdivision for Soda Hall has required quite a bit of hand tuning – we are still investigating techniques for generating spatial subdivisions of appropriate granularity automatically.

4.2 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 portion of the model visible to the observer can be highly variable. Tens of thousands of polygons might be simultaneously visible from some observer viewpoints, whereas just a few can be seen from others. Certainly, there is no upper bound on the complexity of the scene visible from an observer viewpoint. For instance, consider walking through a very detailed model of a fully stocked department store, or viewing an assembly of a complete airplane engine. In our model of Soda Hall, there are some viewpoints from which an observer can see more than eighty thousand polygons. Clearly, visibility processing alone is not sufficient to guarantee a uniform, interactive frame rate. As a result, a simple display algorithm that renders all potentially visible polygons with some predetermined quality may generate frames at highly variable rates, with no guaranteed upper bound on any single frame time.

We have developed an adaptive algorithm for interactive visualization that guarantees a user-specified target frame rate. The idea behind the algorithm is to trade image quality for interactivity in situations where the environment is too complex to be rendered in full detail at the target frame rate. We perform a constrained optimization that selects a level of detail and a rendering algorithm with which to render each potentially visible object to produce the "best" image possible within a user-specified target frame time. In contrast to previous culling techniques, this algorithm supports a uniform, bounded frame rate, even during visualization of very large, complex models.

4.2.1 Levels of Detail

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 3.1.2), 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 [18], and has been used by numerous commercial visualization systems [48]. 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 techniques for choosing a level of detail at which to render each visible object use static heuristics, most often based on a threshold regarding the size or distance of an object to the observer [10, 44, 45, 48, 59], or the number of pixels covered by an average polygon [24]. These simple heuristics can be very effective at improving frame rates in cases where most visible objects are far away from the observer and map to very few pixels on the workstation screen. In these cases, simpler representations of some objects can be displayed, reducing the number of polygons rendered without noticeably reducing image quality.

Although static heuristics for visibility determination and LOD selection improve frame rates in many cases, they do not generally produce a *uniform* frame rate. Since LODs are computed independently for each object, the number of polygons rendered during each frame time depends on the complexity of the scene viewed by the observer. For instance, in some cases, the observer might be looking closely at a very complex scene, containing many detailed objects (e.g., a flower garden). If many complex objects appear large enough to pass the static size threshold for detail reduction and are rendered at the highest LOD, the frame rate is very slow. In other cases, the scene visible to the observer might be very simple, consisting of just a few objects that appear small to the observer. These objects are rendered at the lowest LOD, resulting in a very fast frame rate. Using static heuristics, the frame rate can vary dramatically from frame to frame, depending on the complexity and size of the objects visible to the observer.

Furthermore, static heuristics for visibility determination and LOD selection do not even guarantee a *bounded* frame rate. The frame rate can become arbitrarily slow, as the scene visible to the observer can be arbitrarily complex. In many cases, the frame rate may become so slow that the system is no longer interactive. Instead, a LOD selection algorithm should adapt to overall scene complexity in order to produce uniform, bounded frame rates.

4.2.2 Adaptive Detail Elision

In an effort to maintain a specified *target frame rate*, some commercial flight simulators use an adaptive algorithm that adjusts the size threshold for LOD selection based on feedback regarding the time required to render previous frames [47]. If the previous frame took longer than the target frame time, the size threshold for LOD selection is increased so that future frames can be rendered more quickly.

This adaptive technique works reasonably well for flight simulators, in which there is a large amount of coherence in scene complexity from frame to frame. However, during visualization of more 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 more quickly to achieve the target frame rate.

In order to *guarantee* a bounded frame rate during visualization of discontinuous virtual environments, an adaptive algorithm for LOD selection should be *predictive*, based on the complexity of the scene to be rendered in the current frame, rather than *reactive*, based only on the time required to render previous frames. A predictive algorithm might estimate the time required to render every object at every level of detail, and then compute the largest size threshold that allows the current frame to be rendered within the target frame time. Unfortunately, implementing a predictive algorithm is non-trivial, since no closed-form solution exists for the appropriate size threshold.

4.2.3 Optimization Detail Elision

Our approach is a generalization of the predictive approach. Conceptually, every potentially visible object can be rendered at any level of detail, and with any rendering algorithm (e.g., flat-shaded, Gouraud-shaded, texture mapped, etc.). Every combination of objects rendered with certain levels of detail and rendering algorithms takes a certain amount of time, and produces a certain image. We aim to find the combination of levels of detail and rendering algorithms for all potentially visible objects that produces the "best" image possible within the target frame time.

More formally, 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: $\mathit{Cost}(O, L, R)$ and $\mathit{Benefit}(O, L, R)$. 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 define S to be the set of object tuples rendered in each frame. Using these formalisms, our approach for choosing a level of detail and rendering algorithm for each potentially visible object can be stated:

$$\begin{array}{ll} \text{Maximize :} & \\ & \sum_S \mathit{Benefit}(O, L, R) \\ \text{Subject to :} & \\ & \sum_S \mathit{Cost}(O, L, R) \leq \mathit{TargetFrameTime} \end{array} \tag{4.1}$$

This formulation 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.

For this approach to be successful, we need to find *Cost* and *Benefit* heuristics that can be computed quickly and accurately. Unfortunately, *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.

Cost Heuristic

The $Cost(O, L, R)$ heuristic is an estimate of the time required to render object O with level of detail L and rendering algorithm R. Of course, the actual rendering time for a set of polygons depends on a number of complex factors, including the type and features of the graphics workstation. However, using a model of a generalized rendering system and several simplifying assumptions, it is possible to develop an efficient, approximate *Cost* heuristic that can be applied to a wide variety of workstations. Our model, which is derived from the *Graphics Library Programming Tools and Techniques* document from Silicon Graphics, Inc. [50], represents the rendering system as a pipeline with the two functional stages shown in Figure 4.14:

- *Per Primitive:* coordinate transformations, lighting calculations, clipping, etc.
- *Per Pixel:* rasterization, z-buffering, alpha blending, texture mapping, etc.

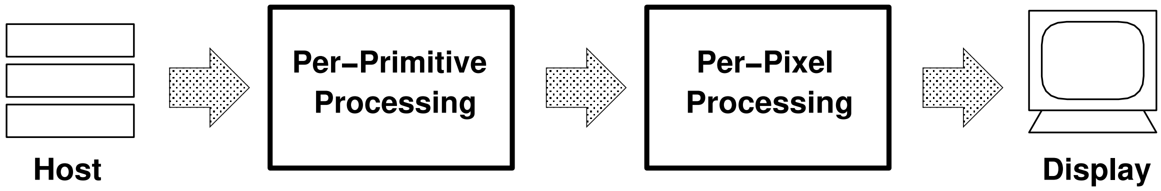


Figure 4.14: Two-stage model of the rendering pipeline.

Since separate stages of the pipeline run concurrently, and must wait only if a subsequent stage is "backed up," the throughput of the pipeline is determined by the speed of the slowest stage – i.e., the bottleneck. If we assume that the host is able to send primitives to the graphics subsystem faster than they can be rendered, and no other operations are being executed that affect the speed of any stage of the graphics subsystem, we can model the time required to render an object tuple as the maximum of the times taken by any of the stages.

We assume that the time required for the *Per Primitive* stage is simply the sum of the times taken for each rendering command sent by the host. We also assume that the host sends rendering commands either *per polygon* or *per vertex*, as is the case in most applications.

```
For each polygon do ...
        { Per polygon commands }
        BeginPolygon;
        For each vertex do ...
                { Per vertex commands }
        EndPolygon;
End;
```

The actual number and type of commands performed *per polygon* and *per vertex* depend on the rendering algorithm used. For instance, if flat shading is used, only one color must be specified *per polygon*, and only coordinates must be specified *per vertex*. On the other hand, if Gouraud shading, hardware lighting, and texture mapping is used, a material and texture must be specified *per polygon*, and a normal vector, texture coordinates, and positional coordinates must be specified *per vertex*.

We model the time taken by the *Per Primitive* stage as a linear combination of the number of polygons and vertices in an object tuple, with coefficients that depend on the rendering algorithm and machine used. Likewise, we assume that the time taken by the *Per Pixel* stage is proportional to the number of pixels an object covers. Our model for the time required to render an object tuple is:

$$Cost(O, L, R) = max \left\{ \begin{array}{c} C_1 Poly(O, L) + C_2 Vert(O, L) \\ C_3 Pix(O) \end{array} \right\}$$

where O is the object, L is the level of detail, R is the rendering algorithm, and C_1, C_2 and C_3 are constant coefficients specific to a rendering algorithm and machine.

For a particular rendering algorithm and machine, useful values for these coefficients can be determined experimentally by rendering sample objects with a wide variety of sizes and LODs, and graphing measured rendering times versus the number of polygons, vertices and pixels drawn. Figure 4.15a shows measured times for rendering four different LODs of the chair shown in Figure 3.3 with flat-shading. The slope of the best fitting line through the data points represents the time required *per polygon* during this test. Similarly, *per pixel* coefficients are derived from plots like the one shown in Figure 4.15b. Using empirical techniques, we have derived cost model coefficients for our Silicon Graphics VGX 320 that are accurate within 10% at the 95% confidence level. A comparison of actual and predicted rendering times for a sample set of frames during an interactive building walkthrough is shown in Figure 4.16. In the future, we hope to extend our system to "learn" appropriate cost coefficients during real-time walkthroughs.

Benefit Heuristic

The *Benefit*(O, L, R) heuristic is an estimate of the "contribution to model perception" of rendering object O with level of detail L and rendering algorithm R. Ideally, it predicts the amount and accuracy of information conveyed to a user due to rendering an object tuple. Of course, it is extremely difficult to model human perception and understanding accurately, so we have developed a simple, easy-to-compute heuristic based on intuitive principles.

Our *Benefit* heuristic depends primarily on the size of an object tuple in the final image. Intuitively, objects that appear larger to the observer "contribute" more to the image (see Figure 4.17). Therefore, the base value for our *Benefit* heuristic is simply an estimate of

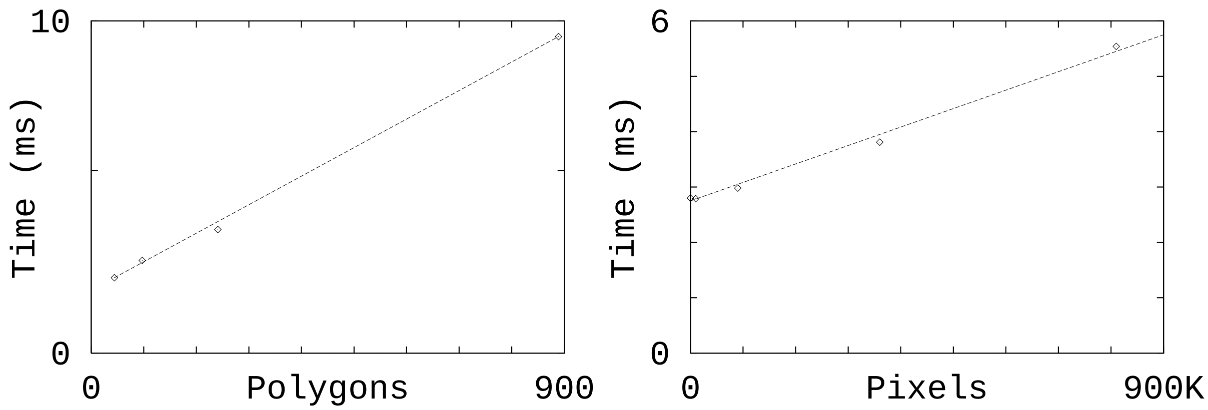


Figure 4.15: Cost model coefficients can be determined empirically. The plots show actual flat-shaded rendering times for the chair shown in Figure 3.3 using different numbers of polygons (on the left), and different sizes (on the right).

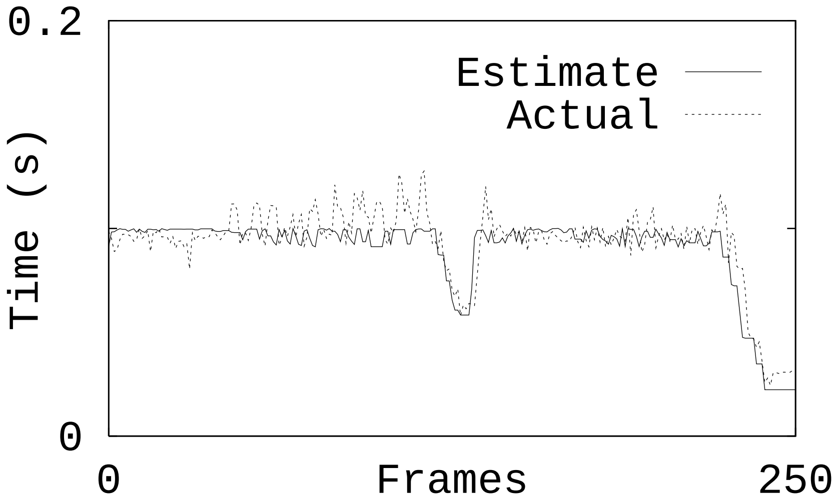


Figure 4.16: Comparison of actual and estimated rendering times of frames during an interactive building walkthrough.

the number of pixels covered by the object, e.g., the area of the projection onto the view plane of a sphere bounding the object.

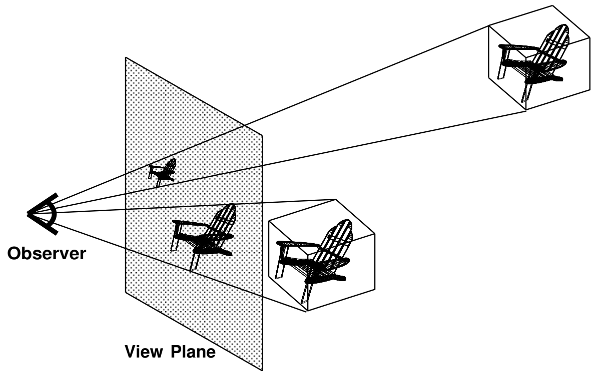
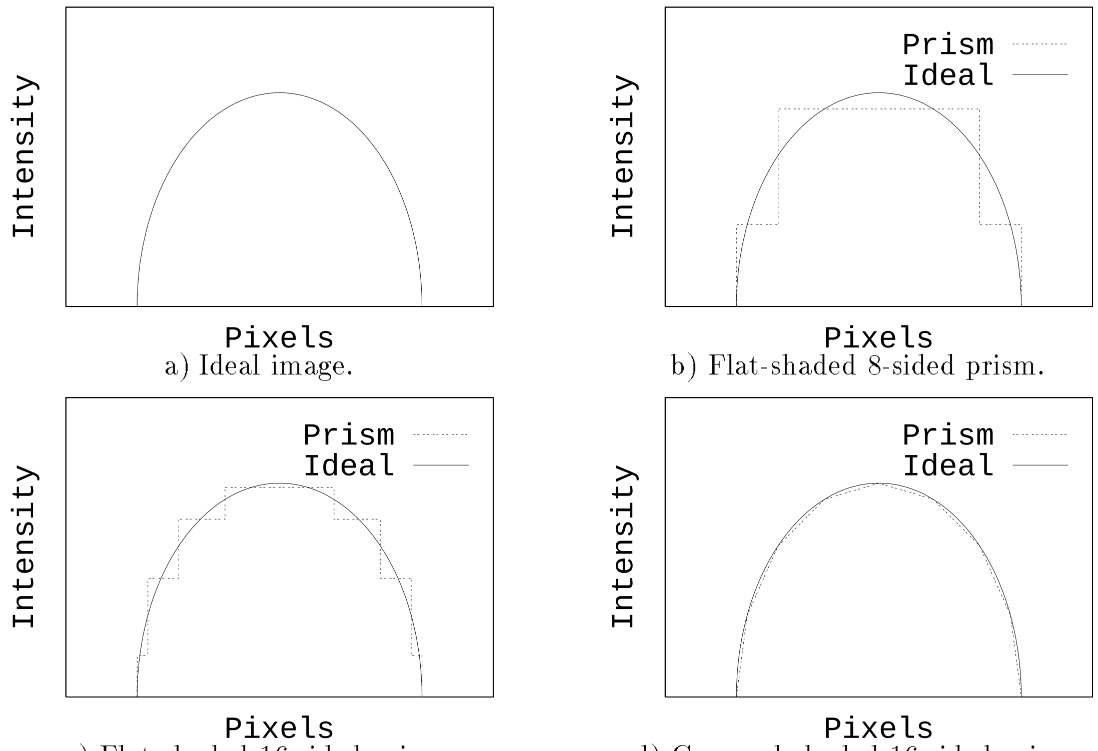


Figure 4.17: Objects that appear larger "contribute" more to the image.

Our *Benefit* heuristic also depends on the "accuracy" of an object tuple rendering. Intuitively, using a more detailed representation or a more realistic rendering algorithm for an object generates a higher quality image, and therefore conveys more accurate information to the user. Conceptually, we evaluate the "accuracy" of an object tuple rendering by comparison to an *ideal image* generated with an ideal camera. For instance, consider generating a gray-level image of a scene containing only a cylinder with a diffusely reflecting Lambert surface illuminated by a single directional light source in orthonormal projection. Figure 4.18a shows an intensity plot of a sample scan-line of an ideal image generated for the cylinder.

Consider approximating this ideal image with an image generated using a flat-shaded, polygonal representation for the cylinder. Since a single color is assigned to all pixels covered by the same polygon, a plot of pixel intensities across a scan-line of such an image is a stair-function. If an 8-sided prism is used to represent the cylinder, at most 4 distinct colors can appear in the image (one for each front-facing polygon), so the resulting image does not approximate the ideal image very well at all, as shown in Figure 4.18b. By comparison, if a 16-sided prism is used to represent the cylinder, as many as 8 distinct colors can appear in the image, generating a closer approximation to the ideal image, as shown in Figure 4.18c.

Next, consider using Gouraud shading for a polygonal representation. In Gouraud shading, intensities are interpolated between vertices of polygons, so a plot of pixel intensities is a continuous, piecewise-linear function. Figure 4.18d shows a plot of pixel intensities



a) Ideal image. b) Flat-shaded 8-sided prism.

c) Flat-shaded 16-sided prism. d) Gouraud-shaded 16-sided prism.

Figure 4.18: Plots of pixel intensity across a sample scan-line of images generated using different representations and rendering algorithms for a simple cylinder.

across a scan line for a Gouraud shaded 16-sided prism. Compared to the plot for the flat-shaded image (Figure 4.18b), the Gouraud shaded image approximates the ideal image much more closely.

More complex representations (e.g., parametric or implicit surfaces) and rendering techniques (e.g., Phong shading, antialiasing, or ray tracing) could be used to approximate the ideal image even more closely. Based on this intuition, we assume that the "error" (i.e., the difference from the ideal image) decreases with the number of samples (e.g., rays/vertices/polygons) used to render an object tuple and is dependent on the type of interpolation method used (e.g., Gouraud/flat). We capture these effects in the *Benefit* heuristic by multiplying by an "accuracy" factor:

$$Accuracy(O, L, R) = 1 - Error = 1 - \frac{BaseError}{Samples(L, R)^m}$$

where Samples(L, R) is #pixels for ray tracing, or #vertices for Gouraud shading, or #polygons for flat-shading (but never more than #pixels); and m is an exponent related to the error of the interpolation method used (flat = 1, Gouraud = 2). The *BaseError* is arbitrarily set to 0.5 to give a strong error for a curved surface represented by a single flat polygon, while still accounting for a significantly higher benefit than not rendering the surface at all.

In addition to the size and accuracy of an object tuple rendering, our *Benefit* heuristic depends on several other, more qualitative, factors, some of which apply to a static image, while others apply to sequences of images:

- **Semantics:** Some types of object may have inherent "importance." For instance, walls might be more important than pencils to the user of a building walkthrough; and enemy robots might be most important to the user of a video game. We adjust the *Benefit* of each object tuple by an amount proportional to the inherent importance of its object type.

- **Focus:** Objects that appear in the portion of the screen at which the user is looking might contribute more to the image than ones in the periphery of the user's view. Since we currently do not track the user's eye position, we simply assume that objects appearing near the middle of the screen are more important than ones near the side. We reduce the *Benefit* of each object tuple by an amount proportional to its distance from the middle of the screen.

- **Motion Blur:** Since objects that are moving quickly across the screen appear blurred or can be seen for only a short amount of time, the user may not be able to see them clearly. So we reduce the *Benefit* of each object tuple by an amount proportional to the ratio of the object's apparent speed to the size of an average polygon.

- **Hysteresis:** Rendering an object with different levels of detail in successive frames may be bothersome to the user and may reduce the quality of an image sequence. Therefore, we reduce the *Benefit* of each object tuple by an amount proportional to the difference in level of detail or rendering algorithm from the ones used for the same object in the previous frame.

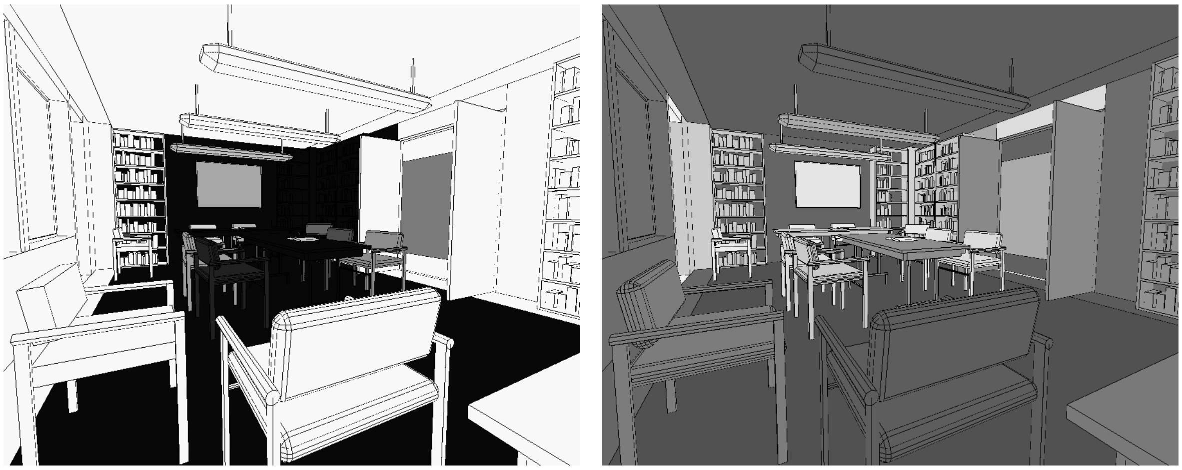
Each of these qualitative factors is represented by a multiplier between 0.0 and 1.0 reflecting a possible reduction in object tuple benefit. The overall *Benefit* heuristic is the product of the following factors:

$$\begin{gathered} \mathit{Benefit}(O,L,R) = \mathit{Size}(O) * \mathit{Accuracy}(O,L,R) * \\ \mathit{Importance}(O) * \mathit{Focus}(O) * \mathit{Motion}(O) * \mathit{Hysteresis}(O,L,R) \end{gathered}$$

This *Benefit* heuristic is a simple experimental estimate of an object tuple's "contribution to model perception." 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 middle of the screen), and more apt to blend with other images in a sequence (i.e., hysteresis). In our implementation, the user can manipulate the relative weighting of these factors interactively using sliders on a control panel (see Figure 3.26), and observe their effects in a real-time walkthrough. For example, Figure 4.19 shows images depicting the relative benefits of objects for different settings for the *Focus* factor of the *Benefit* heuristic – darker shades of gray represent more benefit. Notice that objects near the periphery of the image have less benefit when the *Focus* factor is set to a smaller value. Although our current *Benefit* heuristic is rather ad hoc, it is useful for experimentation until we are able to encode more accurate models for human visual perception and understanding.

Optimization Algorithm

We use the *Cost* and *Benefit* heuristics described in the previous sections to choose a set of object tuples to render each frame by solving equation 4.1 in Section 4.2.3.



a) Focus = 1.0 b) Focus = 0.1

Figure 4.19: Images depicting the relative benefit of objects with the *Focus* factor of the *Benefit* heuristic set to a) 1.0 and b) 0.1. Darker shades of gray represent higher values for the *Benefit* heuristic.

Unfortunately, this constrained optimization problem is NP-complete. It is the Continuous Multiple Choice Knapsack Problem [26, 32], 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. In this case, the set S of object tuples rendered is the knapsack, the object tuples are the elements to be placed into the knapsack, the target frame time is the size of the knapsack, the sets of object tuples representing the same object are the candidate sets, and the *Cost* and *Benefit* functions specify the "size" and "profit" of each element, respectively. The problem is to select the object tuples that have maximum cumulative benefit, but whose cumulative cost fits in the target frame time, subject to the constraint that only one object tuple representing each object may be selected.

We have implemented a simple, greedy approximation algorithm for this problem that selects object tuples with the highest *Value*, defined as $Benefit(O, L, R)/Cost(O, L, R)$. Logically, we add object tuples to S in descending order of *Value* until the maximum cost is competely claimed. However, if an object tuple is added to S which represents the same object as another object tuple already in S, only the object tuple with the maximum benefit of the two is retained. The merit of this approach can be explained intuitively by noting that each subsequent portion of the frame time is used to render the object tuple with the best available "bang for the buck." It is easy to show that a simple implementation of this greedy approach runs in $O(n \log n)$ time for n potentially visible objects, and produces a solution that is at least half as good as the optimal solution [26].

Rather than computing and sorting the *Benefit*, *Cost*, and *Value* for all possible object tuples during every frame, as would be required by a naive implementation, we have implemented an incremental optimization algorithm that takes advantage of the fact that there is typically a large amount of coherence between successive frames. The algorithm works as follows: At the start of the algorithm, an object tuple is added to S for each potentially visible object. Initially, each object is assigned the LOD and rendering algorithm chosen in the previous frame, or the lowest LOD and rendering algorithm if the object is newly visible. In each iteration of the optimization, the algorithm first increments the accuracy attribute (LOD or rendering algorithm) of the object that has the highest subsequent *Value*. It then decrements the accuracy attributes of the object tuples with the lowest current *Value* until the cumulative cost of all object tuples in S is less than the target frame time. The algorithm terminates when the same accuracy attribute of the same object tuple is both incremented and decremented in the same iteration. Psuedocode for this algorithm is shown in Figure 4.20.

This incremental implementation finds an approximate solution that is the same as the solution found by the naive implementation, provided that *Values* of object tuples decrease monotonically as tuples are rendered with greater accuracy (i.e., there are diminishing returns with more complex renderings). In this case, the *Value* of the tuple both incremented and decremented in the final iteration separates the tuple space into two groups: ones with higher *Value* (all of which have been selected), and ones with lower *Value* (all of which have not been selected).

In any case, the worst-case running time for the algorithm is $O(n \log n)$. 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; they do not increase the effective frame rate as long as the time required for computation is not greater than the time required for display.

```
// Initialize set of object tuples
S = InitialTupleSet();
COST = CumulativeCost(S);

// Build increment/decrement priority queues of tuples
NEXT = CreatePriorityQueue(S, MaxValueAfterIncrement);
CURRENT = CreatePriorityQueue(S, MinCurrentValue);

// Iteratively increment/decrement tuples
while not Done do
  // Increment tuple with maximum next value
  if (COST ≤ MAXCOST) then
    // Get tuple with maximum next value
    BEST = GetPriorityQueue(NEXT);
    if (!BEST) Done;

    // Increment BEST tuple
    IncrementTuple(BEST);
    UpdatePriorityQueue(NEXT, BEST);
    UpdatePriorityQueue(CURRENT, BEST);
    UpdateCost(COST, BEST);
  endif

  // Decrement tuples with minimum current value
  while (COST > MAXCOST) do
    // Get tuple with minimum current value
    WORST = GetPriorityQueue(CURRENT);
    if (!WORST) Done;

    // Decrement WORST tuple
    DecrementTuple(WORST);
    UpdatePriorityQueue(NEXT, WORST);
    UpdatePriorityQueue(CURRENT, WORST);
    UpdateCost(COST, WORST);

    // Check termination criterion
    if (Δ BEST == -Δ WORST) Done;
  endwhile
endwhile
```

Figure 4.20: Pseudocode for the constrained optimization algorithm.

4.2.4 Results

To test whether this new cost/benefit optimization algorithm produces more uniform frame rates than previous LOD selection algorithms, we ran a series of tests with our building walkthrough application using four different LOD selection algorithms:

- **No Detail Elision:** Each potentially visible object is rendered at the highest LOD.
- **Static:** Each potentially visible object is rendered at the highest LOD for which an average polygon covers at least 1024 pixels on the screen.
- **Feedback:** Similar to *Static* test, except the size threshold for LOD selection is updated in each frame by a feedback loop, based on the difference between the time required to render the previous frame and the target frame time of one-tenth of a second.
- **Optimization:** Each potentially visible object is rendered at the LOD chosen by the cost/benefit optimization algorithm described in Section 4.2.3 in order to meet the target frame time of one-tenth of a second. For comparison sake, the *Benefit* heuristic is limited to consideration of *object size* in this test, i.e., all other *Benefit* factors are set to 1.0.

All tests were performed on a Silicon Graphics VGX 320 workstation with two 33MHz MIPS R3000 processors and 64MB of memory. We used the (CTO, ETC, ETO) visibility algorithm described in Chapter 4.1 to determine a set of potentially visible objects to be rendered in each frame. The application was configured as a two-process pipeline with one process for visibility and LOD selection computations and another separate process for rendering. Timing statistics were gathered using a $16\mu s$ timer.

In each test, we used the sample observer path shown in Figure 4.21 through a model of an auditorium on the third floor of Soda Hall. This path was chosen because the auditorium contains a set of objects complex enough to differentiate the characteristics of various LOD selection algorithms. At the observer viewpoint marked 'A', many complex objects are simultaneously visible, some of which are close and appear large to the observer; at the viewpoint marked 'B', there are very few objects visible to the observer, most of which appear small; and at the viewpoint marked 'C', numerous complex objects become visible suddenly as the observer spins around quickly. We refer to these marked observer viewpoints in the analysis, as they are the viewpoints at which the differences between the various LOD selection algorithms are most pronounced.

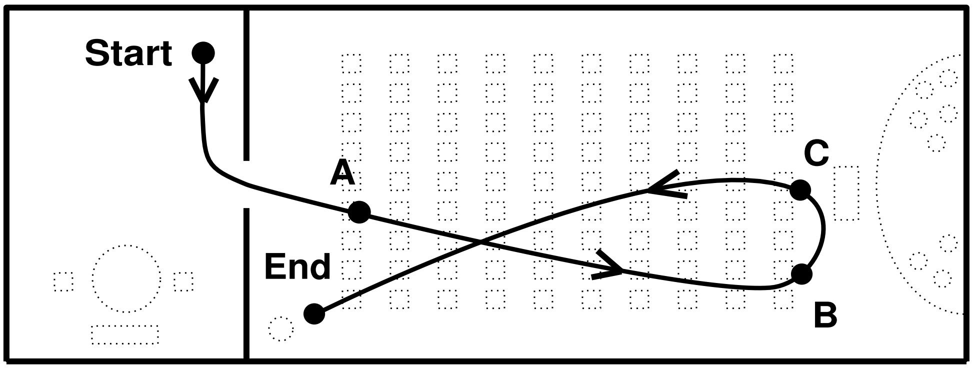


Figure 4.21: Test observer path through an auditorium on the third floor of Soda Hall.

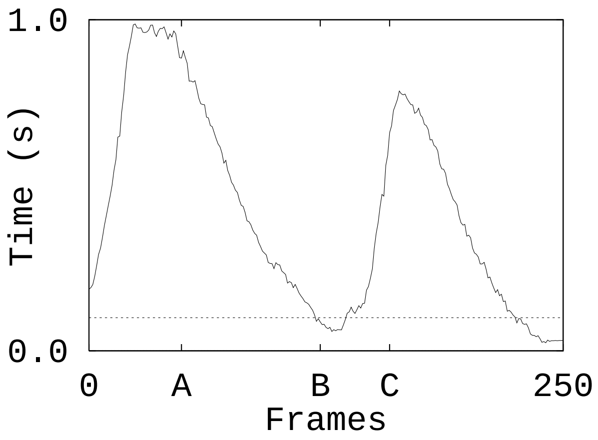
Figure 4.22 shows plots of the frame time (seconds per frame) for each observer viewpoint along the test path for the four LOD selection algorithms tested. Table 4.3 shows cumulative compute time (i.e., time required for execution of the LOD selection algorithm) and frame time statistics for all observer viewpoints along the test path.

LOD Selection Algorithm	Compute Time (s)			Frame Time (s)			
	Min	Mean	Max	Min	Mean	Max	Std. Dev.
None	0.00	0.00	0.00	0.025	0.43	0.99	0.305
Static	0.00	0.00	0.01	0.025	0.11	0.20	0.048
Feedback	0.00	0.00	0.01	0.025	0.10	0.16	0.026
Optimization	0.00	0.01	0.03	0.075	0.10	0.13	0.008

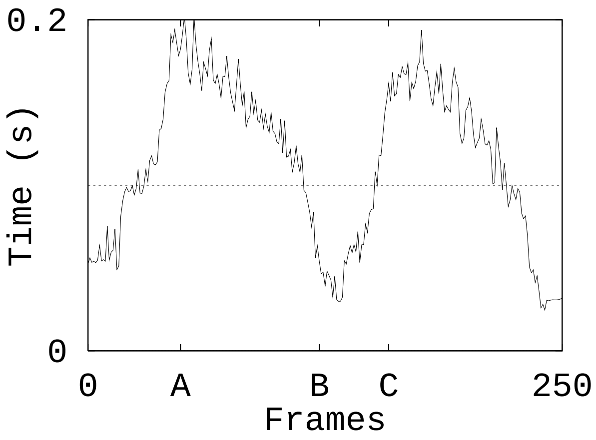
Table 4.3: Minimum, mean, and maximum timing statistics collected during tests with various detail elision algorithms (in seconds).

No Detail Elision

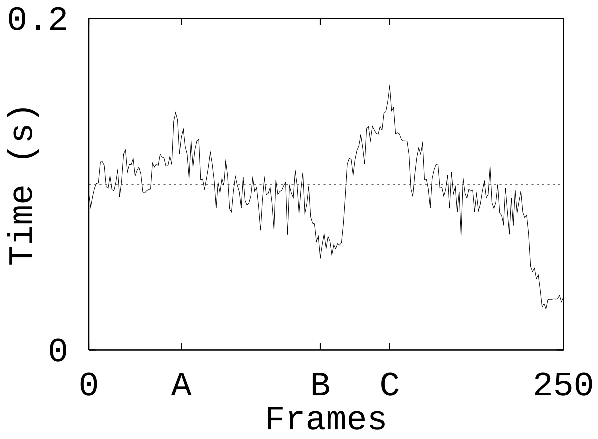
If no detail elision is used, and all potentially visible objects are rendered at the highest LOD, the time required for each frame is generally long and non-uniform, since it depends directly on the number and complexity of the objects visible to the observer (see Figure 4.22a). In our test model, far too many polygons are visible from most observer viewpoints to generate frames at interactive rates without detail elision. For instance, at the observer



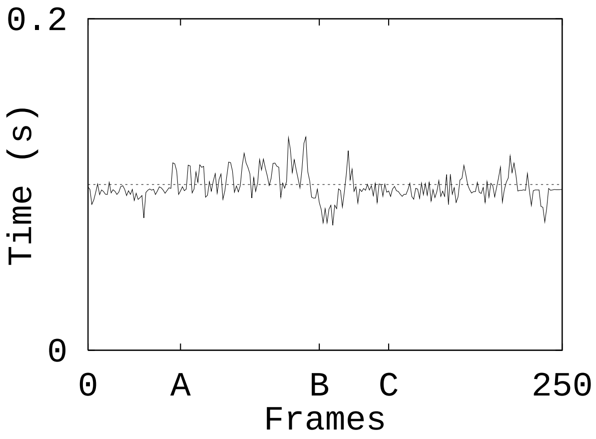
a) No detail elision.



b) *Static* algorithm.



c) *Feedback* algorithm.



d) *Optimization* algorithm.

Figure 4.22: Plots of frame time for every observer viewpoint along test observer path using a) no detail elision, b) static algorithm, c) feedback algorithm, and d) optimization algorithm. Note: the "Frame Time" axis in plot (a) is five-times larger than the others.

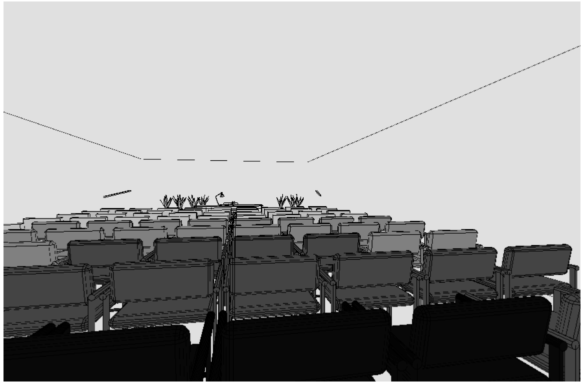
viewpoint marked 'A' in Figure 4.21, 72K polygons are simultaneously visible, and the frame time is 0.98 seconds. Overall, the mean frame time for all observer viewpoints on the test path is 0.43 seconds per frame (2.3 frames per second).

Static Algorithm

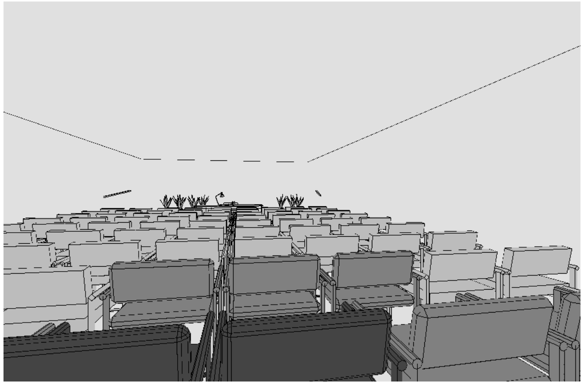
If the *Static* LOD selection algorithm is used, objects whose average polygon is smaller than a size threshold fixed at 1024 pixels per polygon are rendered with lower LODs. Even though the frame rate is much faster than without detail elision, there is still a large amount of variability in the frame time, since it depends on the size and complexity of the objects visible from the observer viewpoint (see Figure 4.22b). For instance, at the observer viewpoint marked 'A', the frame time is quite long (0.19 seconds) because many visible objects are complex and appear large to the observer. A high LOD is chosen for each of these objects independently, resulting in a long overall frame time. This result can be seen clearly in Figure 4.23a which depicts the LOD selected for each object in the frame for observer viewpoint 'A' – higher LODs are represented by darker shades of gray. On the other hand, the frame time is very short in the frame at the observer viewpoint marked 'B' (0.03 seconds). Since all visible objects appear relatively small to the observer, they are rendered at a lower LOD even though more detail could have been rendered within the target frame time. In general, it is impossible to choose a single size threshold for LOD selection that generates uniform frame times for all observer viewpoints.

Feedback Algorithm

The *Feedback* algorithm adjusts the size threshold for LOD selection adaptively based on the time taken to render previous frames in an effort to maintain a uniform frame rate. This algorithm generates a fairly uniform frame rate in situations of smoothly varying scene complexity, as evidenced by the relatively flat portions of the frame time curve shown in Figure 4.22c (frames 1–125). However, in situations where the complexity of the scene visible to the observer changes suddenly, peaks and valleys appear in the curve. Sometimes the frame time generated using the *Feedback* algorithm can be even longer than the one generated using the *Static* algorithm, as the *Feedback* algorithm is lured into an inappropriately low size threshold during times of low scene complexity. For instance, just before the viewpoint marked 'C', the observer is looking at a relatively simple scene containing just



a) *Static* algorithm



b) *Optimization* algorithm

Figure 4.23: Images depicting the LODs selected for each object at the observer viewpoints marked 'A' using the *Static* and *Optimization* algorithms. Darker shades of gray represent higher LODs.

a few objects on the stage, so frame times are very short, and the size threshold for LOD selection is reduced to zero. However, at the viewpoint marked 'C', many chairs become visible suddenly as the observer spins around quickly. Since the adaptive size threshold is set very low, inappropriately high LODs are chosen for most objects (see Figure 4.24a), resulting in a frame time of 0.16 seconds. Although the size threshold can often adapt quickly after such discontinuities in scene complexity, some effects related to this feedback control (i.e., oscillation, overshoot, and a few very slow frames) can be quite disturbing to the user.

Optimization Algorithm

The *Optimization* algorithm predicts the complexity of the model visible from the current observer viewpoint, and chooses an appropriate LOD and rendering algorithm for each object to meet the target frame time. As a result, the frame time generated using the *Optimization* algorithm is much more uniform than using any of the other LOD selection algorithms (see Figure 4.22d). For all observer viewpoints along the test path, the longest frame time is 0.13 seconds, and the shortest is 0.075 seconds, while the standard deviation in the frame time is 0.008 seconds, less than one third of any of the other three algorithms tested. The shortest frame times occur in situations where drawing every potentially visible object at the highest LOD with the most realistic rendering algorithm available still does

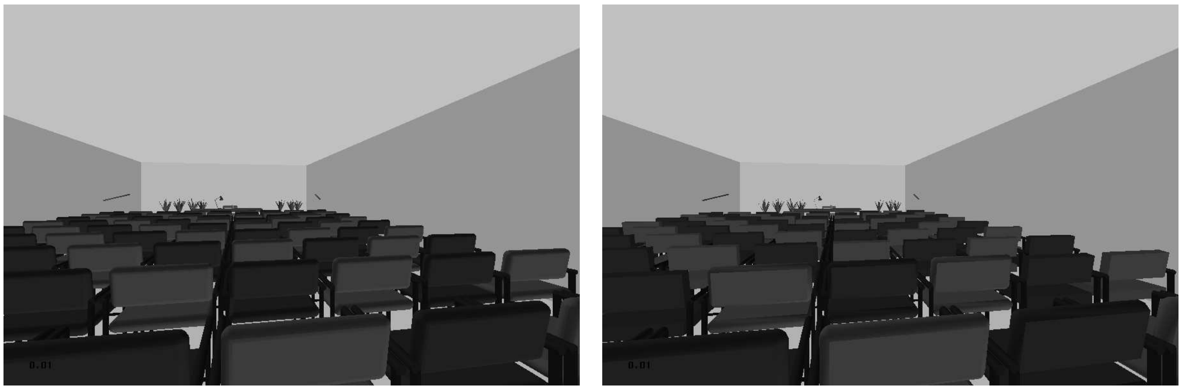
a) *Feedback* algorithm b) *Optimization* algorithm

Figure 4.24: Images depicting the LODs selected for each object at the observer viewpoints marked 'C' using the *Feedback* and *Optimization* algorithms. Darker shades of gray represent higher LODs.

not take long enough to fill the entire target frame time (e.g., Frames 125 - 150). In these cases, the system should employ even more realistic rendering algorithms (e.g., adaptive ray tracing). Currently, our system either stalls to fill the extra time (as in this test), or renders frames faster than the target frame time while adjusting the observer's step length to maintain an illusion of constant velocity navigation through the environment.

As the *Optimization* algorithm adjusts image quality to maintain a uniform, interactive frame rate, it attempts to render the "best" image possible within the target frame time for each observer viewpoint. As a result, there is usually little noticeable difference between images generated using the *Optimization* algorithm and ones generated with no detail elision at all. A comparison of images for observer viewpoint 'A' generated using a) no detail elision and b) using the *Optimization* algorithm to meet a target frame time of one tenth of a second are shown in Figure 4.25. Figure 4.25a has 72,570 polygons and took 0.98 seconds to render, whereas Figure 4.25b has 5,300 polygons and took 0.10 seconds. Even though there are less than a tenth as many polygons in Figure 4.25b, the difference in image quality is barely noticeable. For reference, the LOD chosen for each object in Figure 4.25b is shown in Figure 4.23b. Note that reduction in rendering time does not map to a linear reduction in polygon count since polygons representing lower levels of detail tend to be larger on average.

The *Optimization* algorithm is more general than other detail elision algorithms in that it also adjusts the rendering algorithm (and possibly other attributes in the future) for each



a) No detail elision b) *Optimization* algorithm (0.10 seconds)

Figure 4.25: Images for observer viewpoint 'A' generated using a) no detail elision (72,570 polygons), and b) the *Optimization* algorithm with a 0.10 second target frame time (5,300 polygons).

object independently. Examine Figures 4.26–4.29 which show four images of a small library on the sixth floor of Soda Hall containing several textured surfaces generated using a Silicon Graphics Reality Engine. Figure 4.26 shows an image without any detail elision – it contains 19,881 polygons and took 0.22 seconds to render. Figures 4.27*a*, 4.28*a*, and 4.29*a* show images generated for the same observer viewpoint using the *Optimization* algorithm with target frame times of 0.10 seconds, 0.05 seconds, and 0.02 seconds, respectively. Although the *Optimization* algorithm uses simpler levels of detail and rendering algorithms for many objects (see Figures 4.27*b*, 4.28*b*, and 4.29*b*), and generates images that are quite different from the one generated with no detail elision (see Figures 4.27*c*, 4.28*c* and 4.29*c*), all three images look very similar. Notice the reduced tessellation of chairs further from the observer, and the omission of texture on the bookshelves in Figure 4.28*a*. In Figure 4.29, texture has been removed from all surfaces except the large polygons that make up the floor and ceiling. Since the *Optimization* algorithm uses reduced levels of detail and rendering algorithms for only the least important objects, differences between the three images are not very noticeable, even though rendering times differ by a factor of ten.

4.2.5 Discussion

The *Optimization* algorithm maintains a more uniform frame rate and produces higher quality images than other detail elision algorithms. However, it is particularly prone to

Figure 4.26: Image of library generated using no detail elision (19,821 polygons).

hysteresis during interactive visualization since it varies LODs and rendering algorithms of potentially visible objects to fill the *entire target frame time* during every frame. As objects become newly invisible (or visible) during an interactive walkthrough, there becomes more (or less) frame time available to render other potentially visible objects. The LOD or rendering algorithm used for some object(s) may be increased (or decreased) to fill the available frame time. In this manner, the LOD and rendering algorithms used for each object may vary according to the number and complexity of other objects visible to the observer. Therefore, although the frame rate is kept nearly constant using the *Optimization* algorithm, the LODs and rendering algorithms used for some objects may change from frame to frame.

This effect can be minimized by adjusting the *Hysteresis* factor of the *Benefit* heuristic. Decreasing the benefit of objects whose LOD or rendering algorithm is different than the one used to render the object in the previous frame directs the *Optimization* algorithm to favor fewer LOD and rendering algorithm switches between successive frames. However, since the algorithm always fills the entire target frame time in each frame, the LOD or rendering algorithm of some object is necessarily switched whenever an object becomes newly visible or invisible. This effect is disturbing when a few objects are visible to the observer continuously, while others become visible and invisible intermittantly. Possible solutions to this problem might be to modify the *Optimization* algorithm to require objects to be rendered with each set of rendering attributes for a minimum number of frames,

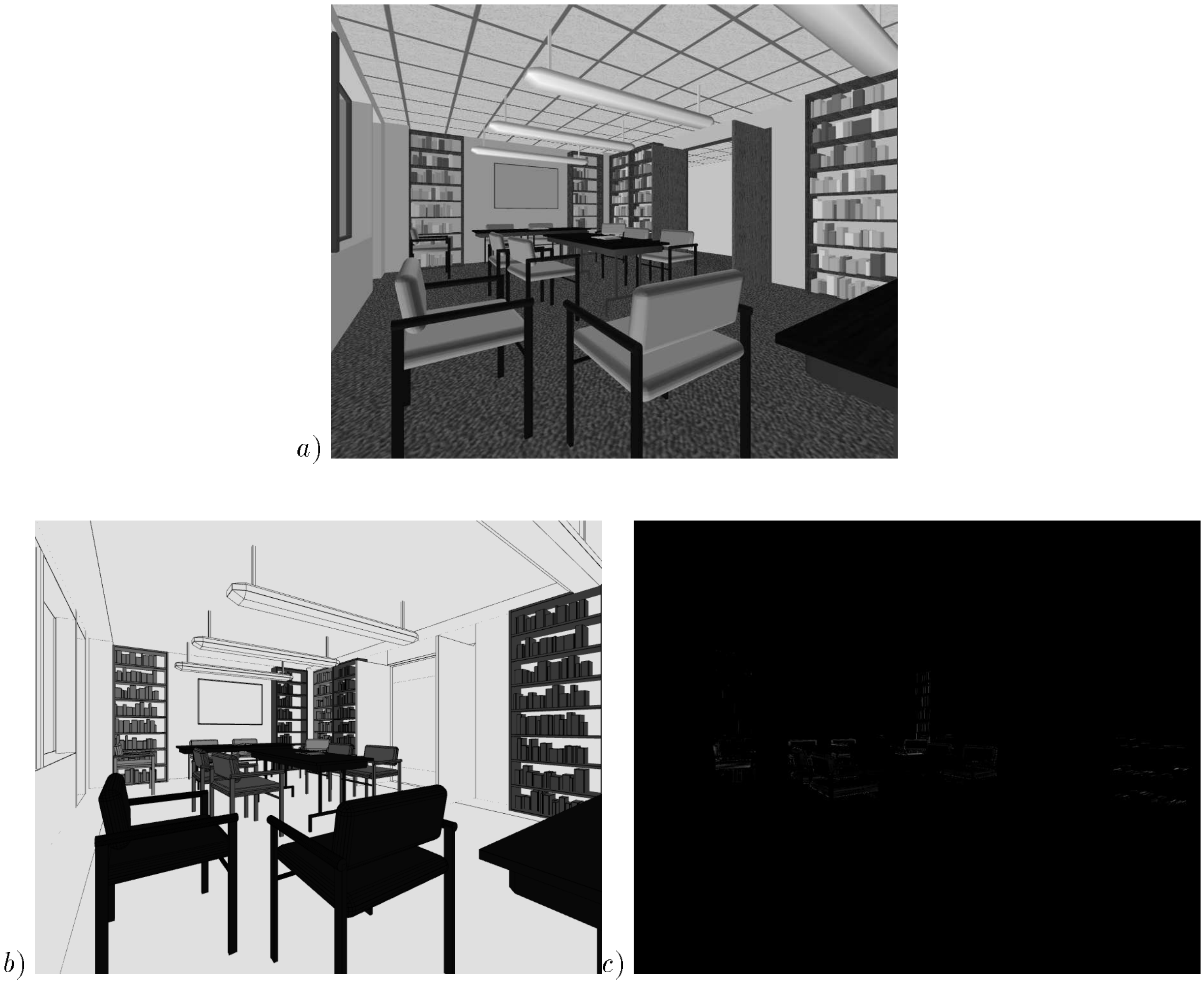


Figure 4.27: Images of library generated using the *Optimization* detail elision algorithm with a target frame time of 0.10 seconds (8,882 polygons). LODs chosen for objects in (a) are shown in (b) – darker shades of gray represent higher LODs. Pixel-by-pixel differences from Figure 4.26 are shown in (c) – brighter colors represent greater difference.

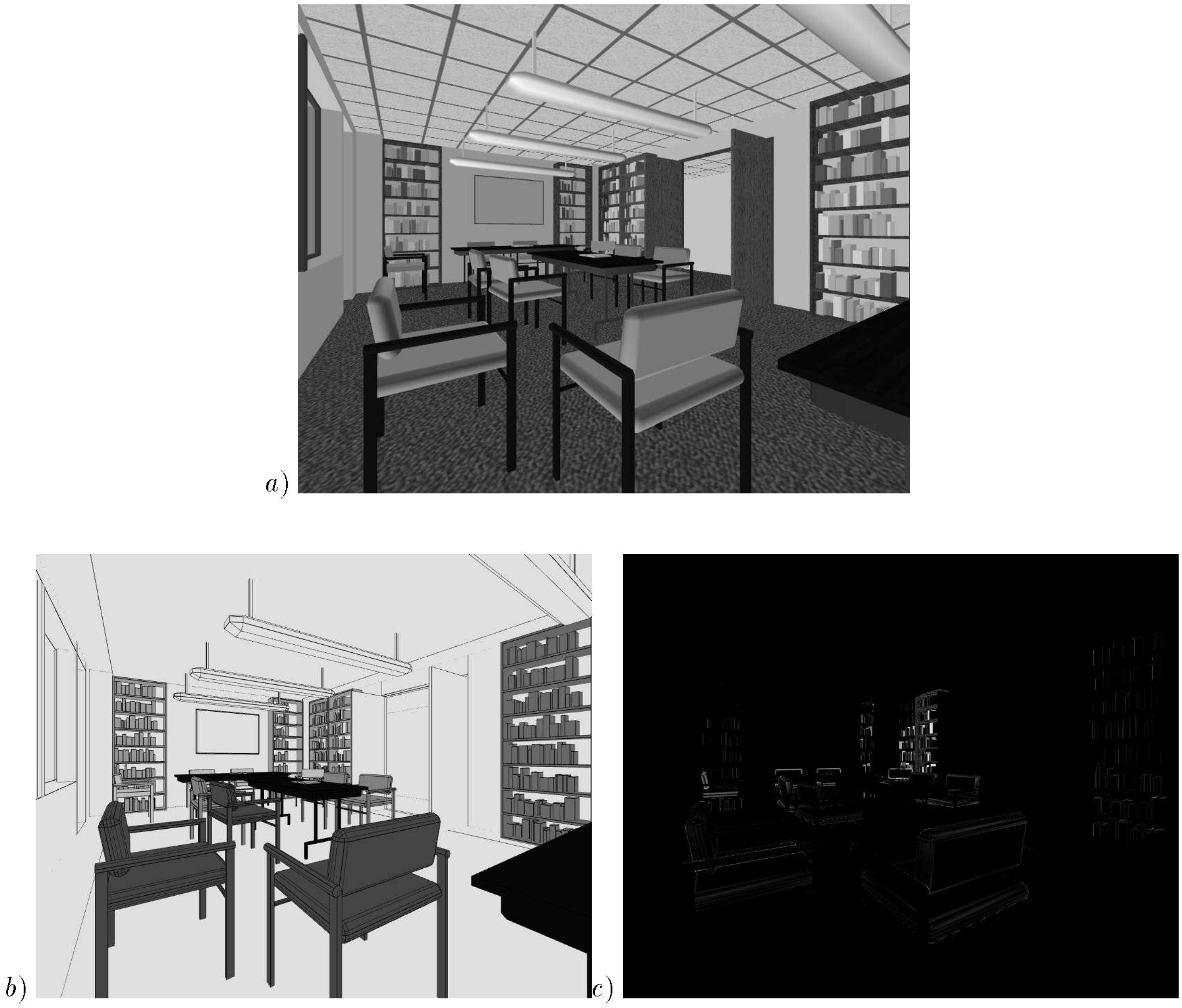
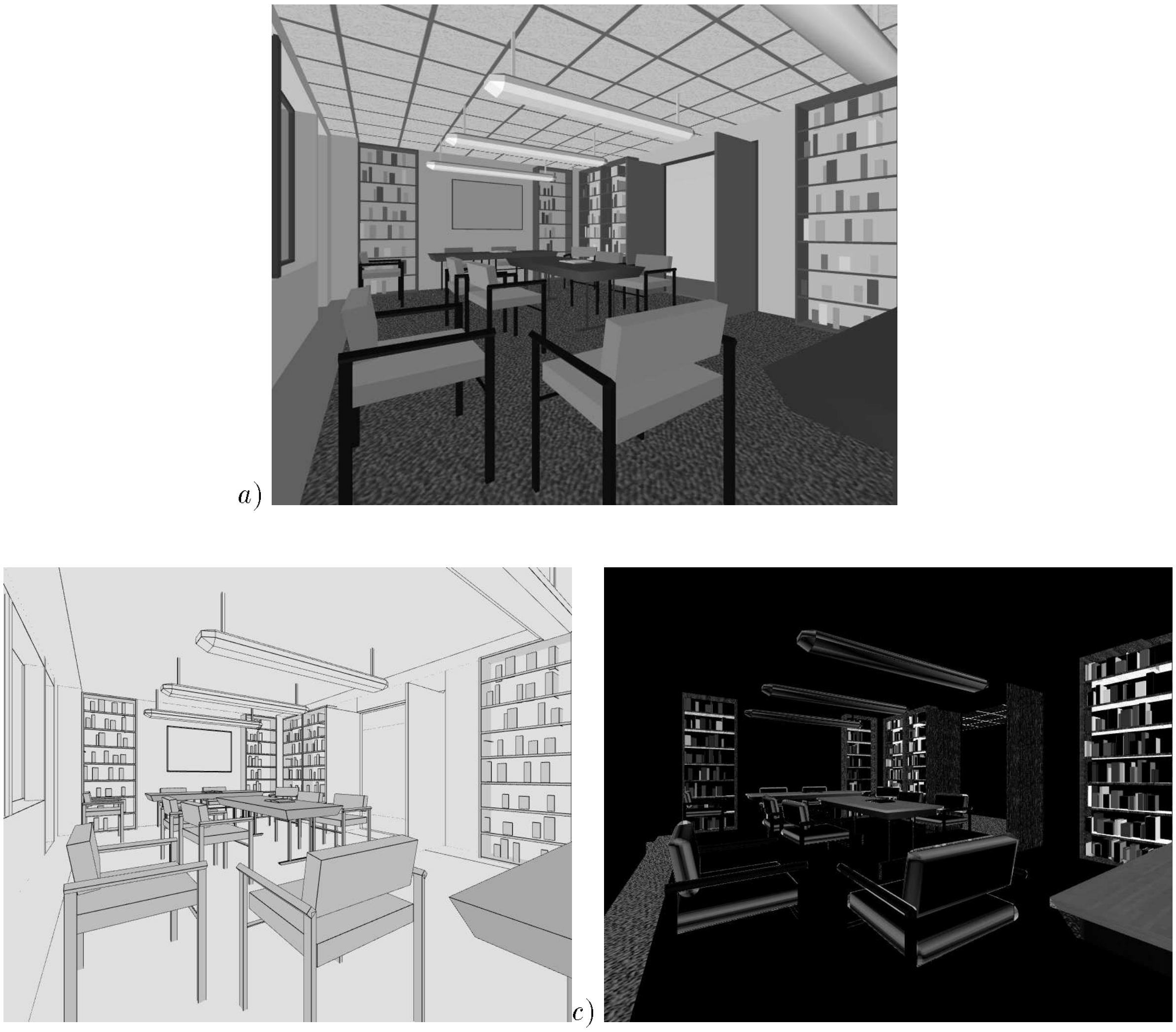


Figure 4.28: Images of library generated using the *Optimization* detail elision algorithm with a target frame time of 0.05 seconds (3,568 polygons). LODs chosen for objects in (a) are shown in (b) – darker shades of gray represent higher LODs. Pixel-by-pixel differences from Figure 4.26 are shown in (c) – brighter colors represent greater difference.



a) b) c)

Figure 4.29: Images of library generated using the *Optimization* detail elision algorithm with a target frame time of 0.02 seconds (1,161 polygons). LODs chosen for objects in (a) are shown in (b) – all objects are rendered using the lowest LOD. Pixel-by-pixel differences from Figure 4.26 are shown in (c) – brighter colors represent greater difference.

to allow a range of target frame times, or to perform an unconstrained optimization which includes consideration for variation from the target frame time in the optimization objective function. Further experimentation is required to determine the merit of these possible approaches.

In order to make switches between different LODs or rendering algorithms less noticeable, we plan to use a transparency blending technique in which objects in transition between successive representations, A and B, are rendered twice: once with representation A, and once with representation B. The resulting images are blended pixel-by-pixel using transparency factors which sum to one, and then painted into the frame buffer. We have not yet included this feature in our rendering system because the hardware we currently use does not support the necessary blending operation (more recently available hardware does).

To include transparency blending in our system, we must extend our *Optimization* algorithm to consider object tuples (O, L, R) in transition between successive LODs or rendering algorithms – i.e., with "non-integral" values for L and R. Although few conceptual changes are necessary to support transparency blending, the optimization algorithm must be extended to increment and decrement rendering attributes by fractional values; the *Cost* heuristic must be updated to account for the time required to draw an object in transition (i.e., rendered twice and then blended); and the *Benefit* heuristic must be modified to consider the "contribution" of rendering a blended object tuple image. Although these extensions are possible in principle, further research is required to evaluate the effects of blending on the *Optimization* detail elision algorithm.

There are still many unanswered questions regarding detail elision for interactive visualization applications. However, having experimented with several LOD selection algorithms, we are optimistic that variation in image quality is less disturbing to a user than variation in frame times, as long as different representations for each object appear similar, and transitions between representations are neither very frequent nor noticeable.

Chapter 5

Memory Management

Detailed building models with radiosity data are too large to fit into main memory all at once on typical graphics workstations. Therefore, we must choose a subset of the model to store in memory, and swap different portions of the model in and out of memory in real-time as the observer moves through the model. We always must store in memory at least the portion of the model to be rendered in the current frame. However, since it takes a relatively large amount of time to load data from disk into memory, we also must predict which parts of the model might be rendered in future frames and begin loading them into memory in advance. An extremely efficient database system running under asynchronous control must be used for loading object descriptions from disk into memory in real-time. Otherwise, frame updates might be delayed, waiting for data to be read from disk before it can be rendered.

5.1 Database System

The challenge of a database system for interactive visualization applications is to provide efficient storage and real-time access to the complex, dynamic data structures required for display of large three dimensional models at interactive frame rates.

Most visualization systems store three dimensional models in graphical display lists under the assumption that the models fit entirely into physical memory of the workstation. If the size of a graphical model exceeds the capacity of physical memory, the operating system uses standard virtual memory techniques to swap portions of the model out to disk. Unfortunately, this hands-off approach to memory management is often inadequate for visualization of very large models for several reasons. First, references to data in a virtual memory system are generally unique to a specific address space, so cannot be stored on disk and then used later during a different program execution. Second, data elements may grow in size, and consequently be forced to move in the virtual address space. Since references are only via direct pointers in virtual memory, all references to a data element must be found and updated whenever a data element is moved. Such updates are both inefficient and difficult to implement unless explicit back-references are stored, which is often impractical. Third, many small blocks of memory allocated for related data elements may be scattered over the virtual memory address space resulting in poor physical memory utilization and swapping performance. Finally, virtual memory provides no synchronization point to control simultaneous access to data elements by separate concurrent processes. Therefore, display lists stored in virtual memory are most practical for access by a single process to static models that fit in memory entirely.

To manage storage of large, dynamic three dimensional models in a visualization system, a more complete database system is required that allows data elements to be modified, moved, and accessed by separate processes concurrently. Such a database system must perform many operations standard in traditional database systems, such as buffer pool management, clustering of related data elements, concurrency control, and persistent updates. However, other functional and performance requirements of a database system for interactive visualization applications are quite different than for traditional database applications.

In visualization applications, data location typically is performed by functional queries that traverse sequences of references through data stored in the database. During every second of execution, the application may perform hundreds of thousands of references and execute tens of thousands of functions on data stored in the database (e.g., rendering a polygon requires references to data stored in the database). Some of these functions may be recursive, executed on another processor asynchronously, or depend on special-purpose hardware (e.g., the graphics engine). Therefore, general purpose query techniques requiring parsing a query language, executing a query engine, copying data into application buffers, converting data between database and application formats, or executing functions in the database system address space are not flexible or fast enough for visualization applications. Even the overhead of a single function call or "if" check per reference is too inefficient. Instead, arbitrary application-defined functions must be allowed to access data stored in the database memory pool directly, by the most efficient means possible in the programming language (i.e., direct pointer dereference).

As the user explores a large three dimensional model, the data describing the portions not already resident in memory must be loaded from disk in real-time while the system continues to render images at interactive frame rates. Since the portion of the database required to be in memory can be highly dynamic and difficult to predict far in advance, memory management functions that pre-fetch data into memory must be defined by the application, execute asynchronously, and support very high performance transfers of data from disk into memory – i.e., must have both low latency and high throughput. Therefore, very little dynamic memory allocation or data replication can be done during disk to memory transfers, and asynchronous read operations must be used so that frame refreshes are not delayed waiting for slow disk i/o operations to complete.

In all, the essential requirements of a database system for three dimensional visualization applications are: 1) store very large models (larger than will fit into main memory); 2) support persistent addition, deletion, and modification of data; 3) support efficient access to data by application-defined functions; 4) allow asynchronous, application-defined memory management functions; and 5) perform efficient transfers from disk into memory.

Most traditional database systems do not provide the necessary combination of extensibility, flexibility and performance to be used in high-performance, interactive visualization applications. This is partially due to support for a multitude of services that are not typically essential in a visualization application, such as crash recovery, security, and general-purpose queries. Many of these features reduce database system performance significantly: 1) crash recovery requires logging updates to disk, 2) security requires application-defined functions to run outside the database system's address space, and 3) general purpose queries require execution of a query engine and often copy query results into application buffers.

We have developed a database system designed specifically for interactive visualization applications. It is NOT a general purpose database management system. Instead, it is a low-level storage system that provides management of arbitrarily moving/growing groups of bytes in a persistent database file. High-level database features and policies, such as crash recovery, security, and query execution are left to the application programmer so that no unnecessary overhead is incurred by the database system. In this regard, our database system is similar to Exodus [14] and Genesis [6] – it is a modular, low-level storage system suitable for use directly by an application program, or by a complete object-oriented database system.

5.1.1 Segments

Our database system stores persistent application data in *segments*, which contain arbitrary groups of bytes that are manipulated by the database system as a unit. The content of a segment is completely application-defined, and is not interpreted by the database system. So, any data structure, including pointers, may be stored inside a persistent segment and managed by the database system.

An important aspect of our database system is that application-defined functions can access data stored in memory resident segments directly via pointers in the native programming language. This feature is important for several reasons. First, applications can access persistent data stored in the database very efficiently. No copy into an application buffer (e.g., [38]), or indirection via an object identifier (e.g., [19]) is required.

Second, database queries can be executed directly by the application using functions defined in the native programming language, rather than by the database system using a query language. As a result, greater expressive power is available to the application. Queries can be iterative or recursive, partitioned among multiple processors, take advantage of application-specific semantics, or utilize special-purpose hardware. For instance, in our building walkthrough application, we execute a query to "Find all objects potentially visible to the observer" during every frame of an interactive walkthrough (see Section 4.1). This query requires execution of a recursive search of the cell-adjacency graph to find cells potentially visible to the observer, and performs a linear program for each object incident upon a potentially visible cell. In future versions of our system, the query may search different paths through the cell-adjacency graph in parallel, or render polygons directly. Clearly, the native programming language is better suited for this type of query than a traditional database query language.

Third, application code must not be rewritten to access data stored in persistent database segments. Visualization applications often are written using main memory data structures first, and then converted to use persistent storage of a database system. In our case, the display management algorithms described in Chapter 4 were developed in the C programming language using small, memory resident models. Later, when we began using building models that were too large to fit into memory, the visualization data structures were mapped directly into database segments, and no existing code had to be rewritten or modified at all.

Another important feature of our database system is that memory allocation and I/O operations are performed only for entire segments, which may contain numerous related data elements, rather than for individual data elements separately. For example, all polygons representing the same object at the same level of detail might be grouped into a single segment. Then, only one memory allocation and read operation is required to transfer many polygons from disk into memory, rather than suffering the latency resulting from a separate memory allocation and read operation for each polygon. This feature improves performance and simplifies data management significantly.

Using the segment abstraction, database management gets divided into two parts: 1) the application part that defines the data types to be stored in each segment and uses segment operations to dictate high-level memory management policies and 2) the database system part that manages low-level storage of persistent segment data and references.

5.1.2 Data Types

In order to use our database system, a programmer must partition application data structures into segments. Since segments reside either entirely in memory or entirely on disk, and can grow in size and/or move, there are certain rules and guidelines an application programmer must follow when partitioning data into segment types. First, application data pointers are *required* to reference addresses that are either: 1) within the same segment (i.e., *intra-segment references*) or 2) the base address of another segment (i.e., *inter-segment references*) and MUST NOT reference addresses internal to another segment. This requirement, shown in Figure 5.1, is necessary to insure that every data reference can be stored in a single pointer field (e.g. four bytes), regardless of whether the referenced data moves or is resident in memory or on disk. If references to addresses internal to other segments were allowed, separate segment address and offset fields would be required for each reference, thereby forcing the application and/or the database system to allocate and manage extra meta-data for each reference.

Second, application programmers are *advised* to group related data elements into segments together. Since disk I/O may be more efficient for larger segments, all data elements that are likely to be needed in memory at the same time should be stored in the same

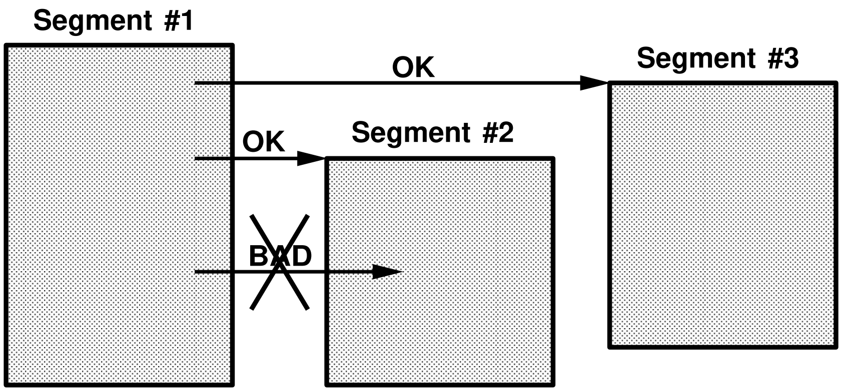


Figure 5.1: References to addresses internal to other segments are not allowed.

segment. However, since all data elements stored in a segment must be read and stored in memory together as a unit, I/O performance and memory utilization is compromised if unrelated data elements are grouped together in the same segment inappropriately.

Finally, application programmers are *advised* to store data elements that are likely to grow (e.g. lists) in segments separate from data that is likely to remain fixed in size. Since space in the database file and/or memory must be re-allocated for an entire segment whenever any data element stored inside the segment grows, storing data that is likely to grow in a separate segment can have significant space utilization advantages.

For example, the data structures used in our building walkthrough application are partitioned into segments as shown in Figure 5.2. Note that many static, related data elements, such as the polygons describing an object at a particular LOD, are grouped into a single segment. However, dynamic data elements, such as the list of objects incident upon a cell, are stored in separate segments since they are likely to grow and shrink as objects are added, deleted, or moved in the database.

An application programmer specifies how data structures are partitioned into segments by defining a set of *segment types*, with six *callback* functions for each type:

DFCreateSegmentData (void *data)

Initialize segment data at address *data*.

DFDeleteSegmentData (void *data)

Delete segment data at address *data*.

DFPackSegmentData (void *src, void *dst)

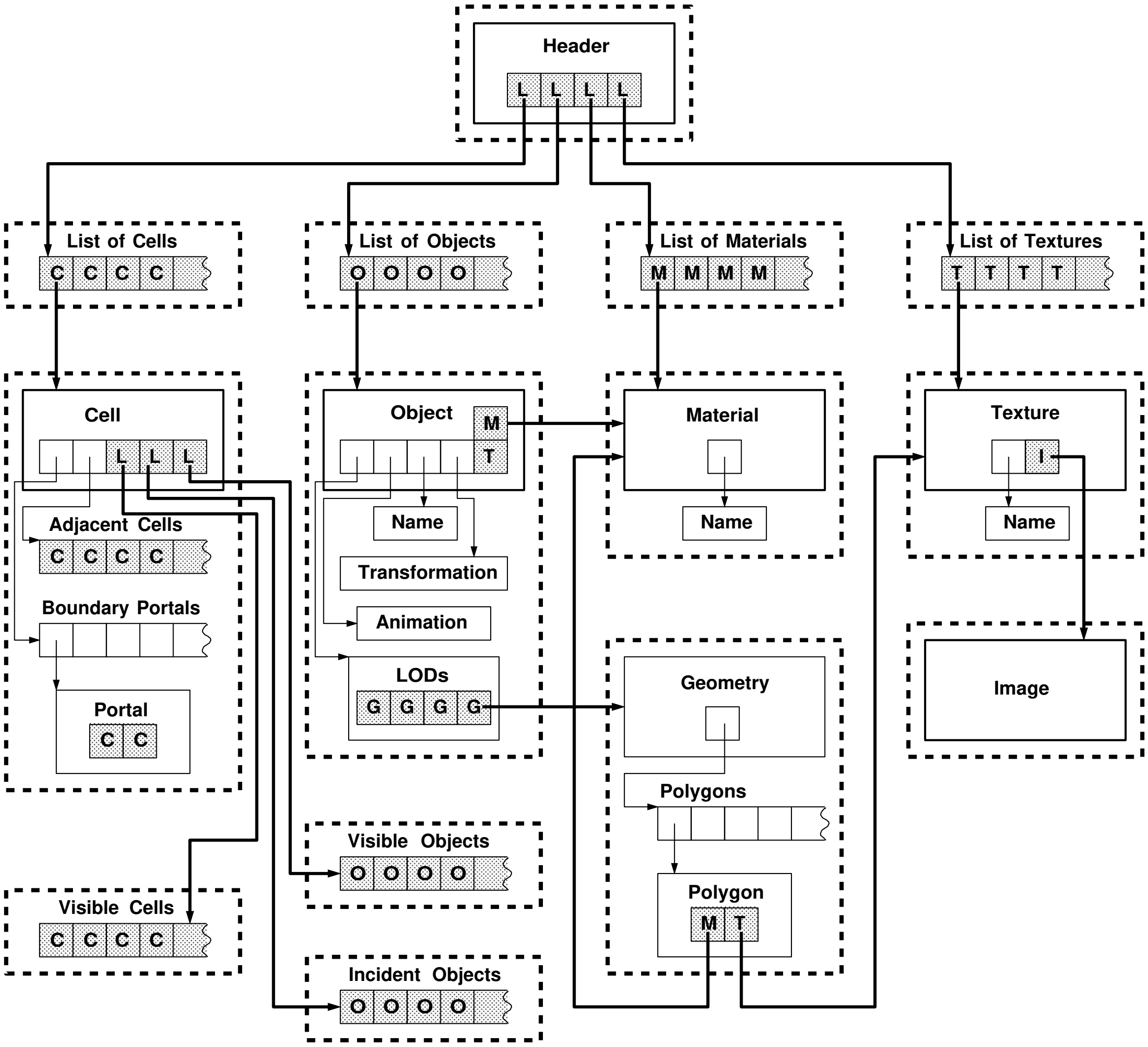


Figure 5.2: The building walkthrough data structures partitioned into segments. Segment boundaries are represented by thick, dashed lines. Inter-segment references are depicted by stipple gray squares labeled by the type of segment referenced (e.g., 'C' = Cell, 'O' = Object, 'G' = Geometry, 'M' = Material, 'T' = Texture, 'I' = Image, and 'L' = List). Intra-segment references are depicted by hollow squares.

Copy segment data at address *src* into contiguous memory at address *dst*.

DFUnpackSegmentData (void *src, void *dst)

Copy segment data at address *src* into segment data at address *dst*.

DFFixupSegmentData (void *data, DF_FIXUP_FUNCTION before, DF_FIXUP_FUNCTION after, DF_FIXUP_FUNCTION external)

Apply fixup functions to all references within the segment data at address *data*. The *before* and *after* functions are applied to each intra-segment reference before (after) the reference is used. The *external* function is applied to each inter-segment reference.

int DFGetSegmentDataSize (void *data)

Return the size of the segment data at address *data* in bytes.

These six callback functions provide an abstraction used by the database system to manage the application-dependent data of each segment type in an application-independent manner. The most important aspect of this abstraction is the conversion of references (i.e., pointers) stored in the application data structures as data is transferred between memory and disk. In particular, inter-segment references must be converted to direct memory pointers when segment data is read from disk, and converted back to *unique segment ids* (OIDs) when segment data is written to disk (i.e., swizzled). This conversion is accomplished via the DFFixupSegmentData callback in which the application calls specified functions for each intra- and inter-segment reference contained in the segment's data.

Like a data description language [19, 13] or tag table [37], this callback mechanism allows the database system to manipulate data and references in application-defined data types transparently. However, callbacks are more general than these previously described declarative methods because data manipulation is procedural. Reference fixup can be conditional, iterative, or even recursive, and depend on values of application-specific data. Therefore, very complex data structures can be stored in a single segment type (e.g., a cell with lists of its neighbors and portals), and segment types can be described in any programming language (e.g., C).

5.1.3 Operations

The database system supports a simple set of operations to create, open, and close database files, and to create, delete, and manipulate database segments:

Database Operations:

DFNewDatabase(DF_DATABASE *database, char *name)

Creates and initializes a new database.

DFOpenDatabase(DF_DATABASE *database, char *name)

Opens an existing database.

DFCloseDatabase(DF_DATABASE *database)

Closes an open database. Any dirty, memory resident segments are rewritten to the database disk file.

Segment Operations:

DFCreateSegment(DF_DATABASE *database, DF_SEGMENT *segment, DF_SEGMENT_TYPE type, void *data)

Creates a segment with the specified type and data, and adds it to the database. New memory is allocated for the segment's data, and the data passed as an argument is copied into it.

DFDeleteSegment(DF_DATABASE *database, DF_SEGMENT *segment)

Removes a segment from the database.

DFWriteSegment(DF_DATABASE *database, DF_SEGMENT *segment)

Writes a segment's data from memory into the database disk file.

DFReadSegment(DF_DATABASE *database, DF_SEGMENT *segment)

Reads a segment's data from the database disk file into memory.

DFReleaseSegment(DF_DATABASE *database, DF_SEGMENT *segment)

Releases a segment's data from memory.

The DFReadSegment, DFWriteSegment, and DFReleaseSegment operations control which segments are resident in memory at any given time. Unlike some object-oriented database systems, which at most allow an application to provide hints regarding buffer pool management [14], our database system allows an application to control segment loading and replacement policies explicitly. For example, our building walkthrough application uses these operations to implement a rather complex pre-fetch memory management algorithm (see Section 5.2). Read, write and release operations access shared memory synchronized by locks, and segment loading is executed from a separate, asynchronous process in order to avoid waiting for slow disk i/o operations during image generation.

5.1.4 Implementation

The database system uses memory allocation and file input/output features provided by the operating system, along with the application-defined callback functions described in Section 5.1.2, to implement the i/o operations described in Section 5.1.3.

The database system stores all data in the database in a single file, as shown in Figure 5.3. The first 512 bytes of the database file are reserved for the *database prologue*, which stores the application-independent state of the database (e.g., magic number, version number, revision number, and text descriptor) and references to segments containing the *database header* and the *segment index*. The remainder of the database file is used to store application segment data.

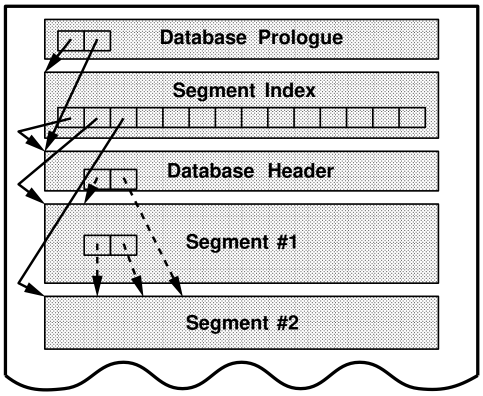


Figure 5.3: Layout of database file. Mandatory inter-segment references are shown as solid arrows, while possible application-defined inter-segment references are shown as dashed-arrows.

The *database header* is a segment, which is specified by the application as the top-level access point to the data stored in the database. It is possible for a database to contain only one segment, i.e. the database header. However, the database system is intended to be used such that the database header contains references to other segments, which contain references to still other segments, etc. In this way, a complex structure of segments and references can be built, originating from the data stored in the database header segment.

The *segment index* is a segment that contains an array of *entries* that store the state of all other segments in the database. Each segment index entry contains the following fields:

- **OID** - specifies the segment's unique integer identifier. The OID for the segment associated with the i*th* segment index entry is constructed by $OID_i = (i << 1)|1$, which allows a segment's index entry to be located from its OID quickly, and allows a segment OID to be distinguished from a pointer, since pointers are constrained to reference only even memory addresses in our implementation.

- **Type** - specifies the segment's application-defined type. As described in Section 5.1.2, the segment type indicates which set of application-defined callback functions should be called by the database system during manipulation of the segment.

- **Size** - specifies the number of bytes reserved for the segment's data in the database file. If the segment's data is resident in memory, its actual size can exceed the number of bytes reserved in the database file. In this case, a new location in the database file is found when the segment is rewritten to the database file.

- **File offset** - specifies the location reserved for the segment's data in the database file. The size and file offset fields together indicate the space set aside for the segment's data in the database file.

- **Memory pointer** - specifies the memory address of the segment's data, if the segment is resident in memory.

- **Reference count** - specifies whether or not a segment's data is resident in memory. The reference count starts at zero, and is incremented each time the segment is "read," and decremented each time it is "released." If the reference count is greater than zero, the segment's data is resident in memory.

- **Status** - specifies the segment's status. This field currently contains bits indicating whether the segment data is "dirty" (i.e., needs to be updated on disk) or "packed" (i.e., contiguous in memory).

A segment's data can either be memory resident or not, as indicated by the *reference count* field. If a segment's data is memory resident, the *memory pointer* field specifies the location of the segment's data in virtual memory, and the first four bytes of the segment's data contain a pointer to the segment index entry. This back-pointer is used to convert direct memory references back to OIDs when a segment is written to or released from memory. If a segment's data is not memory resident, the *file offset* field specifies the segment's data location in the database disk file, and the memory pointer is NULL.

The layout of segment data on disk is the same as in memory, so transfers from disk into memory are very efficient. In order to read a segment's data from disk into memory, the database system performs the following: 1) allocates memory for the segment data (using `malloc`), 2) executes a `read` system call, and 3) swizzles intra-segment references into direct memory pointers using the DFFixupSegmentData application-defined callback function. Inter-segment references are not swizzled during read operations by default – the referenced segments may or may not be read later by the application. In order to write a segment's data from memory into the database disk file, the database system performs the following: 1) allocates space in the database disk file (preferably at the same location from which it was read initially), 2) packs the segment data into contiguous memory using the application-defined DFPackSegmentData callback function, 3) converts each inter-segment reference into an appropriate OID and each intra-segment reference into an offset from the beginning of the segment using the DFFixupSegmentData application-defined callback function, and 4) executes a `write` system call. Reference counts and dirty bits are maintained by the database system in order to avoid redundant or unnecessary read/write operations.

Note that our database system reads and writes data in segment-size blocks, rather than page-size blocks like many object-oriented database management systems [14, 37, 19]. Page-size blocks are preferable when only a small portion of a segment must be accessed, whereas segment-sized blocks are preferable when entire segments are accessed each time any part is, as long as greater performance can be derived from larger i/o operations.

5.2 Predictive Memory Management

Since realistic-looking three dimensional models may be much larger than can fit into main memory, 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. However, since it takes a relatively large amount of time to load data from disk into memory, we must prefetch parts of the model that might be rendered in future frames. Consequently, we have implemented a predictive memory management algorithm that forecasts a range of possible observer viewpoints during the next N future 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 future 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.

5.2.1 Observer Range

An ideal memory management algorithm predicts the observer viewpoint in each future frame perfectly. Then it can use the visibility determination and detail elision algorithms described in Chapters 4.1 and 4.2 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.

Given a particular observer viewpoint in the current frame and constraints on observer movement and rotation enforced by the user interface, we can determine an *observer range* that contains a superset of all observer viewpoints possible during the next N future frames. For example, if the observer is allowed to move and turn in any direction, but is constrained by maximum positional and rotational velocities, v_p and v_r, the upper bound on the observer range during the next N frames is a sphere centered at the observer eye position with radius Nv_p, as shown in Figure 5.4. All possible observer view directions are enclosed in a *range frustum* whose eye position is directly behind the observer, and whose view angle is widened by Nv_r, and which contains the range sphere. If the observer is prevented from moving directly through solid walls (a parameter in our user interface), the observer range is further constrained as shown in Figure 5.5.

5.2.2 Object Lookahead

In order to pre-fetch objects into memory before they are rendered, we must determine which objects are likely to become visible to the observer in advance. This observation suggests a predictive memory management algorithm similar to the visibility determination algorithm described in Section 4.1, but extended to compute visibility for a range of possible future observer viewpoints. Such an algorithm might cast visibility beams from the observer range through the portals of the spatial subdivision to determine a set of potentially visible objects. Unfortunately, real-time visibility determination for a finite, non-zero volume of

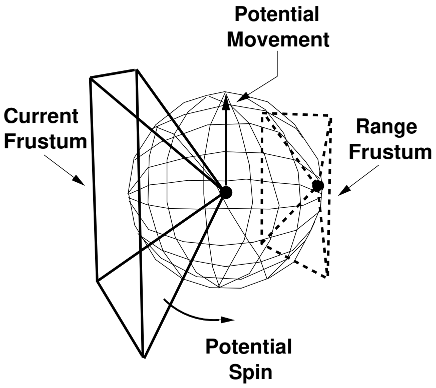


Figure 5.4: The observer range contains all observer view positions (inside sphere) and view directions (inside range frustum) possible during the upcoming N frames.

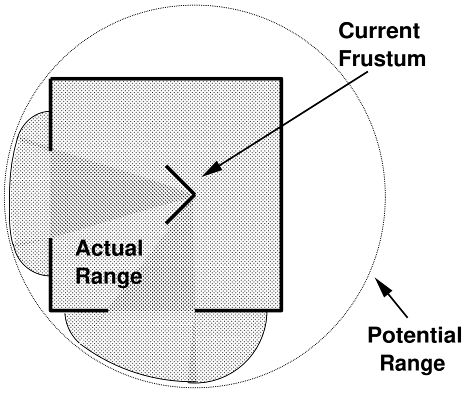


Figure 5.5: Observer range is reduced if the user interface prevents traversal through solid walls.

space (even an axial box) seems to be too compute intensive to be practical for interactive walkthroughs [55].

Fortunately, memory management does not typically require as exact a visibility solution as display management. During display management, the bandwidth of the graphics rendering system is generally the limiting resource. Typically, a system must cull away over 99% of a detailed building model in order to generate images at interactive frame rates. In addition, each potentially visible object must be passed through the graphics rendering system in every frame. Therefore, it is extremely important that the set of objects selected for display comprise a very small superset of the objects actually visible to the observer.

In contrast, during memory management, the capacity of workstation memory and the bandwidth of disk to memory transfers are generally the limiting factors. Typically, a workstation must cull away only 90% of a detailed building model to select a portion that fits into memory. Furthermore, potentially visible objects must be loaded into memory only once, and then can be resident in memory for several frames. Therefore, it is sufficient, and even advantageous, for the set of objects stored in memory to be a rather large superset of the objects potentially visible to the observer range, as long as it is a small enough subset of the entire model to fit in main memory and the difference between successive frames is small.

Our visibility determination algorithm for memory management finds a set of cells whose cumulative boundary contains the observer range. It then utilizes precomputed cell-to-cell and cell-to-object visibility information for those range cells to compute a superset of the objects potentially visible from the observer range.

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 future frames by performing a shortest path search of the cell adjacency graph. The search, implemented using Dijkstra's method [21], adds cells to the range set in order of 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.

Since there is a large amount of coherence in observer motion from frame to frame, we use the current direction of observer movement to help predict future observer eye positions. During the shortest path search, we add cells to the range set only if they satisfy certain directional constraints specified by the *range cull algorithm* parameter. Currently supported range cull algorithms include: 1) *any direction* – all cells that can contain the observer in the next N frames are included in the range set, 2) *observer plane* – range cells must be at least partially in front of the observer, 3) *observer frustum* – range cells must be at least partially inside the current observer frustum, and 4) *observer direction* – range cells must be hit by a ray traced from the current observer eye position along the current observer view direction.

Figure 5.6 shows an example shortest path search result in which the observer range is not limited to any particular direction. 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.

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 future frames. When a new range cell is discovered during the shortest path search, we check each cell in its cell-to-cell visibility to see if the cell satisfies the directional constraints specified by the *lookahead cull algorithm* (similar to the range cull algorithm). If so, we add all potentially visible objects incident upon the cell to the lookahead set. The *lookahead granularity* parameter specifies whether we add all objects incident upon the cell to the lookahead set (i.e., *cell granularity*), or only

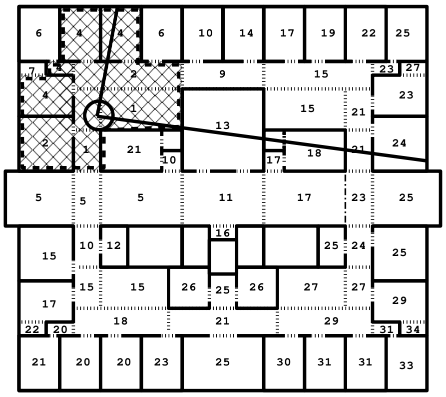


Figure 5.6: The observer range cells (shown in cross-hatch) contain all observer view positions possible during the upcoming $N = 4$ frames. Each cell is labeled by the number of frames before the observer can be resident in it.

the objects in the cell-to-object visibility of the range cell (i.e., *object granularity*).

Figure 5.7 shows an example computation of the lookahead set of objects in which both the range cull algorithm and lookahead cull algorithm are *any direction*, and the lookahead granularity is *cell granularity*. 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.

Figure 5.7: The lookahead cells (shown in 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.

As each object is added to the lookahead set, we mark and claim memory for all LODs for the object that can possibly be rendered during the next N future frames. We use a size threshold for static detail elision, along with precomputed information regarding which objects can be drawn at a given LOD for an observer inside a particular cell, to choose a maximum LOD at which to store each potentially visible object. The effect is that objects near the observer range are stored in memory up to higher LODs than ones further away, as shown in Figure 5.8.

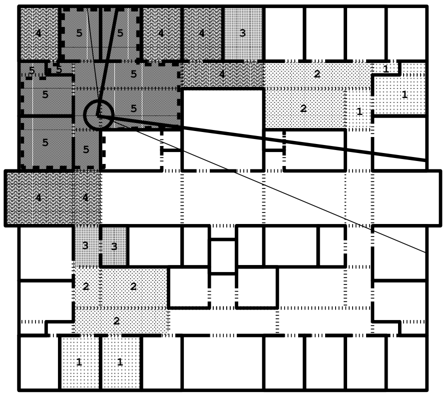


Figure 5.8: Lookahead objects are stored in memory only up to the LOD at which they can possibly be rendered during the next N frames. Each cell is labeled and shaded by the maximum level of detail any object incident upon it is stored in memory – darker shades of gray represent higher levels of detail.

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, if the range cull algorithm and lookahead cull algorithm are both *any direction*, the range set is guaranteed to contain the cells that the observer can enter soonest (since cells are added to the range set in order of minimum distance from the current observer position), and the lookahead set is guaranteed to contain objects represented at LODs that can potentially be rendered for an observer viewpoint within a range cell within the next N frames. If the algorithm terminates due to condition (1), the set of lookahead objects is a provable superset of the objects that can possibly be rendered during the next N frames, and fits into available memory. Otherwise, if the algorithm terminates due to condition (2), the set of lookahead objects is certainly a superset of the objects visible from the current observer viewpoint, as well as a good estimate of the objects that are most likely to be rendered in upcoming frames.

5.2.3 Cache Management

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

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

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

Figure 5.9 shows results of the cache management algorithm for a particular observer path. Each cell is labeled by the number of frames since objects incident upon it were included in the lookahead set. The shade of each cell indicates whether or not it contains objects in the memory resident cache (stipple gray), read set (left-hatch), or release set (right-hatch).

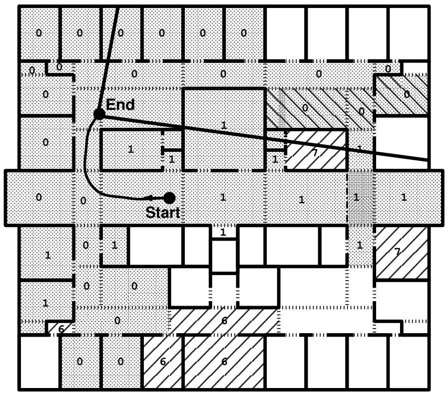


Figure 5.9: Cache management algorithm results. Each cell is labeled by the number of frames since objects incident upon it were included in the lookahead set. Shading for each cell indicates whether or not it contains objects in the memory resident cache (stipple gray), read set (left-hatch), and resident set (right-hatch).

5.2.4 Fault Tolerance

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

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

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

5.2.5 Results

To evaluate the effectiveness of the database system and predictive memory management algorithms, we ran a series of tests using our building walkthrough application with various combinations of the following parameters:

Lookahead Depth:

- **N (1, 4, 8, 16, 32, 64):** The memory management algorithm terminates its lookahead search when it has found all objects potentially visible during the next N frames.

Lookahead Granularity:

- **Object Granularity (Obj):** Objects in the cell-to-object visibility of a range cell are included in the lookahead set.
- **Cell Granularity (Cell):** All objects incident upon a cell in the cell-to-cell visibility of a range cell are included in the lookahead set.

Lookahead Cull Algorithm:

- **Any Direction (Any):** All objects incident upon cells potentially visible from a range cell can be included in the lookahead set.
- **Observer Plane (Plane):** Lookahead objects must be incident upon cells at least partially in front of the observer.
- **Observer Frustum (Frust):** Lookahead objects must be incident upon cells at least partially inside the current observer frustum.

Range Cull Algorithm:

- **Any Direction (Any):** All cells that can contain the observer in the next N frames are included in the range set.
- **Observer Plane (Plane):** Range cells must be at least partially in front of the observer.
- **Observer Frustum (Frust):** Range cells must be at least partially inside the current observer frustum.
- **Observer Direction (Dir):** Range cells must be hit by a ray traced from the current observer eye position along the current observer view direction.

In these tests, we used a flattened version of the Soda Hall model (i.e., without shared object definitions) that requires 349.5MB of storage. Although this flattened model is somewhat artificial since every instance of a particular object type just stores a flattened copy of the object definition's geometries (i.e., using a shared object hierarchy is far more practical in this situation), we believe that the storage requirements are representative of models containing radiosity information – i.e., each object instance may be colored and/or meshed independently after radiosity, thereby requiring a separate description in the database.

Each test was performed on a Silicon Graphics VGX 320 workstation with two 33MHz MIPS R3000 processors, 64MB of memory, a $16\mu s$ timer, and a local disk. 6.9MB of memory was reserved to store the spatial subdivision cell structure; 9.4MB was reserved to store the precomputed visibility stab lists; 4.6MB was reserved to store the segment, object, material, and texture headers; and 12MB was reserved to store object geometries in the memory resident cache. During each test, we used the (CTO,ETC,ETO) visibility determination algorithm described in Section 4.1, and the *Optimization* detail elision algorithm described in Section 4.2 with a target frame rate of ten frames per second combined with a static detail elision size threshold of 4096 pixels/face. The application was configured as a four-process pipeline with one process used for visibility determination and detail elision, a second process for rendering, a third process for memory management computations, and a fourth process for database input/output operations.

We collected statistics as the observer navigated along the path through the fifth floor of Soda Hall shown in Figure 5.10. This path was chosen because it spans a large part of the model and visits the same portion of the model more than once, and thus tests both the lookahead and caching features of our memory management algorithm.

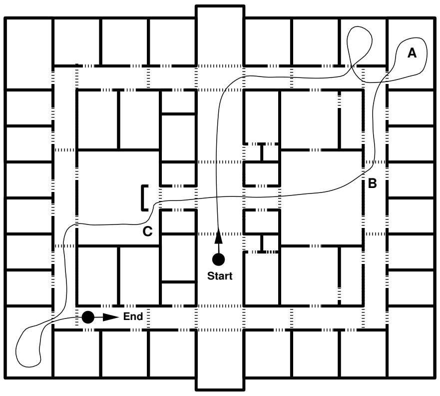


Figure 5.10: Test path through the fifth floor of Soda Hall.

For each frame along the test walkthrough path, we measured the numbers of range cells and lookahead objects, the compute time (i.e., the time required to perform memory management computations), the read time (i.e., the time required to load objects into memory from disk), and the frame time (i.e., the total time between successive frames) – these statistics give quantitative insights into the behavior of the pre-fetch algorithm. We also measured the number of LODs skipped by the rendering process due to failure of the system to load the appropriate LOD into memory in time before it is selected for rendering – this statistic gives a qualitative insight into how well the pre-fetch algorithm is performing its job.

Lookahead Depth

We first tested the effect of lookahead depth (i.e., the number of frames the pre-fetch algorithm looks in advance to load objects into memory) on the performance of our memory management algorithms. Results of tests using a variety of lookahead depths are shown in Tables 5.1 and 5.2. During these tests, object lookahead granularity was used, and the range and lookahead sets were not constrained to any particular direction.

Using the larger lookahead depths, the pre-fetch algorithm found a larger number of range cells, which could see a larger number of lookahead objects. As a result, the system was able to load objects into memory further in advance of when they potentially could become visible to the observer. This behavior is evidenced by the plot in Figure 5.11 which shows the amount of data required to store geometries of objects in the lookahead set during each frame along a portion of the test walkthrough path for a variety of lookahead depths. The lowest curve (labeled "0 Frames") represents the size (in MB) of the set of objects determined to be potentially visible from the actual observer viewpoint using the (CTO, ETC, None) method described in Section 4.1 – it provides a base-line indication of the complexity of the scene potentially visible to the observer viewpoint during each frame. Curves representing the size of the lookahead set using larger lookahead depths rise earlier before frames with large potentially visible sets.

Mean and maximum compute and read times were longer for larger lookahead depth because the pre-fetch algorithm traversed more nodes of the cell adjacency graph during the shortest path search and added more objects to the lookahead set. However, memory management computations and database input/output operations were executed in separate processes, asynchonous from the rest of the walkthrough system. The walkthrough system did not wait for the database input/output process to complete loading lookahead objects during each frame. Instead, it continued to render frames while the lookahead set was computed and objects were loaded from the database asynchronously. The walkthrough system did not wait even if an object to be rendered had not yet been loaded into memory – it just skipped to the highest LOD that was memory resident. Therefore, increased

Look Depth	Range Cells		Look Objects		LODs Skipped	
	Mean	Max	Mean	Max	Mean	Max
1	1.11	3	439	955	1.20	19
4	1.44	5	464	955	1.07	19
8	2.20	8	502	1,024	0.90	19
16	4.64	15	588	1,037	0.79	19
32	11.24	35	1,000	3,345	1.42	19
64	22.95	63	2,012	3,851	2.29	29

Table 5.1: Mean and maximum set statistics collected during tests with various lookahead depths.

Look Depth	Compute Time (s)		Read Time (s)		Frame Time (s)	
	Mean	Max	Mean	Max	Mean	Max
1	0.001	0.056	0.034	0.483	0.066	0.260
4	0.001	0.062	0.035	0.518	0.067	0.214
8	0.002	0.068	0.040	0.536	0.067	0.208
16	0.002	0.084	0.040	0.544	0.068	0.210
32	0.004	0.175	0.068	0.475	0.070	0.221
64	0.010	0.137	0.105	0.609	0.074	0.258

Table 5.2: Mean and maximum timing statistics collected during tests with various lookahead depths.

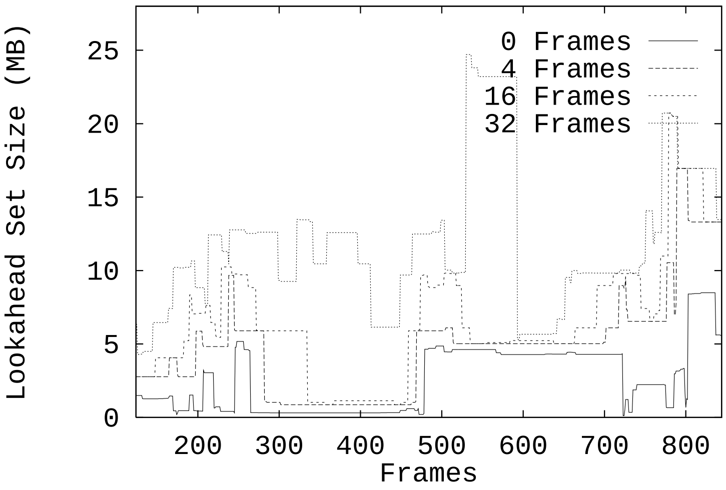


Figure 5.11: Size of the lookahead set (in MB) for various lookahead depths.

compute and read times did not result in an equal increase in the effective frame time. In this test, the frame time increased slightly with larger lookahead depths, most likely due to the fact that only two physical processors were available to run four concurrent processes.

The mean number of LODs skipped during rendering of each frame did not decrease monotonically as the lookahead depth increased. The reason for this behavior is that there was a point at which looking too far in advance was detrimental. Mean read time increased with larger lookahead depths, and lookahead objects claimed a larger percentage of the memory resident cache. As a result, precious read time and memory resources were spent loading and storing objects that were determined to be potentially visible many frames in advance, but actually never were rendered. Also, unnecessarily high LODs were loaded for some objects since the observer range included viewpoints closer to the objects. Evidence of these effects is shown in Figure 5.11 – the curve representing the size of the lookahead set while looking 32 frames in advance (labeled "32 Frames") has a peak between frames 500 and 600, even though the curve representing the size of the objects potentially visible from the actual observer viewpoint (labeled "0 Frames") does not rise during or after those frames. In these tests, looking 16 frames in advance caused the fewest LODs to be skipped (0.79 LODs per frame on average).

Lookahead Granularity

We also tested the effects of different lookahead granularities on the performance of our predictive memory management algorithms. Results of tests using object and cell lookahead granularities with a variety of lookahead depths are shown in Tables 5.3 and 5.4. During these tests, the range and lookahead sets were not constrained to any particular direction.

The difference between object and cell lookahead granularities pertains to the grouping of elements in the lookahead, read, and release sets. If object granularity is used, objects are added and removed from the sets independently – e.g., appropriate LODs for objects are added to the lookahead set when the object becomes visible from a range cell (i.e., in the cell-to-object visibility of the range cell) and a cell containing the object satisfies the lookahead cull. In contrast, using cell granularity, LODs for all objects incident upon a cell are added and removed from the sets as a unit – e.g., appropriate LODs for all objects incident upon a cell are added to the lookahead set when the cell becomes visible from a range cell and satisfies the lookahead cull. The motivation behind the cell granularity approach is that a few very large read operations may be far quicker than many small ones for some storage devices. For instance, our walkthrough system may eventually connect to a RAID storage system that supports very high throughput and long latency read operations for large chunks of data [16, 17, 35, 42]. For such storage devices, it may be advantageous to group read operations into large units. Unfortunately, there is no support for reading more than one object description in a single read operation in the current implementation of our system.

As expected, we found that the mean and maximum number of lookahead objects and read times were significantly greater using cell granularity than using object granularity. This is because all objects incident upon a cell were loaded from disk into memory together, even though many of them were not likely to become visible to the observer. These extra objects (i.e., the ones not in the cell-to-object visibility of any range cell) were added to the lookahead set only because they were incident upon a cell containing some other object that was likely to become visible, and were often read and released without ever being rendered. Using cell granularity, read times for some sequences of frames were so long (i.e., more than 2 seconds per frame) that the entire input queue for the database input/output process became full (64 frames in these tests), at which point the rest of the walkthrough system waited for the database input/output process to finish execution of its read operations before

Look Gran	Look Depth	Range Cells		Look Objects		LODs Skipped	
		Mean	Max	Mean	Max	Mean	Max
Obj	1	1.11	3	439	955	1.20	19
	4	1.44	5	464	955	1.07	19
	8	2.20	8	502	1,024	0.90	19
	16	4.64	15	588	1,037	0.79	19
	32	11.24	35	1,000	3,345	1.42	19
	64	22.95	63	2,012	3,851	2.29	29
Cell	1	1.11	3	764	1,629	0.96	19
	4	1.43	5	833	1,830	0.84	19
	8	2.08	8	840	1,830	2.15	33
	16	4.13	15	949	1,854	1.90	34
	32	8.09	26	1,488	4,950	1.94	33
	64	11.90	44	1,941	5,172	3.97	76

Table 5.3: Mean and maximum set statistics collected during tests with various lookahead granularities.

Look Gran	Look Depth	Compute Time (s)		Read Time (s)		Frame Time (s)	
		Mean	Max	Mean	Max	Mean	Max
Obj	1	0.001	0.056	0.034	0.483	0.066	0.260
	4	0.001	0.062	0.035	0.518	0.067	0.214
	8	0.002	0.068	0.040	0.536	0.067	0.208
	16	0.002	0.084	0.040	0.544	0.068	0.210
	32	0.004	0.175	0.068	0.475	0.070	0.221
	64	0.010	0.137	0.105	0.609	0.074	0.258
Cell	1	0.001	0.052	0.063	1.559	0.074	0.473
	4	0.001	0.029	0.063	1.507	0.075	0.780
	8	0.001	0.031	0.082	2.373	0.089	2.317
	16	0.002	0.026	0.112	2.089	0.100	1.605
	32	0.004	0.905	0.152	2.395	0.115	1.654
	64	0.011	1.332	0.187	2.301	0.134	2.476

Table 5.4: Mean and maximum timing statistics collected during tests with various lookahead granularities.

another frame was added to its input queue. These waiting periods are reflected in long frame times following long read times. Figure 5.12 shows a plot of read times and frame times measured during each frame along a portion of the test path using cell granularity while looking ahead 16 frames in advance. Sharp peaks in the read time curve occur during frames in which a large amount of data was read into memory all at once – peaks in the frame time curve often appear exactly 64 frames later. See Chapter 6 for more information on process synchronization in the walkthrough system.

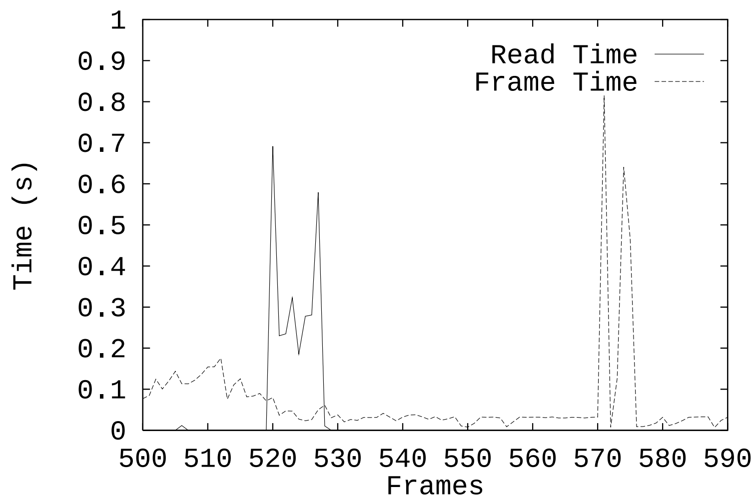


Figure 5.12: Read times and frame times using cell granularity looking 16 frames in advance.

The mean number of LODs skipped using cell granularity was less than using object granularity for small lookahead depths. Figure 5.13 shows a plot of the number of levels skipped during each frame along a portion of the walkthrough path using cell and object granularities while looking only 1 frame in advance. During this sequence of frames, the number of levels skipped was small in both cases during frames in which the observer stayed within the same cell, but was less for cell granularity during frames in which the observer crossed a cell boundary and moved into a new cell (marked by tick marks on the "Frames" axis). Fewer LODs were skipped for very small lookahead depths using cell granularity because there is a large amount of coherence in the cell-to-cell visibilities of neighboring cells.

On the other hand, the mean number of LODs skipped using cell granularity was greater than using object granularity for large lookahead depths because the sizes of the lookahead and read sets increased dramatically with larger lookahead depths using cell granularity. Since there was no advantage to reading more than one object description at once in our current implementation, precious resources, such as disk i/o bandwidth and space in the memory resident cache, were simply wasted using cell granularity. This behavior caused descriptions for objects that did become visible to the observer eventually to be removed from the memory resident cache, or to not be read into memory in time for display by the rendering process. In general, object granularity provided finer control over memory and disk-to-memory bandwidth utilization.

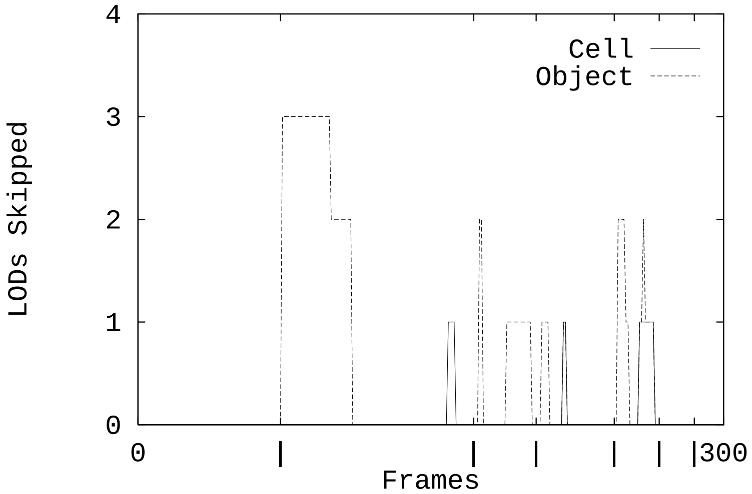


Figure 5.13: Number of LODs skipped using cell granularity and object granularity looking 1 frame in advance. Frames in which the observer crosses a cell boundary are marked by a dash on the "Frames" axis.

Range and Lookahead Culls

We also measured the effect of using the observer's current direction of movement to predict the range of future observer viewpoints and future observer visibility. Results of tests using different combinations of *range cull* and *lookahead cull* methods with object lookahead granularity are shown in Tables 5.5-5.10.

We found results obtained using the observer's current direction of movement to predict

Range Cull	Look Depth	Look Cull: Any Mean	Any Max	Plane Mean	Plane Max	Frustum Mean	Frustum Max
Any	1	1.11	3	1.11	3	1.11	3
	4	1.44	5	1.44	5	1.44	5
	8	2.20	8	2.20	8	2.20	8
	16	4.64	15	4.64	15	4.65	15
	32	11.24	35	11.46	35	11.78	39
	64	22.95	63	23.55	63	27.61	71
Plane	1	1.10	3	1.10	3	1.10	3
	4	1.35	4	1.35	4	1.35	4
	8	1.86	8	1.86	8	1.86	8
	16	3.43	13	3.44	13	3.44	13
	32	7.16	26	7.22	28	7.24	28
	64	13.48	46	15.01	46	17.61	58
Frust	1	1.10	3	1.10	3	1.11	3
	4	1.34	4	1.34	4	1.34	4
	8	1.72	6	1.72	6	1.72	6
	16	2.71	10	2.71	10	2.71	10
	32	5.23	19	5.24	19	5.24	19
	64	10.26	33	10.92	33	11.55	35
Dir	1	1.05	3	1.05	3	1.06	3
	4	1.22	4	1.22	4	1.22	4
	8	1.38	4	1.38	4	1.38	4
	16	1.84	7	1.84	7	1.84	7
	32	2.53	8	2.53	8	2.53	8
	64	3.27	14	3.31	14	3.65	14

Table 5.5: Mean and maximum numbers of range cells during tests with various combinations of range and lookahead cull methods.

Range Cull	Look Depth	Look Cull: Any Mean	Any Max	Plane Mean	Plane Max	Frustum Mean	Frustum Max
Any	1	439	955	323	943	238	577
	4	464	955	357	946	269	683
	8	502	1,024	402	1,020	305	746
	16	588	1,037	506	1,031	379	878
	32	1,000	3,345	895	3,356	721	3,040
	64	2,012	3,851	1,876	3,815	1,536	4,444
Plane	1	438	955	320	943	237	577
	4	457	955	348	946	263	683
	8	478	973	377	971	287	700
	16	526	1,037	445	1,031	335	773
	32	880	3,194	684	2,217	538	1,961
	64	1,201	3,296	1,319	4,372	1,005	3,104
Frust	1	439	955	323	943	237	577
	4	459	955	352	946	264	683
	8	482	1,024	381	1,020	291	746
	16	523	1,024	446	1,020	329	750
	32	787	3,194	632	2,121	481	1,727
	64	1,064	3,199	897	3,394	753	2,439
Dir	1	434	955	317	943	234	577
	4	447	955	338	946	256	671
	8	460	955	358	946	273	683
	16	483	957	392	955	298	705
	32	602	3,194	465	946	357	1,060
	64	748	3,194	593	1,972	423	1,447

Table 5.6: Mean and maximum numbers of lookahead objects during tests with various combinations of range and lookahead cull methods.

Range Cull	Look Depth	Look Cull: Any Mean	Any Max	Plane Mean	Plane Max	Frustum Mean	Frustum Max
Any	1	1.20	19	1.12	19	0.80	23
	4	1.07	19	1.05	19	1.48	58
	8	0.90	19	0.96	19	1.69	52
	16	0.79	19	0.82	19	1.11	24
	32	1.42	19	1.31	19	0.93	19
	64	2.29	29	2.77	29	4.70	60
Plane	1	1.20	19	1.10	19	0.81	23
	4	1.07	19	1.05	19	1.48	60
	8	0.90	19	0.96	19	1.69	51
	16	0.78	19	0.82	19	1.08	24
	32	1.32	19	1.32	19	1.33	19
	64	2.60	19	2.96	28	2.08	22
Frust	1	1.20	19	1.12	19	0.81	23
	4	1.05	19	1.04	19	1.45	57
	8	0.95	19	0.95	19	1.54	32
	16	0.76	19	0.82	19	1.12	24
	32	1.04	19	1.01	19	1.07	19
	64	1.73	17	2.51	19	1.81	21
Dir	1	1.19	19	1.12	19	0.79	23
	4	1.05	19	1.03	19	1.43	57
	8	0.96	19	0.98	19	1.52	32
	16	0.87	19	0.89	19	1.07	22
	32	0.77	19	0.80	19	1.94	21
	64	0.92	18	0.92	19	1.01	19

Table 5.7: Mean and maximum numbers of LODs skipped during tests with various combinations of range and lookahead cull methods.

Range Cull	Look Depth	Look Cull: Any Mean	Any Max	Plane Mean	Plane Max	Frustum Mean	Frustum Max
Any	1	0.001	0.056	0.002	0.047	0.002	0.044
	4	0.001	0.062	0.002	0.047	0.002	0.035
	8	0.002	0.068	0.003	0.053	0.003	0.041
	16	0.002	0.084	0.004	0.065	0.004	0.056
	32	0.004	0.175	0.008	0.184	0.009	0.087
	64	0.010	0.137	0.017	0.120	0.024	0.113
Plane	1	0.002	0.061	0.002	0.045	0.002	0.032
	4	0.002	0.058	0.002	0.057	0.003	0.036
	8	0.002	0.064	0.003	0.054	0.003	0.042
	16	0.002	0.085	0.004	0.067	0.004	0.067
	32	0.004	0.151	0.007	0.083	0.007	0.075
	64	0.008	0.242	0.015	0.227	0.020	0.177
Frust	1	0.001	0.064	0.002	0.045	0.002	0.032
	4	0.001	0.054	0.002	0.047	0.002	0.035
	8	0.002	0.061	0.003	0.056	0.003	0.042
	16	0.002	0.085	0.003	0.068	0.003	0.051
	32	0.004	0.136	0.005	0.103	0.006	0.068
	64	0.006	0.235	0.011	0.168	0.013	0.119
Dir	1	0.001	0.058	0.002	0.048	0.002	0.032
	4	0.001	0.060	0.002	0.049	0.002	0.033
	8	0.001	0.048	0.002	0.045	0.003	0.046
	16	0.002	0.062	0.003	0.054	0.003	0.049
	32	0.002	0.134	0.004	0.081	0.004	0.076
	64	0.003	0.211	0.005	0.118	0.006	0.092

Table 5.8: Mean and maximum compute times during tests with various combinations of range and lookahead cull methods. All times are in seconds.

Range Cull	Look Depth	Look Cull: Any Mean	Any Max	Plane Mean	Plane Max	Frustum Mean	Frustum Max
Any	1	0.034	0.483	0.034	0.443	0.032	0.418
	4	0.035	0.518	0.036	0.518	0.033	0.447
	8	0.040	0.536	0.042	0.516	0.036	0.594
	16	0.040	0.544	0.041	0.542	0.040	0.567
	32	0.068	0.475	0.070	0.488	0.054	0.459
	64	0.105	0.609	0.104	0.592	0.092	0.501
Plane	1	0.035	0.624	0.035	0.664	0.032	0.460
	4	0.036	0.401	0.036	0.677	0.033	0.523
	8	0.040	0.471	0.042	0.735	0.035	0.614
	16	0.042	0.503	0.043	0.543	0.041	0.532
	32	0.065	0.486	0.070	0.466	0.051	0.508
	64	0.097	0.606	0.096	0.535	0.084	0.564
Frust	1	0.035	0.411	0.034	0.499	0.032	0.520
	4	0.034	0.461	0.037	0.490	0.034	0.478
	8	0.040	0.524	0.041	0.619	0.039	0.473
	16	0.041	0.576	0.044	0.808	0.040	0.624
	32	0.057	0.417	0.055	0.445	0.044	0.428
	64	0.083	0.508	0.087	0.432	0.063	0.573
Dir	1	0.034	0.396	0.034	0.465	0.032	0.463
	4	0.036	0.580	0.034	0.388	0.033	0.559
	8	0.037	0.406	0.035	0.431	0.034	0.463
	16	0.038	0.598	0.038	0.481	0.037	0.536
	32	0.054	0.415	0.049	0.443	0.049	0.450
	64	0.056	0.532	0.050	0.526	0.044	0.504

Table 5.9: Mean and maximum read times during tests with various combinations of range and lookahead cull methods. All times are in seconds.

Range Cull	Look Depth	Look Cull: Any Mean	Any Max	Plane Mean	Plane Max	Frustum Mean	Frustum Max
Any	1	0.066	0.260	0.067	0.220	0.068	0.242
	4	0.067	0.214	0.067	0.215	0.068	0.231
	8	0.067	0.208	0.068	0.208	0.071	0.223
	16	0.068	0.210	0.067	0.208	0.068	0.216
	32	0.070	0.221	0.071	0.213	0.071	0.216
	64	0.074	0.258	0.076	0.280	0.077	0.354
Plane	1	0.067	0.214	0.069	0.269	0.069	0.223
	4	0.067	0.218	0.068	0.214	0.069	0.219
	8	0.069	0.228	0.067	0.233	0.070	0.238
	16	0.068	0.209	0.068	0.214	0.071	0.235
	32	0.070	0.216	0.071	0.213	0.072	0.214
	64	0.077	0.285	0.075	0.258	0.076	0.220
Frust	1	0.068	0.243	0.068	0.216	0.070	0.232
	4	0.068	0.216	0.069	0.217	0.070	0.222
	8	0.069	0.221	0.069	0.212	0.071	0.212
	16	0.069	0.222	0.071	0.219	0.071	0.218
	32	0.072	0.222	0.073	0.226	0.071	0.210
	64	0.074	0.225	0.080	0.369	0.074	0.222
Dir	1	0.070	0.225	0.069	0.216	0.069	0.227
	4	0.068	0.296	0.068	0.234	0.070	0.224
	8	0.069	0.227	0.068	0.226	0.069	0.220
	16	0.069	0.233	0.069	0.216	0.070	0.300
	32	0.071	0.214	0.072	0.216	0.071	0.305
	64	0.071	0.217	0.071	0.215	0.072	0.248

Table 5.10: Mean and maximum frame times during tests with various combinations of range and lookahead cull methods. All times are in seconds.

future observer range cells and lookahead objects to be mixed. On the one hand, directing the shortest path search to consider only range cells and lookahead objects in front of the observer caused the pre-fetch algorithm to find fewer lookahead objects during each frame. Less cache memory was required to store the lookahead objects, and less read time was required to load them. Therefore, the algorithm could look further in advance while using the same amount of cache memory and disk bandwidth resources. Accordingly, the mean and maximum numbers of LODs skipped were less using more constrained range and lookahead cull methods for large lookahead depths.

On the other hand, for small lookahead depths, the mean and maximum numbers of LODs skipped were greater using more constrained range and lookahead cull methods. This difference was isolated to a few frames along the test walkthrough path – i.e., the ones after the observer switches direction of movement suddenly. For instance, at the observer viewpoint marked 'B' in Figure 5.10, 57 LODs were skipped using the *observer direction* range cull method and *observer frustum* lookahead cull method with a lookahead depth of 4 frames, while only 19 LODs were skipped using the *any direction* method for both range and lookahead culls. As we expected, the more constrained cull methods did not predict future observer viewpoints adequately at times when there is a sudden change of direction.

Clearly, the best combination of range and lookahead cull methods depends on the path taken by the observer. Since observer's tend to turn more quickly than they move, the default for our system is to use the most constrained range cull method (i.e., *observer direction*) and the least constrained lookahead cull method (i.e., *any direction*). Other factors, such as the size of the memory resident cache and the speed at which objects can be loaded from disk into memory, certainly may affect these choices as well.

5.2.6 Discussion

In order to compute a set of objects to store in memory during each frame of an interactive walkthrough, we predict a range of observer viewpoints during the upcoming N future frames (i.e., the observer range), and then determine a superset of objects potentially visible from that observer range. Unfortunately, we currently are not able to perform an exact observer range visibility determination (e.g., range-to-object visibility) in real-time due to the high computational complexity of our algorithm for computing visibility from a finite, non-zero volume of space. For instance, computing the region of space visible from a box requires execution of an $O(n)$ algorithm for each portal encountered through a sequence of N portals [55] (compared to $O(1)$ for each portal encountered during computation of visibility from a single point). Instead, we over-estimate the range-to-object visibility set using the precomputed cell-to-object visibility information of the cells containing some viewpoint in the observer range (i.e., the range cells). Using this technique, expensive computations for determining visibility from a finite, non-zero volume of space are performed only off-line, during the precomputation phase.

As described in Section 4.1.4, the amount of data in the cell-to-object visibility for an average cell can depend on the granularity of the spatial subdivision. During our tests, we found that the objects in the lookahead set for a single range cell can require as much as 23MB of storage (e.g., large cells in the exterior space around the building). Therefore, the visibility of the range cells can be quite a large over-estimate of the true visibility of the observer range, especially if the spatial subdivision granularity is rather coarse.

Perhaps it would be better to perform an approximate, but efficient, real-time visiblity determination for the observer range to determine a lookahead set rather than using the cumulative visibility of the range cells. Such an algorithm might cast visibility beams from a bounding box around the observer range through the portals of the spatial subdivision to determine a set of potentially visible objects (see Figure 5.14), checking the feasibility of a sight-line for portal sequences only up to a certain length, or only through the last portal of the portal sequence. Further experimentation is required to determine exactly which types of observer range visibility determination computations are possible in real-time.

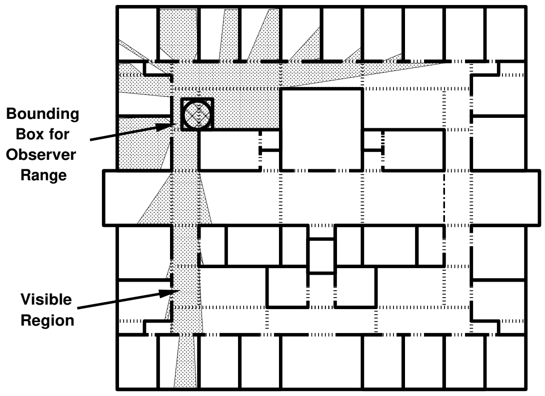


Figure 5.14: Visibility from a box bounding the observer range.

Chapter 6

Concurrent Processing

In this chapter, we study the effects of concurrent processing in an interactive visualization application. The motivation behind multiprocessing is obvious: faster frame rates via concurrent processing. Overlapping computations is particularly important in our building walkthrough system, since the display and memory management algorithms described in Chapters 4 and 5 perform a significant amount of computation.

In order to add concurrent processing to our building walkthrough system, we must decide how to partition the system into concurrent processes, and choose a communication protocol for process synchronization. Unfortunately, our choices are constrained by the limitations of the Silicon Graphics workstations used by our system. Although Silicon Graphics workstations may have many general purpose processors, there is a limitation that the graphics engine may only be accessed by one process at once. Accordingly, we use a single rendering process, which is responsible for all communication with the graphics engine.

A further constraint is due to data dependencies in certain computational steps of our building walkthrough system. The *Rendering* step relies upon information computed during the *Detail Elision* step, which, in turn, relies upon information computed during the *Visibility Determination* step. These constraints imply a sequential flow of data through the system, making a coarse-grained pipeline the obvious parallel system architecture.

6.1 Pipelining

We added concurrency to the building walkthrough system by mapping the functional operations of the walkthrough phase (see Figure 3.24) into separate stages of a pipeline. Each functional operation (except *Cache Management*) can run asynchronously in a separate thread on a separate CPU. A thread is spawned for each independent pipeline stage using the system call `sproc`, which creates a lightweight process in the same address space as the original heavyweight process. All concurrent threads access the same copy of the building model stored in shared memory. The threads are locked onto separate physical CPUs using the system call `sysmp`.

Each stage of the pipeline is implemented using an input queue and a computational unit, as shown in Figure 6.1. The input queue, which stores *frames* waiting to be processed by the computational unit, is synchronized by *full* and *empty* counting semaphores. The full semaphore is initialized to the depth of the queue, and is decremented each time a frame is added to the queue and incremented each time a frame is removed from the queue. The empty semaphore does the opposite – it is initialized to zero, incremented each time a frame is added to the queue and decremented each time a frame is removed from the queue. These semaphores control asynchronous access to the queue. The caller (i.e., the previous stage in the pipeline) must wait if the queue is full (i.e., the full semaphore is zero), and the computational unit must wait if the queue is empty (i.e., the empty semaphore is zero).

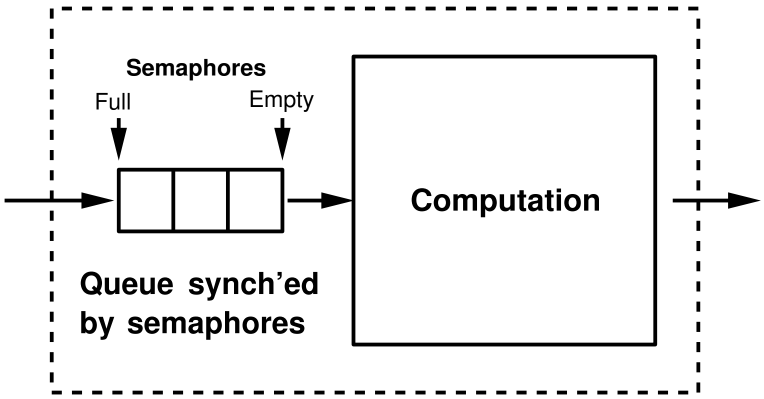


Figure 6.1: Pipeline stage implementation.

The pipeline is implemented in such a way that any grouping of functional operations into pipeline stages can be tested. In our notation for stage partitioning, each functional operation is represented by a single letter: "U" = *User Interface*, "V" = *Visibility Determination*, "D" = *Detail Elision*, "R" = *Rendering Operations*, "M" = *Lookahead Determination* and *Cache Management*, and "I" = *Database Input/Output Operations*. Mappings of functional operations into pipeline stages are represented by groupings of these letters in parentheses. Input queue depths are represented by numbers proceeding the letters (no number indicates a queue depth of one frame). For instance, each functional operation might be mapped to a separate pipeline stage with a one frame input queue (i.e., `(U)(V)(D)(R)(M)(I)`). Alternatively, the *User Interface*, *Visibility Determination*, and *Detail Elision* operations might be grouped into one stage, while the *Rendering*, *Memory Management*, and *Database Input/Output* operations execute in separate stages with queue depths 1, 12 and 64, respectively (i.e., `(UVD)(R)(12M)(64I)`). Our application is implemented such that the grouping of functional operations into pipeline stages can be controlled by the user, which allows us to experiment with different pipeline partitions.

6.2 Results

To test the performance of various pipeline architectures in our building walkthrough system, we ran a set of tests on a Silicon Graphics GTX workstation with four 33MHz MIPS R3000 processors and 32MB of shared memory. Note that this machine is different than the one used in all other tests described in this thesis – it has four processors and a slower graphics engine.

To simplify the characterization of performance during our tests, we experimented with pipeline parameters for only the lower-fork of the double pipeline shown in Figure 3.24. The lower-fork performs the display management operations to generate images on the screen of the graphics workstation during each frame (i.e., *Visibility Determination*, *Detail Elision*, and *Rendering*). Therefore, the performance of the lower-fork dictates the apparent frame times and response times of the system. In order to eliminate the effects of memory management in these tests, we used a model of Soda Hall with shared object definitions which fit entirely in memory.

In each test, we gathered statistics as the simulated observer "walked" along the path shown in Figure 6.2. This path was chosen due to its cyclical nature, which allows certain performance characteristics to be more easily identified. The path alternates between relatively restricted views encountered inside offices, and wide-open views in the hallway.

Inside an office, the times required by the *Visibility Determination* and *Detail Elision* computations are small relative to the time required for *Rendering*. Once we enter the hallway, the time required for *Visibility Determination* becomes more significant and sometimes surpasses *Rendering* time. *Visibility Determination* becomes less significant as the observer progresses further down the hallway, since there are fewer cells and objects in the observer's view.

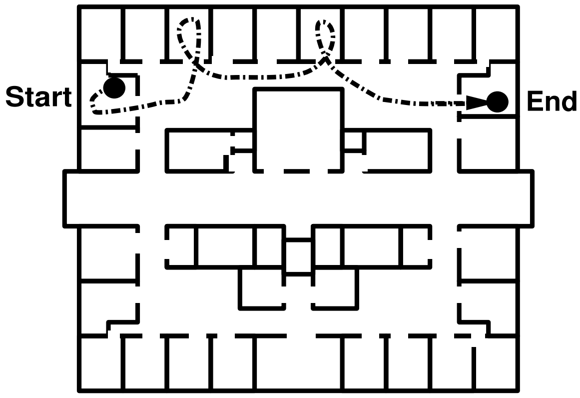


Figure 6.2: Test path through the seventh floor of Soda Hall.

During each frame of each test, we measured the time required for computation during each functional operation, the overall frame time (i.e., the inverse of frame rate), and the response time (i.e., the real-time expired between the time an observer viewpoint is specified by the user and the time the corresponding image appears on the workstation screen). Timing statistics were gathered using a 16μs timer.

Pipeline Partition

We first experimented with different groupings of functional operations into pipeline stages using a one frame input queue for each stage in each test. For the four functional operations of display management, there are eight possible groupings of functional operations into pipeline stages. There is one possible one-stage pipeline (all in the same stage), three possible two-stage pipelines, three possible three-stage pipelines, and one possible four-stage pipeline. Mean and maximum compute time, frame time, and response time statistics for each of these eight possible partitions are summarized in Tables 6.1 and 6.2. Plots of the frame time and response time measured in each frame along a portion of the walkthrough path for representative 1-, 2-, 3-, and 4-stage pipelines are shown in Figures 6.3 and 6.4, respectively.

This experiment confirmed many expected results. First, the mean throughputs (1/frame time) of all concurrent pipeline partitions were higher than the throughput of the sequential pipeline partition (i.e., the mean frame time for `(UVDR)` was the highest). This result can be explained by noting that the frame time in a concurrent pipeline (i.e., stages on separate processors) typically equals the time required for the slowest stage, whereas the frame time in a sequential pipeline (i.e., all stages on the same processor) equals the sum of the times for its stages. This behavior can be seen in Figure 6.5, which shows the time required for each stage of the pipeline at each frame along the test walkthrough path. The overall frame time was almost exactly equal to the time required for the slowest stage in each frame along the path. The speed-up due to concurrency was much less than four times for a 4-way pipeline because the *User Interface* and *Detail Elision* operations generally required far less time than the *Visibility Determination* and *Rendering* operations.

Second, the mean response times (time from start of user interaction to completion of drawing) of all concurrent pipeline partitions were higher than the sequential pipeline partition (i.e., the response time for `(UVDR)` is the lowest). This result can be explained by noting that the pipeline stages contain queues that can buffer frames inside the pipeline. As a result, in cases where the last stage is the limiting step, frames get queued up inside the pipe, adding to the latency of the system (i.e., longer response time).

However, this experiment generated an interesting and unexpected result. One would expect that increasing the number of pipeline stages would necessarily increase overall pipeline throughput. But, examining the "Frame Time" column of Table 6.2, we found that overall pipeline throughput was highest in the `(U)(VD)(R)` three-stage pipeline partition, rather than the `(U)(V)(D)(R)` four-stage partition. Furthermore, it seems that throughput was not correlated with the number of pipeline stages at all! Instead, the throughputs of the eight pipeline partitions appear to be correlated with the grouping of stages in each partition. For instance, partitions with the *Visibility Determination*, *Detail Elision*, and *Rendering* operations in the same pipeline stage (i.e., `(UVDR)` and `(U)(VDR)`) were the slowest (0.16 seconds); partitions with the *Visibility Determination* and *Detail Elision* operations in different stages (i.e., `(UV)(DR)`, `(UV)(D)(R)`, `(U)(V)(DR)`, and `(U)(V)(D)(R)`) were just a little faster (0.14 seconds); and partitions with the *Visibility Determination* and *Detail Elision* operations in the same stage, and the *Rendering* operation in a separate stage (i.e., `(UVD)(R)` and `(U)(VD)(R)`) were the fastest (0.12 seconds).

# Stages	Pipeline Partition	Visibility		Detail Elision		Rendering	
		Mean	Max	Mean	Max	Mean	Max
1	(UVDR)	0.069	0.384	0.004	0.012	0.083	0.142
2	(U)(VDR)	0.075	0.410	0.005	0.015	0.083	0.137
2	(UV)(DR)	0.125	0.493	0.005	0.013	0.083	0.160
2	(UVD)(R)	0.094	0.507	0.005	0.049	0.083	0.159
3	(UV)(D)(R)	0.127	0.502	0.005	0.069	0.083	0.145
3	(U)(V)(DR)	0.123	0.494	0.005	0.016	0.083	0.141
3	(U)(VD)(R)	0.125	0.683	0.005	0.048	0.084	0.161
4	(U)(V)(D)(R)	0.125	0.508	0.005	0.038	0.083	0.155

Table 6.1: Mean and maximum compute time statistics collected during tests with various pipeline stage partitions.

# Stages	Pipeline Partition	Frame Time		Response Time	
		Mean	Max	Mean	Max
1	(UVDR)	0.162	0.516	0.161	0.515
2	(U)(VDR)	0.164	0.541	0.330	1.081
2	(UV)(DR)	0.148	0.494	0.243	0.641
2	(UVD)(R)	0.121	0.517	0.211	0.641
3	(UV)(D)(R)	0.148	0.504	0.269	0.644
3	(U)(V)(DR)	0.145	0.489	0.390	1.124
3	(U)(VD)(R)	0.118	0.645	0.363	1.126
4	(U)(V)(D)(R)	0.144	0.503	0.419	1.135

Table 6.2: Mean and maximum frame time and response time statistics collected during tests with various pipeline stage partitions.

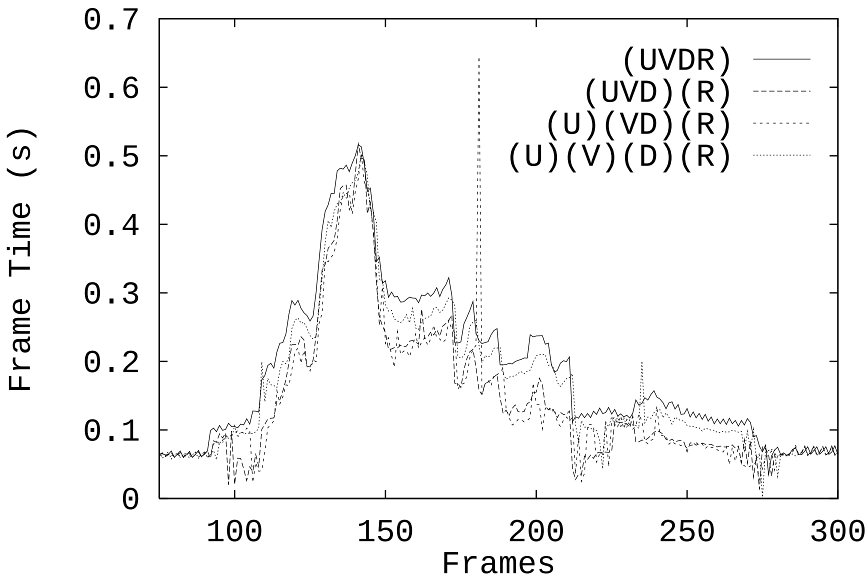


Figure 6.3: Frame time during each frame along a portion of the test walkthrough path for representative 1-, 2-, 3-, and 4-stage pipelines.

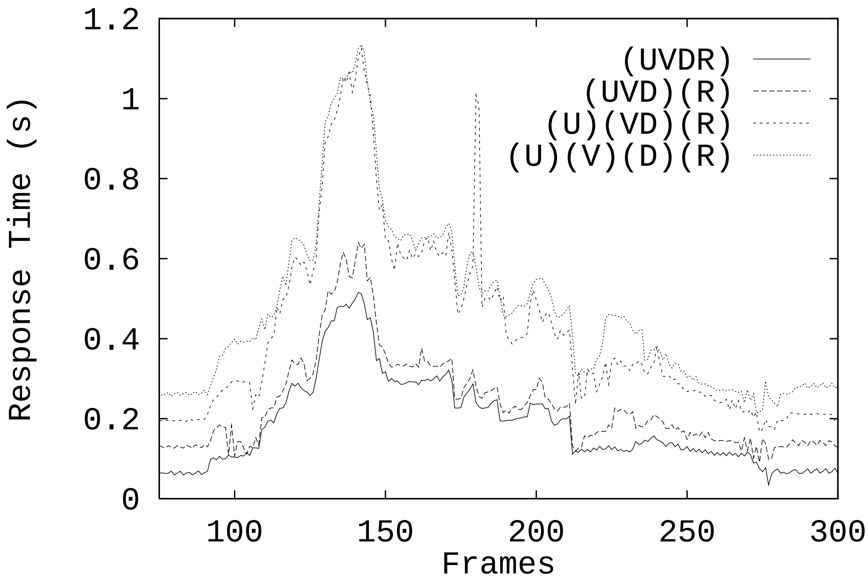


Figure 6.4: Response time during each frame along a portion of the test walkthrough path for representative 1-, 2-, 3-, and 4-stage pipelines.

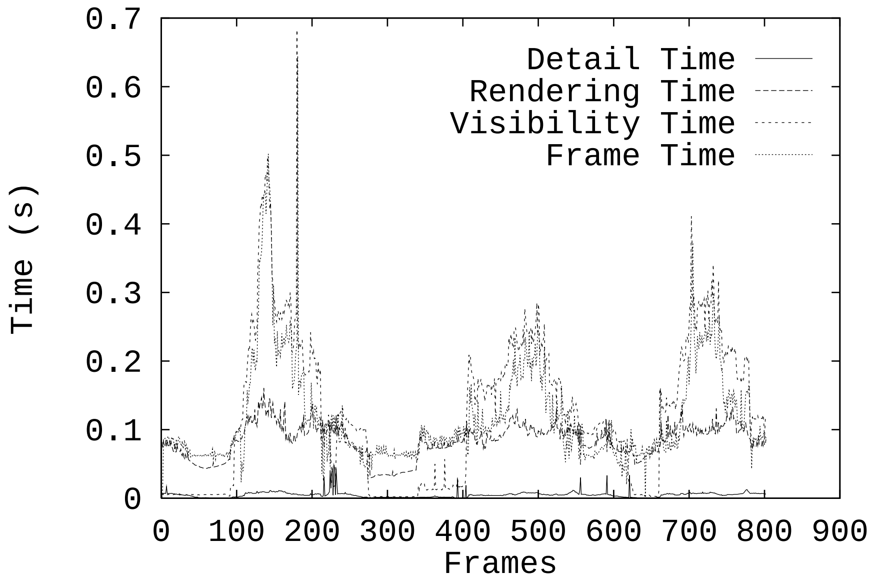


Figure 6.5: Throughput is determined by the slowest stage in a concurrent pipeline.

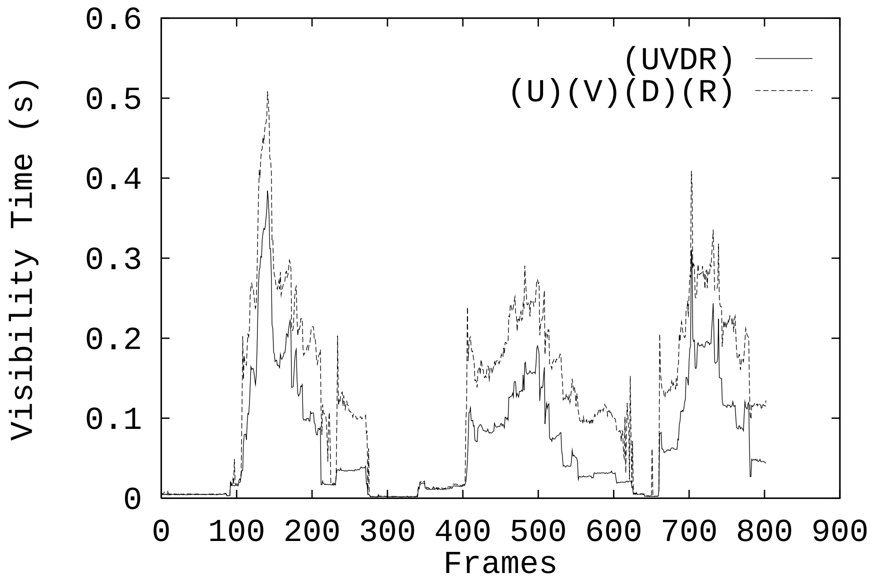


Figure 6.6: Time for the *Visibility Determination* operation is larger when other operations are executing on other processors.

This result can be explained by noting that the execution time of each stage was not independent of the pipeline partition. That is, within a given operation, the same sequence of instructions took a different amount of time depending on how many other stages were executing simultaneously, even though each stage was running on a separate physical CPU. For instance, Figure 6.6 shows the time required for the *Visibility Determination* operation in two cases: 1) when all operations were grouped in the same stage (i.e., `(UVDR)`), and 2) when each operation was in a separate stage executing simultaneously (i.e., `(U)(V)(D)(R)`). The time to execute the visibility operation (i.e., the same sequence of instructions) was different in these two cases. This effect was most likely due to contention between concurrent processes accessing shared resources (e.g., cache flushes and shared memory contention). Further work is necessary to better understand the cause of this effect.

Queue Depth

We also measured the performance of the display management pipeline using various queue depths for the *Rendering* stage of the `(UVD)(R)` two-stage pipeline. We used this 2-stage pipeline partition, with only the *Rendering* operation separated into a separate process, to keep the test as simple as possible. Mean and maximum compute time, frame time, and response time statistics for several queue depths are summarized in Tables 6.3 and 6.4. More detailed statistics describing the frame time and response time for each frame along the walkthrough path are shown in Figures 6.7 and 6.8, respectively.

This experiment also verified many expected results. Most significantly, the mean response time was larger as the depth of the *Rendering* queue increased. This can be explained by noting that the queue buffered frames whenever the *Rendering* operation was the slowest stage. This queueing effect can be seen clearly in the regions of large positive slope in the response time curves of Figure 6.8. During the down slopes in these curves, the *Visibility Determination* operation was the rate limiting step, so the queue was able to drain out.

There was also an unexpected result in this experiment. One would expect that increasing the queue depth for the draw stage would increase overall throughput. This is expected since the slowest stage in the pipeline varied alternately between the *Visibility Determination* and *Rendering* operations for this test walkthrough path. During periods in which the *Rendering* operation was slowest, the queue filled; and during times when

Queue Depth	Pipeline Partition	Visibility		Detail Elision		Rendering	
		Mean	Max	Mean	Max	Mean	Max
0	(UVDR)	0.069	0.384	0.004	0.012	0.083	0.142
1	(UVD)(R)	0.094	0.507	0.005	0.049	0.083	0.159
4	(UVD)(4R)	0.097	0.508	0.005	0.062	0.083	0.159
16	(UVD)(16R)	0.102	0.726	0.006	0.055	0.083	0.139
64	(UVD)(64R)	0.111	0.694	0.006	0.084	0.084	0.491

Table 6.3: Mean and maximum compute time statistics collected during tests with various *Rendering* stage queue depths.

Queue Depth	Pipeline Partition	Frame Time		Response Time	
		Mean	Max	Mean	Max
0	(UVDR)	0.162	0.516	0.161	0.515
1	(UVD)(R)	0.121	0.517	0.211	0.641
4	(UVD)(4R)	0.123	0.519	0.291	0.643
16	(UVD)(16R)	0.122	0.733	0.544	1.629
64	(UVD)(64R)	0.122	0.712	1.158	6.157

Table 6.4: Mean and maximum frame time and response time statistics collected during tests with various *Rendering* stage queue depths.

the *Visibility Determination* operation was slowest, the queue drained. This buffering effect was expected to improve throughput since the overall frame time matched the time required for the *Visibility Determination* operation in situations where the *Rendering* operation was slowest (assuming that the queue eventually drained). This effect can be seen clearly in regions of the frame time curves for `(UVD)(64R)` and `(UVD)(R)` in Figure 6.7 where the *Rendering* operation is the slowest stage (e.g. frames 300 and 650). In these cases, the frame time for `(UVD)(64R)` was less than for `(UVD)(R)`.

However, looking at Table 6.4, the mean frame time did not improve for larger queue depths, even though the frame time was shorter for frames in which the *Rendering* operation was slowest. This result can be explained by noting the large peaks in the frame time curve for (UVD)(64R) at frames 100, 400, and 680 in Figure 6.7. These peaks appear during periods in which the queue was draining, i.e., times when the *Visibility Determination* and *Rendering* operations were both very active. The peaks were caused by a marked increase in the time required by the *Visibility Determination* operation. Once again, this decrease in performance can be explained by the contention introduced by heavy periods of concurrent execution. The fact that the mean frame times for all queue depths greater than one were the same seems to be just a coincidence resulting from the combination of some shorter frame times for some frames due to buffering frames in the queue, and longer frame times for frames in which there was high contention. For a different walkthrough path, the balance between these two phenomena might have been different, resulting in different means for each queue depth.

6.3 Discussion

Using pipelining, we were able to improve mean frame times of the building walkthrough application by 25% – about one third of what could ideally be realized by a balanced four-stage pipeline. Speed-up is sub-linear due in part to poor partitioning of functional operations into pipeline stages – the times required by two of the four stages dominate most frames (i.e., *Visibility Determination* and *Rendering*). Speed-up was further diminished by contention for data between processes executing concurrently.

The small improvement in frame times is achieved at the expense of longer response times. This tradeoff is a consequence of our implementation of the asynchronous pipeline. The *User Interface* enters observer viewpoints into the pipe as soon as the subsequent stage

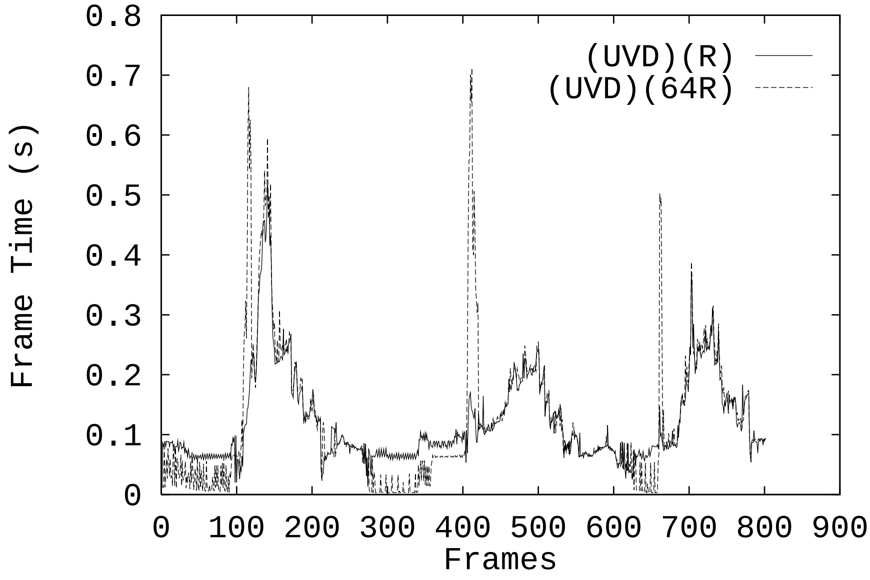


Figure 6.7: Frame time during each frame of test walkthrough path for various queue depths in *Rendering* stage.

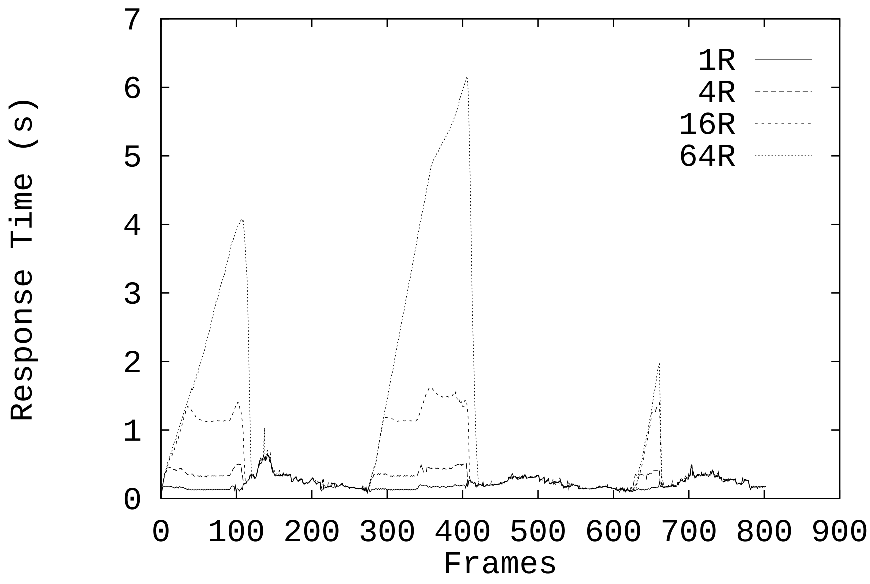


Figure 6.8: Response time during each frame of test walkthrough path for various queue depths in *Rendering* stage.

has room in its input queue. As a result, many frames can be buffered inside the pipeline causing long response times. Alternatively, the *User Interface* could wait until the previous frame has passed some particular stage in the pipeline before entering the next observer viewpoint so as to minimize this effect. In our experience, the tradeoff between frame time and response time seems to favor faster frame times at the expense of slower response times, as long as response times are not so large that the user becomes frustrated or has trouble navigating (e.g., 500 milliseconds).

On Silicon Graphics workstations, pipelining seems to be the best method for distributing computation. However, a significant bottleneck often occurs at the *Rendering* stage, which cannot be further subdivided due to the constraint that only a single process may communicate with the graphics engine. Other forms of rendering parallelism exist in different types of graphics systems, and several of them have the potential to provide substantial, scalable speed gains by eliminating this constraint. Two approaches for coordinating the efforts of more than one drawing process are: 1) division in object space, and 2) division in screen space. In the case of object space parallelism (e.g., [39]), each processor produces a full-screen image of a portion of the scene; in addition to the usual color and pixel coverage information calculated at each pixel, the depth of the closest object (or several objects if rendering of transparent objects is to be implemented) is computed and kept with the pixel data. A separate processor, or more generally a hierarchy of processors, gathers pixel data from the scene processors and composites it by choosing the closest object at each pixel. In the case of image space parallelism (e.g., [43]), each processor is responsible for rendering only objects which map to a particular area of the screen. In either case, the scene is broken up into disjoint pieces, and separate processors work on separate pieces with little or no communication between them. Results of separate processors are composed to form a single image. These approaches to parallelism are promising topics for further study.

Chapter 7

Results

In many of the preceding chapters, we have presented results of tests to evaluate the effectiveness of particular algorithms used in our building walkthrough system. Section 4.1.3 contains results gathered using various visibility precomputation and real-time visibility determination algorithms; Section 4.2.4 contains detail elision results; Section 5.2.5 presents results gathered during memory management experiments; and Section 6.2 contains results of experiments with concurrent processing. In this chapter, we summarize our results and present overall statistics gathered during a "full-blown" interactive walk through the Soda Hall model using combinations of these previously described algorithms.

We ran two experiments: one that tests only display management algorithms (i.e., visibility determination and detail elision), and another that also tests memory management algorithms (i.e., pre-fetching objects into memory). Both experiments were performed on a Silicon Graphics VGX 320 workstation with two 33MHz MIPS R3000 processors, 64MB of memory, and a $16\mu s$ 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 (queue depth = 2) – i.e., (UVD)(2R). 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 (queue depth = 12), and database input/output operations in a separate fourth process (queue depth = 64) – i.e., (UVD)(2R)(12M)(64I).

In both experiments, we used the observer path shown in Figure 7.1. It contains over 8,000 observer viewpoints, and visits the top four floors of Soda Hall.

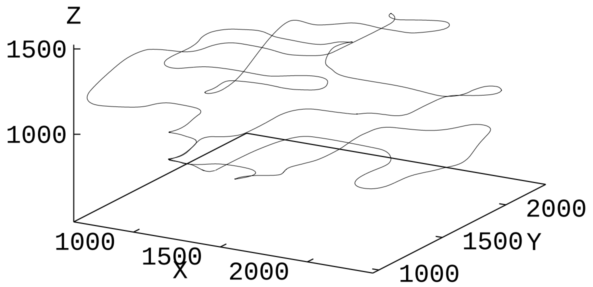


Figure 7.1: Test observer path through the top four floors of Soda Hall.

7.1 Display Management

In the first experiment, we tested the combined effectiveness of the display management algorithms described in this thesis, independent of memory management. We used the 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 four tests using the following combinations of 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.
- **Cell-to-Object:** Renders each object in the cell-to-object visibility of the observer cell at its highest level of detail – i.e., uses (CTO, None, None) visibility determination without any detail elision.
- **Eye-to-Object:** Renders each object in the eye-to-object visibility of observer viewpoint at its highest level of detail – i.e., uses (CTO, ETC, ETO) visibility determination without any detail elision.
- **Detail Elision:** Render each object in the eye-to-object visibility of observer viewpoint (i.e., (CTO, ETC, ETO)) with the level of detail and rendering algorithm (not texture) chosen by the optimization detail elision algorithm to keep the rendering time under one tenth of a second.

During each test, we measured the frame time (i.e., total time between successive frames), the response time (i.e., total time between a user action and the resulting image), the real-time visibility determination compute time, the detail elision compute time, and the rendering time, 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 7.1–7.3. Figure 7.2 contains a plot of the frame time for each observer viewpoint along the test path during the test using detail elision.

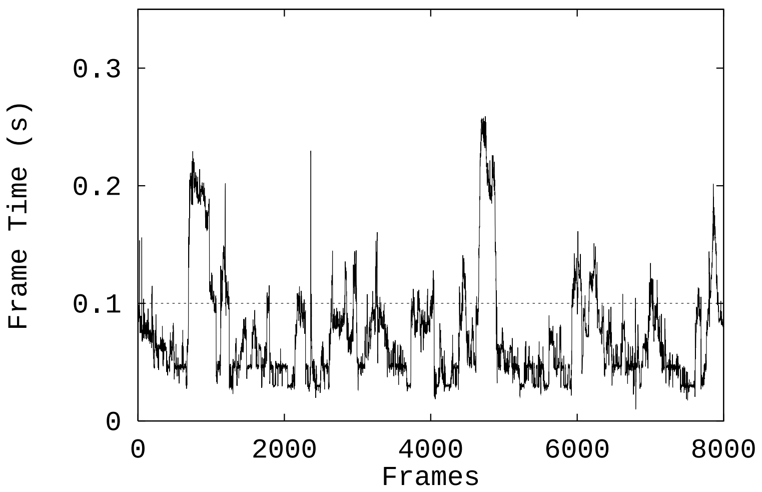


Figure 7.2: Frame time for each observer viewpoint along the entire test observer path during the detail elision test.

During the test in which we rendered every object in the entire model at the highest level of detail, the graphics engine processed all 1,418,807 polygons during every frame and the frame time was 17.937 seconds on average.

During the cell-to-object test, in which we used precomputed visibility information stored in the display database, and considered only objects in the cell-to-object visibility of the observer cell without any real-time visibility determination or detail elision (CTO, None, None), we rendered only 45,691 polygons (3.22% of the model) during each frame and generated an image every 0.616 seconds on average (1.6 frames per second). The maximum frame time was 2.137 seconds.

Cull Method	Frame Time (s)		Response Time (s)	
	Mean	Max	Mean	Max
Entire Model	17.937	19.388	53.619	55.352
Cell-to-Object	0.616	2.137	1.874	7.000
Eye-to-Object	0.104	0.624	0.311	1.872
Detail Elision	0.072	0.259	0.147	0.500

Table 7.1: Mean and maximum frame time and response time statistics collected during display management tests.

Cull Method	Visibility Time (s)		Detail Time (s)		Rendering Time (s)	
	Mean	Max	Mean	Max	Mean	Max
Entire Model	–	–	–	–	17.937	19.388
Cell-to-Object	–	–	–	–	0.617	2.152
Eye-to-Object	0.047	0.217	–	–	0.095	0.619
Detail Elision	0.047	0.215	0.016	0.093	0.044	0.121

Table 7.2: Mean and maximum compute time and rendering 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%	
Cell-to-Object	126	322	545	1,608	45,691	169,102	3.22%	11.92%
Eye-to-Object	19	67	111	469	5,300	45,829	0.37%	3.23%
Detail Elision	19	67	108	465	1,533	5,175	0.11%	0.36%

Table 7.3: Mean and maximum numbers of cells, objects, and polygons rendered during display management tests.

During the eye-to-object test, in which we also used the eye-to-object real-time visibility determination algorithm (CTO, ETC, ETO) but still no detail elision, we rendered only 5,300 polygons (0.37% of the model) on average and the mean frame time was 0.104 seconds (9.6 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 (465 objects containing 45,829 polygons at the highest level of detail). At these viewpoints, the frame time increased to 0.624 seconds (1.6 frames per second).

During the detail elision test, in which we also used the optimization detail elision algorithm to select a possibly reduced LOD for each potentially visible object, we rendered only 1,533 polygons (0.11% of the model) and generated an image every 0.072 seconds (13.9 frames per second) on average. The maximum rendering time was 0.121 seconds, and the the maximum response time was 0.500 seconds – short enough for the system to maintain an interactive feel.

Figure 7.3 shows rendering times for each observer viewpoint along the portion of the walkthrough path in which the scene visible to the observer is most complex. In the eye-to-object and cell-to-object tests, the rendering time was correlated with the complexity of the scene visible to the observer (or observer cell). Whereas, in the detail elision test, the rendering time did not exceed the target frame time of 0.1 seconds during this portion of the walkthrough. Although this fast, uniform rendering time was achieved by rendering simpler representations for some objects during some frames, the visual effects of detail elision were barely noticeable during this test.

Although the maximum rendering time (0.121 seconds) was very close to the target rendering time (0.100 seconds) during the detail elision test, the maximum frame time was much larger (0.259 seconds). Large frame times occurred when the compute time dominated the rendering time. There is currently no adaptive control over the time required for computation in our system. 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 rendering time and the sum of the visibility determination and detail elision compute times. Figure 7.4 shows the measured frame time, rendering time, and compute time for each frame along the same portion of the walkthrough path shown in Figure 7.3. The process performing visibility and detail elision

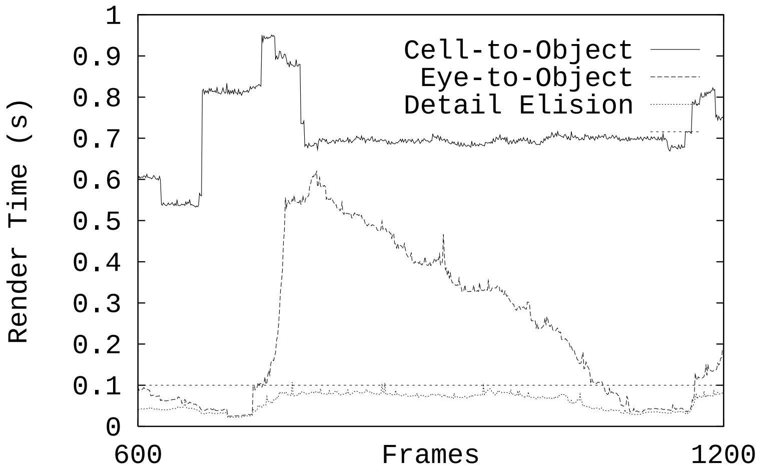


Figure 7.3: Rendering times using cell-to-object, eye-to-object, and detail elision cull methods for each observer viewpoint along complex portion of test observer path.

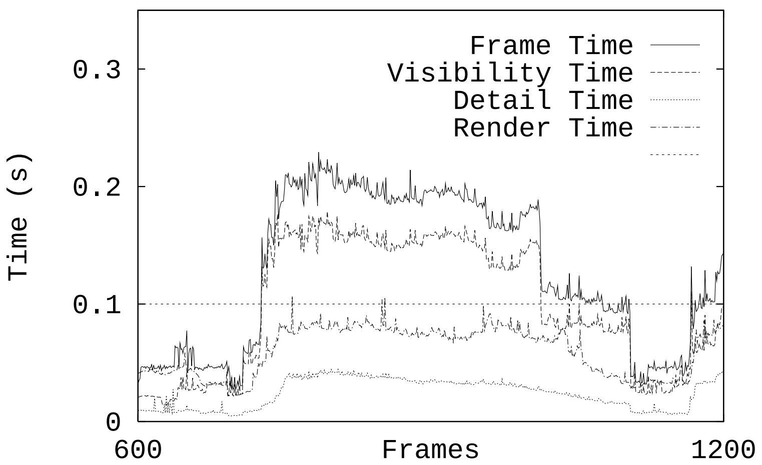


Figure 7.4: Frame time, visibility time, detail elision time, and draw time using detail elision cull method for each observer viewpoint along complex portion of test observer path.

computations was the bottleneck during this portion of the test path, and thus determined the effective frame time. However, note that frame times would have been far greater if either the visibility determination or detail elision computation were not performed. 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.

7.2 Memory Management

In the second experiment, we tested the cumulative effectiveness of the display and memory management algorithms described in this thesis. We used the same display management algorithms as were used in the test labeled "detail elision" in the previous section (i.e., (CTO, ETC, ETO) visibility determination and *Optimization* detail elision). 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 5 to swap objects in and out of memory in real-time as the observer moved along the test walkthrough path. The memory management algorithm was configured to use the *Observer Direction* range cull, the *Observer Frustum* lookahead cull, and *Object* lookahead granularity, while the lookahead depth was set to 1 frame in advance. This combination of parameters causes the lookahead set to be a rather small superset of the objects actually visible to the observer, and thus makes the most frugal use of available memory.

In addition to the statistics measured during the display management experiment, we gathered statistics regarding the lookahead compute time (i.e., time required to execute the pre-fetch memory management algorithm), the read time (i.e., the time required to load objects into memory from disk), the sizes of rendered, lookahead, read and release sets, and the number of LODs skipped by the rendering process during each frame due to failure of the pre-fetch process to load the appropriate LOD into memory in time before it is selected for rendering. Mean and maximum statistics for all observer viewpoints along the test walkthrough path are shown for tests with and without memory management in Tables 7.4–7.5. Table 7.6 contains mean and maximum statistics regarding the lookahead sets collected during the memory management test. Figure 7.5 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, we were able to maintain 13 frames per

Mem Mgmt	Frame Time (s)		Response Time (s)		LODs Skipped	
	Mean	Max	Mean	Max	Mean	Max
No	0.072	0.259	0.147	0.500	–	–
Yes	0.077	0.319	0.131	0.381	2.56	109

Table 7.4: Mean and maximum frame time, response time, and numbers of LODs skipped statistics collected during tests with and without memory management.

Mem Mgmt	Vis Time		Detail Time		Render Time		Look Time		Read Time	
	Mean	Max	Mean	Max	Mean	Max	Mean	Max	Mean	Max
No	0.047	0.215	0.016	0.093	0.044	0.121	–	–	–	–
Yes	0.051	0.229	0.016	0.119	0.048	0.127	0.003	0.220	0.030	1.613

Table 7.5: Mean and maximum visibility determination, detail elision, rendering, lookahead, and read time statistics collected during tests with and without memory management. All times are in seconds.

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%	
Rendered	108	465	1,533	5,175	0.246	0.757	0.07%	0.22%
Lookahead	299	1,229	25,287	124,072	3.857	17.539	1.10%	5.02%
Read	1	190	98	4,929	0.014	0.695	0.004%	0.20%
Released	1	862	88	80,232	0.013	11.346	0.004%	3.25%

Table 7.6: Mean and maximum set statistics collected during memory management tests.

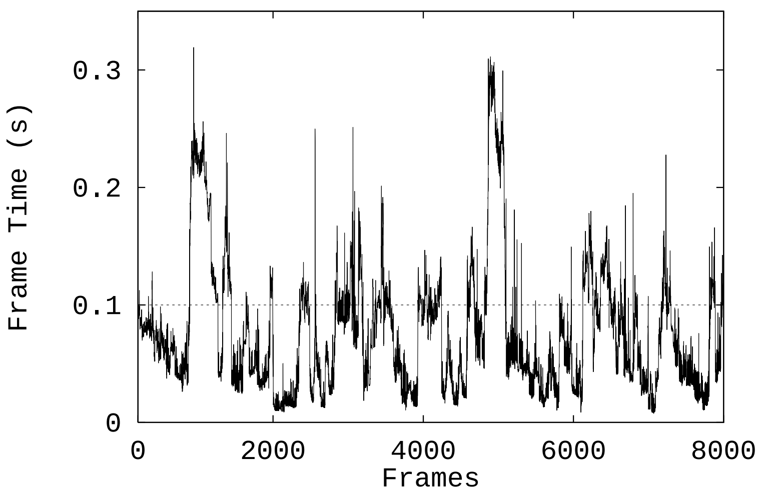


Figure 7.5: Frame time for each observer viewpoint along the entire test observer path during the test with memory management.

second (0.077 seconds per frame) during an interactive walkthrough on a workstation containing only 64MB of memory. Of the 349.5MB of data in the entire model, only 246KB of data was required to describe the polygons rendered during each frame on average. The lookahead algorithm was able to determine a relatively small set of object descriptions to store in memory (3.857MB on average) by predicting which object descriptions were most likely to be rendered during upcoming frames. The lookahead computation required very little time (0.003 seconds per frame on average), and executed in a separate asynchronous process with a 12 frame input queue, so had very little impact on the effective frame time.

During each frame on average, 14KB of data describing objects new to the lookahead set was read from disk into memory, requiring 0.030 seconds per frame. Although the read time was highly variable (the maximum read time was 1.613 seconds), read operations were performed in an asynchronous database input/output process with a 64 frame input queue, 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 any object description 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, 2.56 LODs were skipped on average, and as many as 109 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 several sequences 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.

Chapter 8

Conclusion

8.1 Summary and Reflections

This thesis describes a system for interactive walkthroughs of very large architectural models. It builds a hierarchical display database containing objects represented at multiple levels of detail during a modeling phase, performs a spatial subdivision and visibility analysis during a precomputation phase, and uses real-time display and memory management algorithms during a walkthrough phase to 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 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 uniform, fast 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 that it is possible to swap objects in and out of memory as a simulated observer walks through a model that is much larger than can fit into memory. We use an asynchronous pre-fetch 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 future frames. An approximate least-recently used algorithm determines which objects to replace in the memory resident cache as new objects are added. A database system manages storage of visualization data structures in memory and on disk, swizzling pointers to memory resident data so that direct access by application-defined functions is efficient.

We have implemented a first version of a building walkthrough system, and tested it in real walkthroughs of a furnished model of Soda Hall, the planned Computer Science building at UC Berkeley. Our initial results show that these display and memory management techniques are effective at culling away substantial portions of the model, and make interactive frame rates possible even for very large models.

8.2 Comparison to Other Approaches

Many other approaches using different combinations of precomputation and real-time computation are possible. For instance, at one extreme, a visualization system could perform all calculations during precomputation, generating complete images off-line, and then display a sequence of these precomputed images under interactive user control. The advantage of this approach is that very realistic-looking images can be generated off-line using either view-dependent rendering algorithms too expensive to run in real-time (e.g., ray tracing) or a camera if a physical model exists. Image display time is most critical and does not depend on how an image was generated. On the other hand, precomputed images have several potential disadvantages: they may require a large amount of storage, introducing a real-time memory management problem; there may be limits on the resolution and/or speed at which precomputed images can be displayed in real-time; and the simulated observer may be limited to viewpoints for which images have been precomputed. Also, since precomputed images must be updated when objects move or change appearance, this approach does not seem well-suited for visualization of dynamic models.

At the other extreme, a visualization system could perform no precomputation and render images completely in real-time during an interactive walkthrough. The advantage of this approach is that it supports arbitrary observer viewpoints and highly dynamic models with moving objects. However, currently available rendering systems are generally not able to produce realistic lighting simulations for complex models without any precomputation.

We have taken an intermediate approach – we precompute observer-independent intermediate rendering results that reduce the amount of real-time computation required to generate reasonably realistic-looking images (e.g., with radiosity) and interactive frame rates (e.g., visibility determination and detail elision). Although the storage requirements of precomputed radiosity information can be quite large (i.e., a color for each vertex), they are not nearly as large as would be required for precomputed images (i.e., a color for each pixel of each image). Our approach supports arbitrary observer viewpoints, and can handle moving objects without radiosity illumination.

Since graphics workstations are continually gaining computing and graphics rendering speed, and models are becoming more dynamic, with animated and moving objects, we believe that there will be a trend towards real-time computation during visualization. However, no matter how much faster or bigger computer workstations become, there will always be interesting models that are too large to be rendered at interactive frame rates without some amount of precomputation. For instance, today we are walking through a building – tomorrow we may be walking through a city, or a country, or a universe, etc. The appropriate partition between precomputation and real-time computation will always depend on the goals and constraints of a particular visualization system.

8.3 Final Observation

Although the focus of this thesis is data structures and algorithms for improving frame rates during visualization of complex models, the most important lesson learned during this work is that generating interesting, detailed models is DIFFICULT. In our case, it has taken almost as much time to construct our current model of Soda Hall as it has to develop algorithms to visualize it! Approximately eighteen "person months" were spent during conversion of $2\frac{1}{2}$D models received from the architects into a consistent 3D representation suitable for rendering; six "person months" were spent creating multi-resolution models for various pieces of furniture; and six "person months" were spent placing instances of the furniture models into the building model. Although some of the models can be re-used in other projects (e.g., the furniture models), most of this effort must be replicated for each new building model. Clearly, modeling tools must be developed that are more user-friendly and more automatic in order to make interactive visualization of complex three dimensional environments a reality, even a virtual reality.

Bibliography

[1] Akeley, K. Reality Engine Graphics. To appear in *Computer Graphics (Proc. SIGGRAPH '93)*.

[2] 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.

[3] 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.

[4] 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.

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

[6] Batory, D. GENESIS: A Project to Develop an Extensible Database Management System. *Proceedings of 1986 International Workshop on Object-Oriented Database Systems.* Asilomar, California, September, 1986.

[7] Baum, Daniel, R., Stephen Mann, Kevin P. Smith, and James M. Winget. Making Radiosity Usable: Automatic Preprocessing and Meshing Techniques for the Generation of Accurate Radiosity. *Computer Graphics (Proc. SIGGRAPH '91)*, 25, 4 (August 1991), 51-60.

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

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

[10] 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.

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

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

[13] Butterworth, Paul, Allen Otis, and Jacob Stein. The GemStone Object Management System. *Communications of the ACM*, 34, 10 (October, 1991), 64-77.

[14] Carey, Michael, J., David J. DeWitt, Joel E. Richardson, and Eugene J. Shekita. Object and File Management in the EXODUS Extensible Database System. *Proceedings of the Twelfth International Conference on Very Large Databases*, Kyoto, Japan, August, 1986, 91-100.

[15] Cohen, Michael, F., and Donald P. Greenberg. The Hemi-cube: A Radiosity Solution for Complex Environments. *Computer Graphics (Proc. SIGGRAPH '85)*, 19, 3 (July 1985), 31-40.

[16] Chen, Peter, M., Edward K. Lee, Ann L. Drapeau, Ken Lutz, Ethan L. Miller, Srinivasan Seshan, Ken Shirriff, David A. Patterson, and Randy H. Katz. Performance and Design Evaluation of the RAID-II Storage Server. *International Parallel Processing Symposium Workshop on I/O in Parallel Computer Systems*, April, 1993.

[17] Chervenak, Ann, L., and Randy H. Katz. Perfomance of a Disk Array Prototype. *Proc. SIGMETRICS*, May, 1991.

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

[19] Deux, O., et al. The O_2 System. *Communications of the ACM*, 34, 10 (October, 1991), 34-48.

[20] 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.

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

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

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

[24] 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.

[25] Funkhouser, Thomas A., and Carlo H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments. To appear in *Computer Graphics (Proc. SIGGRAPH '93).*

[26] 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.

[27] Garlick, Benjamin, Daniel R. Baum, and James M. Winget. Interactive Viewing of Large Geometric Databases Using Multiprocessor Graphics Workstations. *SIGGRAPH '90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation).*

[28] 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.

[29] Hanrahan, Pat, and David Salzman. A Rapid Hierarchical Radiosity Algorithm. *Computer Graphics (Proc. SIGGRAPH '91)*, 25, 4 (August 1991), 197-206.

[30] Heckbert, Paul, S. *Simulating Global Illumination Using Adaptive Meshing.* Ph.D. thesis, Computer Science Division (EECS), University of California, Berkeley, 1991. Also available as Technical Report UCB/CSD 91/636.

[31] 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.

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

[33] 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.

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

[35] Katz, Randy, H., Peter M. Chen, Ann L. Drapeau, Edward K. Lee, Ken Lutz, Ethan L. Miller, Srinivasan Seshan, and David A. Patterson. RAID-II: Design and Implementation of a Large Scale Disk Array Controller. *1993 Symposium on Integrated Systems.* Also available as UC Berkeley technical report UCB/CSD 92/705.

[36] 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.

[37] Lamb, Charles, Gordon Landis, Jack Orenstein, and Dan Winreb. The ObjectStore Database System. *Communications of the ACM*, 34, 10 (October, 1991), 50-63.

[38] Lohman, Guy, M., Bruce Lindsay, Hamin Pirahesh, and K. Bernhard Schierfer. Extentions to Starburst: Objects, Types, Functions, and Rules. *Communications of the ACM*, 34, 10 (October, 1991), 94-109.

[39] Molnar, Steven, Eyles, John, and Poulton, John. PixelFlow: High-Speed Rendering Using Image Composition. *Computer Graphics (Proc. SIGGRAPH '92)*, 26, 2 (July 1992), 231-240.

[40] 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.

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

[42] Patterson, David, A., Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proc. ACM SIGMOD*, June, 1988, 109-116.

[43] Potmesil, Michael and Hoffert, Eric M. The Pixel Machine: A Parallel Image Computer. *Computer Graphics (Proc. SIGGRAPH '89)*, 23, 3 (July 1989), 69-78.

[44] 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.

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

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

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

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

[49] 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.

[50] Silicon Graphics, Inc. *Graphics Library Programming Tools and Techniques*, Document #007-1489-01, Silicon Graphics, Inc., Mountain View, CA, 1992.

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

[52] Stonebraker, Michael, and Greg Kemmitz. The POSTGRES Next Generation Database Management System. *Communications of the ACM*, 34, 10 (October, 1991), 78-92.

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

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

[55] 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.

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

[57] Ward, Greg. Lawrence Berkeley Labatories. Personal Communication, 1993.

[58] 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.

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

[60] 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.