Interactive Walkthrough of Large Geometric Databases

Course Notes for SIGGRAPH ’95

Course Organizer

Eric L. Brechner
The Boeing Company

Course Speakers

James Helman
Silicon Graphics Inc.

Ned Greene
Apple Computer

Jarek Rossignac
IBM T.J. Watson Research Center

Thomas A. Funkhouser
AT&T Bell Laboratories

Abstract

This course will focus on techniques, algorithms, data-structures, and databases for displaying very large geometric databases interactively (greater than 1 million polygons drawn at least 10 frames per second). Beginning with a discussion of basic techniques and leading to state-of-the-art algorithms, the speakers will address key issues in walkthrough, including visibility computations, automatic object simplification, and memory management through database subset pre-fetching. Speakers will show real applications of these algorithms to a variety of areas, including visual simulation, virtual reality, architecture, and digital mockup.

Attendees of this course will better understand the key issues in dealing with very large geometric databases. They will be provided basic and state-of-the-art techniques to overcome hardware and software limitations that preclude the high frames rates necessary for interactive inspection of complex geometric scenes. These techniques will be illustrated by real examples of working walkthrough applications using databases with between 1 million and 500 million polygons.

A working knowledge of interactive computer graphics, including the mechanisms of matrix transformation, perspective, and raster graphics, will form a good foundation for the course material.

Course Schedule

A. Introduction - 10 minutes

   Statement of the problem. Introduction of the speakers. Agenda. - Eric Brechner

B. Graphics Techniques for Walkthrough Applications - 75 minutes

   An introduction to techniques common to most walkthrough applications with examples from the IRIS Performer toolkit. The methods include multi-tasking, view frustum culling, level-of-detail, frame rate control, and database paging. - James Helman

- **Morning Break**

C. Hierarchical Visibility and Tiling - 75 minutes

   Hierarchical approaches to accelerating visibility computations in extremely complex scenes. - Ned Greene

- **Lunch**

D. Geometric Simplification - 75 minutes

   The generation and exploitation of multi-resolution graphic models for the interactive visualization of complex mechanical or architectural 3D scenes. - Jarek Rossignac

- **Afternoon Break**

E. Database Management - 75 minutes

   Algorithms for computing and pre-fetching a small subset of a disk-resident database to store in memory during an interactive walkthrough. - Thomas Funkhouser

- **Seventh-inning Stretch**

F. Wrap-up and Future Directions - 20 minutes

   Hierarchical level of detail, culling, and data staging for interactive walkthrough of infinite databases, including a 3D Web browser. - Eric Brechner

Contents

List of Figures

List of Tables

About the speakers

Eric L. Brechner
Senior Principal Scientist
The Boeing Company
P.O. Box 24346, M.S. 7L-43, Seattle, WA 98124-0346
brechner@boeing.com[1]

Eric Brechner is a Senior Principal Scientist in the Research and Technology Division of Boeing Computer Services. He has been with Boeing for five years working in the areas of large scale visualization, computational geometry, network communications, data-flow languages, and software integration. He was the principal architect of FlyThru™, the walkthrough program for the 20GB, 500+ million polygon model of the Boeing 777 aircraft, authoring three invention disclosures relating to that work. He was also one of the principal architects of FlyThru™ Core and CAD Core. He is currently the Principal Investigator for the Access Manager Integration Framework and a new parametric solid modeler. Before coming to Boeing, Eric had worked in computer graphics and CAD for Silicon Graphics Inc., GRAFTEK, the Rensselaer Design Research Center (formally the Center for Interactive Computer Graphics), and the Jet Propulsion Laboratory. He holds a BS and MS in mathematics, and a PhD in applied mathematics from Rensselaer Polytechnic Institute.

James Helman
Member Technical Staff
Silicon Graphics Inc.
2011 N. Shoreline Blvd, Mountain View, CA 94043-1389
jimh@sgi.com

Jim Helman works in Silicon Graphics' Advanced Graphics Division as a member of the engineering team for IRIS Performer, SGI's real-time graphics toolkit. Before coming to SGI, he was a student in the Applied Physics Department at Stanford University where he worked on a PhD in data visualization. His interests include virtual environments, game design, and keeping large green cars running.

[1] Eric Brechner is now at Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, *ericbrec@microsoft.com*

Ned Greene

Apple Computer and U.C. Santa Cruz
MS301-3J, 1 Infinite Loop, Cupertino, CA 95014
greene@apple.com

Ned Greene is a member of the Advanced Technology Group at Apple Computer in Cupertino, California, where he conducts research in computer graphics. He holds a masters degree in computer science from New York University and is currently completing a PhD at the University of California at Santa Cruz. From 1980 to 1989 Ned worked at the Computer Graphics Lab at The New York Institute of Technology where he developed software for computer animation and contributed to pioneering animation projects such as The Works and The Magic Egg. Over the years he has been a frequent contributor to the SIGGRAPH technical program and Electronic Theater.

Jarek Rossignac

Senior Manager
IBM T.J. Watson Research Center
PO Box 704, Yorktown Heights, NY 10598
jarek@watson.ibm.com

Jarek Rossignac is Senior Manager of the 3D Graphics and Interactions Department at IBM Research, covering a wide range of activities in CAD, graphics, and virtual reality. He joined IBM Research in 85 with a PhD in EE on Solid Modeling from the University of Rochester. In 89, he created the Interactive Geometric Modeling group and managed research projects in topological representations and algorithms for CAD and in the interactive design and inspection of complex animated assemblies. He published over 40 papers winning three Best Paper external awards (Computer and Graphics'90, Eurographics'91, and IEEE Computer Graphics and Applications'93). He authored 12 invention disclosures in modeling and graphics and received numerous IBM internal awards (including the Best Paper award from Computer Science and a Research Division Award for the 3D Interaction Accelerator software system, which supports graphics and VR interactions with 3D CAD models of industrial complexity, i.e.: comprising millions of faces). Rossignac lectures at SIGGRAPH and at Eurographics. He is Associate Editor of four professional journals (ACM Transactions on Graphics, Computer-Aided Design, The Visual Computer, and Computer Graphics Forum) and Guest Editor for 7 special issues (three in Computer-Aided Design, one in the International Journal of Computational Geometry and Applications, two in the IEEE Computer Graphics and Applications, and one in the ACM transactions on Graphics). He refereed several hundred manuscripts and numerous grant proposals, serves in various international conference committees (including Eurographics 91-93-94 and SIGGRAPH 93-94). Rossignac is also co-chairman of the ACM/SIGGRAPH Symposium on Solid Modeling for 91-95, of the Eurographics'94 Workshop on Graphics Hardware. He is co-chairman of the International Program Committees for Eurographics'96 and for CAD/Graphics'95. Finally, He serves as Program Director for SIAM's Activity Group on Geometric Design.

Thomas A. Funkhouser
MTS
AT&T Bell Laboratories
600 Mountain Avenue, 2A-202, Murray Hill, NJ 07974
funk@research.att.com

Thomas Funkhouser is a member of the technical staff at AT&T Bell Laboratories. His research interests include multi-user systems, global illumination, and algorithms for managing large amounts of three-dimensional data in interactive computer graphics and visualization systems. He is a principal developer of the UC Berkeley Architectural Walkthrough System which is able to maintain thirty frames per second during interactive visualization of a building model containing 1.5 million polygons. He received a B.S. in biological sciences from Stanford University in 1983, a M.S. in computer science from UCLA in 1989, and a PhD in computer science from UC Berkeley in 1993.

Lecture A

Introduction

Eric L. Brechner
The Boeing Company
P.O. Box 24346, M.S. 7L-43, Seattle, WA 98124-0346
brechner@boeing.com[1]

Section 1

Slides

[1]Eric Brechner is now at Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, *ericbrec@microsoft.com*

The Boeing Company
Research and Technology

Interactive Walkthrough of Large Geometric Databases

Eric L. Brechner
May 19, 1995

The Boeing 777

The Boeing Company
Research and Technology

- The Boeing 777 is the first commercial airliner to be designed entirely on a computer.
- Its 3 million parts were all modeled as CATIA solids using ~20GB of storage and consisting of over 500 million polygons.
- To avoid building physical mockups at enormous expense, a digital walkthrough of the database was required to verify the design and make corrections.
- The graphics throughput necessary to view sections of the aircraft was between 150–250 million polygons per second.

How was it done?

The Boeing Company
Research and Technology

- How do you display 250 million polygons per second?

  Don't draw them all.

- How do you choose which ones to draw?

  Sit back, relax –– that's what this course is all about.

Statement of the problem

The Boeing Company
Research and Technology

The problem addressed by this course is that of displaying a large geometric database at an interactive display rate.

For the purposes of this course:

- A geometric database is one that consists of polygons or objects convertible to polygons.
- A large geometric database contains the equivalent of 1 million polygons or more.
- An interactive display rate is at least 10 new frames per second.

Therefore, the problem becomes how to display at least 10 million polygons per second.

Depth of the problem

The Boeing Company
Research and Technology

- Typically, the database will be much larger than 1 million polygons, and the display rate needed will be much higher than 10 frames per second.
- The Boeing 777 has over 500 million polygons.
- Virtual reality displays require 60 frames per second (two displays each running at 30 frames per second).
- In the extreme, a VR display of the entire 777 would require a display rate of over 30 billion polygons per second.

Course Schedule

The Boeing Company
Research and Technology

1. Introduction – 10 minutes
   Eric Brechner, The Boeing Company
2. Graphics Techniques – 75 minutes
   James Helman, Silicon Graphics Inc.

- Morning Break

3. Hierarchical Visibility and Tiling – 75 minutes
   Ned Greene, Apple Computer

- Lunch

4. Geometric Simplification – 75 minutes
   Jarek Rossignac, IBM T.J. Watson Research Center

- Afternoon Break

5. Database Management – 75 minutes
   Thomas Funkhouser, AT&T Bell Laboratories

- Seventh–inning Stretch

6. Wrap–up and Future Directions – 20 minutes
   Eric Brechner, The Boeing Company

Section 2

Introduction

Earlier this year, the first commercial airliner to be designed entirely on a computer was delivered to United Airlines. The Boeing 777 marked a new era for The Boeing Company. Before this plane, several complete physical mockups were needed for a new airplane design. At enormous cost, these mockups would be built to verify the design and then moved to some corner of the production bay to test new configurations and gather dust. With the 777, the first airplane built was delivered to the customer for passenger travel.

Using a new process to design an airplane introduces new problems. The Boeing 777 was designed on a cluster of IBM mainframes running CATIA computer-aided design software. All 3 million parts were modeled as solids. In the end there was roughly 20GB of CATIA data defining more than 500 million polygons. Although CATIA provided functionality to view and edit several hundred parts at once, engineers needed to view many tens of thousands of parts together to verify the design and make corrections. Without that capability, complete physical mockups of the airplane would have been required.

In addition, the engineers needed to move through the design, examining many aspects and alternatives in order to spot problems and take corrective actions. From the standpoint of graphics performance, they needed a system to display between 150-250 million polygons per second of airplane geometry. This level of hardware performance is simply not available today, nor was it available three and a half years ago when the 777 was going through design reviews.

How was it done? That is the subject of this one day course. Although the actual algorithms used remain proprietary, the basic techniques are the same as those published in recent years. These course notes have been prepared by some of the leaders in what has become known as high-performance walkthrough.

2.1 Statement of the problem

The problem addressed by this course is how to draw a geometric database, i.e., a database consisting of polygons or objects convertible to polygons, that contains more polygons than can be drawn by current hardware at interactive rates. Current commercial hardware can support up to 5 million polygons per second. For the purposes of this discussion, a geometric database containing over 1 million polygons will be considered large, and an interactive display rate will be considered 10 new frames per second. By these measures, a large geometric database requires drawing at least 10 million polygons per second, twice the rate available on current commercial hardware.

Typically, however, the database will be much larger than 1 million polygons, and/or the update rate will need to be much higher than 10 frames per second. Either way, a small improvement to the hardware will not account for the large disparity between what is required and what is available.

As was stated earlier, the Boeing 777 database contains over 500 million polygons. Other databases discussed in these notes include the 42 thousand polygon model of the new SGI building, the 167 million polygon model of the Empire State Building, the 3 million polygon model of a factory, and the 1.4 million polygon model of UC Berkeley's Computer Science Building. In addition, some applications require higher frame rates than 10 frames per second. Most virtual reality applications require as many as 60 frames per second (two displays each running at 30 frames a second). Thus, in the extreme, to display the entire 777 airplane in a VR environment requires a rate of over 30 billion polygons per second.

2.2 Overview of the notes

Jim Helman, the first speaker, will address the basic issues and algorithms of walkthrough. Included is a further discussion of the requirements for walkthrough, key performance improvement measures, maintenance of constant frame rates, handling database paging, and a description of the Performer commercial walkthrough toolkit.

The speakers that follow will go into greater depth on key problem areas. Ned Greene, the second speaker, will discuss object visibility and introduce a variety of hierarchical methods to reduce the number of objects that require rendering. Jarek Rossignac, the third speaker, will discuss object simplification and present a wide range of techniques to reduce the number of polygons needed to render each object. Tom Funkhouser, the fourth speaker, will discuss how database management issues interact with visibility computations, multiple levels of detail for objects, and movement through a scene.

Finally, Eric Brechner, the organizer, will provide a wrap-up of the issues presented and a discussion of future directions. The principle problem introduced will be that of displaying an infinite database of information, followed by the application of the solution to an interactive 3D Web browser.

Lecture B

Graphics Techniques for Walkthrough Applications

James Helman
Silicon Graphics Inc.
2011 N. Shoreline Blvd, Mountain View, CA 94043-1389
jimh@sgi.com

Section 3

Graphics Techniques for Walkthrough Applications

3.1 Introduction

Improvements in the performance and cost are enabling practical applications of computer graphics in many new areas. Interactive manipulation and walkthrough of very complex geometric databases are currently pushing the limits of technology. A number of application domains fall into this category. Architectural visualization has long been a research topic [5] only recently has the combination of quality and cost enabled practical use in areas such as urban planning[16]. Another related problem is the design and verification of large mechanical systems. Because the combination of simulation and visualization offers to substantially reduce the expense of prototyping and design verification, this area of simulation-based design is among the most promising areas. With a complete 3D graphics representation of a design that can be interacted with in real-time, it becomes possible to verify aspects of design ranging from aesthetics to ergonomics, performance and maintainability. In this section we concern ourselves primarily with one aspect of the problem, namely using computer graphics to generate the visual display in real time. But in actual applications, this is just one small portion of a much larger task that can involve elaborate simulations, connections to manufacturing databases and interaction with powerful M/CAD systems.

As with most problems in graphics, rendering these large databases in real time would be much more tractable if computation were instantaneous, data transfers infinitely fast and memory free. But comparing the requirements of large-scale M/CAD and walkthrough applications against the capabilities of even the most powerful of today's computer graphics systems, we find a number of interesting challenges posed by each of the finite limitations of our tools.

Many of these challenges are similar those encountered in visual simulation. The frame rate and fidelity requirements of visual simulation have always pushed the limits of technology and have driven the development of new techniques in computer graphics; texturing, hardware antialiasing and level-of-detail are among the more prominent.

The requirements of virtual environment technology are also relevant because many walkthrough and simulation-based design systems require visual immersion.

3.2 Motivation of Performance Requirements

Of all the sensory information we need to provide, the visual component is certainly the most important in producing the illusion of a virtual environment. The ideal would be to provide simulated input with fidelity that matches or exceeds the limits of human visual perception in all aspects. Fortunately, in practice, this is not a prerequisite for producing a usable virtual environment. But in designing a system, we need to always keep the capabilities of the human visual system in mind because for the foreseeable future the cost and limitations of graphics technology will force many tradeoffs to be made.

The human visual system is complex and even basic perceptual thresholds defy simple characterization, usually depending on a variety of factors. The following briefly discusses some of the perceptual limits and how they have been addressed in visual simulation systems.

3.2.1 Visual Acuity

Spatial resolution limits can be measured in many ways and depend strongly on many factors including brightness, color, contrast, off-axis eccentricity, length of exposure and retinal illumination.

Visual acuity is commonly measured in terms of the angle subtended at the eye. For reasonably bright objects, on-axis, the limit is around 1 minute of arc. This corresponds roughly to a 20/20 result in a standard vision test, which indicates the ability to recognize letters which subtend 5 minutes of arc. Sensitivity to spatial frequencies is measured by the contrast required to perceive vertical bars and is largest at around 5 cycles/degree [6]. This indicates that resolutions of around 5 minutes of arc are required to get into the region of peak sensitivity, a limit important for creating sharp edges.

For comparison, a first generation LCD-based head-mounted display (HMD) with around 185 RGB triads across an 75 degrees field of view per eye [23] yields a resolution of around 24 arcmins, or around 20/480 vision. To make resolution matters worse, typically optical blurring was often employed to help fuse the red, green and blue pixels [26].

Even a "high-resolution" display with 1280 pixels across a narrower 60 degrees field of view per eye achieves only around 3 arcmins, which also happens to be the standard for out-the window commercial flight simulators (Level C) set by the Federal Aviation Administration (FAA) requires 3 arcmin resolution [10].

But acuity falls off rapidly as the object moves outside the central 2 degree region, and at 10 degree of off-axis eccentricity, it has dropped to around 10 arcmin. So displays that generate a high resolution constant pixel density are wasting a lot of pixels wherever the user is not looking.

3.2.2 Field of View

A sense of "immersion" is important for a number of applications ranging from architectural visualization (for a sense of scale and presence) to task training or usability verification. Since field-of-view must usually be traded off against angular resolution, the value added by a wide field-of-view needs to be carefully weighed against the cost.

Each eye has approximately a 150 degrees horizontal field of view (60 degrees towards the nose and 90 degrees to the side) and 120 degrees vertically (50 degrees up and 80 degrees down) [6]. Again using visual simulation experience as a benchmark albeit for one particular task, the FAA standard for commercial out-the-window flight simulators (Level C) requires 75 degrees horizontal and 30 degrees vertical fields of view [10].

Head-mounted displays vary in the amount of overlap they provide between the left and right eye's fields of view. Spreading the fields-of-view creates a wider overall field of view, but usually at some cost in image quality because of disparate distortion from the display optics in the overlap region. Because of other problems such as blending of brightness at the borders of overlap, one of the few studies in this area has found that 100performance on a visual search task [9].

3.2.3 Latency

Latency is the time measured from the setting of an input until the corresponding output is manifested. Many factors contribute to latency: input devices, software architecture, rendering time, display scanout. Different portions of a system may have different latencies, e.g. the response of the visuals to changes in eye point might differ from the collision response. Any path from an input to an output in Figure 3.1 could take a different amount of time.

For the rendering portion of the system, the latency is typically taken as the time after some value, such as the eyepoint, is set until the last pixel of the corresponding frame is scanned out by the display device. While most of the literature on the impact of latency on task performance come from visual simulation, they at least provide some rough guidelines for other tasks with similar time responses. Reports from flight training indicate that the user-perceived quality of the simulation degrades steadily for total latencies over 100ms [3]. Other studies have indicated that latencies below 100ms have little effect [7]. The required latencies are application dependent. According to Padmos, in out-the-window simulators, latencies typically range from 40-80ms for driving simulators to 100-150ms for low-maneuverability flight simulation [21]. As is well known from aircraft control theory, excessive latency can lead to over-compensation and control oscillations induced by the operator.

3.2.4 High Frame Rate

Generally, most displays have refresh rates close to or above the flicker limit (60-80Hz) of the human visual system, so the display itself introduces no significant problems. But frame rate is another issue. Low frame rates make motion choppy and are especially problematic when rapid motion is possible as with a head-mounted display. Perhaps

Figure 3.1: Sources of latency in a hypothetical virtual environment system.

most significantly, a low frame rate increases latency which can dramatically affect task performance in a virtual environment.

But there are also some purely visual effects. When going from an update rate that matches the video rate, e.g. 60Hz, to some fraction of the video rate, e.g. 30Hz, there is a qualitative difference in the fluidity of motion. The reason for the qualitative difference has to do with the same image being scanned out several times before being changed. In general, the lower the update rate, the choppier the motion. But there is another effect that occurs when the eye is tracking a moving object. Because the eye is moving continuously, but the same image is being scanned out multiple times on the display, objects appear to split into multiple copies along the direction of motion with the number of ghosts equaling the refresh rate divided by the update rate. As shown in Figure3.2, a simulation running at 20Hz update on a display refreshing at 60Hz, the object will appear tripled as shown in Figure 3.2. On large objects such as horizon silhouette, the effect manifests itself as multiple edges.

A similar effect occurs on field-sequential color displays found in some high-resolution head-mounted displays. Since each field is being drawn at 3X the video rate, even when the rendering update rate is equal to the video frame rate, color ghosting occurs and especially noticeable on high contrast edges.

Temporal antialiasing or motion blur would mitigate these effect, but poses other problems both in performance (since most methods are multipass) and in realism (since in reality an object which your eye is following does not blur). In practice, smooth mo-

Figure 3.2: Multiple image artifact when fixating on moving object.

tion and the absence of ghosting are best achieved by having the update rate equal the display refresh rate.

3.2.5 Constant Frame Rate

Varying frame rates pose a number of additional perceptual and practical issues. Consequently, most visual simulators require fixed frame rates. Frame rate variations tend to distract the user from the task at hand and are particularly problematic for tasks involving interaction with objects or performance evaluation of physical tasks.

Frame rate variations also cause temporal inaccuracies because the change in frame rate affects the latency. An unanticipated change in the latency means that frame is not displayed at the time for which it was planned. This creates jerkiness in motion seen when most systems make frame rate transitions. In many cases when a graphics overload is about to occur, it's better to force a degradation in frame rate so that the change in latency can be taken into account, rather than having an uncorrected change in latency.

3.3 Performance in Graphics Pipelines

Rendering on most workstation-based 3D graphics subsystems can be regarded as a pipeline with one or more stages that can include database traversal, polygon processing and pixel processing. Maximizing frame rate and image quality becomes a problem of making the best use of each of the available stages and avoiding bottlenecks. A detailed treatment of performance issues in pipelined graphics systems can be found in [11].

Database traversal is typically done on a host CPU and can become a bottleneck if the traversal is unable to keep the graphics pipeline full. This commonly happens because of high traversal overhead or poorly optimized loops that send commands to the graphics subsystem.

Polygon processing including vertex transformations, lighting calculations and 2D triangle setup is sometimes done by host the CPU, but on high-performance systems of interest here such as RealityEngine[2], the processing is usually done by one or more specialized transformation engines in the graphics subsystem. Performance of these systems usually improves when dealing with long meshes of triangles, relatively few mode changes such as texture or material switches, and relatively few, infinite lights. Performance is usually lower for independent triangles, surface primitives such as NURBS, scene with many transformations, many lights, and many mode changes.

Polygon performance is often reduced to a simple number of triangles per second. While such numbers can provide rough information about relative performance, real application performance, particularly in scenes with many transforms, textures and materials, these numbers are only part of the story. To achieve peak performance, attention must be paid to optimizing both the geometry and the associated graphics modes.

Pixel processing involves operations such as depth buffer testing, multisample antialiasing, transparency blending and texturing, stenciling and typically involves intensive memory access. Performance is most often affected by the number of memory accesses required and hence can depend on the number and type of pixel modes that are enabled, e.g. whether the geometry is textured and if so, what order interpolation is being done. Pixel fill rates can also depend on the size of the primitive being rendered, since there is often some setup cost to be amortized.

When depth buffering is used for hidden surface removal, some pixels on the screen may be painted or at least depth tested many times. The average over the rendering area of the number of times a pixel is touched is known as the depth complexity. On many graphics systems, clearing the rendering area counts as one layer of depth complexity, but some architectures have specialized clearing modes that take close to zero time, e.g. RealityEngine's fast multisample depth clear. Pixel fill is rated in millions of pixels per second. The pixel fill requirement for an application can be estimated by multiplying the frame rate times the deph complexity.

3.4 Reducing Geometric Complexity

3.4.1 Texture Mapping

The widespread use of hardware texture mapping originated in visual simulation as a technique to provide a great deal of visual complexity with a much smaller amount of geometry than would otherwise be required. Texture mapping allows the rendering of fine detail within an object down to the pixel resolution of the display without requiring a geometry down to the level of a triangle per pixel. The use of partially transparent

Figure 3.3: View into a simple town using texture mapping and billboards.

texture maps also allows objects with complex outlines to be rendered with little geometry.

Rich scenery can be generated by combinations of modeled geometry and textures. But even with texture mapping a large amount of geometry can be required, particularly in areas of mechanical design where shape and geometry are of principal interest.

3.4.2 Billboards

Billboards are a specialized rendering trick from visual simulation that involve using texture mapping on a geometry that rotates to face the eye point, most commonly a single polygon. Billboards are most often used for rendering distant objects that are radially or spherically symmetric. The rotation is constrained about an axis for objects that are nominally radially such as the trees shown in figure 3.3 or left unconstrained about a point for spherically symmetric objects. The use of billboard geometry improves performance primarily by reducing geometric complexity. In simulation-based design, billboards are most useful for radially symmetric objects such as pipes and for textual annotations using either geometric or texture mapped fonts, e.g. the pfText node in IRIS Performer.

3.4.3 Level of Detail

3.4.4 Geometric

The single most important technique developed for managing scene complexity is providing multiple levels of detail for objects in the scene so that simpler versions can be rendered when an object is small on the display. In visual simulation, typically every object in the scene is modeled with multiple levels of detail and ranges specified at which the different models should be selected. Typically a object might be modeled at 3-5 different levels of detail. The view of a town shown in Figure 3.3 contains 9500 triangles, without level-of-detail the number would be 16000.

But man large-scale database, the models to be rendered come not from polygonal modeling but are imported from an M/CAD system, most frequently as a surface representation such as NURBS or as polygonal data. And they are too numerous and too complex for manual generation of level-of-detail models to be practical, so automatic generation of mulitple resolution models from the available representations becomes very important.

Surface representations such as NURBS lend themselves to automatic generation of different model resolutions by virtue of retessellation, at least to a point. For coarse representations, the separate patches that make up an object must often be combined.

A significant amount of recent investigation has done into the development of techniques that can be applied to polygonal data [28], [29], [14], [25].

3.4.5 Texturing

In addition to merely simplifying geometry, level of detail can be accomplished using texture maps to reduce geometric complexity. The simplest form of this involves rendering objects onto the faces of a box, and then using the texture mapped box instead of the original geometry[19].

3.5 Optimizing Runtime Rendering

Database optimization can be divided somewhat into two categories, those which can be done as a preprocessing stage and those which must be done at run time, although in some applications tasks that would normally be done in a preprocessing stage must be done at run time because of changes in a dynamic database. We first consider some of the run-time optimizations available for use by graphics applications. In many cases, we'll make reference to how a particular feature is handled in IRIS Performer[24], which is Silicon Graphics toolkit for real-time graphics applications.

3.5.1 Visibility Culling

In visual simulation and virtual environments, the user is often in the midst of the database but each visual channel (HMD eye or projection screen) only looks out in one direction. Sending all the information to the graphics system and letting figure out what's visible is extremely wasteful. Culling of the geometric objects to the viewing frustum

Figure 3.4: Transition from a higher level-of-detail model to a lower one using fade (top) and morphing (bottom).

can easily be done on the host CPU and substantially increases performance by reducing the amount of data transfer and view testing done in the graphics subsystem. This feature is essential for any graphics toolkit designed to support virtual environments. Since IRIS Performer maintains a hierarchical scene graph, cull testing is accomplished by testing the viewing frustum against a set of hierarchical bounding volumes which are spheres at internal nodes and boxes at leaf nodes.

More advanced types of visibility culling incorporate occlusion of objects within the database. Common methods for architectural interiors are based on determining what is visible through portals such as windows or doors[1][27]. In environments with many large occluders such as walls or compartments, advanced culling is essential for high performance.

Simpler techniques with looser acceptance are also possible, for example simple frustum culling can be continued based upon the screen aligned bounding box of a portal[18].

Another simple approach is to divide the database into cells and compute or specify which other cells are visible from each cell. At run time, portions of the scene graph are turned on or off depending on what cell the viewer is in. This technique is most applicable to highly compartmentalized databases such as mazes or tunnels found in many 3D game[17].

3.5.2 Level of Detail Switching

Unless the viewer is restricted to a small range of positions, the size of objects will vary greatly depending among different viewpoints. Using 100 polygons to represent an object which is only 2 pixels high is clearly a waste when a much simpler model would be indistinguishable. If one had several versions of the model suitable for rendering at different resolutions, one could switch to simpler ones as appropriate, and when the object is smaller than a pixel (or a subsample) switch it out entirely. Visual simulators and modeling packages support the specification and automatic switching of such models.

But there are a number of ways to effect an LOD transition:

Hard switching. At some size, the simpler model replace the more complex model. This meets the performance goal, but can cause visual ‘popping’ when the model is changed.

Fading. Instead of simply switching the models, for some range of distances or period of time the both models are drawn blended together. Such a transition is show in the top half of Figure 3.4. This substantially reduces ‘popping’ but temporarily increases the rendering load while both models are being drawn. Also, simple alpha transparency is not adequate for the blending, since hidden surface removal must still be done within the object. One method is the use of complementary multisample masks on Silicon Graphics RealityEngine graphics subsystem.

Morphing. Instead of blending, continuously morph one object into the other as shown in the lower half of Figure 3.4. This probably has the best visual and performance effect, but few modeling packages currently support this method of specifying LOD models.

IRIS Performer currently supports hard switching and fading, with support for morph transitions under development.

3.5.3 Maintaining Constant Frame Rate

Ideally a database would be structured so that for all possible viewpoints in the simulation, it could be rendered in the allotted frame time with the default level-of-detail selection. This is quite possible in artificially created databases in which the modeler has complete freedom, but when the geometry in the database is dictated by a desire to simulate an actual system or mechanical design, objects are frequently distributed in a very non-uniform manner, and so the scene complexity varies dramatically for different eye points.

One approach to maintaining constant frame rates in this case is to simply limit the frame rate so it never runs faster than the rate of the slowest view into the database. Another approach is to try to modify what is being rendered in order to keep up the frame rate. For example, by changing LOD selection as the graphics system becomes “stressed,” constant frame rates can be maintained [13], at least when rendering is geometry rather than pixel fill limited.

IRIS Performer uses this LOD modification mechanism for load balancing. High and low water marks are specified as a fraction of the frame time. When the actual rendering time exceeds the high water mark, the system enters “stress” and lower LODs are switched in until the system returns to the target hysteresis band between low and

Figure 3.5: Graphics Load Management.

high as shown in Figure 3.5. This mechanism works when the graphics rendering is geometry limited. If the rendering is limited by pixel fill, the system may still overload and exceed the allotted frame time. So in the end, constant frame rate still relies heavily on the modeling of the scene.

Some traditional image generators have additional mechanisms for dealing with overload. When rendering from front to back, rendering can sometimes simply be stopped before completion without substantial visual anomaly, especially if the rendering is being fogged. Another approach is to vary the number of pixels per line or even the number of lines being rendered and have the display device compensate correspondingly. Since this reduces the number of pixels to be rendered, it is appropriate when rendering is pixel fill limited and complementary to stress-based LOD selection.

3.5.4 Parallel Processing

In most graphics workstations, the geometric database resides on the host CPU and objects to be rendered are transferred to the graphics subsystem every frame (with the exception of some state such as textures which are often cached in the graphics subsystem). In some architectures, the host CPU carries out some of the rendering operations such as transformation and lighting. But in most real applications this rendering or data transfer operation is only one of the tasks which needs to be done every frame. Some of the tasks are graphics related, such as view frustum culling and level-of-detail selection. Others are application specific, such as behaviors, physics, collision detection, networking, and device I/O. When all of these tasks are performed sequentially, the graphics subsystem may be left idle a significant fraction of the time.

Some of the tasks form a natural pipeline. Most behaviors and application-specific modifications to the scene graph must be completed before culling can be done and culling and level-of-detail selection must be done before transferring primitives to the graphics subsystem. IRIS Performer uses this division to define a set of pipeline processing stages which can happen in parallel. The model includes:

1. APP Application processing and modifications to the scene graph.

2. CULL Scene graph traversal, view frustum culling, level-of-detail selection, graphics mode sorting

3. DRAW Transferring data to the graphics subsystem.

4. ISECT Picking and collision detection based on testing line segments. against geometry.

The concurrent rather than serial processing of the rendering stages allows a processor to be dedicated to keeping the graphics subsystem fully occupied, thereby ensuring full utilization of what is often the most expensive hardware component in the system.

The pipeline stages are configurable so that a single APP process can drive multiple graphics subsystems through the creation of a CULL/DRAW pair per graphics hardware pipeline.

3.5.5 Mode Sorting

One performance optimization handled by IRIS Performer that is rarely done in applications is sorting objects in the scene by graphics mode. This allows data to be sent to the graphics pipe in an order which minimizes the number of mode changes caused by material or texture changes. During the cull or rendering traversal, the objects to be rendered can be accumulated and before being sent to the graphics subsystem, they can be sorted to minimize the number of changes made to the most expensive graphics mode. On current Silicon Graphic's machines, texturing is the most expensive mode to change, so IRIS Performer draws all objects that use the same texture together. And within a particular texture, objects are grouped by other mode attributes. The benefit is strongest in databases with many small pieces of textured geometry in which it can increase performance by up to 2X.

3.5.6 Latency Control

Given that even the highest performance graphics hardware is currently unable to create images which meet the limits of our perceptions, we want to be keeping the graphics subsystem fully occupied.

One way to do this is to design an architecture which dedicates a processor whose sole responsibility is to feed the graphics system. IRIS Performer has a pipelined architecture which allows the application processing, collision testing, culling and drawing

Figure 3.6: Pipelined processing model for multiprocessor graphics workstations.

each to run on a separate processor. Figure 3.6 shows the passage of a single frame through the multiprocessed pipeline running with a frame rate half the video refresh rate, e.g. 30Hz update with 60Hz refresh. As shown by the collision process taking more than a single frame, an overrun by a downstream process does not cause the pipe to back up. When a downstream process starts, it always picks up the most recent upstream result.

This model has high throughput since the application, cull and draw processes can all take up to a frame time without disrupting fixed frame rate operation. Unfortunately the latency is also long, 3 frame times + 1 retrace time from the start of the application process to the last pixel on the screen. This can be improved on, if the arrival of latency-critical information is available synchronously, i.e. the last byte from the head tracker can be guaranteed to arrive just before the start of the cull process. In this case, rendering latency from the arrival of that data can be reduced to 2 frame times + 1 retrace, i.e. 83.3ms at a 30Hz frame rate with 60Hz video.

As mentioned earlier, the synchronous transmission of data can be simulated by measuring response time of the tracker after polling and then regularly polling the device that long before the start of the frame.

Even lower latency can be achieved with only a small decrease in throughput by having the draw process to overlap with the cull process as shown in Figure 3.7. This slightly decreases graphics throughput because the draw process starts later and the cull process cannot make certain reordering optimizations that might otherwise be possible.

Figure 3.7: Trading off throughput for latency in pipeline multiprocessor.

3.6 Optimizing the Graphical Database

3.6.1 Triangle Meshes

Geometric data are often in the form of independent triangles, but may initially be in the form of CSG, boundary representations, surface patches or arbitrary polygons, most graphics hardware is designed to render triangles. So while many graphics libraries accept geometry as trimmed NURBS or non-convex polygons, usually the best performance is obtained by converting geometry into triangles as a preprocessing stage. On many platforms, such as RealityEngine, a significant performance gain can be achieve by generating strips of triangles which allows sharing of vertices of adjacent triangles rather than requiring each vertex to be transferred and processed completely separately.

3.6.2 Spatial Organization

Most 3D graphics applications and toolkits have some set of data structures to traverse and render. In IRIS Performer, the structure is a scene graph with a hierarchical set of bounding volumes. In order to optimize the time spent testing the view frustum against bounding volumes, it's best if the bounding volumes are as small as possible. This means that the database should be organized spatially so that adjacent leaf nodes in the scene graph are also proximal in world space.

Another issue of database structure has to do with the spatial organization and granularity of leaf nodes in the scene graph. In many textured databases, a fairly fine granularity is dictated by the need to change textures or materials fairly frequently. But often the untextured models common in M/CAD have thousands of polygons that could be grouped into a single object. However for purposes of culling, a much finer granularity is usually desirable so that if the large model is only partially in view, only a portion of it is sent to the graphics subsystem.

3.6.3 Graphics Attributes

As mentioned earlier, changes in graphics modes can often limit graphics performance. Sorting at runtime can help performance, but only if the number of different textures and materials is reasonable. In general, performance is optimized by using a few different textures and materials as required by the database. Sometimes applications can even benefit by combining multiple images into a single texture map and using texture coordinates to index into the different subimages.

3.6.4 Eliminating Transformations

Databases are usually modeled using many transformations in the scene graph, sometimes one per object. While important in the modeling process, they are not necessary for static portions of the database. By preprocessing the database on the host to remove all static transformations and transform the geometry in place, many unnecessary matrix operations can be eliminated.

3.6.5 Precomputed Animation

Rather than burdening the graphics system or the host CPU with transforming or morphing vertices for pre-scripted animations, geometry for each of the desired conformations can be generated during preprocessing and simply switched during run-time. Phenomena like an explosion or a flame can be created by animating transparent 2D textures in this manner.

3.7 Database Examples

3.7.1 Architectural Exteriors

The exterior scene shown in Figure 3.8 is a model of a new building recently built for Silicon Graphics. This model has 41,000 triangles and 70 different textures. The view shown in Figure 3.8 takes 110ms to render on a 150MHZ Onyx/RealityEngine2.

3.7.2 Architectural Interiors

Architectural interiors differ from exterior scenes in a number of ways. In particular, lighting effects are extremely important in interior spaces, so often the visual database includes precomputed radiosity solutions. While radiosity can be encoded into either texture maps or more finely subdivided goemetry, the latter tends to be more common because of the limited texture memory available in current graphics hardware.

Figure 3.8: Visualization of a building used for city zoning discussions.

Figure 3.9 shows a building interior with a radiosity solution generated by Lightscape Graphics Software. The database is untextured and contains approximately 11,000 triangles and takes about 16ms to render on a 150MHz Onyx/RealityEngine2. Figure 3.10 shows a more complex building containing 76,000 triangles and 17 different textures and takes about 120ms to render on the same machine.

3.8 Handling Very Large Databases

Another problem with solutions from visual simulation is the need to move through a database that is too "large." Too large can refer either to a database that is to large to fit in the RAM memory of the host computer or one that has too much textured imagery to fit in the graphics subsystems texture cache. Both of these require special consideration in application design.

3.8.1 Paging Geometry and Texture from Disk

When paging geometry and texture from disk into the active scene graph, the main issues are:

Achieving full I/O bandwidth. On most workstation-class machines, this requires structuring the data so that it can be read in large blocks preferably being DMAed straight into the application's address space. The time required for performing an extra data copy operation is particularly an issue when paging in large texture images.

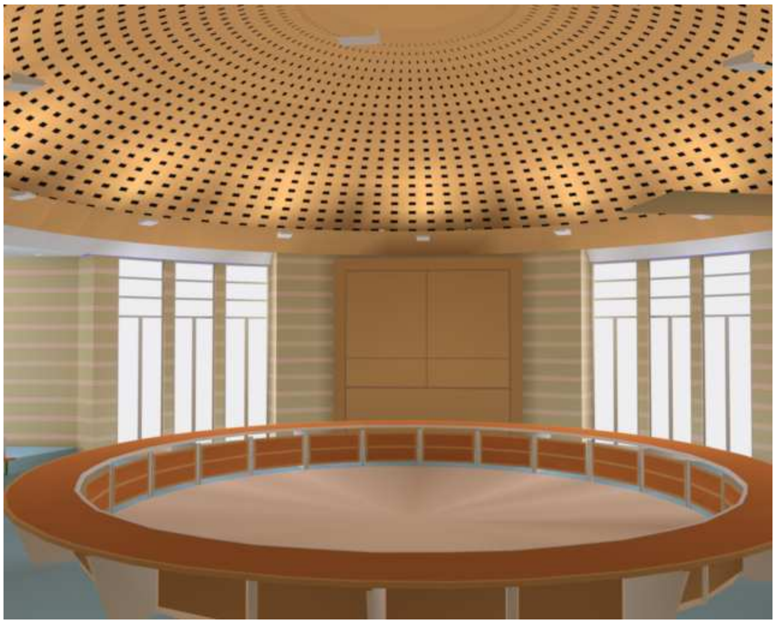


Figure 3.9: Building interior using radiosity solution, no texture mapping.

Figure 3.10: Architectural scene using radiosity solution and texture mapping.

Minimizing impact on frame rate. The paging in of data can either be done synchronously, a little bit every frame, or in a multithreaded application, the loading can be done asynchronously in a another process. When possible, asynchronous loading in a separate process is much less likely to impact the frame rate and is the approach used in IRIS Performer and some walkthrough applications, e.g. [13].

Accurate prediction and timing. In order to avoid arriving at a point in the scene and not having the correct data to render, the application must adequately predict and time the loading of data into the scene graph. In very large worlds with long visibilities, it's also important to have the data structured so that low-resolution level-of-detail models and textures can be loaded in without having to read all resolutions of the model or texture.

3.8.2 Paging Texture to Graphics Subsystem

Fast texture mapping usually requires caching texture images in the graphics subsystem. When the amount of texture memory is sufficient for rendering any one view, but not sufficient for storing the textures for the entire database, texture must be paged down to the graphics subsystem. Unlike paging from disk, this cannot be done asynchronously with rendering because on most graphics architectures texture downloads share the same data paths as normal rendering. So instead, a fraction of the rendering time must be set aside and made available for texture downloads. The issues of prediction and timing are similar to those for paging from disk, including the issues of only having to load the appropriate level-of-detail (e.g. mipmap level). When implementing texture paging it is also important to be aware of possible memory management issues, in particular the risk that repeated allocations and deallocations may produce fragmentation of the already limited texture memory. One rather drastic solution is to only page textures of one particular size.

3.9 Beyond the Rendering

While the graphical aspects of large-scale walkthroughs are undeniably challenging, they are only one element working in the service of a larger goal. This section discusses some of the other issues with which many applications have to deal and which are related to the visual database.

3.9.1 Interface with M/CAD Systems

For simulation-based design applications, one of the most important problems is being able to import and interactively modify portions of the database. In it's simplest form, importation can take the form of a database loader for a particular M/CAD format and modification, can be limited to changing the values of transformations in the scene graph. But more interesting applications of the technology require a tighter coupling between the M/CAD system and the visual application to enable more elaborate editing and manipulation within the environment.

Somewhat related is the need to access and modify non-geometric information related to a project, such as parts lists, engineering documentation, manuals, etc. This also requires an interface to an external database. But the relation to the visual database is weaker, except perhaps the need to bring up textual and 3D annotations within the scene.

3.9.2 Collision Detection

One of the purposes of simulation-based design is the verification of aspects of a design for ergonomics, accessibility, maintainability, etc. While most M/CAD packages have their own elaborate systems for interference testing, interaction in a virtual environment also requires real-time collision detection for everything from keeping one's feet on the floor to opening an compartment hatch. IRIS Performer provides a simple mechanism that tests line segments against the visible geometry using the same hierarchical bounding volumes that are used for cull testing. More elaborate methods for interference checking are also possible[8].

Some applications also need accurate computations of physical behavior. most typically some subset or approximation to Newtonian mechanics[20].

3.9.3 Networking

A common requirement in visual simulation and virtual environments is the ability to have more multiple participants collaborate (or shoot at each other). This requires maintaining a consistent state of the visual and other databases across a network. Simple protocols from visual simulation such as SIMNET [22] and DIS [15] provide a useful starting point, especially in the area of dead reckoning. But these protocols were designed specifically for applications including vehicles and combat personnel and have limited utility in other areas. Network protocols for virtual environments have been developed using broadcast, point-to-point[4] and server-based[12] approaches. But so far there is no standard outside of military simulation.

3.10 Conclusions

The requirements of real application will usually have many task specific requirements both in and outside of the domain of the graphics issues that we have focused on here. In particular, the users of applications in simulation-based design and architectural walkthroughs often require different types of information and interaction than is available within the visual database.

But many of the computer graphics techniques developed for visual simulation are applicable to the interactive walkthrough of large databases. In particular the well-established use of culling, level-of-detail, frame rate control, texture mapping, and database paging all carry over directly to large-scale virtual environments. Consequently, a real-time graphics toolkit like IRIS Performer that supports all these visual simulation features can provide a useful framework for developing walkthrough applications.

Bibliography

[1] John M. Airey, John H. Rohlf, and Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics (Proceedings 1990 Symposium on Interactive 3D Graphics)*, 24(2):41–50, March 1990.

[2] Kurt Akeley. Reality engine graphics. *Computer Graphics (Proceedings of SIGGRAPH '93)*, 27:109–116, August 1993.

[3] Peter Beckett. Effective cueing during approach and touchdown - comparison with flight. In *Piloted Simulation Effectiveness AGARD Conference Proceedings 513 (Brussels, 14-17 Oct 1991)*, pages 30.1–30.11, 1992.

[4] Brian Blau, Charles E. Hughes, Michael J. Moshell, and Curis Lisle. Networked virtual environments. In *Proceedings 1992 Symposium on Interactive 3D Graphics*, pages 157–160, June 1992.

[5] Frederick P. Brooks. A dynamic graphics system for simulating virtual buildings. In *Proceedings of the 1986 Workshop on Interactive Computer Graphics (October 23-24, 1986)*, October 1986.

[6] Pierre Buser and Michel Imbert. *Vision*. MIT Press, Cambridge, Massachusetts, 1992.

[7] Frank M. Cardullo. Virtual system lags: The problem, the cause, the cure. In Eric G. Monroe, editor, *Proceedings of the 1990 Image V Conference (Phoenix, 19-22 June 1990)*, pages 31–42. Image Society, 1990.

[8] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha, and Madhav Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings 1995 Symposium on Interactive 3D Graphics*, pages 189–196, June 1995.

[9] G. K. Edgar, K. T. Carr, M. Williams, J. Page, and A. L. Clarke. The effects upon visual performance of varying binocular overlap. In *Helmet Mounted Displays and Night Vision Goggles, AGARD Conference Proceedings 517 (Pensacola, Florida, 2 May 1991)*, pages 8.1–8.15, 1991.

[10] FAA. Airline Simulator Qualification. Advisory Circular 120-40B, Federal Aviation Administration, May 1989.

[11] Sharon Fischler. Performance trade-offs and database tuning. In *Designing Real-Time Graphics for Entertainment*. SIGGRAPH '94 Course Notes, August 1994.

[12] Thomas A. Funkhouser. Ring: A client-server system for mulit-user virtual environments. In *Proceedings 1995 Symposium on Interactive 3D Graphics*, pages 85–92, June 1995.

[13] Thomas A. Funkhouser and Carlo H. Sequin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *SIGGRAPH 93 Conference Proceedings (Computer Graphics, Annual Conference Series 1993)*, pages 247–254, August 1993.

[14] Hughes Hoppe, Tony DeRose, Tom DuChamp, John McDonald, and Werner Stuetzle. Mesh optimization. In *SIGGRAPH 93 Conference Proceedings (Computer Graphics, Annual Conference Series 1993)*, pages 19–26, August 1993.

[15] Ansi/ieee pstd 1278-1993 - standard for information technology, protocols for distributed interactive simulation. Technical report, March 1993.

[16] William Jepson, Robin Liggett, and Scott Friedman. An environment for real-time urban simulation. In *Proceedings 1995 Symposium on Interactive 3D Graphics*, pages 165–166, June 1995.

[17] Eric Johnston. High quality computer graphics in entertainment. In *Designing Real-Time Graphics for Entertainment*. SIGGRAPH '94 Course Notes, August 1994.

[18] David Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *Proceedings 1995 Symposium on Interactive 3D Graphics*, pages 105–114, June 1995.

[19] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *Proceedings 1995 Symposium on Interactive 3D Graphics*, pages 95–102, June 1995.

[20] Brian Mirtich and John Canny. Impuls-based simulation of rigid bodies. In *Proceedings 1995 Symposium on Interactive 3D Graphics*, pages 181–188, June 1995.

[21] Pieter Padmos and Maarten Milders. Checklist for outside-world images of simulators. In *Proceedings of ITEC '92, International Training Equipment Conference and Exhibition (Luxembourg, 7-9 April 1992)*, pages 2–14, 1992.

[22] Arthur Pope. The simnet network and protocols. Technical report, BBN Systems and Technologies, July 1989.

[23] Warren Robinett and Jannick P. Rolland. A computational model for the stereoscopic optics of a head mounted display. In *Stereoscopic Displays and Applications II*, pages 140–160. SPIE Proceedings Vol. 1457, May 1991.

[24] John Rohlf and James Helman. Iris performer: A high performance multiprocessing toolkit for real-time 3d graphics. In *SIGGRAPH 94 Conference Proceedings (Computer Graphics, Annual Conference Series 1994)*, pages 381–394, July 1994.

[25] Jarek Rossignac and P. Borrel. Multi-resolution 3D approximation for rendering complex scenes. In *Second Conference on Geometric Modelling in Computer Graphics*, pages 453–465, June 1993. Genova, Italy.

[26] Michael A. Teitel. The eyephone, a head mounted stereo display. In *Stereoscopic Displays and Applications*, pages 168–171. SPIE Proceedings Vol. 1256, February 1990.

[27] Seth Teller and Carlo Sequin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proceedings SIGGRAPH '92)*, 26(2):61–69, August 1992.

[28] Greg Turk. Re-tiling polygonal surfaces. *Computer Graphics (Proceedings SIGGRAPH '92)*, 26(2):55–64, August 1992.

[29] William E. Lorensen William J. Schroeder, Jonathan A. Zarge. Decimation of triangle meshes. In *SIGGRAPH 93 Conference Proceedings (Computer Graphics, Annual Conference Series 1993)*, pages 65–70, August 1993.

Lecture C

Hierarchical Visibility and Tiling

Ned Greene
Apple Computer and U.C. Santa Cruz
MS301-3J, 1 Infinite Loop, Cupertino, CA 95014
greene@apple.com

Section 4

Introduction

These notes are preliminary versions of two chapters from my PhD thesis, tentatively titled *Hierarchical Rendering of Complex Environments*, which I expect to complete in June of 1995. The thesis presents hierarchical methods to accelerate visibility and tiling in densely-occluded scenes composed of geometric primitives. The first section of these notes is a longer version of a paper on accelerating z-buffer tiling that I presented at Siggraph '93. The paper, called *Hierarchical Z-Buffer Visibility*, was written with two of my colleagues at Apple Computer, Michael Kass and Gavin Miller [13]. The second section presents work in progress on a novel hierarchical polygon tiling algorithm. The full thesis will be available by ftp from the University of California at Santa Cruz.[1] Microfilm copies and photocopies will be available from University Microfilms.[2]

Hierarchical Z-Buffer Visibility introduced what we call the *hierarchical visibility* algorithm, which may be summarized as follows. We maintain hierarchical data structures in both object space and image space which facilitate hierarchical culling of hidden geometry. To enable hierarchical culling in object space, we organize scene geometry in an object-space octree. Using a simple recursive subdivision procedure, we traverse octree nodes in front-to-back order, testing them for visibility, culling nodes that are hidden and rendering the geometry contained in visible nodes. This object-space culling procedure is very efficient because it traverses only visible octree nodes and their children, and only primitives contained in visible octree nodes need to be rendered. Thus, it very effectively culls most hidden geometry in densely-occluded scenes. However, some geometry will still overlap on the screen when primitives are projected. To cull remaining hidden geometry we perform Warnock-style hierarchical culling in image space [29], using an image-space hierarchy to maintain visibility information about previously-rendered geometry. Since the recursive subdivision procedures of this basic algorithm find visible geometry by logarithmic search, it is very efficient. It should be noted that our use of object-space and image-space hierarchies is very similar to Meagher's [18], even though his methods are only directly applicable to volume models rather than geometric models. The main limitation of the basic algorithm is its performance in animating dynamic scenes, since this requires maintaining the object-space

[1] Computer and Information Sciences at UCSC: 408-459-2320.

[2] University Microfilms: 800-521-0600. Probably not available until early 1996.

hierarchy, which can be costly if a large number of primitives are moving independently.

Within the hierarchical visibility algorithm, the basic visibility operations that cull hidden geometry and determine visible geometry are performed by *tiling*. The two variations of the basic algorithm presented here perform tiling in different ways. The algorithm of section 5 performs *z-buffer* tiling [6] in a hierarchical manner and generates standard point-sampled z-buffer images. The algorithm of section 6 employs a novel tiling algorithm called *hierarchical polygon tiling*, which is capable of producing antialiased images much more efficiently than oversampling with the z-buffer algorithm of section 5.

Section 5

Hierarchical Z-Buffer Visibility

5.1 Overview

In this section we present a very efficient algorithm for generating z-buffer images of very complex, densely-occluded scenes. The algorithm achieves efficiency by applying hierarchical methods to culling hidden geometry in both object space and image space, and by rendering visible geometry with the speed of z-buffer scan conversion.

To enable hierarchical culling of geometry in hidden regions of object space, we organize scene geometry in an octree. To render a frame, we recursively subdivide the octree in front-to-back order as with Meagher's volume rendering algorithm [18]. During traversal of the octree, octree nodes hidden by the z-buffer are culled, and geometry inside visible octree nodes is rendered into the z-buffer. Culling hidden octree nodes removes most hidden geometry, but some geometry still overlaps when projected onto the screen. To facilitate culling of remaining hidden geometry, z-buffer depth samples are maintained in a pyramid, which permits hierarchical culling in image space. Thus, the algorithm exploits coherence with both object-space and image-space hierarchies. For animation, the algorithm is also able to exploit temporal coherence by using geometry that was visible in the previous frame to construct a starting point for the algorithm. This appears to be the first visibility algorithm that materially profits from object-space, image-space, and temporal coherence simultaneously. For very densely-occluded scenes, the algorithm sometimes achieves orders of magnitude acceleration compared with ordinary z-buffer scan conversion. Although the algorithm performs general-purpose z-buffer rendering, we refer to it as a *visibility* algorithm because its novel features accelerate visibility operations and are unrelated to shading.

5.2 Introduction

Extremely complex scenes offer interesting challenges for visibility algorithms. Consider, for example, an interactive walk-through of a detailed geometric model describing an entire city complete with gardens and trees, buildings with furnishings, etc. Traditional visibility algorithms running on currently available hardware cannot come close to rendering scenes of this complexity at interactive rates and it will be a long time before faster hardware alone will provide the needed performance. In order to get the most out of available hardware we need faster algorithms that exploit properties of the visibility computation itself. Our ultimate objective is a visibility algorithm that does work proportional to the *visible complexity* of the scene in the output image, rather than the complexity of the overall geometric model. While this objective is unattainable because it would allow no work to be expended in culling hidden geometry, culling can be done very efficiently using hierarchical methods.

The key to accelerating visibility computations is exploiting three forms of coherence: object-space coherence, image-space coherence, and for animation sequences, temporal coherence. Ideally, a visibility algorithm should be able to exploit all of these forms of coherence. However, no traditional visibility algorithm succeeds in doing this. Traditional z-buffering effectively exploits image-space coherence but not object-space coherence. Conversely, ray casting through a spatial subdivision effectively exploits object-space coherence but not image-space coherence. Although image-space culling with a ZZ-buffer [25] improves the efficiency of ray casting, it is still necessary to traverse every primitive in the scene at every frame, which impairs performance for densely-occluded scenes. Prior work includes various attempts to simultaneously exploit both object-space and image-space coherence. In the domain of volume rendering, Meagher's algorithm effectively exploits both of these forms of coherence, but this method is not directly applicable to rendering geometric models [18]. Potentially visible set methods effectively exploit both object-space and image-space coherence (assuming z-buffering of primitives), but their usefulness is limited to models having certain geometric characteristics [28, 27]. Naylor's BSP-tree visibility algorithm culls in both object space and image space, but it has not actually been demonstrated to efficiently process densely-occluded scenes [20]. In short, no existing visibility algorithm has been demonstrated to effectively exploit both object-space and image-space coherence in processing arbitrary geometric models. For animation, there is the additional challenge of harnessing temporal coherence, which traditional algorithms rarely exploit in practice.

In this section we present a z-buffer visibility algorithm that exploits object-space coherence as effectively as ray casting through a spatial subdivision, exploits image-space coherence even more effectively than traditional incremental scan conversion, and for animation sequences is also able to exploit frame-to-frame coherence. To exploit object-space coherence, we use an octree spatial subdivision of the type commonly used to accelerate ray tracing [21, 23, 12, 17, 16, 15]. To exploit image-space coherence, we augment traditional z-buffer scan conversion with an image-space z-pyramid that allows us to cull hidden geometry very quickly. To exploit temporal coherence, we use the geometry that was visible in the previous frame to construct a starting point for the algorithm. The result is a z-buffer visibility algorithm which is orders of magnitude faster than traditional z-buffering for some densely-occluded scenes we have experimented with. The algorithm is not difficult to implement and it works for arbitrary geometric models consisting of polygons and other primitives that can be efficiently scan converted. Moreover, the algorithm has only modest memory requirements and it is amenable to parallel computation and to implementation in hardware.

In the section 5.3 we present the hierarchical z-buffer visibility algorithm, beginning with the data structures that it employs to exploit object-space, image-space, and temporal coherence. In section 5.4 we discuss methods for building and maintaining the octree spatial subdivision. In section 5.5 we describe our implementation and show results for some complex models containing hundreds of millions of polygons. Finally, in section 5.6 we state our conclusions.

5.3 The Hierarchical Z-Buffer Visibility Algorithm

The hierarchical z-buffer visibility algorithm uses an octree spatial subdivision to exploit object-space coherence, a z-pyramid to exploit image-space coherence, and to exploit temporal coherence in animation sequences, it keeps track of octree nodes that were visible in the preceding frame. While the full advantage of the algorithm is realized by using all three of these together, the octree and the z-pyramid can also be used separately. Whether used separately or together, these data structures make it possible to compute the same result as ordinary z-buffering at less computational expense.

5.3.1 The Object-Space Octree

Octrees have been used previously with great effectiveness to accelerate ray casting and ray tracing [23, 12, 17, 16, 15] and rendering of volume models [18]. With some important modification, the principles of this previous work can be applied to z-buffer rendering. The result is an algorithm which can accelerate z-buffering by orders of magnitude for models with sufficient depth complexity.

In order to be precise about the algorithm, we begin with some simple definitions. We will say that a polygon or other primitive is hidden with respect to a z-buffer if no depth samples on the tiled primitive are closer to the observer than the corresponding z values already in the z-buffer. Similarly, we will say that a cube is hidden with respect to a z-buffer if all of its faces are hidden polygons. Finally, we will call an octree node hidden if its bounding cube is hidden. Note that these definitions depend on the sampling of the z-buffer. A primitive that is hidden at one z-buffer resolution may not be hidden at another.

Testing Octree Cubes for Visibility

With these definitions, we can state the basic observation that makes it possible to combine z-buffering with an octree spatial subdivision. As schematically illustrated in figure 5.1, if a cube is hidden with respect to a z-buffer, then all geometry fully contained in the cube is also hidden. Actually, this principle applies to any bounding volume, not just a cube. It follows that if we tile the faces of a cube and determine that it is hidden, we can safely ignore all the geometry contained in that cube. The following pseudocode outlines a procedure for determining whether an octree cube is visible, returning either TRUE or FALSE.

```
IsCubeVisible(OctreeNode N)
{
```

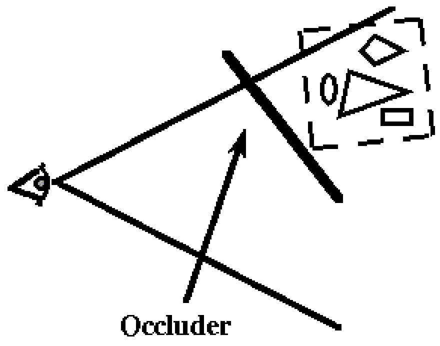


Figure 5.1: If a cube is hidden, then all geometry it contains is also hidden.

```
    if N is completely outside the viewing frustum
    then return FALSE
    if viewpoint is inside N
    then return TRUE
    if N intersects the near face of the viewing frustum
    then return TRUE
    for each front face F of N {
        if F is visible at one or more pixels
        then return TRUE
    }
    return FALSE
}
```

This procedure's first step is determining whether a node's bounding cube intersects the viewing frustum. We have developed a fast algorithm for the more general problem of detecting intersection of an axis-aligned rectangular solid and a convex polyhedron. Our implementation uses this method, which is presented in [14]. With this method, determining whether a cube intersects the viewing frustum requires evaluating between one and thirty inequalities derived from line and plane equations.

The Basic Rendering Algorithm

Given this cube visibility test, the basic rendering algorithm is easy to construct. We begin by organizing scene geometry into an octree. There are various ways of building an octree, but for the moment let's assume that each primitive is associated with the smallest enclosing octree cube. At the beginning of a frame, we clear the image buffer to the background color (or image) and clear the z-buffer to the far clipping plane. Then we start at the root node of the octree and process it using the following recursive steps.

If the octree node is not visible, either outside the viewing frustum or hidden by the z-buffer, we are done. Otherwise, we tile any primitives associated with the octree node into the z-buffer and then recursively process its children, if any, in front-to-back order using this same procedure. When the recursion finishes, we have a standard z-buffer image of the scene. This depth-first recursive subdivision procedure for rendering a scene is outlined in pseudocode below.

```
RenderScene(OctreeNode Root)
{
   clear image buffer to background
   clear z-buffer to far clipping plane
   ProcessOctreeNode(Root)
}

ProcessOctreeNode(OctreeNode N)
{
   if IsCubeVisible(N) returns FALSE
   then return
   for each primitive P associated with N
      tile P into the z-buffer
   for each child C of N in front-to-back order
      ProcessOctreeNode(C)
}
```

Note that procedure `ProcessOctreeNode()` requires front-to-back ordering of the octants of an octree node with respect to the viewpoint. Front-to-back ordering of octants during recursive subdivision guarantees strict front-to-back traversal of nodes in the octree. This is crucial to the algorithm's efficiency, because it guarantees that any geometry that can occlude a node is tiled into the z-buffer before that node is processed. As a result, even though the z-buffer is usually only partially formed when cube-visibility tests are made, these tests are definitive and succeed in culling all octree nodes that are hidden.

The nested, rectilinear structure of an octree makes it easy to establish front-to-back ordering of octants. We use the method described in [Foley-et-al90], whereby the octant corresponding to the nearest corner of the cube is known to be "frontmost," and the three octants which share a face with the frontmost octant all have the same visibility priority, just behind frontmost. Symmetrically, the octant opposite frontmost is "backmost," and the three octants which share a face with the backmost octant all have the same visibility priority, in front of backmost and behind all the other octants. Thus, this procedure clusters the eight octants of a cube into four groups of equivalent visibility priority, as illustrated in figure 5.2. This ordering algorithm works regardless of the cube's orientation and whether or not the viewpoint lies inside the cube.

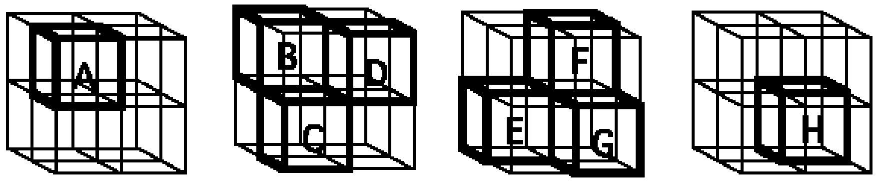


Figure 5.2: Ordering of a cube's octants into four clusters of equivalent visibility priority: {A},{B,C,D},{E,F,G},{H}

The basic rendering algorithm outlined above has some desirable properties. First of all, the algorithm only traverses and renders primitives contained in octree nodes that are visible. Some of the rendered primitives may be hidden, but as illustrated in figure 5.3, each primitive in a visible octree cube is "nearly visible" in the following sense: there is some place we could move it where it would be visible which is no further away than the length of the diagonal of its bounding cube. Thus, the algorithm only renders primitives which are visible or nearly visible. In addition, the algorithm only visits visible octree nodes and their hidden children, so it does not waste time on irrelevant portions of the octree. Finally, the algorithm never visits an octree node more than once during the rendering of a frame. This stands in marked contrast to ray casting through an octree, where the root node is visited for every pixel rendered, and other nodes may be visited thousands of times. As a result of these properties, the algorithm is very efficient at both culling hidden geometry and traversing visible geometry.

Building the Octree

Recall that scene geometry must be organized into an octree prior to rendering. We can construct the octree with a simple recursive procedure. Beginning with a root cube large enough to enclose the entire model and the complete list of geometric primitives, we perform the following steps recursively. If the number of primitives is sufficiently small, say, less than or equal to m, we associate all of the primitives with the cube and return. Otherwise, we associate with the cube any primitive that intersects any of the three axis-aligned planes that bisect the cube. We then subdivide the octree cube and call the procedure recursively with each of the eight child cubes and the list of primitives that fit entirely in that cube. When this recursive procedure finishes, each primitive is associated with the smallest enclosing octree cube in the hierarchy and each leaf node contains a maximum of m primitives.

One weakness of this algorithm for building octrees is that it associates some small primitives with large cubes if the primitives happen to intersect the planes that separate the cube's children. For example, a small triangle that crosses the center of the root cube will be associated with the root cube and it will need to be rendered anytime the entire model is not hidden. To avoid this behavior, there are two basic choices. One alternative is to clip the problematic small primitives so they fit in much smaller octree cells. This has the disadvantage of increasing the number of primitives in the model. The other al-

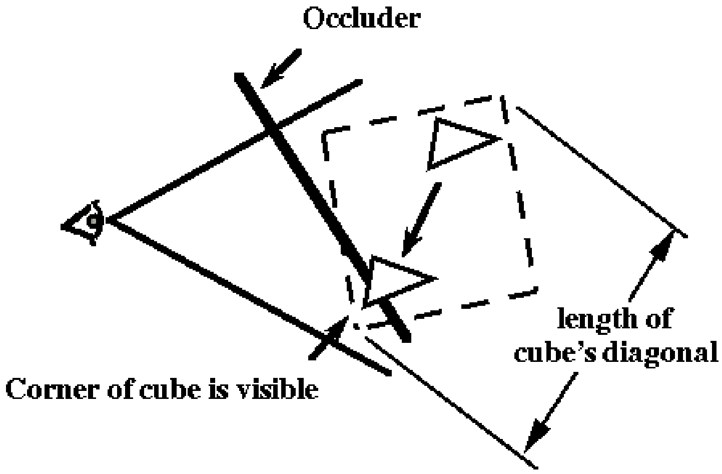


Figure 5.3: Any primitive inside a visible cube is within the length of the cube's diagonal of being visible.

ternative is to place some primitives in multiple octree cells. We chose to implement the latter alternative. To do this, we modify the recursive construction of the octree as follows. If we find that a primitive intersects a cube's dividing planes but is small compared to the cube, then we no longer associate the primitive with the whole cube. Instead we associate it with each of the cube's children that the primitive intersects.[1] This subdivision procedure continues recursively until the primitive is associated with cubes of the appropriate size. The same strategy can also be used to place long, skinny primitives into multiple cubes.

Primitives that are associated with more than one octree node may be encountered more than once during rendering. To avoid rendering a primitive more than once, we mark it with the frame number when it is rendered. This permits us to know whether a primitive has already been rendered in the current frame, without having to clear a flag for each primitive at the beginning of each frame.

5.3.2 The Image-Space Z-Pyramid

The basic rendering algorithm described in the preceding section spends most of its time tiling cube faces and primitives. The object-space octree allows us to cull large hidden portions of the model at the cost of tiling the faces of the visible octree cubes and their hidden children. Actually, only hidden cube faces need to be tiled completely, since encountering a visible pixel on a face establishes that a cube is visible, permitting tiling to stop. Even so, tiling of cube faces requires considerable computation because many

[1] For polygonal primitives, we test for cube-polygon intersection using the method presented in [14].

faces are large and faces may overlap densely on the screen, due to nesting of cubes. Primitives associated with visible cubes must also be tiled, and they also overlap on the screen, sometimes densely. Even if we employ fast incremental scan conversion, tiling of cube faces and primitives is very expensive, because it requires traversing each primitive pixel by pixel, even if it is entirely or mostly hidden. Thus, there is a need to reduce the tiling requirements of the algorithm. To accomplish this, we apply hierarchical methods to accelerate culling of hidden geometry in image-space.

To support hierarchical culling in image space, we maintain z-buffer depth samples in a *z-pyramid*, an image pyramid similar to those used in texture mapping (e.g. [30]) and image processing (e.g. [3]). Frequently, the z-pyramid makes it possible to conclude very quickly that a cube face or primitive is hidden, making pixel-by-pixel traversal unnecessary.

Figure 5.4 shows a z-buffer image of a densely-occluded scene and its corresponding z-pyramid. The finest level of the pyramid is an ordinary z-buffer. At all other levels, each z sample is the farthest z from the observer in the corresponding 2×2 window of the next finer level. Every entry in the pyramid therefore represents the farthest z for a square region of the screen. At the coarsest level of the pyramid there is a single z value that is the farthest z from the observer in the whole image.

Maintaining the z-pyramid is an easy matter. Every time we modify the z-buffer, we propagate the new z value through to coarser levels of the pyramid. As soon as we reach a level where the entry in the pyramid is already as far away as the new z value, propagation can stop since coarser levels of the pyramid will not be affected.

The method we use to test the visibility of primitives with respect to the z-pyramid is illustrated schematically in figure 5.5. First, we find the finest-level sample of the pyramid whose corresponding image region covers the screen-space bounding box of the primitive. Then, if the nearest z value of the primitive is farther away than this sample in the z-pyramid, we know immediately that the entire primitive is hidden. Very often, we are able to show with this single depth comparison that an entire cube, cube face, or primitive is hidden.

While the basic z-pyramid test can cull a substantial fraction of hidden primitives, it suffers from a similar difficulty to the basic octree method. Because of the structure of the pyramid, a small primitive covering the center of the image will be compared to the z value at the coarsest level of the pyramid. While the test is still accurate in this case, it is not very powerful.

A definitive visibility test can be constructed by applying the basic test recursively through the pyramid. If the basic test fails to show that a primitive is hidden, we go to the next finer level in the pyramid where the previous pyramid region is divided into four quadrants. Here we attempt to prove that the primitive is hidden in each of the quadrants that it intersects. For each of these quadrants, we compare the closest z value of the primitive in the quadrant to the value in the pyramid. If the pyramid value is closer, we know the primitive is hidden in the quadrant. If we fail to prove that the primitive is hidden in one of the quadrants, we go to the next finer level of the pyramid for that quadrant and try again. Ultimately, we either prove that the entire primitive is hidden, or we recurse down to the finest level of the pyramid and find a visible pixel. This recursive procedure is essentially a logarithmic search for a visible pixel. Note that if we find all visible pixels on a primitive this way, we are tiling the primitive hierarchically.

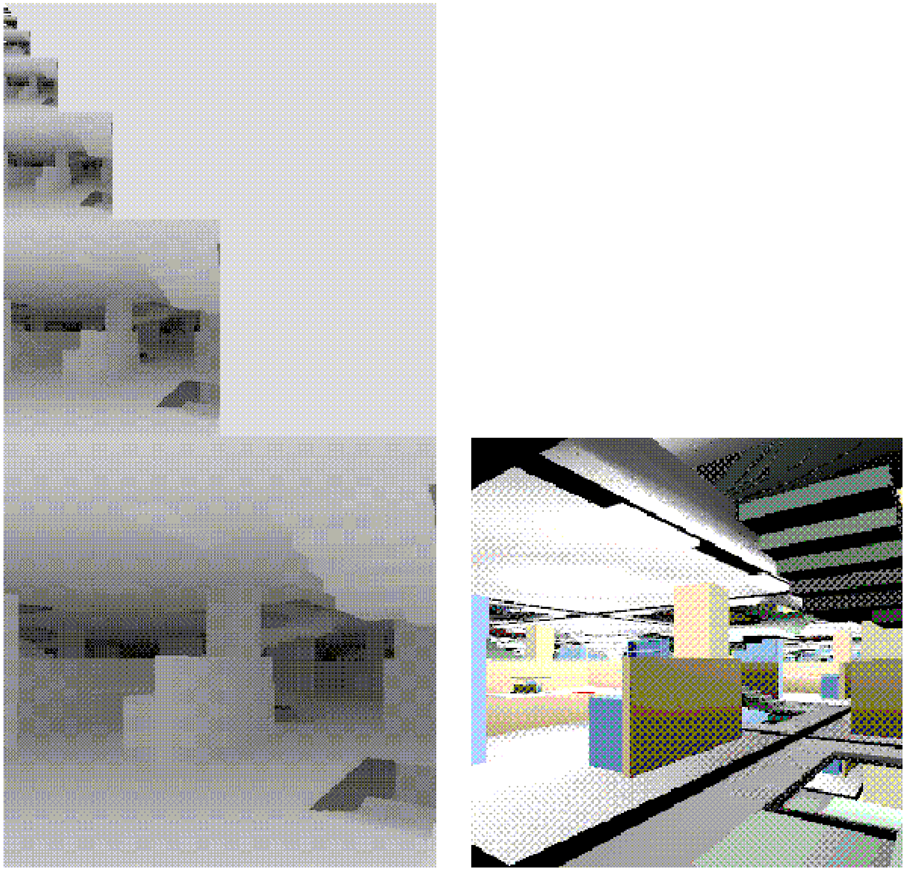


Figure 5.4: A scene and its corresponding z-pyramid. The finest level of the pyramid is the ordinary z-buffer. At all other levels, each z sample is the farthest z from the observer in the corresponding 2×2 window of the next finer level. Every entry in the pyramid therefore represents the farthest z for a square region of the screen.

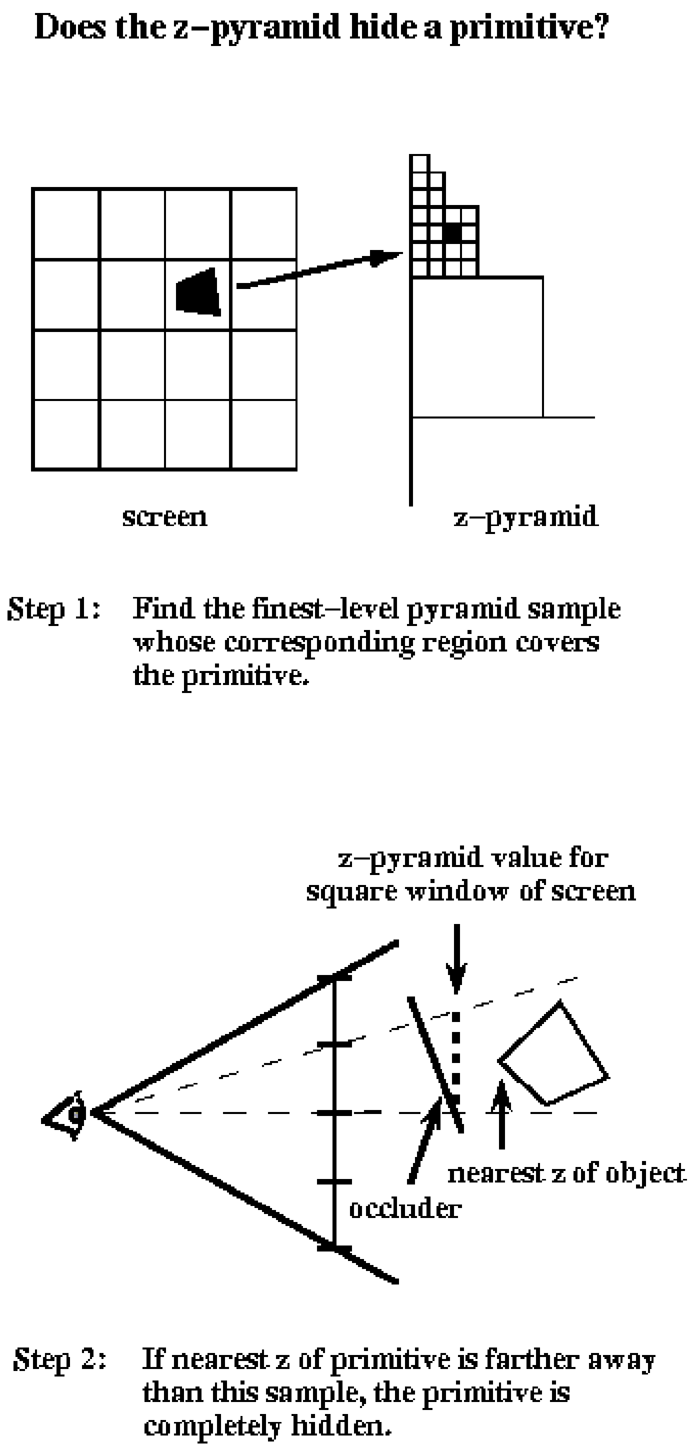

Does the z−pyramid hide a primitive?
screen
z−pyramid
Step 1: Find the finest−level pyramid sample whose corresponding region covers the primitive.
z−pyramid value for square window of screen
nearest z of object
occluder
Step 2: If nearest z of primitive is farther away than this sample, the primitive is completely hidden.

Figure 5.5:

In practice, it is quite expensive to perform definitive visibility tests on primitives in this manner because it is necessary to determine which quadrants intersect the primitive and to find the farthest z value of the primitive within each intersected quadrant. An alternative is to perform a much faster but non-definitive test which is often able to detect hidden primitives. According to this method, we construct a primitive's screen-space bounding rectangle and set its depth to the primitive's closest z value. Then the recursive subdivision procedure described above can quickly establish whether there is at least one visible pixel on the bounding rectangle. If not, we know the corresponding primitive is hidden. When this test fails to prove that a primitive is hidden, our implementation reverts to ordinary scan conversion to establish visibility of cube faces and to tile primitives into the z-buffer. Our current implementation uses this method.

5.3.3 The Temporal Coherence Scheme

We turn now to the problem of exploiting frame-to-frame coherence when rendering frames of an animation sequence. When we render an image of a densely-occluded scene with the hierarchical visibility algorithm, typically only a small fraction of the cubes in the octree are visible. When we render the next frame, most of the cubes that were visible in the last frame will probably still be visible. Some of the cubes visible in the last frame will become hidden and some cubes hidden in the last frame will become visible, but the frame-to-frame coherence that is typical of most animation ensures that there will be relatively few changes in cube visibility for most frames, except for scene changes and camera cuts. We exploit this fact in a simple way with the hierarchical visibility algorithm.

We maintain a list of the visible cubes from the previous frame, the *temporal coherence list*, and simply render all of the geometry on the list, marking the listed cubes as rendered, before commencing the usual algorithm. We then take the resulting z-buffer and use it to build the initial z-pyramid. If there is sufficient frame-to-frame coherence, most of the visible geometry will already be rendered, so the z-pyramid test will be much more effective than when we start from scratch. The z-pyramid test will be able to prove with less recursion that octree cubes and model primitives are hidden. As we will see in section 5.5.2, this can accelerate the rendering process substantially. After rendering a frame, we update the temporal coherence list by checking each of the cubes on the list for visibility using the z-pyramid test. This prevents the temporal coherence list from growing with each frame that is rendered.

One way of thinking about the temporal coherence strategy is that we begin by guessing the final solution. If our guess is very close to the actual solution, the hierarchical visibility algorithm can use the z-pyramid to verify the portions of the guess that are correct much faster than it can construct them from scratch. Only the portions of the image that it cannot verify as being correct require further processing.

5.4 Building and Maintaining The Octree

In this section we show that the cost of building an octree from n unorganized primitives using either variation of the algorithm presented in section 5.3.1 is normally $O(n \log(n))$.

By comparison, generating a fixed-resolution image of a scene with n primitives requires only $O(n)$ time with naive z-buffering, since each primitive can be tiled in constant time. Thus, the cost of building and maintaining the scene octree is an important issue, particularly for dynamic scenes.

5.4.1 Building an Octree from Unorganized Primitives

First we consider the cost of building an octree from an unorganized list of n primitives using either variation of the algorithm presented in section 5.3.1. To estimate the asymptotic cost of these algorithms it is necessary to make some assumptions. First, we assume that the average depth of a leaf node in the octree is $O(\log(n))$ and that, on average, the depth of insertion of a primitive is also $O(\log(n))$. We also assume that each primitive is inserted into no more than some constant number of nodes at each level. Given these assumptions, the cost of building the octree is $O(n\log(n))$, since each of the n primitives must be inserted into, on average, $O(\log(n))$ levels. Our assumption that the average depth of a leaf node is $O(\log(n))$ is valid, unless the octree is very poorly balanced, which is unlikely to occur unless the underlying geometry is distributed in a very unusual way. Average depth is lowest for a perfectly-balanced octree. In this case, assuming that all n primitives are associated with leaf nodes and each leaf node has m primitives, the tree's depth is $\log_8(n) - \log_8(m) + 1$. Although average depth is higher for octrees that are not so well balanced, as Aho et al. show for binary trees [2], even octrees with "random" branching patterns have an average depth of $O(\log(n))$. In the worst case, the octree's branching structure is "linear," average depth is $O(n)$, and the cost of building the octree is $O(n^2)$.

When the scene model is static, the cost of building the octree is usually not an issue, because it may be considered a precomputation expense which can be amortized over all of the frames. For most interactive applications, the primary concern is maintaining rapid frame rate, and it may be acceptable to pay a high precomputation cost. The same considerations apply if the scene is predominantly static, with only a small number of moving objects. In this case, we can precompute an octree for the static components, render this part of the scene with the usual recursive subdivision procedure, and then complete the frame by tiling the moving objects into the z-buffer. The ability to render objects in any order is one of the advantages of z-buffering.

The cost of building and maintaining an octree is more problematic when animating *dynamic scenes* having numerous moving primitives. Although the static component of the scene, if any, can be handled as described above with a *static octree*, this leaves numerous *dynamic primitives* that must be organized each frame. One strategy is to build a *dynamic octree* from the dynamic primitives at the beginning of each frame and, assuming that it is registered with the static octree, traverse the static and dynamic primitives simultaneously during the recursive rendering procedure. However, building the dynamic octree from d unorganized primitives normally requires $O(d\log(d))$ time and we would like to reduce this cost if possible. Fortunately, octree construction can often be accelerated by exploiting frame-to-frame coherence or by using lazy evaluation.

To exploit coherence in building the dynamic octree, we note that due to frame-to-frame coherence in the motion of primitives, it is usually faster to update the dynamic octree for the preceding frame than to build a new octree from scratch. Typically, a dynamic primitive moves only a short distance from one frame to the next, and in this case the primitive usually remains in the same octree node or moves to a nearby octree node. On average, such repositioning would be expected to require much less work than insertion into the $\log(d)$ levels of a new octree. While this approach can save considerable computation, the cost of updating an octree is at least linear in the number of dynamic primitives, since each primitive must be individually processed.

Now let's consider lazy evaluation as a strategy for reducing the cost of building an octree from unorganized primitives. With lazy evaluation, instead of building the complete octree and then traversing its visible nodes, we organize primitives within octree nodes as they are encountered during the recursive rendering procedure. We begin by associating all scene primitives with a root node having no children. During recursive subdivision, we determine that an octree node is visible before organizing its primitives into octants. This method avoids organizing primitives in hidden octree nodes, so in densely-occluded scenes it can avoid most of the work required to build the fully-subdivided octree. However, even with lazy evaluation the work required to build the octree is at least linear with respect to the number of primitives, since each primitive must be considered when the root node is processed, except in the trivial case that the root cube is hidden.

Thus far, the discussion of building and updating octrees has presumed that scenes are represented as an unorganized list of primitives. Consequently, the methods presented need to consider every dynamic primitive at every frame, and their performance is, at best, linear in the number of dynamic primitives. Fortunately, it is usually possible to obtain better performance, but this generally requires some higher-level structure in the scene model, some form of *coherence* that can be exploited. For example, some primitives could be organized in bounding volumes, the model could include instances of replicated geometry, there could be limits on the range of motion of primitives or instances, and so forth. Let's consider each of these circumstances in turn, bearing in mind that the proposed methods may require modifying the basic rendering algorithm.

If the model includes bounding volumes, it's not necessary to insert primitives individually into the octree. Rather, the cluster of primitives bounded by a volume can be inserted all at once into the octree node that encloses the volume. If the model includes instances of a repeating module, the module can be organized into bounding volumes, and transformed instances of the module can then be organized within the octree by transforming and inserting their component bounding volumes. To exploit limited range of motion, if we know that an object moves a maximum of some fixed distance each frame, we can construct a bounding volume that is guaranteed to enclose the object over a sequence of k frames. It follows that the octree entry for this object only needs to be updated every k frames. In the case that the range of motion of a cluster of objects is limited to a particular bounding volume over all frames, the objects can be inserted into the static octree and never updated. This would be appropriate, for example, for an anchored, articulated object such as a robot arm.

In principle, it should be possible to perform space-time culling of hidden dynamic geometry by combining some of the methods outlined above. The central idea is to cull geometry within regions of space that are hidden over intervals of time. For example, if the motion of a cluster of primitives is predetermined, then for any sequence of frames we can construct a bounding volume for the primitives. Then, when processing any frame in the sequence, we can safely cull all primitives in the cluster if that volume is hidden (or if an octree node enclosing the volume is hidden). This approach can be applied hierarchically in both space and time. If the bounding volume is visible in a given frame, we can subdivide in either space or time as appropriate to construct a smaller bounding volume and then proceed as before. While the details remain to be worked out, this appears to be a promising approach to culling dynamic geometry.

In conclusion, a variety of strategies for exploiting coherence can accelerate building and maintaining octrees for dynamic scenes, and avoid the expense of considering every dynamic primitive at every frame of animation.

5.5 Implementation and Results

Our implementation of the hierarchical visibility algorithm is based on general-purpose, portable C code and software scan conversion. This implementation uses the object-space octree, the image-space z-pyramid, and the temporal coherence scheme. Even for relatively simple models the pure software algorithm is often faster than traditional software z-buffering, and for complex models the acceleration can be very large.

To test the algorithm, we constructed an office module consisting of 15,000 polygons and then replicated the module in a three-dimensional grid. Each module includes a stairway with a large open stairwell making it possible to see parts of the neighboring floors. The office walls do not extend to the ceiling, so from a high enough point in any of the cubicles, it is possible to see parts of most of the other cubicles on the same floor.

For simple models with low depth complexity, the hierarchical visibility method can be expected to take somewhat longer than traditional scan conversion due to the overhead of performing visibility tests on octree cubes and the cost of maintaining a z-pyramid. To measure the algorithm's overhead on simple models, we rendered a single office module consisting of 15,000 polygons at a viewpoint from which a high proportion of the model was visible. On a 50Mhz Crimson, rendering time for a 512×512 image was 1.52 seconds with the hierarchical visibility method and 1.30 seconds with traditional scan conversion, indicating a performance penalty of 17%. When we rendered three instances of the model (45,000 polygons), the running time was 3.05 seconds for both methods, indicating that this level of complexity was the break-even point for this particular model. Hierarchical visibility rendered nine instances of the same model (135,000 polygons) in 5.17 seconds, while traditional scan conversion took 7.16 seconds.

The chief value of the hierarchical visibility algorithm is, of course, for scenes of much higher depth complexity. To illustrate the point, we constructed a $33 \times 33 \times 33$ replication of the office module which consists of 538 million polygons. The model is shown rendered in the upper left panel of 5.6. In the lower left panel of figure 5.6, we show a top view of the viewing frustum and the octree subdivision. The cubes shown in wireframe are the octree nodes visited during rendering, that is, the visible octree nodes and their hidden children. Note that the algorithm is able to prove that large octree nodes in the distance are hidden. The z-pyramid for the scene is shown in figure 5.4. Even at fairly coarse resolutions, the z-pyramid contains a recognizeable representation of the major occluders in the scene.

59.7 million polygons lie in the viewing frustum from this viewpoint, about one tenth of the entire model. Using the hierarchical visibility method, the z-pyramid test was invoked on 1746 octree cubes and culled about 27% of the polygons in the viewing frustum. The bounding boxes of 687 octree cubes were scan converted which culled nearly 73% of the model polygons in the viewing frustum, leaving only 83,000 polygons of which 41,000 were front facing (.000076 of the total model) to be scan converted in software. On a 50MHz SGI Crimson Elan, rendering took 6.45 seconds. Rendering this model using traditional z-buffering on the Crimson Elan hardware took approximately one hour and fifteen minutes, nearly three orders of magnitude longer. Rendering it using traditional z-buffer software scan conversion on the Crimson would probably take days.

The center left panel of figure 5.6 shows the depth complexity processed by the algorithm for the image at upper left. The depth complexity displayed in this image is the number of times each pixel was accessed in a cube visibility test or in z-buffer polygon scan conversion. Note the bright regions corresponding to portions of the image where it is possible to see far into the model; these are regions where the algorithm has to do the most work. The two brightest regions in the depth complexity image correspond to the two long strings of finely divided cubes in the octree subdivision image shown at lower left. In this image, the average depth complexity due to cube scans is 7.23, and due to polygon scan conversion is 2.48 for a total of 9.71. The maximum depth complexity is 124. Dividing the number of times the z-pyramid is accessed by the number of pixels on the screen lets us assign a value of .43 for the “depth complexity” of the z-pyramid tests. Thus, the total average depth complexity of z-pyramid tests, cube scans, and polygon scans is 10.14. Note that this is not the depth complexity of the model itself, but only the depth complexity of the hierarchical visibility computation. Computing the true depth complexity of the scene would require scan converting the entire model of 538 million polygons in software, which we have not done.

The office environment of figure 5.6 was chosen in part because of the difficulties it presents the potentially visible set methods [28, 27]. Recall that these methods partition model space into disjoint regions and establish the potentially visible set of primitives that are visible from each region. Then, only primitives in the potentially visible set for the region that contains the current viewpoint need to be rendered for a given frame. While potentially visible set methods often work well for architectural models, they would not work effectively with our model because from every office cubicle there are points from which almost every other cubicle on the same floor is visible. As a result, if the cubicles were used as cells, the potentially visible set for each cell would have to include nearly all the cells on its floor and many on other floors. Since each floor contains about four million polygons, potentially visible set methods would probably have to render many more polygons than the hierarchical method.

Admittedly, the replication of a single cell in the model means that it may not be a representative example, but it will be some time before it is practical to use models of this complexity without a great deal of instancing. The hierarchical visibility program we used for this example makes use of the replication in only two ways. First, the algorithm does not need to store half a billion polygons in main memory. Second, the algorithm only needs to consider a single cell in constructing the octree. These same simplifications would apply to any complex model that uses a great deal of instancing.

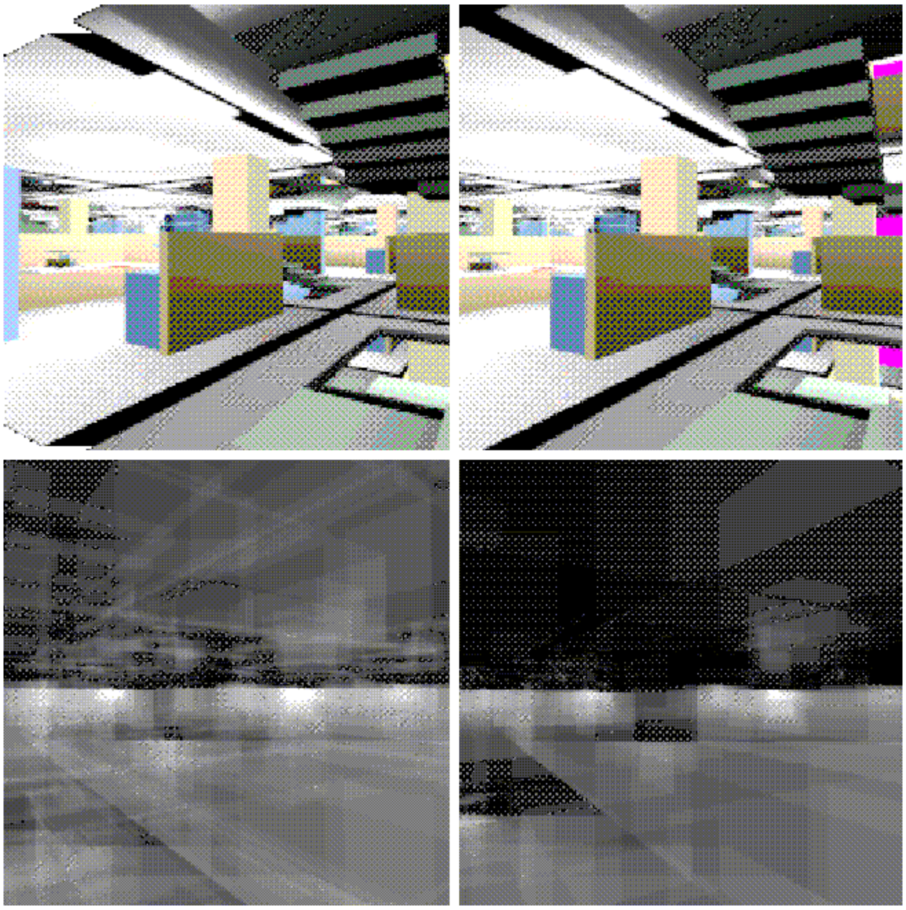


Figure 5.6: A 538 million polygon office environment rendered with hierarchical visibility. Upper left: Rendered image. Center left: Depth complexity of the hierarchical visibility computation. Lower right: Z-pyramid used to cull hidden geometry. Upper right: Image rendered with temporal coherence. Polygons not rendered in the previous frame are shown in magenta. Center right: Depth complexity of the hierarchical visibility computation for the frame rendered using temporal coherence.

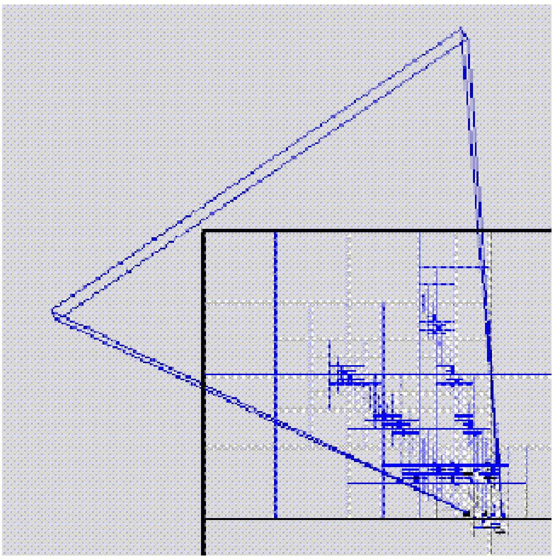


Figure 5.7: Top view of viewing frustum and octree cubes visited while rendering the office model of figure 5.6.

Figure 5.8 shows the hierarchical visibility method applied to an outdoor scene consisting of a terrain mesh with vegetation replicated on a two dimensional grid. The model shown in the lower left image consists of 53 million polygons, but only about 25,000 polygons are visible from this point of view. Most of the model is hidden by the hill or is outside the viewing frustum. The corresponding depth complexity image for hierarchical visibility computations is shown at the upper left. The algorithm works hardest near the horizon where cube visibility is most difficult to establish. This frame took 7 seconds to render with software scan conversion on an SGI Crimson. In the lower right, we show a model consisting of 5 million polygons. Even though the model is simpler than the model in the lower left, the image is more complicated and took longer to render because a much larger fraction of the model is visible from this point of view. This image took 40 seconds to render with software scan conversion on an SGI Crimson. The average depth complexity for the scene is 7.27, but it reaches a peak of 85 in the bright areas of the depth complexity image in the upper right. These outdoor scenes have very different characteristics from the building interiors shown in figure 5.6 and are poorly suited to potentially visible set methods because (a) the cell-to-cell visibility is not nearly as limited as in an architectural interior, and (b) the polygons are not axis-aligned. Nonetheless, the hierarchical visibility algorithm continues to work effectively.

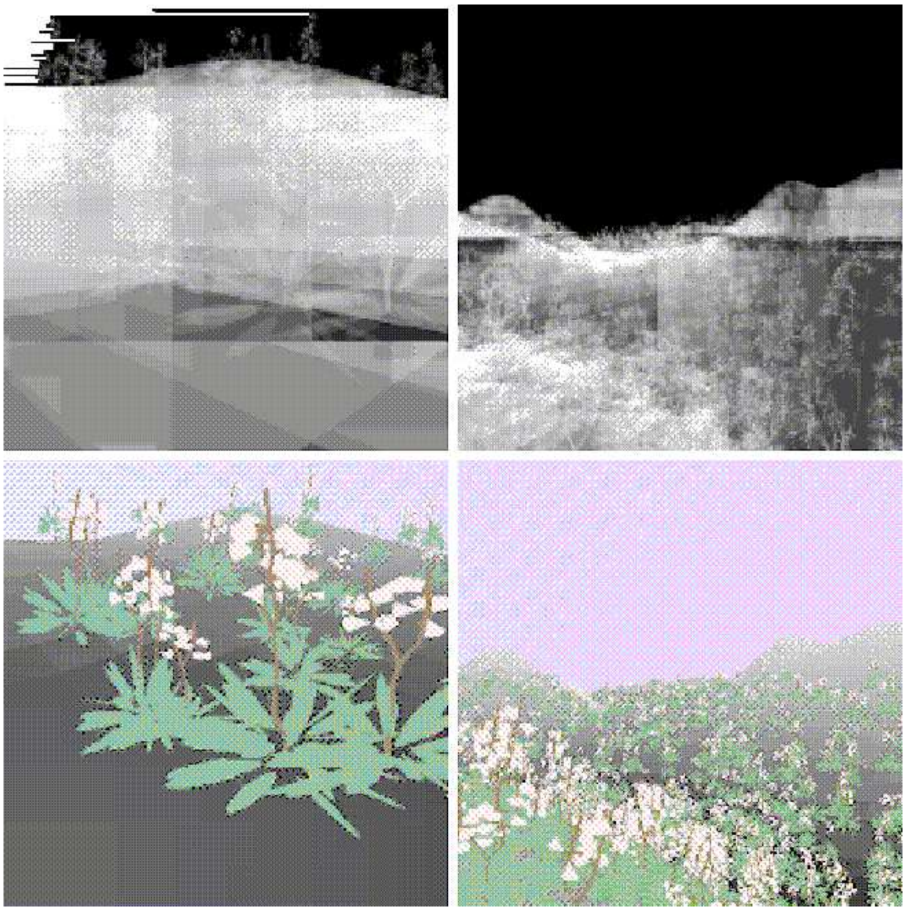


Figure 5.8: Lower left: Image of a 53 million polygon model (mostly hidden) rendered using hierarchical visibility. Upper left: Corresponding depth complexity for the hierarchical visibility computation. Lower right: Image of a 5 million polygon model. Upper right: Corresponding depth complexity for the hierarchical visibility computation.

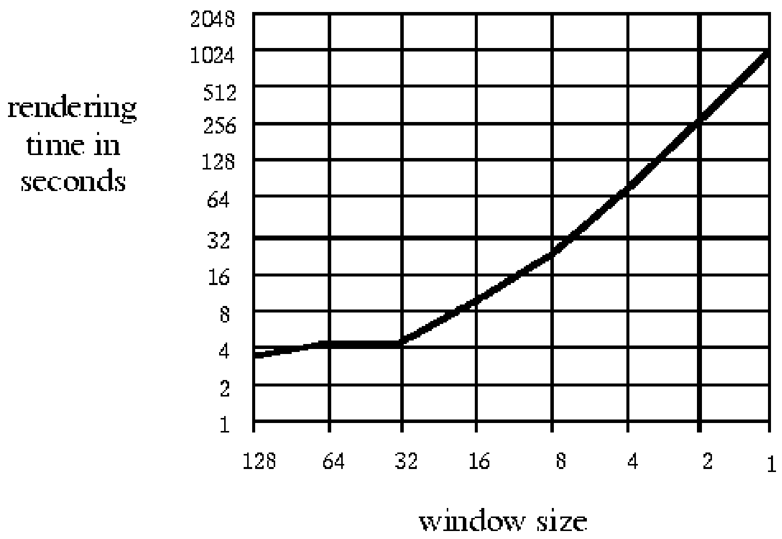


Figure 5.9: Total time in seconds to render all windows of a frame versus window size expressed as the number of pixels on the side of each window.

5.5.1 Parallel Performance

We have made our hierarchical visibility implementation capable of dividing the image into a grid of smaller windows, rendering them individually and compositing them into a final image. The performance of the algorithm as the window size is varied tells us about the parallel performance of the algorithm and the extent to which it makes use of image-space coherence. If, like most ray tracers, the algorithm made no use of image-space coherence, we could render each pixel separately at no extra cost. Then it would be fully parallelizable. At the other extreme, if the algorithm made the best possible use of image-space coherence, it would render a sizeable region of pixels with only a small amount more computation than required to render a single pixel. Then it would be difficult to parallelize. Note that if we shrink the window size down to a single pixel, the hierarchical visibility algorithm resembles a ray caster using an octree subdivision.

Figure 5.9 graphs the rendering time for a frame from a walk-through of the model shown in figure 5.6 as a function of the window size. For window sizes from 32×32 on up, the curve is relatively flat, indicating that the algorithm should parallelize fairly well. For window sizes below 32×32, however, the slope of the curve indicates that the time to render a window is almost independent of the window size. The algorithm can, for example, render a 32×32 region for only slightly more than four times the computational expense of ray-casting a single pixel with this algorithm.

5.5.2 Use of Graphics Hardware

In addition to the pure software implementation, we have attempted to modify the algorithm to make the best possible use of available commercial hardware graphics accelerators. This raises some difficult challenges because the hierarchical visibility algorithm makes somewhat different demands of scan-conversion hardware than traditional z-buffering. In particular, the use of octree object-space coherence depends on being able to determine quickly whether any pixel of a polygon would be visible if it were scan converted. Unfortunately, the commercial hardware graphics pipelines we have examined are either unable to answer this query at all, or can take milliseconds to answer it. One would certainly expect some delay in getting information back from a graphics pipeline, but hardware designed with this type of query in mind should be able to return a result in microseconds rather than milliseconds.

We have implemented the object-space octree on a Kubota Pacific Titan 3000 workstation with Denali GB graphics hardware. The Denali supports an unusual graphics library call that determines whether or not any pixels in a set of polygons are visible given the current z-buffer. We use this *z query* feature to determine the visibility of octree cubes. The cost of a z query depends on the screen size of the cube, and it can take up to several milliseconds to determine whether or not a cube is visible. Our implementation makes no use of the z-pyramid because the cost of getting the required data to and from the z-buffer would exceed any possible savings. On a walk-through of a version of the office model with 1.9 million polygons, the Titan took an average of .54 seconds per frame to render 512×512 images. Because of the cost of doing the z query, we only tested visibility of octree cubes containing at least 800 polygons. Even so, 36.5% of the running time was taken up by z queries. If z query were faster, we could use it effectively on octree cubes containing many fewer polygons and achieve substantial further acceleration. The Titan implementation has not been fully optimized for the Denali hardware and makes no use of temporal coherence, so these performance figures should be considered only suggestive of the machine's capabilities.

The other implementation we have that makes use of graphics hardware runs on SGI workstations. On these workstations there is no way to inquire whether or not a polygon is visible without actually rendering it, so we use a hybrid hardware/software strategy. We render the first frame of an animation sequence entirely with software scan conversion. Starting with the second frame, we use the hardware pipeline to render the polygons on the temporal coherence list, that is, all polygons contained in octree nodes that were visible in the preceding frame. Then we read the image and the z-buffer from the hardware, build a z-pyramid, and continue rendering with software. With this implementation, temporal coherence typically reduces the running time by a factor of between 1.5 and 2 for the models we have experimented with.

In the course of a walk-through of our office model, we rendered the frame in the upper left panel of figure 5.6 without the temporal coherence scheme and then the next frame, shown at upper right panel, with it. In the temporal coherence frame at upper right, polygons rendered with the hardware pipeline are shown with normal shading and the remaining polygons rendered with software scan conversion are shown in magenta to distinguish them. For the most part, these are polygons that came into view as a result of panning the camera. These magenta polygons are the only geometry visible in this frame that is not associated with an octree node that was visible in the preceding frame. This very high degree of frame-to-frame coherence of visible octree nodes is typical of walking through this environment.

The center right panel of figure 5.6 shows the depth complexity of the hierarchical visibility computation for this frame. The image is much darker in most regions than the depth complexity image corresponding to the upper right panel, shown at center left, because the algorithm has much less work to do given the previous frame as a starting point. This temporal coherence frame took 3.96 seconds to render on a Crimson Elan, as compared with 6.45 seconds to render the same frame without the temporal coherence scheme.

Current graphics accelerators are not designed to support the rapid feedback from the pipeline needed to realize the full potential of octree culling in the hierarchical visibility algorithm. Hardware designed to take full advantage of the algorithm, however, could make it possible to interact very effectively with extremely complex environments as long as only a manageable number of the polygons are visible from any point of view. The octree subdivision, the z-pyramid, and the temporal coherence strategy are all suitable for hardware implementation.

5.6 Conclusion

As more and more complex models become commonplace in computer graphics, it becomes increasingly important to exploit the available coherence in the visibility computation so that the amount of work expended on visibility is roughly proportional to *visible complexity* rather than *scene complexity*. Here we present a z-buffer visibility algorithm which exploits object-space coherence as effectively as ray casting, exploits image-space coherence even more effectively than traditional incremental scan conversion, and for animation sequences is also able to exploit frame-to-frame coherence. It appears to be the first practical algorithm which materially profits from object-space, image-space, and temporal coherence simultaneously. The algorithm has been tested and shown to work effectively on indoor and outdoor scenes with up to half a billion polygons.

The hierarchical visibility algorithm can make use of existing graphics accelerators without modification. Small changes in the design of graphics accelerators, however, would make a large difference in the performance of the algorithm. We hope that the appeal of this algorithm will induce hardware designers to adapt future graphics hardware to facilitate hierarchical visibility computations.

Section 6

Hierarchical Polygon Tiling with Coverage Masks

6.1 Overview

While the hierarchical z-buffer visibility algorithm has excellent performance, the images that it produces suffer from the usual aliasing problems caused by point-sampling algorithms – jaggies and other disturbing visual artifacts. The most popular approach to antialiasing is oversampling and filtering. Applying this approach with the hierarchical z-buffer algorithm is very straightforward. Tiling of primitives and visibility testing of octree cubes is simply performed on a higher-resolution grid, which entails maintaining a higher-resolution image buffer and z-pyramid. After the scene has been tiled, each pixel in the output image is obtained by filtering a neighborhood of the higher-resolution image. While this approach is reasonably efficient, the cost of tiling and the size of image memory increase in approximate proportion to the degree of oversampling, so image generation is many times slower than with the corresponding point-sampling algorithm. Consequently, we explored alternative methods of performing tiling within the framework of the basic hierarchical visibility algorithm. In doing so, we discovered a novel polygon tiling algorithm which we call *hierarchical polygon tiling* that is capable of producing antialiased images much more efficiently than hierarchical z-buffering, while requiring much less image memory.

Hierarchical tiling traverses convex polygons in front-to-back order, tiling them into an image pyramid, using specially-constructed masks that we call *triage coverage masks* to accelerate tiling. Triage masks are tiling patterns represented by a pair of bit masks that facilitate classifying a convex polygon as *inside*, *outside*, or *intersecting* cells in the image pyramid. They permit Warnock-style subdivision of image space with its ability to find visible geometry by logarithmic search to be driven very efficiently by boolean mask operations. The resulting tiling procedure performs subdivision and visibility operations very rapidly while only visiting cells in the image hierarchy that are crossed or nearly crossed by visible edges in the output image. Visible samples are never overwritten. Hierarchical z-buffering, on the other hand, must visit every image sample covered by every visible polygon,[1] and it frequently overwrites image samples. Moreover, when antialiasing is performed by oversampling, hierarchical tiling requires only a small fraction of the image memory required by hierarchical z-buffering. As a consequence of these advantages in tiling efficiency and memory usage, hierarchical tiling can generate high-quality antialiased images at approximately the same speed as hierarchical z-buffering generates the corresponding point-sampled images. The main drawback of hierarchical tiling as compared with hierarchical z-buffering is that polygons must be traversed in strict front-to-back order, which for dynamic scenes necessitates additional sorting computations. This is usually not a serious problem for predominantly static scenes, since they can be conveniently maintained in a BSP tree [11, 19], but it can severely impair performance if a very large number of polygons are moving independently.

We begin our presentation of the hierarchical polygon tiling algorithm by placing it in the general context of polygon tiling algorithms, and then describe how it can be integrated with the basic hierarchical visibility algorithm.

6.2 Introduction

The purpose of a polygon tiling algorithm is to determine which point samples on an image raster are covered by the visible portions of each of the polygons composing a scene. This has been an important topic in computer image synthesis since the advent of raster graphics some two decades ago. Currently, polygon tiling software running on inexpensive computers can render point-sampled images of simple scenes at interactive rates. The fastest tiling algorithms have been carefully tuned to exploit image coherence by using incremental methods wherever possible. However, they fail to exploit opportunities for precomputation and they waste time tiling hidden geometry. There is a need for more efficient tiling algorithms that effectively exploit precomputation and coherence to enable efficient culling of hidden geometry and efficient tiling of visible geometry.

The dominant polygon tiling algorithm in use today is incremental scan conversion. Typically, raster samples on a polygon's perimeter are traversed with an incremental line-tiling algorithm such as the Bresenham algorithm [4]. Edge samples on each intersected scanline define spans within a polygon, which are then crossed pixel-by-pixel, permitting incremental update of shading parameters and, in the case of z-buffering, depth values. Visibility of samples can be determined by a) maintaining a z-buffer and performing depth comparisons [6], b) traversing primitives back to front and writing every pixel tiled [10], or c) traversing primitives front to back and overwriting only vacant pixels [10]. With incremental scan conversion, the cost per pixel tiled is very low because incremental edge and span traversal effectively exploits image coherence. Overall, the cost of tiling a scene is roughly proportional to its depth complexity, so traditional incremental scan conversion is an efficient way to tile shallowly-occluded scenes.

[1] Recall that the hierarchical z-buffer algorithm reverts to traditional z-buffer scan conversion in tiling visible polygons.

One problem with traditional incremental scan conversion is that it must tile every sample on every primitive, whether or not it is visible, and so it wastes time tiling hidden geometry. This is not a big problem for simple scenes, but for densely-occluded scenes it severely impairs efficiency. Ideally, a tiling algorithm should cull hidden geometry efficiently so that running time is proportional to the visible complexity of the scene and independent of the complexity of hidden geometry.

The Warnock subdivision algorithm [29] approaches this goal, performing logarithmic search for visible tiles in the quadtree subdivision of a polygon. If scene primitives are processed front to back, only visible tiles and their children in the quadtree are visited. Although Warnock subdivision satisfies our desire to work only on visible (or nearly visible) regions of primitives, the traditional subdivision procedure is relatively costly and consequently, this approach is slower than incremental scan conversion except for densely-occluded scenes. Neither traditional incremental scan conversion nor Warnock subdivision is well suited to tiling scenes of moderate depth complexity.

A second shortcoming of incremental scan conversion is that it spends most of its time tiling edges and spans, traversing these features pixel by pixel, even though all possible tiling patterns for an edge crossing a block of samples can be precomputed and stored as bit masks called *coverage masks*. Then the samples that a convex polygon covers within a block can be quickly found by compositing the coverage masks of its edges. Previously, this technique has been applied to estimating coverage of polygonal fragments within a pixel to accelerate filtering [5, 24, 8, 9].

Here we present a polygon tiling algorithm that combines the best features of traditional algorithms. The key innovation that makes this integration possible is the generalization of coverage masks to permit their application to image hierarchies. The generalized masks, which we call *triage coverage masks*, classify cells in the hierarchy as *inside*, *outside*, or *intersecting* an edge. This permits them to drive Warnock-style subdivision. The result is a hierarchical tiling algorithm that finds visible geometry by logarithmic search, as with the Warnock algorithm, that exploits precomputation of tiling patterns, as with filtering with coverage masks, and that also uses incremental methods to exploit image coherence, as with incremental scan conversion. The algorithm is well suited to high-resolution tiling (e.g. 4096×4096), so it naturally supports high-quality antialiasing by oversampling. A-buffer-style antialiasing with coverage masks [5] is particularly convenient.

For densely-occluded scenes we combine hierarchical tiling with the basic hierarchical visibility algorithm to permit hierarchical culling of hidden regions of object space. This combination of algorithms enables very rapid rendering of complex polygonal scenes with high-quality antialiasing. This method has been tested and shown to work effectively on densely-occluded scenes. On a test scene having upwards of 500 million replicated polygons, the algorithm computed visibility on a 4096×4096 grid nearly as rapidly as the hierarchical z-buffer visibility algorithm of section 5 tiled a 512×512 grid.

In section 6.3, we survey the most relevant previous work on efficient polygon tiling. In section 6.4, we introduce triage coverage masks, and in section 6.5 we present the hierarchical tiling algorithm in which they are applied. In section 6.6, we discuss how the algorithm can be accelerated with object-space culling methods. In section 6.7, we describe the implementation and show results for a densely-occluded scene. Finally, we present our conclusions in section 6.8.

6.3 Previous Work

Previous approaches to accelerating polygon tiling have included exploiting image coherence with incremental methods, hierarchical culling with Warnock subdivision, and precomputation of tiling patterns with coverage masks. However, no previous algorithm has combined all of these features.

6.3.1 Warnock Subdivision

Incremental methods that exploit image coherence in traversing edges and spans are summarized in the introduction, so we begin our discussion of previous work with the Warnock algorithm [29]. The Warnock algorithm is a recursive subdivision procedure that finds the quadtree subdivision of visible edges in a scene by logarithmic search [26]. Scene primitives are inserted into a quadtree data structure beginning at the root cell. At each level of subdivision, the algorithm classifies the quadrants of the current quadtree cell as inside, outside, or intersecting a primitive, and only intersected quadrants are subdivided. Quadrants which are entirely covered by primitives are identified, permitting hidden geometry within them to be culled. The Warnock algorithm is actually a family of algorithms based on a common subdivision procedure, and the control structure varies from implementation to implementation [22]. A typical implementation processes primitives in no particular order, maintains lists of potentially visible primitives at quadtree cells, and expends considerable work performing depth comparisons in order to cull hidden geometry.

When circumstances permit convenient front-to-back traversal of primitives, as with presorted static polygonal scenes, a simpler and more efficient variation of the Warnock algorithm can be employed. In this case, we insert primitives into the quadtree one at a time in front-to-back order. As subdivision proceeds, we mark cells that a primitive completely covers as *occupied* and ignore cells that are already occupied, since any geometry that projects to them is known to be hidden. We complete subdivision of one primitive down to the finest level of the quadtree before processing the next. This version of the Warnock algorithm is simpler because it need not maintain lists of primitives or perform depth comparisons. It is more efficient because, unlike the traditional algorithm, it only subdivides cells crossed by edges that are visible in the output image. To the best of our knowledge, this is a novel variation of the Warnock algorithm, although we haven't researched the question in depth. In any case, our tiling algorithm is based on this variation, which we will refer to as the *depth-priority Warnock algorithm*.

6.3.2 Coverage Masks

We turn now to reviewing how filtering algorithms exploit precomputation with coverage masks [5, 24, 8, 9]. The underlying idea is that all possible tiling patterns for a single edge crossing a grid of raster samples within a pixel can be precomputed and later retrieved, indexed by the points where the edge intersects the pixel's border [8, 24]. These tiling patterns can be stored as bit masks, permitting samples inside a convex polygon to be determined by ANDING together the coverage masks for its edges. Moreover, if polygons are processed front to back or back to front, visible-surface determination within

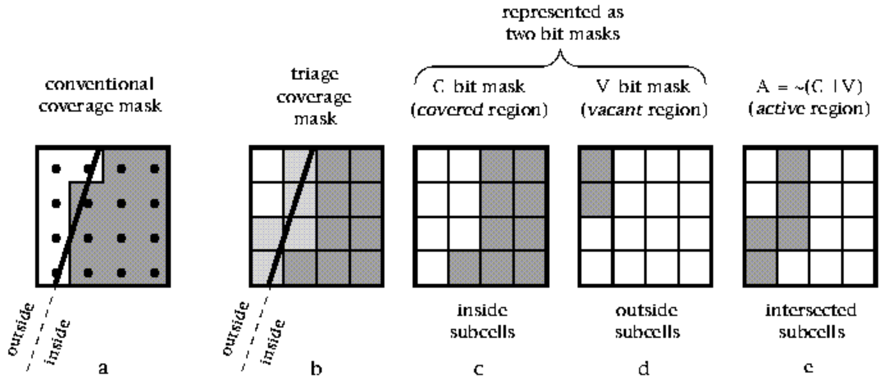


Figure 6.1: A conventional coverage mask classifies grid points as inside or outside an edge (panel a). A *triage* coverage mask classifies subcells as inside, outside, or intersecting an edge (panel b). We refer to these regions as *covered* (panel c), *vacant* (panel d), and *active* (panel e), respectively. We represent triage masks as the pair of bit masks (C,V) indicating the covered and vacant regions. In our illustrations of triage masks (e.g. panel b), the C bit mask corresponds to dark gray subcells and the V bit mask corresponds to white subcells. In practice, we use 8×8 masks rather than 4×4 masks.

a pixel can also be performed with boolean mask operations. For example, Carpenter's A-buffer algorithm [5] clips polygons to pixel borders, sorts the polygonal fragments front to back, and determines the visible samples of each fragment on a 4×8 grid by compositing coverage masks. For each visible fragment, a single shading value is computed, weighted by the bit count of its mask, and added to pixel color. This method is a very efficient way of approximating *area sampling* [26, 7] and effectively antialiases edges. Abram, Westover, and Whitted advance similar methods that permit convolution with arbitrary filter kernels, jitter, and evaluation of simple shading functions by table lookup [1].

6.4 Triage Coverage Masks

To accelerate polygon tiling, the hierarchical tiling algorithm generalizes coverage masks to operate on image hierarchies, thereby enabling Warnock-style subdivision of image space to be driven by boolean mask operations. We precede the discussion of the tiling algorithm by explaining how these *triage* masks work.

6.4.1 Triage Edge Masks

A conventional coverage mask for an edge classifies each grid point within a square as inside or outside the edge, as shown in figure 6.1a. In the context of Warnock subdivision, the analogous operation is classifying subcells of an image hierarchy as inside, outside, or intersecting an edge, as shown in figure 6.1b for an edge crossing a square containing a 4×4 grid of subcells. We call such masks *triage*[2] *coverage masks* because the three states that they distinguish correspond to trivial rejection, trivial acceptance, and "do further work." We represent each triage mask as a pair of bit masks, one indicating outside subcells, the other indicating inside subcells, as shown in figures 6.1c and 6.1d. We will refer to the bit mask for inside subcells as the "C" mask (for *covered*) and the bit mask for outside subcells as the "V" mask (for *vacant*). We call the intersected subcells the *active* region of the mask, because this is the region that requires further work and will later be subdivided. The bit mask for the active region is $A = \tilde{}(C \mid V)$, as shown in figure 6.1.[3] Hereafter, we will often refer to triage coverage masks simply as "masks" and we will refer to conventional one-bit masks as "bit masks" or "conventional coverage masks." In practice, we use 8×8 masks rather than the illustrated 4×4 masks.

6.4.2 Compositing Triage Masks

The representation of masks as (C,V) pairs was chosen to facilitate the basic tiling and visibility operations: finding the mask of a convex polygon from the masks of its edges, and compositing a polygon mask together with a mask representing the screen. Both of these operations can be performed with simple boolean mask operations.

Polygon Masks from Edge Masks

We make a mask for convex polygon P within a square region S of the screen as follows. Assume that we have already created a lookup table for edge masks that is indexed by edge-square intersection points, as we will describe later in section 6.4.3. For each edge of P that intersects S, we find its two intersection points on S's perimeter and look up the precomputed edge mask. Suppose that P has N edges which intersect S, and that their edge masks are $(E1_C, E1_V), (E2_C, E2_V), ..., (EN_C, EN_V)$. Then P's mask is simply (P_C, P_V), where $P_C = E1_C \;\&\; E2_C \;\&\; ... \;\&\; EN_C$ and $P_V = E1_V \mid E2_V \mid ... \mid EN_V$. For example, figure 6.2a shows a polygon P having two edges, $E1$ and $E2$, that intersect square S. We ignore the edges of P that don't intersect S's perimeter. Combining the edge masks for $E1$ (figure 6.2b) and $E2$ (figure 6.2c) as just described produces P's mask (figure 6.2d).

Compositing Rules

Now for compositing masks within a square. Let (S_C, S_V) be the mask for a square region S of the screen that is initially vacant with S_C=*all_zeros*, S_V=*all_ones*. We process

[2] Pronounced *TREE-ahj*, meaning to divide into three parts.

[3] We use standard notation for boolean mask operations: & for bitwise AND, | for bitwise OR, and ˜ for bitwise complement.

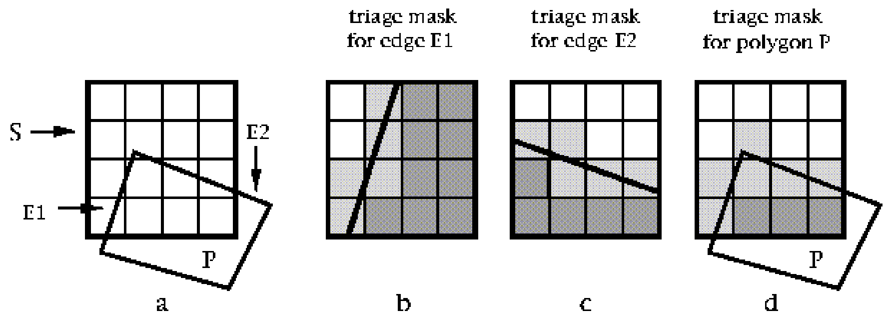


Figure 6.2: An example illustrating how the triage mask of a convex polygon is constructed from the triage masks of its edges that intersect the mask. The polygon's C bit mask is obtained by ANDing together the edges' C bit masks, and the polygon's V bit mask is obtained by ORing together the edges' V bit masks. In this example, the mask of polygon P (panel d) is made from the masks of edges $E1$ (panel b) and $E2$ (panel c) as follows: $P_C = E1_C \,\&\, E2_C$, $P_V = E1_V \mid E2_V$. As in the other illustrations of triage masks, C bit masks correspond to dark gray subcells and V bit masks correspond to white subcells.

case	screen subcell s	polygon subcell p	resulting screen subcell s'	action
1	covered	covered	covered	do nothing
2	covered	active	covered	do nothing
3	covered	vacant	covered	do nothing
4	active	covered	active*	**SUBDIVIDE**
5	active	active	active†	**SUBDIVIDE**
6	active	vacant	active	do nothing
7	vacant	covered	covered	**DISPLAY**
8	vacant	active	vacant‡	**SUBDIVIDE**
9	vacant	vacant	vacant	do nothing

* propagation will change status to covered
† propagation may change status to covered
‡ propagation will change status to active

Table 6.1: Rules for front-to-back compositing of triage masks, subcell by subcell. Since polygons are traversed front to back, in compositing a polygon's mask with the existing screen mask, screen subcell s is composited "over" polygon subcell p. The hierarchical tiling algorithm actually performs compositing with boolean mask operations rather than subcell by subcell.

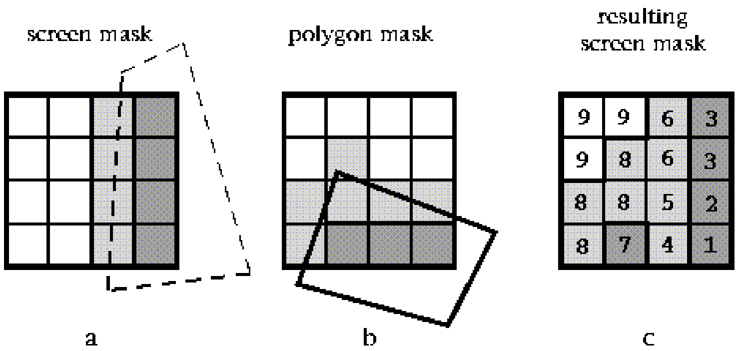


Figure 6.3: Example illustrating compositing of triage masks according to the compositing rules of table 6.1. The screen mask of panel a was produced by the dashed polygon. This screen mask is composited "over" the polygon mask of panel b, producing the new screen mask of panel c. In panel c, each subcell is numbered with the corresponding compositing case of table 6.1.

polygons in front-to-back order, compositing their masks with S's mask. For each polygon, our task is to identify the subcells that it covers and the subcells in which it is active, and then update (S_C, S_V) as necessary. In analysing this procedure, we first consider the rules for compositing on a subcell-by-subcell basis, even though actual compositing is done with boolean mask operations.

Let s be any subcell of S and let p be the corresponding subcell of the mask for polygon P. Since each mask represents three states, there are nine possible compositing cases which are summarized in table 6.1. With cases 1, 2, and 3, s is already *covered*, so P is not visible and can be ignored. With cases 6 and 9, p is *vacant*, so P is not visible and can be ignored. Thus, only cases 4, 5, 7, and 8 require any action. With case 7, s is *vacant* and p is *covered*, so we know that P is visible at all raster samples within the subcell. Thus, we shade and display them (or tag them as belonging to P) and then change the status of s to *covered*. Cases 4, 5, and 8 require subdivision to determine where P is visible within the subcell and what the new status of s should be, as we will discuss in the context of hierarchical tiling in section 6.5.

Figure 6.3 presents an example which includes all nine compositing cases. Suppose we have already composited the polygon shown in dashed lines into a vacant square on the screen, creating the screen mask of panel a. Now we composite with this mask the mask for polygon P, shown in panel b. Applying the compositing rules of table 6.1, we obtain the new screen mask, shown in panel c, where each subcell is labeled with the number of the corresponding compositing case.

Next we consider how the compositing rules can be efficiently executed with boolean mask operations. Let (S_C, S_V) be the mask for a region of the screen and (P_C, P_V) be the mask for a polygon P that we are compositing. Recall that only compositing cases 4, 5, 7, and 8 require any action. First we handle case 7 by creating a bit mask for the subcells where S is *vacant* and P is *covered*: W = S_V & P_C. We call this mask "W" because all raster samples in this region of the screen are covered by P and may now be *written* (or tagged). We update the status of the *covered* subcells in S as follows: S_C = $S_C \mid W$, S_V = S_V &˜W. Cases 4, 5, and 8 require subdivision and can only be fully described in the context of the hierarchical tiling algorithm. However, the first step to is construct the bit mask for *active* subcells which require subdivision, which is done as follows: $A = \tilde{}(W \mid P_V \mid S_C)$. As we will see in section 6.5, the hierarchical tiling algorithm recursively subdivides *active* subcells and updates the corresponding subcells of S by propagation from finer levels.

6.4.3 Building Lookup Tables

Before presenting the recursive tiling procedure in which these procedures are applied, we describe how lookup tables for triage masks are constructed. Although the method is generally analogous to building tables for conventional coverage masks, there are some important differences. In both cases, we divide the perimeter of a canonical square into some number of equal intervals and create an entry in a two-dimensional table for each pair of intervals not sharing a common edge. Once this table has been constructed, to obtain the mask for an arbitrary edge we determine which two intervals it crosses and look up the corresponding table entry. To conserve storage, we can use the same table entry for edges with opposite directions, since complementing the (C,V) bit masks in a triage mask corresponds to reversing an edge.

With conventional coverage masks, the accuracy of the table depends on the length of the intervals, the shorter the intervals, the more accurate the table. However, there can be classification errors even if intervals are very short. The problem is illustrated in figure 6.4a, where we see that within the quadrilateral defined by the endpoints of a pair of intervals, classification of grid points is ambiguous. Thus, for some edges, some grid points will be mis-classified by the precomputed edge mask in the lookup table. In practice, small errors in classification are not a problem when applying coverage masks to filtering subpixel samples. In fact, caching performance is generally considered more important than accuracy, so typical implementations use low-resolution tables that routinely mis-classify a few samples. For example, Sabella and Wozny use 28×28 tables [24].

The hierarchical tiling algorithm, however, depends on accurate classification of *vacant* and *covered* regions in triage masks. Accordingly, in constructing a mask we classify any subcell which intersects the quadrilateral defined by interval endpoints as *active* (see figure 6.4b). Thus, cells classified as *covered* are completely covered and cells classified as *vacant* are completely vacant. This conservative procedure guarantees accurate results, regardless of interval length. The only advantage of dividing the square's edges into shorter intervals is that, on average, a smaller fraction of subcells will be classified as *active*, which reduces the amount of subdivision the tiling algorithm needs to perform. In addition to creating a table of triage masks, we also create a table of conventional one-bit coverage masks to determine coverage of raster samples at finest level of the mask pyramid.

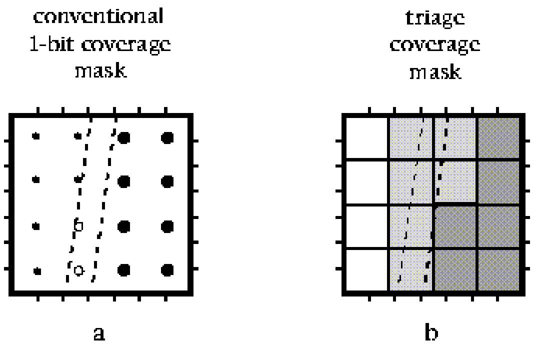


Figure 6.4: The endpoints of the pair of intervals used to index a coverage mask define a quadrilateral. With conventional one-bit coverage masks, the status of gridpoints inside the quadrilateral is ambiguous, for example, the gridpoints marked with small circles in panel a. With triage masks, however, any subcells intersected by the quadrilateral are classified *active*, guaranteeing that cells classified *covered* are completely covered and cells classified *vacant* are completely vacant (panel b). In practice, edges of triage masks are more finely divided than shown here.

6.5 The Hierarchical Tiling Algorithm

As discussed in section 6.3, the "depth-priority" variation of the Warnock algorithm that processes primitives in front-to-back order is simpler and more efficient than the traditional algorithm, provided that front-to-back traversal of primitives is convenient. While depth-priority Warnock is a big improvement over the traditional algorithm, it is still relatively slow due to the computational expense of classifying primitives as inside, outside, and intersecting quadrants. We use the basic recursive subdivision procedure of depth-priority Warnock as the core of our hierarchical tiling algorithm and circumvent the classification bottleneck by precomputing classification patterns and storing them as triage coverage masks. The mechanism by which the masks control subdivision is a straightforward application of the compositing rules for masks. We will discuss this procedure after describing the underlying data structures: the mask pyramid, the image array, and the model tree.

6.5.1 Data Structures

We maintain visibility information about previously-tiled polygons in an image-space pyramid. The finest level of the pyramid indicates visibility at each point sample in the image raster. Since we traverse polygons front to back, it is only necessary to maintain one bit of visibility information for each raster sample, indicating whether it has been covered. At coarser levels of the pyramid, elements correspond to square regions of the screen, indicating whether they are *covered*, *vacant*, or *active*. Distinguishing these three states requires two bits. Total memory requirements for the pyramid are very modest, only slightly more than one bit per raster sample. Note that storing depth values in a z-buffer requires much more memory. We use a pyramid rather than an adaptively-subdivided hierarchy to exploit the efficiency of static memory allocation.

To permit Warnock subdivision to be driven by boolean mask operations, we build the image-space pyramid from coverage masks. As schematically illustrated in figure 6.5, a single triage mask represents the whole screen, triage masks at the next level of the pyramid correspond to subcells in the root mask, and so forth. Thus, this *mask pyramid* is a hierarchical representation of the screen with the C and V bits for each subcell in the triage masks indicating whether a square region of the screen is *covered*, *vacant*, or *active*. At the finest level of the pyramid only, we use conventional one-bit coverage masks to indicate whether or not point samples in the image raster have been covered. If we are performing A-buffer-style filtering, each of these one-bit masks corresponds to the 8×8 grid of raster samples within a pixel. The appropriate pyramid for a 512×512 image with 8×8 oversampling at each pixel has four levels, three arrays of triage masks with dimensions 1×1, 8×8, and 64×64, and one 512×512 array of one-bit masks. Alternatively, if we are point sampling rather than filtering, each one-bit mask corresponds to an 8×8 block of pixels. In this case, the pyramid for a 512×512 image would have two arrays of triage masks with dimensions 1×1 and 8×8, and one 64×64 array of one-bit masks.

The other image-space data structure that our algorithm requires is an image array with an element for each color component at each pixel. If we perform A-buffer-style filtering [5], shading contributions from 64 subpixel samples accumulate in each array element. Thus, elements in this *accumulation buffer* require considerable depth. We use 16 bits per pixel per color channel. If no filtering is performed, pixel values do not accumulate, so a conventional image array is employed.

Now for representing the model. Our algorithm requires front-to-back traversal of polygons in the scene, so we represent the scene as a binary space partitioning tree (*BSP tree*) [11], which permits very efficient traversal in depth order. If some polygons are in motion, it is necessary to update the BSP tree every frame. This does not require much computation if the scene is predominantly static [19], but if a large number of polygons are moving, maintaining the tree can be expensive. In extreme cases, the time spent maintaining the tree can exceed the time saved by our hierarchical tiling procedure, in which case our algorithm offers no advantage over traditional methods such as z-buffering. This potential difficulty with dynamic scenes is the biggest limitation of our algorithm.

6.5.2 The Basic Hierarchical Tiling Algorithm

Now that the primitive mask operations and the underlying data structures have been presented, the procedure for creating a frame is easy to describe. Details of the algorithm vary depending on filtering procedure, if any. First we present the algorithm in the context of A-buffer-style area sampling, saving discussion of other variations until later.

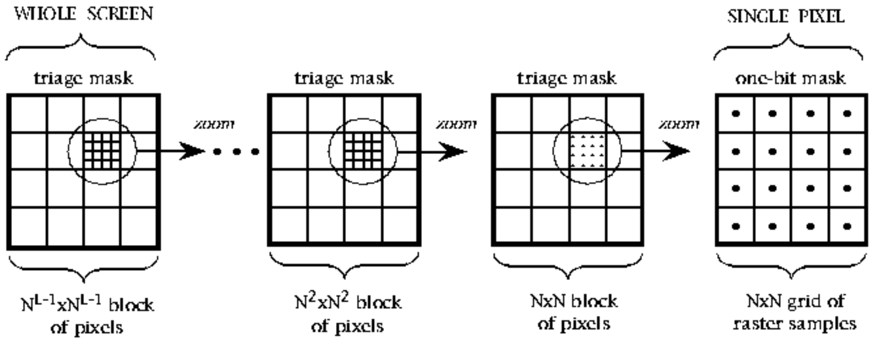


Figure 6.5: Schematic diagram of a pyramid of $N \times N$ masks with L levels for an image with $N \times N$ oversampling at each pixel. This *mask pyramid* is built from triage masks, except at the finest level where a conventional one-bit coverage mask is associated with each pixel. In this hierarchical representation of the screen, the C and V bits for each subcell in triage masks indicate whether a square region of the screen is *covered*, *vacant*, or *active*. At the coarsest level, a single triage mask represents the whole screen (left), and at the finest level, a single one-bit mask represents the raster samples within a pixel (right). The corresponding diagram for a point-sampled image is the same, except that the masks represent an $N \times N$ block of pixels, an $N^2 \times N^2$ block of pixels, and so forth. In practice, we use 8×8 masks. A four-level pyramid of 8×8 masks corresponds to a 512×512 image with 8×8 oversampling at each pixel.

Generating a Frame

In a precomputation step, we build a BSP tree for the model and build lookup tables for both triage and conventional coverage masks. We begin a frame by clearing entries in the accumulation buffer to zero and clearing masks in the pyramid to *vacant*. We traverse polygons in the BSP tree front to back, tiling them one at a time into the mask pyramid, adding color values for each of their visible samples into the accumulation buffer. The tiling procedure also identifies *covered* and *active* regions of the screen and maintains this information in the mask pyramid. After all polygons have been processed, we obtain the output image by scanning through the accumulation buffer and dividing each color value by 64, the number of raster samples per pixel.

Tiling a Polygon with Area Sampling

Within this overall algorithm, the procedure for tiling a polygon with A-buffer-style filtering [5] is as follows. We clip each polygon P to the front clipping plane in object space, if necessary, and project its vertices into the image plane. There is no need to preserve depth information. Now we begin the recursive subdivision procedure that tiles P into the mask pyramid, beginning with the root mask which corresponds to the whole screen. The procedure summarized here is also outlined in pseudocode in LISTING 6.6. We create P's mask by finding the intersection points of its edges with the border of the pyramid mask (initially, the root mask), looking up the corresponding edge masks, and combining them as described in section 6.4.2. We then composite P's mask with the pyramid mask using the compositing rules of section 6.4.2. At subcells where the "W" bit mask (see section 6.4.1) indicates that P is entirely visible, we evaluate the shading function for each pixel in the corresponding region of the screen and update the accumulation buffer. Since P covers all 8×8 samples at these pixels, we add 64 times the evaluated color to the accumulation buffer. No visible edges cross these pixels, so edge filtering is not called for. We also change the status of the corresponding subcells in the mask pyramid to *covered*. At subcells where the "A" bit mask (see section 6.4.1) indicates that subdivision is required, we call this same tiling procedure recursively with the appropriate pyramid mask and the edges of P that intersect it. In regions of the screen crossed by visible edges, recursive subdivision continues down to the pixel level where A-buffer-style filtering is performed on visible samples using conventional one-bit coverage masks. At these pixels, shading is evaluated and color values are multipled by the bit count of the coverage mask before they are added to the accumulation buffer. In addition to updating the accumulation buffer, we also need to update the mask pyramid by propagating coverage information to coarser levels, which is accomplished easily with recursively-applied mask operations (see pseudocode). When this procedure finishes, all visible raster samples covered by P have been evaluated and added to the accumulation buffer, and the appropriate cells in the mask pyramid have been updated. Since visibility is computed on an 8×8 grid of raster samples within each pixel, the resulting image has very high-quality antialiasing.

```
/*
Recursive subdivision procedure for tiling a convex polygon P.

After clipping P to the near clipping plane in object space, if necessary, and
projecting P's vertices into the image plane, we call tile_poly with the root
mask of the mask pyramid, P's edge list, and "level" set to 1.

arguments:
(Sc,Sv): pyramid mask (input and output)
edge_list: P's edges that intersect pyramid mask
level: pyramid level: 1 is root, 2 is next coarsest, etc.
*/

tile_poly((Sc,Sv), edge_list, level)
{
   set active_edge_list to nil

   /* build P's mask (Pc,Pv) */
   Pc = all_ones
   Pv = all_zeros
   for each edge on edge_list {
      find intercepts on square perimeter of mask
      if square is outside edge, then return /* polygon doesn't intersect mask */
      if edge intersects square, then {
         append edge to active_edge_list
         /* Note: at pixel level, Ec is conventional coverage mask, Ev = ~Ec */
         look up edge mask (Ec, Ev)
         Pc = Pc & Ec
         Pv = Pv | Ev
      }
   }

   /* make "write" bit mask and update pyramid mask */
   W = Sv & Pc
   Sc = Sc | W
   Sv = Sv & ~W

   if level is the pixel level, then {
      /* filter pixel with coverage mask W */
      evaluate shading and add bitcount(W)*(pixel color) to accumulation buffer
      return
   }

   for each TRUE bit in W { /* compositing case 7 */
      for each pixel in this square region of screen /* all 64 samples covered */
         evaluate shading and add 64*(pixel color) to accumulation buffer

   /* RECURSION */

   /* make "active" bit mask and subdivide active subcells */
   A = ~(W | Pv | Pc)
   for each TRUE bit in A { /* compositing cases 4, 5, 8 */
      /* call corresponding subcell C; call its pyramid mask (Cc,Cv) */
      copy all edges on active_edge_list that intersect C to C_edge_list
      tile_poly((Cc,Cv), C_edge_list, level+1)
      /* propagate coverage status to coarser levels of mask pyramid */
      if Cc==all_ones then Sc = Sc | active_bit /* set covered status */
      if Cv!=all_ones then Sv = Sv & ~active_bit /* clear vacant status */
   }
}
```

Figure 6.6: PSEUDOCODE LISTING

Other Filtering Methods

The filtering method outlined above performs A-buffer-style *area sampling*, a term used to describe convolution of visible samples with a pixel-sized box filter [7]. As with the original A-buffer algorithm [5], to improve efficiency only one shading evaluation is performed for each primitive at each pixel. This generally produces acceptable results, although results are not as accurate as when shading is performed at all subpixel samples. It is interesting to note that for the important special case of Gouraud shading, the contributions of all visible samples within a pixel can be computed all at once using precomputed coefficients stored in lookup tables. This technique was proposed by Abram, Westover, and Whitted [1], who also experimented with coverage masks for jittered samples and table-driven convolution with arbitrary convolution filters. All of these refinements are compatible with our algorithm and would be expected to improve accuracy, although we have obtained good results with A-buffer-style area sampling, which is simpler and faster.

Two other variations on the algorithm outlined in pseudocode should be mentioned. First, image resolution doesn't need to be a power of eight. For example, if the resolution of the raster grid is 4096×4096, instead of filtering 8×8 blocks of samples to produce a 512×512 image, we could instead filter 4×4 blocks of a 1024×1024 image. Second, modifying the algorithm to produce point-sampled rather than filtered images is straightforward. In this case, each mask at the finest level of the pyramid corresponds to an 8×8 block of pixels. So for each TRUE subcell in the "W" mask (see pseudocode), we evaluate the shading function at the corresponding pixel and write the result to the image buffer. Since pixel values correspond to point samples, color values do not accumulate, so we use a conventional image array rather than an accumulation buffer. Note that it is not necessary to clear the image array at the beginning of a frame. Instead, after tiling all scene polygons, we composite a screen-sized polygon of the desired background color (or texture) with the root mask, thereby tiling all remaining vacant pixels in the image.

Properties of the Hierarchical Tiling Algorithm

Hierarchical polygon tiling combines desirable features of traditional tiling algorithms: logarithmic search for visible geometry, precomputation of tiling patterns, and incremental evaluation. It is the only tiling algorithm that effectively exploits all of these methods. Since the recursive subdivision procedure only visits cells in the mask pyramid that are crossed (or nearly crossed, in some rare cases) by visible edges in the output image, it finds visible geometry by logarithmic search. Logarithmic search can be extended to finding visible subcells within masks by using bit patterns to control a switch statement. Once visible samples are found, front-to-back traversal of polygons guarantees that they are never overwritten, so shading evaluation is never wasted. Regarding precomputation, since tables of triage tiling patterns are precomputed, the only geometric operations required to classify cells as inside, outside, or intersecting an edge are finding intercepts of edges on mask borders, and this can be done incrementally, except at the root mask. In fact, the only other geometric operation that must be performed on polygons is clipping to the near clipping plane, if necessary, and projecting vertices into the image plane. Polygons are automatically clipped to the screen border when they are composited with the root mask. The algorithm's geometric simplicity makes it an attractive candidate for hardware implementation. Regarding incremental evaluation, the algorithm identifies hierarchical blocks of pixels covered by the same polygon, so it is straightforward to apply standard incremental methods for evaluating shading functions. Moreover, indices into image data structures can be computed incrementally, as can edge-mask intercepts, as mentioned above. Finally, we note that the algorithm is probably well suited to parallel implementation on a moderate scale, since it exploits image-space coherence in much the same way as the hierarchical z-buffer visibility algorithm, which has good parallel performance as discussed in section 5.5.1.

6.6 Hierarchical Object-Space Culling

Because of its ability to cull hierarchically in image space, the hierarchical tiling algorithm processes densely-occluded scenes much more efficiently than conventional tiling methods, which must traverse hidden geometry pixel by pixel. Nonetheless, it must consider every polygon in a scene, doing some work even on those that are entirely hidden. To avoid this behavior, we integrate our algorithm with the basic hierarchical visibility algorithm to enable object-space culling of hidden regions of the model. This can be done by substituting hierarchical tiling for z-buffering in the hierarchical z-buffer algorithm of section 5, which requires some changes in both the object-space and image-space hierarchies. In image space, instead of using a z-pyramid of depth samples to maintain visibility information, we use a mask pyramid. In object space, we modify the octree to permit strict front-to-back traversal of polygons. Note that the z-buffer algorithm traverses octree cubes in front-to-back order, but not the primitives contained in them. And since octree cubes are nested, it is not sufficient to simply organize the primitives inside each cube in a BSP tree. Instead we use the following algorithm for building an *octree of BSP trees* that does permit strict front-to-back traversal.

Building an Octree of BSP Trees

Starting with a root cube which bounds model space, we insert polygons one at a time into the cube. If the polygon count in the cube reaches a specified threshold (e.g. 80), we subdivide the cube into eight octants and insert each of its polygons into each octant that it intersects, clipping to the cube's three median planes. When all polygons in the scene have been inserted into the root cube and propagated through the tree, we have an octree where all polygons are only associated with leaf nodes, thereby circumventing the ordering problem caused by nesting. The last step is to organize the polygons in each leaf node of the octree into a BSP tree [10]. Now scene polygons can be traversed in strict front-to-back order by traversing octree cubes front to back and traversing their BSP trees front to back.

Combining Hierarchical Tiling with Hierarchical Visibility

Now combining hierarchical tiling with the basic hierarchical visibility algorithm is very straightforward. As with the hierarchical z-buffer visibility algorithm, we traverse octree cubes in front to back order, testing them for visibility and culling those that are hidden. To test cubes for visibility, we modify the tiling procedure of LISTING 6.6 to report visibility status of polygons, returning TRUE whenever a polygon's mask indicates that it covers a subcell or a point in the image raster. Cube-visibility testing can be done much more efficiently with hierarchical tiling than with z-buffering, because tiling is strictly two dimensional, so the visibility of a cube can usually be established by tiling its polygonal silhouette. By comparison, z-buffering often needs to tile three faces of a cube to establish its visibility. Once we have established that an octree cube is visible, we traverse the polygons in its BSP tree front to back, tiling them into the mask pyramid and updating the image buffer when visible samples are encountered. When we finish traversing the octree, all visible polygons have been tiled and the image is complete.

This version of the hierarchical visibility algorithm has very efficient traversal properties in both object-space and image-space. Like the z-buffer algorithm of section 5, in object space the algorithm only visits visible octree nodes and their children, and it only renders polygons that are in visible octree nodes. In image space, when tiling polygons into the mask pyramid it only visits cells that are crossed by visible edges in the output image. Visible samples are never overwritten. As a result of these properties, the hierarchical visibility algorithm is very efficient in culling hidden geometry and tiling visible geometry.

Accelerating Image-Space Culling

In addition to enabling object-space culling, organizing polygons in bounding boxes can also accelerate image-space culling. A polygon can only be visible at samples where its bounding box is visible, and this can be exploited in two ways. First, tiling a polygon can begin at the smallest mask in the pyramid that contains all of its bounding box's visible samples. Secondly, there is a simple strategy that avoids attempting to tile any polygon into any mask in which it has no visible samples, which works as follows. When a cube is tested for visibility, we traverse all visible samples on the cube and store the count of visible samples with each mask in the pyramid that is traversed. When polygons contained in the cube are tiled, this count is decremented when visible samples are covered. If the count for a particular mask falls to zero, we know that no visible samples remain within the cube's silhouette, so the corresponding region of the screen can be ignored. This strategy will often prevent unnecessary subdivision, because masks will often contain visible samples that are outside the cube's silhouette. And if the count falls to zero at the coarsest level of subdivision, indicating that all samples within the cube's silhouette are covered, we know that all remaining polygons in that subtree of the octree are hidden and can be culled.

6.7 Implementation and Results

Our initial implementation of hierarchical tiling is programmed in C and is integrated with the hierarchical visibility algorithm as described in the preceding section. It renders either point-sampled or filtered images with either flat or Gouraud shading. Fil-

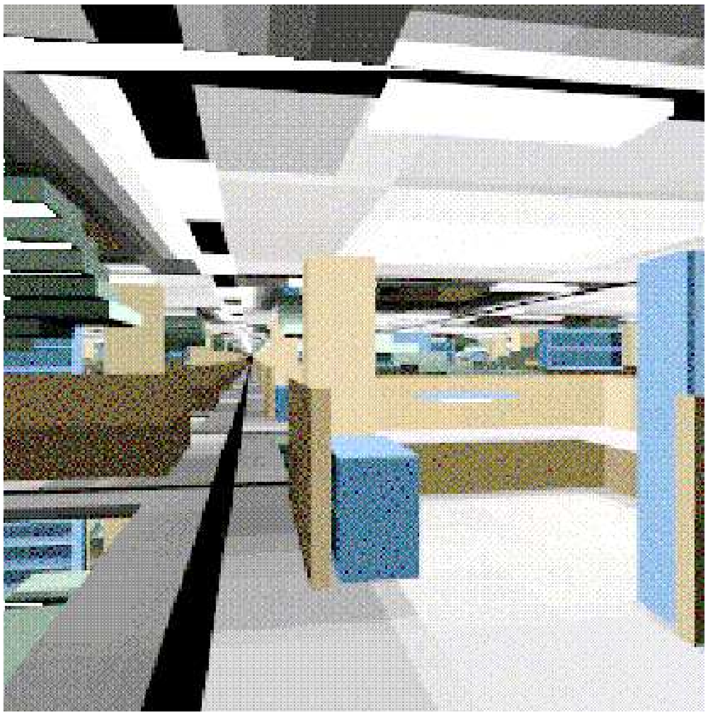


Figure 6.7: A frame from a motion sequence through an scene containing upwards of 500 million replicated polygons. Our unoptimized implementation of hierarchical tiling with object-space culling rendered this flat-shaded 512×512 antialiased image in approximately 11 seconds on a 50mz R4000 Crimson. Visibility was determined on a 4096×4096 raster to permit A-buffer-style filtering on an 8×8 grid within each pixel.

tering is done using the area-sampling method outlined in pseudocode. The accumulation buffer for filtered images is 16 bits deep in each color channel. Triage masks were constructed with 64 intervals along an edge. We tested our algorithm by generating antialiased motion sequences of a densely-occluded scene, and observed good-quality antialiasing. Figure 6.7 is a frame from this animation.

We tested our implementation on a version of the office-interior model described in section 5. We built an octree of BSP trees for the repeating module, each BSP tree containing approximately 16,000 polygons, and replicated the octree in a 33×33×33 grid to simulate a densely-occluded scene with upwards of 500 million polygons. The hierarchical tiling algorithm with object-space culling rendered 512×512 antialiased frames with flat shading in approximately 11 seconds on a 50mz R4000 Crimson. As our area-sampling method calls for, a single shading value was computed for each visible primitive at each pixel, and visibility was determined at all samples on a uniform

4096×4096 grid. Rendering this same scene with Gouraud shading took an additional two seconds per frame. By comparison, the hierarchical z-buffer algorithm of section 5 took 6.5 seconds to render point-sampled flat-shaded 512×512 images of this model. Although this is 1.7 times faster than our implementation of hierarchical tiling, hierarchical tiling determines visibility at 64 times as many samples. When this is taken into consideration, we see that the hierarchical tiling implementation processed visible samples 38 times faster than the z-buffer algorithm. It should also be noted that many more primitives are visible on the finer raster. As previously mentioned, we observed good-quality antialiasing in motion sequences of this scene with our implementation of hierarchical tiling. By comparison, images produced with the z-buffer algorithm aliased badly as expected.

It should be noted that these results were produced by our initial implementation which fails to exploit a number of optimizations and opportunities to exploit incremental methods. For example, polygon tiling always begins at the root mask and we don't use the image-space bounds of octree cubes to prevent unnecessary subdivision as described in section 6.6. Also, we don't use traditional incremental evaluation of Gouraud-shading values, even though the algorithm facilitates this by identifying blocks of pixels in which a single polygon is visible. In addition, we use floating-point rather than fixed-point arithmetic throughout. All things considered, our preliminary results suggest that a careful implementation of the hierarchical tiling algorithm on the latest generation of microprocessors would be able to render frames of our replicated office model at frame rates permitting convenient interaction.

6.8 Conclusion

Hierarchical polygon tiling in combination with the basic hierarchical visibility algorithm simultaneously exploits precomputation, hierarchical culling, and the parallelism inherent in boolean mask operations. To exploit precomputation, triage tiling patterns are precomputed and stored in lookup tables and the scene model is presorted in an octree of BSP trees. The main limitation of the algorithm is that it can be costly to maintain the octree of BSP trees if numerous primitives are moving independently. These data structures permit tiling and visibility operations to be performed hierarchically, with triage masks driving recursive subdivision of image space. The result is a very efficient subdivision procedure that only visits visible nodes in the octree, only visits cells in the image-space hierarchy that are crossed by visible edges, and never overwrites a visible sample. The algorithm is compact and straightforward to implement, has very modest memory requirements, and is amenable to parallel implementation. In short, hierarchical visibility and tiling methods working together offer the prospect of rendering very complex scenes at interactive rates with modest computing resources.

Bibliography

[1] G. Abram, L. Westover, and T. Whitted, "Efficient Alias-Free Rendering using Bit-Masks and Look-Up Tables," *Proceedings of SIGGRAPH '85*, July 1985, 53–59.

[2] A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[3] P. J. Burt and E. H. Adelson, "A Multiresolution Spline with Applications to Image Mosaics," *ACM Transactions on Graphics*, 2(4), Oct. 1983, 217–236.

[4] J. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal*, 4(1), 1965, 25–30.

[5] L. Carpenter, "The A-buffer, an Antialiased Hidden Surface Method," *Proceedings of SIGGRAPH '84*, July 1984, 103–108.

[6] E. Catmull, "A Subdivision Algorithm for Computer Display of Curved Surfaces," PhD Thesis, Report UTEC-CSc-74-133, Computer Science Dept., University of Utah, Salt Lake City, Utah, Dec. 1974.

[7] E. Catmull, "A Hidden-Surface Algorithm with Anti-Aliasing," *Proceedings of SIGGRAPH '78*, Aug. 1978, 6–11.

[8] E. Fiume, A. Fournier, and L. Rudolph, "A Parallel Scan Conversion Algorithm with Anti-Aliasing for a General-Purpose Ultracomputer," *Proceedings of SIGGRAPH '83*, July 1983, 141-150.

[9] E. Fiume, "Coverage Masks and Convolution Tables for Fast Area Sampling," *Graphical Models and Image Processing*, 53(1), Jan. 1991, 25–30.

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

[11] H. Fuchs, J. Kedem, and B. Naylor, "On Visible Surface Generation by a Priori Tree Structures," *Proceedings of SIGGRAPH '80*, June 1980, 124-133.

[12] A. S. Glassner, "Space Subdivision for Fast Ray Tracing," -Wozny83 4(10), Oct. 1984, 15–22.

[13] N. Greene, M. Kass, and G. Miller, "Hierarchical Z-Buffer Visibility," *Proceedings of SIGGRAPH '93*, July 1993, 231–238.

[14] N. Greene, "Detecting Intersection of a Rectangular Solid and a Convex Polyhedron," *Graphics Gems IV*, Ed: P. Heckbert, 1994, 71–79.

[15] D. Jevans and B. Wyvill, "Adaptive Voxel Subdivision for Ray Tracing," *Proceedings of Graphics Interface '89*, June 1989, 164–172.

[16] M. Kaplan, "The Use of Spatial Coherence in Ray Tracing," in *Techniques for Computer Graphics, etc.*, Ed: D. Rogers and R. A. Earnshaw, Springer-Verlag, New York, 1987.

[17] T. Kay and J. Kajiya, "Ray Tracing Complex Scenes," *Proceedings of SIGGRAPH '86*, Aug. 1986, 269–278.

[18] D. Meagher, "The Octree Encoding Method for Efficient Solid Modeling," PhD Thesis, Electrical Engineering Dept., Rensselaer Polytechnic Institute, Troy, New York, Aug. 1982.

[19] B. Naylor, "Interactive Solid Geometry Via Partitioning Trees," *Proceedings of Graphics Interface '92*, 1992, 11–18.

[20] B. Naylor, "Partitioning Tree Image Representation and Generation from 3D Geometric Models," *Proceedings of Graphics Interface '92*, 1992, 201–212.

[21] D. R. Reddy and S. M. Rubin, "Representation of Three-Dimensional Objects," Technical Report CMU-CS-78-113, Computer Science Dept., Carnegie-Mellon University, April 1978.

[22] D. Rogers, *Procedural Elements for Computer Graphics,* McGraw-Hill, New York, 1985.

[23] S. M. Rubin and T. Whitted, "A 3-Dimensional Representation for Fast Rendering of Complex Scenes," *Proceedings of SIGGRAPH '80*, July 1980, 110-116.

[24] P. Sabella and M. Wozny, "Toward Fast Color-Shaded Images of CAD/CAM Geometry," *IEEE Computer Graphics and Applications*, Nov. 1983, 60–71.

[25] D. Salesin and J. Stolfi, "The ZZ-buffer: A Simple and Efficient Rendering Algorithm with Reliable Antialiasing," *Proceedings of the PIXIM '89 Conference*, Hermes Editions, Paris, Sept. 1989, 451–466.

[26] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A Characterisation of Ten Hidden-Surface Algorithms," ACM Computing Surveys, 6(1), March 1974, 1–55.

[27] S. Teller and C. Séquin, "Visibility Preprocessing for Interactive Walkthroughs," *Proceedings of SIGGRAPH '91*, July 1991, 61–69.

[28] S. Teller, "Visibility Computations in Densely Occluded Polyhedral Environments," PhD Thesis, Univ. of California at Berkeley, Report No. UCB/CSD 92/708, Oct. 1992.

[29] J. Warnock, "A Hidden Surface Algorithm for Computer Generated Halftone Pictures," PhD Thesis, Computer Science Dept., University of Utah, TR 4-15, June 1969.

[30] L. Williams, "Pyramidal Parametrics," *Proceedings of SIGGRAPH '83*, July 1983, 1–11.

Lecture D

Geometric Simplification

Jarek Rossignac
IBM T.J. Watson Research Center
PO Box 704, Yorktown Heights, NY 10598
jarek@watson.ibm.com

Section 7

Geometric Simplification

Abstract

Interactive graphics response is a must for many 3D applications. Yet, because of the geometric complexity of the 3D models, it may require expensive hardware, complex algorithmic preprocessing, a compromise on the quality and accuracy of the images, or combinations of these. The relative impact of software techniques for accelerating the rendering of 3D scenes depends on the characteristics of the model and on the hardware configuration. After a brief review of various techniques, we present a simple, effective, and efficient solution for automatically computing geometric approximations of arbitrary polyhedra. Our technique is based on face triangulation and on vertex clustering. It is currently exploited in IBM's 3D Interaction Accelerator to pre-compute several simplified models of each object in the imported databases. These models resemble the original objects with decreasing accuracy and have decreasing geometric complexity (i.e. triangle count). They are used as levels of detail during interactive navigation and manipulation of the models. Using a lower level of detail when displaying small, distant, or background objects improves graphic performance without a significant loss of perceptual information, and thus enables real-time inspection of complex scenes. The simplification process is more efficient, more robust, and simpler to implement than alternative approaches, because it does not require building and maintaining a topological adjacency graph. Although it is not adaptive and does not produce optimal simplifications for a given complexity reduction, it appears particularly well suited for mechanical and architectural CAD model, because it automatically groups and simplifies features that are geometrically close, but need not be topologically close nor even part of a single connected component.

7.1 Introduction

Computer graphics is an effective means for communicating three-dimensional concepts, for inspecting CAD models, and for interacting with education and entertainment software. Although photo-realistic images and video sequences, typically computed off-line, have their own merits, interactive graphics is required when direct manipulation is important. Direct manipulation increases ease-of-use and enhances productivity by exploiting our natural ability to use visual cues when controlling our gestures. The closed loop involving the hand, the modeling or animation software, the graphics, and the human vision is only effective if graphic feedback is instantaneous. Delays between the gestures and the resulting image, lead to overshooting, reduce the feeling of control, and thus make the user less productive.

Interactive manipulation of 3D models and interactive inspection of 3D scenes cannot be effective when the graphics feedback requires more than a fraction of a second. Although the graphics performance, benefiting from faster CPUs, leaner APIs, and dedicated graphics subsystems, has significantly increased in the recent years, it is lagging behind the galloping complexity of CAD models found in industrial and consumer applications and also of scientific datasets [15]. 3D models of typical mechanical assemblies (appliances, engines, cars, aircrafts...) contain thousands of curved surfaces. Accurate polyhedral approximations of these exact models, typically constructed to interface with popular graphics APIs, involve millions of polygonal faces. Such complexity exceeds by one or two orders of magnitude the rendering performance of commercially available graphics adapters. The solution is not likely to come from hardware development alone. Algorithmic solutions must be employed to reduce the amount of redundant or unessential computation.

We define redundant computation as the processing steps that could be omitted without affecting the resulting images. Unessential computation is defined as the steps which, when omitted, affect the resulting image only to a moderate degree, so that the visual discrepancies between the correct image and the one produced do not hinder the user's perception and understanding of the scene. The elimination of all redundant and unessential computations remains an open research issue, because the computational cost associated with the identification of what is redundant and what is not may often offset the benefits of eliminating redundancy. Several techniques are reviewed below.

7.2 Rendering cost

To better identify redundant and unessential steps for rendering polyhedral scenes, we use an over-simplified model of the rendering cost. The scene is described by its geometry (the location of the polygons and the location and characteristics of the view) and by the associated photometric properties (light sources, surface colors, surface properties, textures). Rendering techniques compute, for each pixel, the amount of light reflected towards the eye by the objects in the scene. This computation may be arranged in various ways to cater to the desired degrees of photorealism and to take advantage of the available graphics hardware or libraries. We focus here on rasterization techniques, which visit each polygon in the scene and combine the results in a frame buffer using a z-buffer for hidden surface elimination. This choice is dictated by the prevalence and performance of affordable 3D graphics hardware accelerators and rasterizers, and by the success of associated APIs.

Assume that the geometry of our scene is composed of T triangles. (We use triangle counts in these arguments, instead of polygons, for simplicity and because polygons are typically converted into triangles, either during preprocessing or during rendering, and are an accepted unit of graphics performance.)

The cost of rendering a scene comprising T triangles depends on a number of factors, such as the window size, the lighting model, the amount of memory paging involved, or how the triangles project onto the screen.

Rendering T triangles involves several costs: - F: the cost of fetching the necessary triangles into cache memory - X: the cost of transforming their vertices and lighting them using associated normals - C: the cost of clipping the triangles and computing slopes - R: the cost of rasterizing them

The overall rendering cost (i.e. performance) depends on the particular architecture of the 3D graphics subsystem. Three families of uniprocessor architectures are popular for graphics: - all software rendering - hardware rasterization with software geometric processing, - all hardware rendering

For purely software rendering architecture, the total cost is the sum of all the costs: $F(T) + X(T) + C(T) + R(T)$. For an architecture based on a hardware rasterizer, the total cost is the maximum of $max(F(T) + X(T) + C(T), hR(T))$, where h provides the boost factor of the graphics rasterizer hardware. Because the hardware or the software may be bottlenecks, we use max instead of a sum when combining the individual costs. Similarly, because an all hardware rendering subsystem is pipelined, its cost is the maximum of the costs at each stage, i.e.: $max(F(T), kX(T) + kC(T), hR(T))$, where k provides the boost factor of the hardware supported transformation, lighting, and clipping.

7.3 Graphics acceleration techniques

The performance enhancing techniques reviewed in this section help reduce the different aspects of the overall rendering cost.

7.3.1 Meshing or storing transformed vertices

Note that in a polyhedral scene, the number of triangles is roughly twice the number of vertices. For example, for a triangulated manifold polyhedron with genus zero, the Euler-Poincarre formula becomes: $T - E + V = 2$, where E is the number of edges and V the number of vertices. Notice that $2E = 3T$, because there are three edges for each triangle, but each edge is shared by two adjacent triangles. Combining these two equalities yields: $T = 2V + 4$.

Therefore, independently processing the vertices of each triangle may unnecessarily increase $X(T)$, and maybe even $F(T)$, by a factor of 6, since each vertex is processed six times.

To reduce this computational redundancy, 3D graphics adapters and associated APIs support triangle meshes, where each vertex is used in conjunction with two of the recently processed vertices to define the next triangle. If very long meshes were used, the majority of vertices, and associated normals would only be processed twice, reducing by three the geometric cost of transforming and lighting the vertices. Note however that half of this cost is still redundant. Furthermore, constructing long meshes is algorithmically expensive.

In architectures where the geometric transformation and lighting is performed in software, all the vertices could be transformed only once and the resulting coordinates stored for clipping and rasterization, as needed for the triangles. Similarly, normals may be transformed and the associated colors computed and stored (depending on the lighting model, colors may be defined by normals only or by combinations of normals and vertices). For compatibility reasons with hardware graphics adapters, this possibility is not systematically exploited by popular APIs.

7.3.2 Frustum culling

Graphics performance may in general be significantly improved by avoiding to process the geometry of objects that do not project on the screen. Such objects may lie behind the viewpoint or outside of the viewing frustum. Simple bounds, such as min-max boxes or tight spheres around the objects may be easily pre-computed, updated during model editing or animation, and used for efficient culling. For scenes involving a large number of objects or for scenes where individual objects are relatively large and involve large numbers of triangles, pre-computed hierarchical spatial directories (see [1], [2] or [19]) are used to quickly prune portions of the scene outside the viewing frustum. These early culling techniques may have no effect when the entire models is examined and fits in the viewing frustum. In average however, they may significantly affect $F(T)$, because one needs not fetch models whose bounds is outside the frustum, and of course also $X(T)$ and $C(T)$. They do not affect $R(T)$, because these triangles would have been rejected through clipping anyway. For example, culling may reduce $X(T)$ and $C(T)$ 6 times if the viewpoint is in the center of the scene and if the viewing frustum spans a 90 degrees angle.

7.3.3 Smart caching and pre-fetching

Arranging the data so that contiguous memory locations are accessed by the graphics subsystem and using secondary processors for pre-fetching data may eliminate the delays caused by page faults and reduce $F(T)$. This is particularly effective if the size of the model that is not culled out exceeds the available memory [6].

7.3.4 Visibility

Culling objects or individual triangles that intersect the frustum but are invisible in the current view, because hidden by other objects, would directly impact the overall cost for all three types of architectures. Visibility information may be pre-computed for a given granularity of the space and of the models. For example, [20] uses the model's geometry to subdivide space into cells, associating with each cell c the list of other cells visible from at least one point in c. Cells may correspond to a natural, semantic, partition of the scene (floors, rooms, corridors) or to more general partitions induced by planes that contain the faces of the model. The preprocessing is slow and requires storing vast amounts of visibility information, unless the number of cells and the number of objects in the scene is small. A different approach was used in [7], where a hierarchical quadtree representation of the z-buffer generated after displaying the front most objects is used to quickly cull hidden objects. Front most objects candidates are computed from the previous frame. Culling compares a hierarchical bound system of the 3D scene against the z-buffer quadtree. Both techniques are effective under the appropriate conditions, but are of little help in scenes, such as factories or exterior views, where very few objects are completely hidden.

7.3.5 Polygon count reducing techniques

Using simplified models with a lower triangle count instead of the original models may in certain cases reduce paging F, and will in general reduce X and C significantly. A comparative discussion of some of these technique may be found in [8]. We summarize here several trends.

Polyhedral models are often constructed to approximate curved shapes through tessellation. Polyhedral approximations with fewer vertices and faces can thus in principle be produced by using coarser tessellation parameters. Emerging high-end graphic architectures support adaptive tessellation for trimmed NURBS surfaces [12]. However, The simplification process should be made independent of the design history and should be automatic [3].

Surface fitting techniques to regularly spaced scanner data points [17] may also be used for producing levels of detail polyhedral approximations [4]. These techniques have been recently extended to unorganized data points [9] [5] and to new sparse points automatically distributed over an existing triangulated surface (either evenly or according to curvature) [21].

Several simplification techniques of dense triangular meshes approximating smooth surfaces have been proposed. Flat vertices (defined in terms of the normals to abutting triangles [11] or in terms of the distance to a locally approximating plane [18]) may be removed and the so created hole triangulated. The process may be repeated by progressively increasing the tolerance until a sufficiently low number of vertices is reached. The approach proposed in [10] performs local topological operations and geometric optimization techniques, which amount to pinching selected edges so as to eliminate pairs of triangles and finding locally optimal positions of the combined vertices. All these techniques may be viewed as replacing clusters of vertices by single vertices. Clusters are sets of vertices that are topologically connected and that satisfy certain smoothness properties. These approaches do not identify clusters explicitly, but rather proceed to merge vertices incrementally.

The geometric simplification introduced in [14] is aimed at very complex and fairly irregular CAD models of mechanical parts. In contrast to the techniques mentioned above, it explicitly clusters the vertices of the original model. Furthermore, it uses only geometric criteria in the clustering process, as opposite to topological criteria. These differences lead to suboptimal approximations, but also to a very efficient, robust, and algorithmically simple process. It will be presented in details in the next section.

7.4 Geometric simplification

We present here a new simplification technique that operates on boundary representations of an arbitrary polyhedron and generates a series of simplified models with a decreasing number of faces and vertices. The resulting models do not necessarily form valid boundaries of 3D regions–for example, an elongated solid may be approximated by a curve segment. However, the error introduced by the simplification is bounded (in the Hausdorff distance sense) by a user-controlled accuracy factor and the resulting shapes exhibit a remarkable visual fidelity considering the data-reduction ratios.

The original model of each object is represented by a vertex table containing vertex coordinates and a face table containing references to V, sorted and organized according to the edge-loops bounding the face. The simplification involves the following processing steps: - triangulation of faces - grading of vertices as to their visual importance - clustering of vertices - selecting representative vertices - elimination of redundant triangles, edges, vertices - adjustment of normals and remeshing the new triangles

7.4.1 Grading

A weight is computed for each vertex. The weight defines the subjective perceptual importance of the vertex. We favor (1) vertices that have a higher probability of lying on the object's silhouettes from an arbitrary viewing direction and (2) vertices that bound large faces that should not be affected by the removal of small details. The first factor may be efficiently estimated using the inverse of the maximum angle between all pairs of incident edges on the candidate vertex. The second factor may be estimated using the length of the longest among all of the edges incident upon the vertex.

7.4.2 Triangulation

Each face is decomposed into triangles supported by its original vertices. Because CAD models typically contain faces bounded by a large number of edges, a very efficient, yet simple triangulation technique is used [13]. The resulting table of triangles contains 3 vertex-indices per triangle.

7.4.3 Clustering

Based on geometric proximity, the vertices are grouped into clusters. With each vertex, we associate the corresponding cluster's id. Although a variety of clustering techniques was envisioned, we have opted for a simple clustering process based on the truncation of vertex coordinates. A box, or other bound, containing the object is uniformly subdivided into cells. After truncation of coordinates, the vertices falling within a cell will have equal coordinates. The clustering procedure takes as parameters the box in which the clustering should occur and the maximum number of cells along each dimension. The solid's bounding box or a common box for the entire scene may be used. The number of cells in each dimension is computed so as to achieve the desired level of simplification. A particular choice may take into account the geometric complexity of the object, its size relative size and importance in the scene, and the desired reduction in triangle count.

7.4.4 Synthesis

The vertex/cluster association is used to compute a vertex representative for each cluster. A good choice is the vertex closest to the weighted the average of the vertices of the cluster, where the results of grading are used as weights. Less ambitious choices, permits to compute the cluster's representative vertices without reading the input data twice, which leads to important performance improvements when the input vertex table is too large to fit in memory. Vertex/cluster correspondence yields a correspondence between the original vertices and the representative vertices of the simplified object. Thus, each triangle of the original object references three original vertices, which in turn reference three representative vertices. (Note that representative vertices are a subset of the original vertices.) The representative vertices define the geometry of the triangle in the simplified object.

7.4.5 Elimination

Many triangles may have collapsed as the result of using representative vertices rather than the original ones. When, for a given triangle, all three representative vertices are equal, the triangle degenerates (collapses) into a point. When exactly two representative vertices are equal the triangle degenerates into an edge. Such edges and points, when they bound a triangle in the simplified object are eliminated. Otherwise, they are added to the geometry associated with the simplified model. Duplicated, triangles, edges, and vertices are eliminated during that process. Efficient techniques may be invoked, which use the best compromise between space and performance.

7.4.6 Adjustment of normals

A final step computes new normals for all the triangles coordinates. It uses a heuristic to establish which edges are smooth. The process also computes triangle meshes.

7.4.7 Exploitation

Several levels of detail may be pre-computed for each object and used whenever appropriate to speed up graphics. In selecting the particular simplification level for a given object, it is important to take into accounts the architecture of the rendering subsystem so as not to oversimplify in situations where the rendering process is pixel bound. For example, the cost of rendering in software and in a large window an object that has a relatively low complexity but fills most of the screen is dominated by R. Consequently, simplification will have very little performance impact, and may reduce the image fidelity without benefit. On the other hand, displaying a scene with small, yet complex objects, via a software geometric processing on a fast hardware rasterizer will be significantly improved by simplification before the effects of using simplified models become noticeable.

7.4.8 Advantages

The process described above has several advantages over other simplification methods.

The computation of the simplification does not require the construction of a topological adjacency graph between faces, edges, and vertices. It works of a simple array of vertices and of an array of triangles, each defined in terms of three vertex-indices.

The algorithm for computing the simplification is very time efficient. In its simplest form, it needs to traverse the input data (vertex and triangle tables) only once.

The tolerance (i.e. bound on the Hausdorff distance between the original and simplified model) may be arbitrarily increased reducing the triangle count by several orders of magnitude.

To further reduce the triangle count, the simplification algorithm may produce non-regularized models. Particularly, when using the appropriate tolerance, thin plates may be simplified to dangling faces, long objects to isolated edges, and (groups of) small solids into isolated points.

The approach is not restricted by topological adjacency constraints and may merge features that are geometrically close, but are not topologically adjacent. Particularly, an arbitrary number of small neighboring isolated objects may be merged and simplified into a single point.

7.5 Experiments

The simplification algorithm was combined with the data import modules of IBM's 3D Interaction Accelerator and exercised on thousands of models of various complexity. It exhibits a remarkable performance characteristics, making it faster to recompute simplifications than to read them from disk. The algorithm pre-computes several levels of detail, which are then used at runtime to accelerate graphics during interaction with models comprising millions of triangles. The particular simplification level of a given object is computed so as to match a user specified performance or quality target while allocating more geometric complexity (and thus more rendering cost) to objects which a higher visual importance.

Our experience shows that typical CAD models of mechanical assemblies comprise dozens of thousands of objects. The relative size and complexity of the objects may vary greatly. A typical object may have a thousand triangles in its original boundary. The simplification process described here may be used to automatically reduce the triangle count by an average factor of 5 without impacting the overall shape and without hindering the users ability to identify the important features. Further simplifications lead to further reduction of the triangle count, all the way down to a single digit, while still preserving the overall shape of the object and making it recognizable in the scene. This simplification process has been further improved to yield a better fidelity/complexity ratio by incorporating topological and curvature considerations in the clustering process. However, these improvements only lead to additional computational costs and more complex code.

7.6 Conclusion

Although the performance of state of the art hardware graphics is growing rapidly, price constraints and the growing needs of industrial customers call for algorithmic solutions that improve the speed of visualizing complex 3D scenes. We have presented a new technique, which automatically computes one or several simplified graphics representations of each object. These representations may be used selectively in lieu of the original model to accelerate the display process while preserving the overall perceptual information content of the scene. The described method clusters vertices of the model and produces an approximate model where original faces are approximated with fewer faces defined in terms of selected vertices. Several simplified representations with different simplification factors may be stored in addition to the original model. Actual viewing conditions are used to establish automatically for each object which representation should be used for graphics.

Bibliography

[1] Airey, J., Rohlf, J. and Brooks, F. *Towards image realism with interactive update rates in complex virtual building environments*. ACM Proc. Symposium on Interactive 3D Graphics, 24(2):41-50, 1990.

[2] Clark, J. *Hierarchical geometric models for visible surface algorithms*. Communications of the ACM, 19(10):547-554, October 1976.

[3] Crow, F. *A more flexible image generation environment*. Computer Graphics, 16(3):9-18, 1982.

[4] DeHaemer and Zyda, M. *Simplification of objects rendered by polygonal approximations*. Computers and Graphics, 15(2):175-184, 1991.

[5] DeRose, T., Hoppe, H., McDonald, J. and Stuetzle, W. *Fitting of surfaces to scattered data*. In J. Warren, editor, SPIE Proc. Curves and Surfaces in Computer Vision and Graphics III, 1830:212-220, November 16-18, 1992.

[6] Funkhouser, T. *Database and Display Algorithms for Interactive Visualization of Architectural Models*, PhD Thesis, CS Division, UC Berkeley, 1993.

[7] Greene, N., Kass, M., and Miller, G., *Hierarchical z-buffer visibility*, ACM SIGGRAPH'93 Proceedings, pp:231-238, 1993.

[8] Heckbert P. and Garland M. *Multiresolution modeling for fast rendering*, Proc Graphics Interface'94, pp:43-50, May 1994.

[9] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J. and Stuetzle, W. *Surface reconstruction from unorganized points*. Computer Graphics, 26(2):71-78, July 1992.

[10] Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. *Mesh optimization*, Proceedings SIGGRAPH'93, pp:19-26, August 1993.

[11] Kalvin, A., Cutting, C., Haddad, B. and Noz, M. *Constructing topologically connected surfaces for the comprehensive analysis of 3D medical structures*. SPIE Image Processing, 1445:247-258, 1991.

[12] Rockwood, A., Heaton, K. and Davis, T. *Real-time Rendering of Trimmed Surfaces*. Computer Graphics, 23(3):107-116, 1989.

[13] Ronfard, R. and Rossignac, J. *Triangulating multiply-connected polygons: A simple, yet efficient algorithm*, Proc. Eurographics'94, Oslo, Norway, Computer Graphics Forum, 13(3):C281-C292, 1994.

[14] Rossignac, J. and Borrel, P. *Multi-resolution 3D approximations for rendering complex scenes*, pp. 455-465, in Geometric Modeling in Computer Graphics, Springer Verlag, Eds. B. Falcidieno and T.L. Kunii, Genova, Italy, June 28-July 2, 1993.

[15] Rossignac J. and Novak M. *Research Issues in Model-based Visualization of Complex Data Sets*, IEEE Computer Graphics and Applications, 14(2):83-85, March 1994.

[16] Scarlatos, L. and Pavlidis, T. *Hierarchical triangulation using cartographic coherence*. CVGIP: Graphical Models and Image Processing, 54(2):147-161, 1992.

[17] Schmitt, F., Barsky, B. and Du, W. *An adaptive subdivision method for surface-fitting from sampled data*. Computer Graphics, 20(4):179-188, 1996.

[18] Schroeder, W., Zarge, J. and Lorensen, W. *Decimation of triangle meshes*. Computer Graphics, 26(2):65-70, July 1992.

[19] Teller, S.J. and Sequin, C.H. *Visibility Preprocessing for interactive walkthroughs*. Computer Graphics, 25(4):61-69, July 1991.

[20] Teller, S. *Visibility Computations in Densely Occluded Polyhedral Environments*, PhD Thesis, UCB/CSD-92-708, CS Division, UC Berkeley, October 1992.

[21] Turk, G. *Re-tiling polygonal surfaces*. Computer Graphics, 26(2):55-64, July 1992.

Lecture E

Database Management for Models Larger Than Main Memory

Thomas A. Funkhouser
AT&T Bell Laboratories
600 Mountain Avenue, 2A-202, Murray Hill, NJ 07974
funk@research.att.com

Section 8

Slides

Interactive Visualization of Geometric Models Larger than Main Memory

Thomas A. Funkhouser
AT&T Bell Laboratories

Interactive Visualization

Goal:

- Interactive program that simulates the visual experience of walking through a large 3D model (e.g., building).

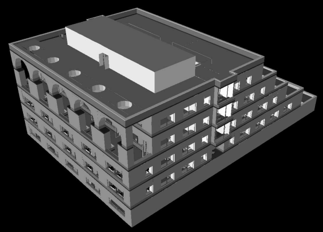
Related systems:

- 3D CAD packages: small models, slow frame rates.
- Flight simulators: special purpose processors.
- Commercial visualization systems: memory resident.
- Database systems: not interactive, not real−time.

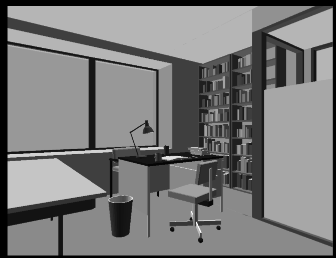
Challenges:

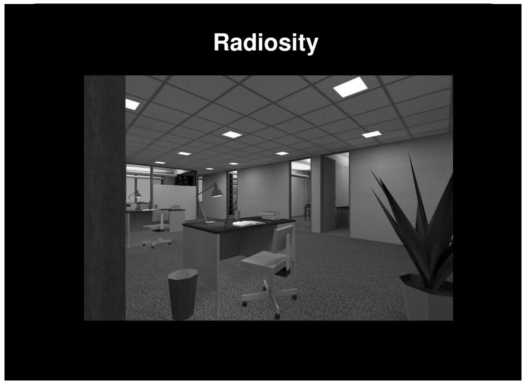
- Scale: large, detailed models.
- Fidelity: realistic looking images.
- Interactivity: fast, constant frame rates.
- Hardware: off−the−shelf computers.

Soda Hall Model

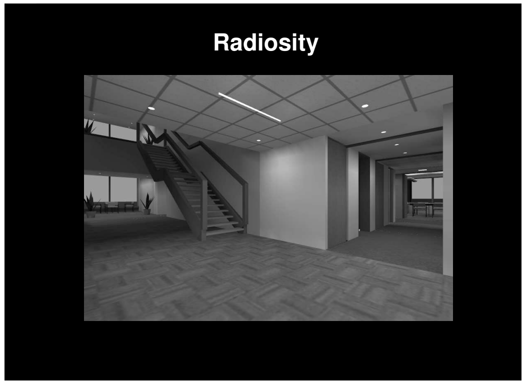


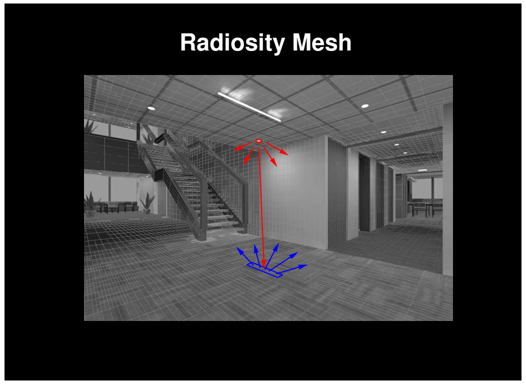
Soda Hall Model

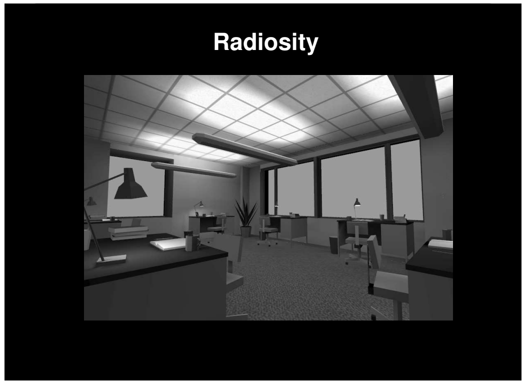


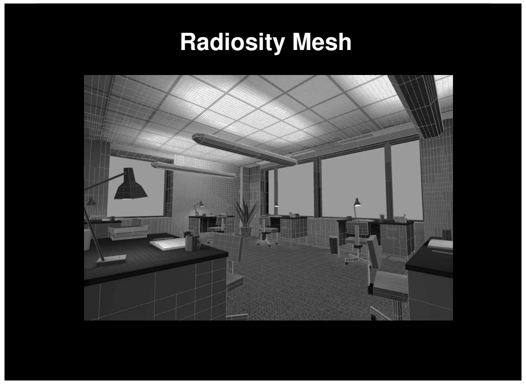

Radiosity

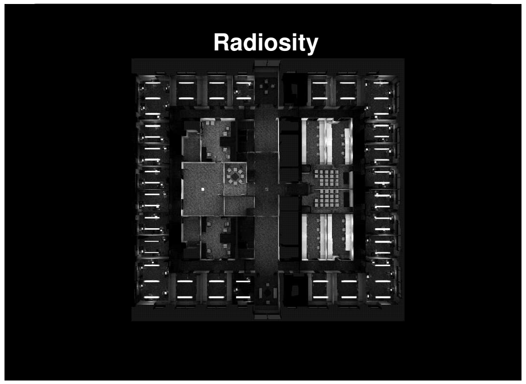

Radiosity

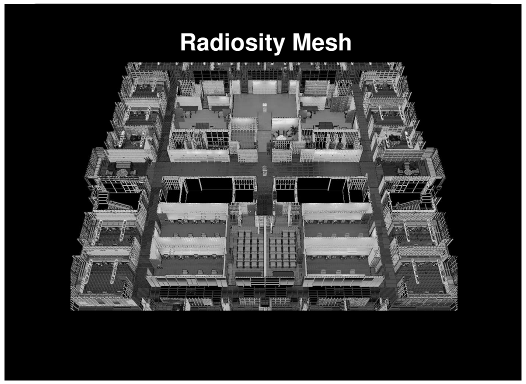

Radiosity


Radiosity Mesh


Radiosity


Radiosity Mesh


Radiosity


Radiosity Mesh

Problem Statement

Display Problem:

- There may be 10^7 polygons in the model.
- Must draw at least 10 frames per second.
- Able to draw at most 10^5 polygons per second.
- Model is 1000 times larger than can be drawn at interactive frame rates.

Database Problem:

- There may be 10^9 bytes in the model.
- Workstation has 10^8 bytes of main memory.
- Model is 10 times larger than can be stored in main memory.

Overview of Approach

→ • **Database Organization**

• **Display Management**

• **Database Management**

Database Organization

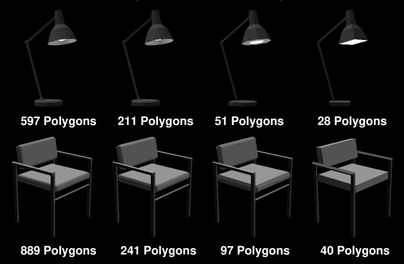
Goal:

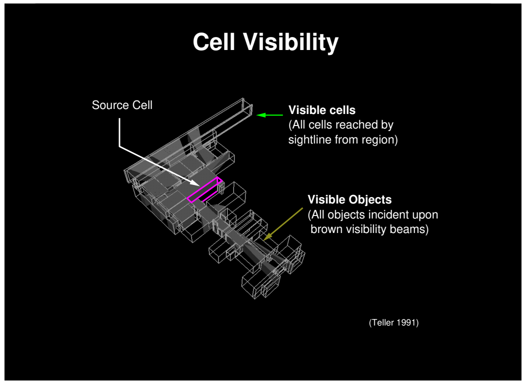
- Represent model in a manner suitable for executing real–time geometric queries, database paging, and display operations.

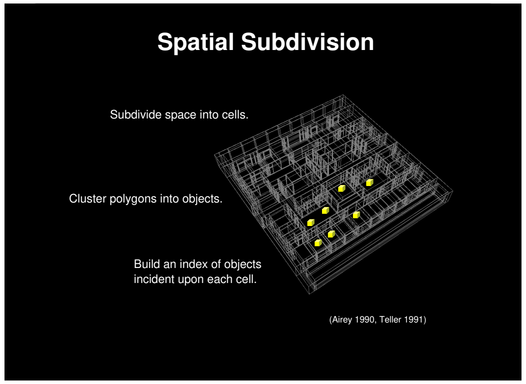
General Approach:

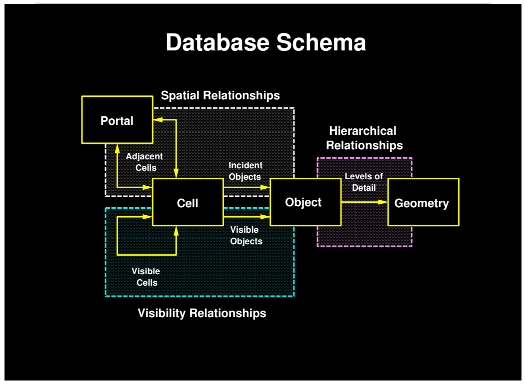
- Hierarchical object representation.
- Explicit, precomputed spatial relationships.
- Efficient data layout for real–time swapping.

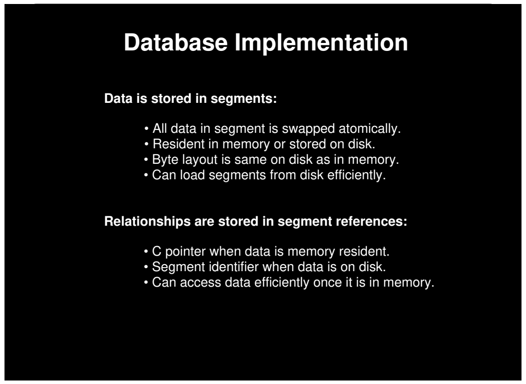
Object Hierarchy




Cell Visibility
Source Cell
Visible cells
(All cells reached by
sightline from region)
Visible Objects
(All objects incident upon
brown visibility beams)
(Teller 1991)


Spatial Subdivision
Subdivide space into cells.
Cluster polygons into objects.
Build an index of objects
incident upon each cell.
(Airey 1990, Teller 1991)


Database Schema
Spatial Relationships
Portal
Adjacent
Cells
Incident
Objects
Hierarchical
Relationships
Levels of
Detail
Cell
Object
Geometry
Visible
Objects
Visible
Cells
Visibility Relationships


Database Implementation
Data is stored in segments:
• All data in segment is swapped atomically.
• Resident in memory or stored on disk.
• Byte layout is same on disk as in memory.
• Can load segments from disk efficiently.
Relationships are stored in segment references:
• C pointer when data is memory resident.
• Segment identifier when data is on disk.
• Can access data efficiently once it is in memory.

Overview of Approach

- Database Organization
- Display Management
- Database Management

Display Management

Goal:

- Draw scene from the observer's viewpoint at interactive frame rates.

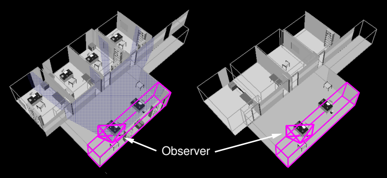
Display Problem:

- Too many polygons to draw at interactive frame rates.

General Approach:

- Cull objects invisible from observer viewpoint.
- Elide detail to maintain interactive frame rate.

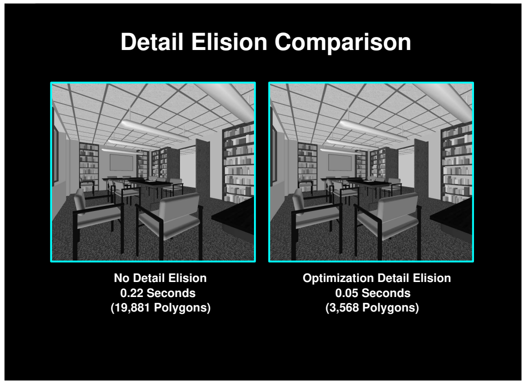
Observer Visibility

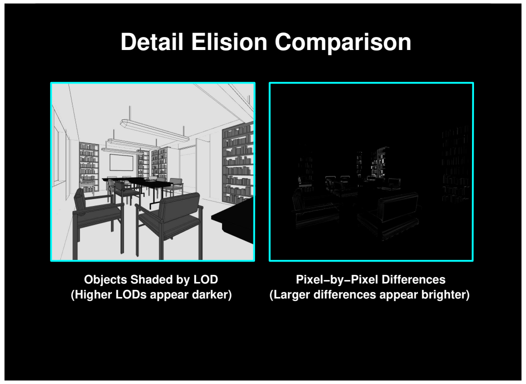


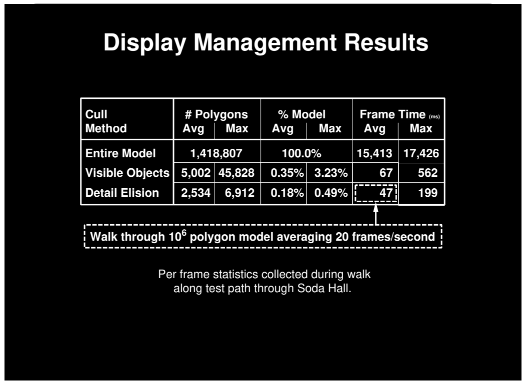
Visible Cells   Visible Objects

Detail Elision

Draw some objects with reduced detail to maintain target frame rate.







Cull Method	# Polygons Avg	# Polygons Max	% Model Avg	% Model Max	Frame Time (ms) Avg	Frame Time (ms) Max
Entire Model	1,418,807		100.0%		15,413	17,426
Visible Objects	5,002	45,828	0.35%	3.23%	67	562
Detail Elision	2,534	6,912	0.18%	0.49%	47	199

Overview of Approach

- Database Organization
- Display Management
- Database Management

Database Management

Goal:

- Make sure relevant portion of model is resident in memory during each frame.

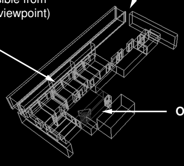
Problem:

- Model is larger than will fit into memory.
- Must swap model memory in real–time.
- Disk I/O operations have long latency.

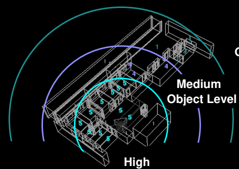
General Approach:

- Pre–fetch into memory all objects that might be relevant during frames in the near future.

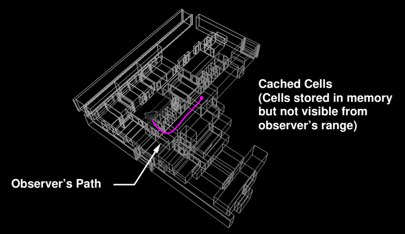
Pre–Fetch Visibility



Pre–Fetch Detail Elision



Database Cache



Database Management Results

Object Set	# Polygons Avg	# Polygons Max	# MBytes Avg	# MBytes Max	% Model Avg	% Model Max
Entire Model	1,418,807		349.5		100%	
Resident	97,081	128,102	14.52	18.10	4.15%	5.18%
Lookahead	66,297	362,831	9.92	54.41	2.84%	15.61%
Rendered	2,534	6,912	0.38	0.99	0.11%	0.28%
Read	266	6,501	0.02	0.80	0.01%	0.23%

Read only 20 KB
per frame on average

Store at most
18.1 MB in memory

Per frame statistics collected during walk
along test path through Soda Hall.

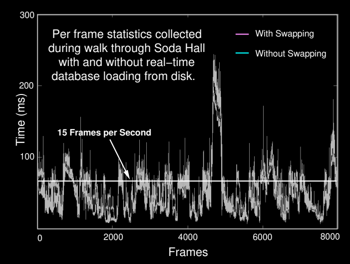
Database Management Results

Swap?	Compute Time (ms) Avg	Compute Time (ms) Max	Render Time (ms) Avg	Render Time (ms) Max	Swap Time (ms) Avg	Swap Time (ms) Max	Frame Time (ms) Avg	Frame Time (ms) Max
No	33	199	32	83	–	–	47	199
Yes	40	244	34	124	20	898	47	244

Average frame time is same with and without swapping.

Per frame statistics collected during walk
along test path through Soda Hall.

Database Management Results



Conclusion

Database Organization:

- Hierarchical object representation.
- Explicit, precomputed spatial relationships.
- Efficient data layout for real−time swapping.

Display Management:

- Cull objects invisible from observer viewpoint.
- Elide detail to maintain interactive frame rate.

Database Management:

- Pre−fetch into memory all objects that might be relevant during frames in the near future.

Interactive visualization of model larger than main memory.

Section 9

Database Management for Models Larger Than Main Memory[1]

Abstract

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

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

This paper describes an interactive building walkthrough system that uses these data structures and algorithms to maintain interactive frame rates during visualization of very large models. So far, the implementation supports models whose major occluding surfaces are axis-aligned rectangles (e.g., typical buildings). This system is able to maintain over twenty frames per second with little noticeable detail elision during interactive walkthroughs of a building model containing over one million polygons.

[1] Also appearing as: Thomas Funkhouser, Seth Teller, Carlo Séquin, and Delnaz Khorramabadi. The UC Berkeley System for Interactive Visualization of Large Architectural Models. To appear in *Presence: Teleoperators and Virtual Environments*, Volume 5, No. 1, Winter 1996.

9.1 Introduction

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

Today, graphics workstations offer a great potential for real-time simulation of movement through complex environments, particularly large buildings. A computer-based, interactive building walkthrough system can simulate the visual experience of moving through a three dimensional model of a building on the screen of a computer workstation by displaying rendered images of the model as seen from a hypothetical observer viewpoint under interactive control by the user. If images are rendered smoothly and quickly enough, the illusion of real-time exploration of the proposed building can be achieved.

A building walkthrough system might be useful for architects, interior designers, and clients to visualize and evaluate architectural designs before a building has been constructed. For example, an architect may be able to detect errors, such as material interferences; test lighting conditions, evaluating them at different times of day or during different seasons; and check the view out the windows of particular offices. An interior designer might experiment with several furniture arrangements, color schemes, and lighting arrangements in a computer simulation before purchases are made. Most importantly, a building walkthrough system provides a means by which architects, interior designers and clients can communicate their ideas to one another. In particular, it is extremely difficult for a typical client, who has commissioned a building, but has not been trained in visualization of three dimensional spaces, to understand what the inside of a building might actually look like by viewing blueprints or cardboard models. An interactive, three dimensional building walkthrough system allows an architect to show a client a proposed architectural design, and elicit real-time feedback as the client "walks" through the interior of the building interactively. As a result, faults in the architectural plan may be found earlier in the design cycle, thereby saving time and money.

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

9.1.1 Previous Work

Vehicle Simulators

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

Although there are many similarities between vehicle simulators and building walkthrough systems, there are several important differences. First, the types of environments encountered in vehicle simulators are quite different from building interiors. Typical vehicle simulator models contain terrain data augmented with plants, buildings, roads, etc. In these models, space tends to be "sparsely occluded" – i.e., there are few observer viewpoints for which a significant portion of the model is occluded by other parts of the model. In contrast, building models typically contain walls, ceilings, and floors which partition space into rooms. These models tend to be "densely occluded" – i.e., a large portion of the model is occluded by some polygon for observer viewpoints in the interior of the building. Therefore, visibility determination algorithms that cull not only polygons outside the observer's view, but also ones occluded by other polygons (e.g., walls) may be better suited for visualization of building models than for vehicle simulator models.

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

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

Mechanical CAD Systems

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

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

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

Architectural CAD Systems

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

Currently available commercial products that do allow interactive, real-time navigation generally support only small buildings models (e.g., less than one hundred thousand polygons), displayed with simple rendering algorithms (e.g., wire-frame or flat shading)

[5, 39]. These commercial systems generally make little use of sophisticated precomputation, visibility determination, or detail elision, and require that the entire model be resident in memory.

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

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

9.1.2 Goals

Model Size

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

Our test case is a model of Soda Hall, a seven floor academic building containing more than one hundred faculty and student offices, twelve computer laboratories, and six class rooms (Figure 9.1). Almost every room in the model contains a functionally complete set of furniture (Figure 9.2). For instance, each office has at least one desk, a few chairs, a bookshelf, a plant, etc. On each desk, there is at least one book, a desk lamp, and a few pencils (Figures 9.3 and 9.4). 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."

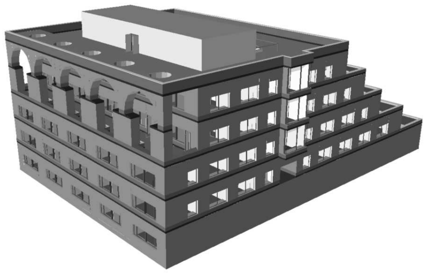


Figure 9.1: Exterior view of Soda Hall.

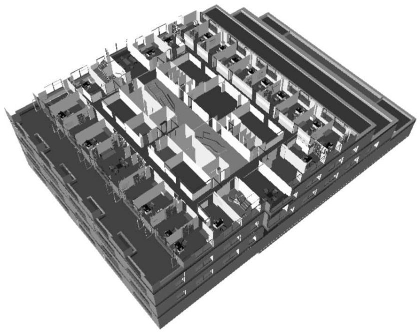


Figure 9.2: Exterior view of Soda Hall "cut open" by a horizontal plane at the sixth floor.

Figure 9.3: Typical office with furniture.

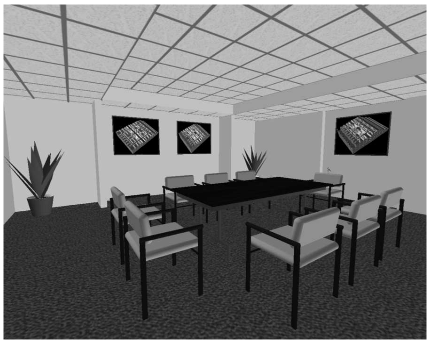


Figure 9.4: Board room with furniture.

Image Quality

A second goal of our work is to use *radiosity* illumination simulation methods [19, 26] to generate realistic-looking images with indirect diffuse reflections and shadows (Figures 9.5 and 9.6). Radiosity methods, based on models of radiative transfer methods in thermaengineering, consider every polygon a potential emitter or reflector of radiance (or luminance). Conceptually, for every pair of polygons, A and B, a form factor is computed which measures the fraction of the energy leaving polygon A that arrives at polygon B. This approach yields a set of simultaneous linear equations which can be solved to obtain the radiance for each polygon.

The advantage of radiosity methods for interactive visualization is that a global radiosity solution can be precomputed. The solution includes only the diffuse component of reflection and does not depend on a particular observer viewpoint (Figure 9.7). Therefore, a radiosity computation can be performed for an entire building model during a precomputation phase in which results are stored in a database for use later during interactive visualization. This approach off-loads the expensive illumination computations required to capture realistic lighting effects, such as shadows, so that rendering during interactive visualization can produce high-quality images quickly.

One difficulty associated with radiosity methods is that a tremendous amount of data may be required to describe the radiosity solution. A separate color is stored for each vertex of each polygon in the model; and large polygons are split into many smaller ones (where the gradient of radiosity is high) in order to capture complex illumination effects, such as highlights and shadow boundaries (Figure 9.8). Furthermore, although many of the polygons in the original model can be shared via hierarchical instancing, each polygon is meshed and illuminated independently during the radiosity computation, and must be stored separately in the resulting model. As a result, a model that originally contains millions of possibly shared polygons can generate a radiosity solution with tens of millions of separate polygons requiring gigabytes of data.

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

Performance

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

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

Short response times (i.e. the time required for the system to react to user input) is

Figure 9.5: Radiosity rendering of hallway.

Figure 9.6: Radiosity rendering of office.

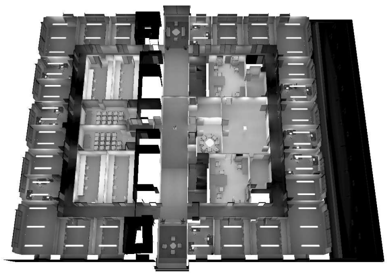


Figure 9.7: Radiosity computation and results are independent of observer viewpoint.

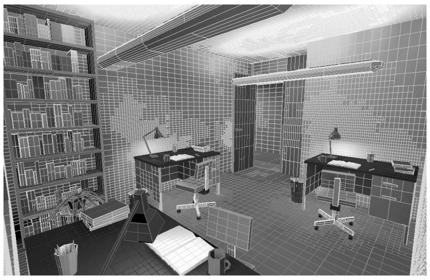


Figure 9.8: Polygonal mesh generated for the office during radiosity computation.

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

9.1.3 Problem Statement

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

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

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

9.1.4 System Organization

Our building walkthrough system is divided into three distinct computational phases as shown in Figure 9.9. First, during the *modeling phase*, we construct the building model from AutoCAD floor plans and elevations and populate the model with furniture. Next, during the *precomputation phase*, we perform a spatial subdivision and indexing, and compute observer-independent lighting and visibility information. Finally, during the *walkthrough phase*, we simulate an observer moving through the building model and render the model as seen from the observer viewpoint in each frame. The *display database* is the link between these three phases. It stores the complete building model, along with the results of the precomputation phase, for use during the walkthrough phase.

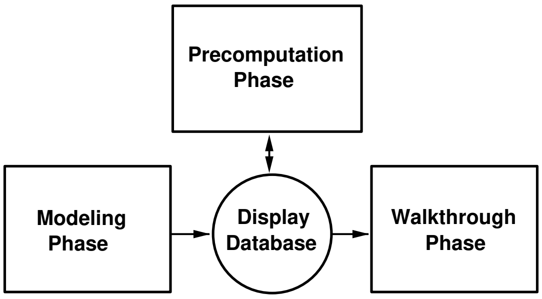


Figure 9.9: System overview.

Since details of many aspects of this system have been published in previous papers [15, 16, 17, 25, 35, 36, 37], this paper focuses on the overall system design. The paper is organized as follows. Sections 9.2 through 9.4 describe the three phases of the system, respectively. Special attention is paid to how the data structures and algorithms used in each phase are integrated into the entire system. Section 9.5 presents results of interactive tests using the walkthrough system. Finally, Section 9.6 contains a summary and conclusions.

9.2 Modeling

9.2.1 Model Loading

A primary concern of any interactive visualization system is to obtain a complete and geometrically consistent electronic model. In the case of Soda Hall, a set of files describing the major structural elements of the building (i.e., walls, ceilings, and floors) were prepared by architects using AutoCAD [4]. We implemented a converter that extracts the geometrical and surface attribute information embedded in AutoCAD DXF files [4] and translates them into the Berkeley UNIGRAFIX format [33] a format suitable for 3D object modeling, and then loads them into a display database, as shown in Figure 9.10.

Unfortunately, the raw architectural models were not true three-dimensional models. They were originally created for the generation of architectural blueprints such as

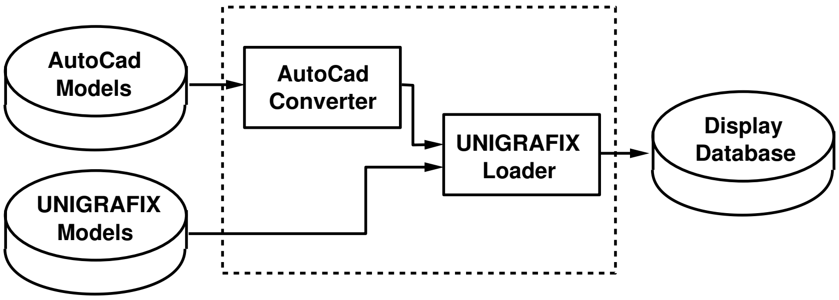


Figure 9.10: Loading operations of the modeling phase.

plan views and crude three dimensional views of buildings. Drafting packages used to prepare them were not written with anticipation that their products might be used in interactive walkthrough systems. Consequently, the initial version of the model that we received from the architects had many problems. Stairs and many other objects were missing; building components were not modeled as closed objects; windows were only line drawings on the walls; polygons were drawn with no consistent orientation; many co-planar polygons coincided, and many polygons were non-planar or intersected without sharing edges. Therefore, an element-by-element conversion of the architectural database to the UNIGRAFIX format would not have produced a viable model. We used automated programs to detect and correct most of these anomalies [25] and then manually corrected the remaining modeling errors with interactive tools.

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

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

9.2.2 Model Representation

The walkthrough display database represents the model as a set of *objects*, each of which can be described at more than one level of detail (LOD). For example, a chair may be described by three different representations, as shown in Figure 9.11: 1) a highly detailed chair containing hundreds of polygons to approximate the curved surfaces of the cushions and rounded edges of the arms and legs, 2) a slightly less-detailed chair with simpler polygonal approximations for the cushions, arms, and legs, and 3) a coarsely detailed chair with just a simple box for each cushion, arm, and leg. Simpler representations for objects are used during the interactive walkthrough phase to improve refresh rates and memory utilization.

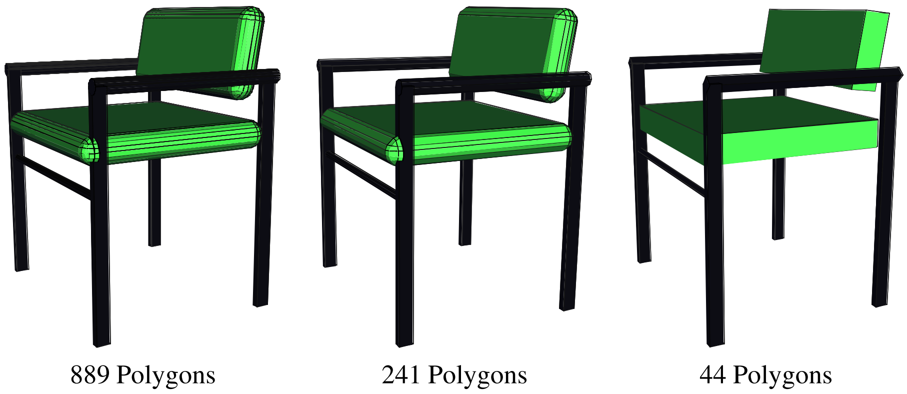


Figure 9.11: Three LODs for a chair.

In general, if there is more than one instance of the same object type (e.g., if the same type of chair appears in many positions throughout the building), all instances share the same *object definition*, which stores the *geometries* (i.e., vertices, polygons, materials, and textures) describing the object at each LOD. Each instance of the object may specify a 4x4 transformation matrix which is to be applied to the object definition, and a material which is to be applied to polygons that do not already have material attributes. An example hierarchy of object instances and definitions is shown in Figure 9.12. The model shown contains four instances of a chair whose definition has three LODs, and two instances of a table whose definition has only one LOD.

Although models are most often stored as a hierarchy with shared object definitions, the display database also allows an object instance to store a specific, separate geometry for any LOD. Geometries stored specifically with an object instance override the geometries for the corresponding LODs stored with the object definition. Other LODs (i.e., ones not explicitly stored with an object instance) are inherited from the object definition. This feature is important for the storage of radiosity information, since different instances of the same object definition are likely to be meshed and illuminated differently after a radiosity computation. An example object hierarchy containing an object instance with its own geometries is shown in Figure 9.13. Chair instance #1 inherits its lowest LOD from the object definition, whereas its medium and high LODs are specified explicitly.

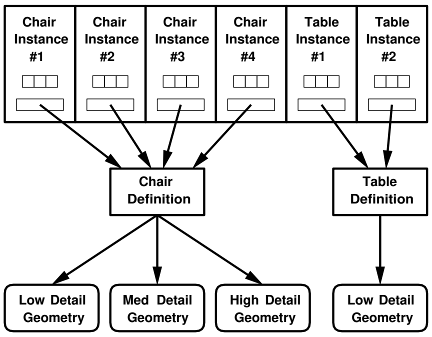


Figure 9.12: Object instances can share geometries stored in an object definition.

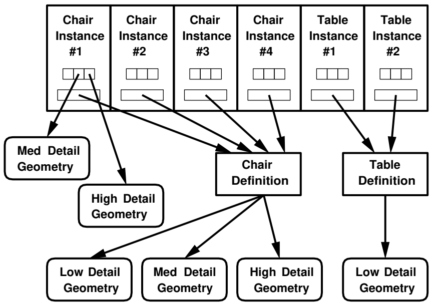
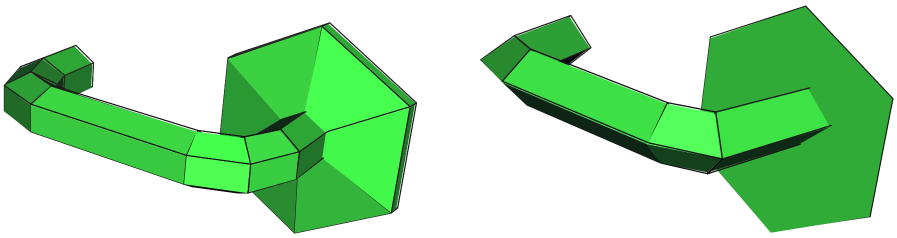


Figure 9.13: An object instance can store its own geometries.

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

9.2.3 Object Abstraction

Although we are currently developing automatic tools for object abstraction, so far we have used a variety of ad hoc techniques to create multiple LODs for each object definition. For objects created by procedural generation programs, it is usually possible to extend the programs to produce not only a very detailed model of the object, but also simpler representations as appropriate. For instance, the program used to generate the door handle shown in Figure 9.14 is parameterized to output segments with a user-specified number of sides.



6-sided prisms. 4-sided prisms.

Figure 9.14: Different door handle representations constructed using a parameterized generator.

For objects described as a CSG hierarchy of high-level shapes (e.g., boxes, spheres, cones, cylinders, etc.), it is possible to create simpler representations by a combination of: 1) choosing simpler representations for some shapes and 2) removing some shapes. Using these techniques, more than one polygonal representation must be constructed for only a few standard shapes, rather than many complex objects. Example results of this abstraction technique is shown in Figure 9.15.

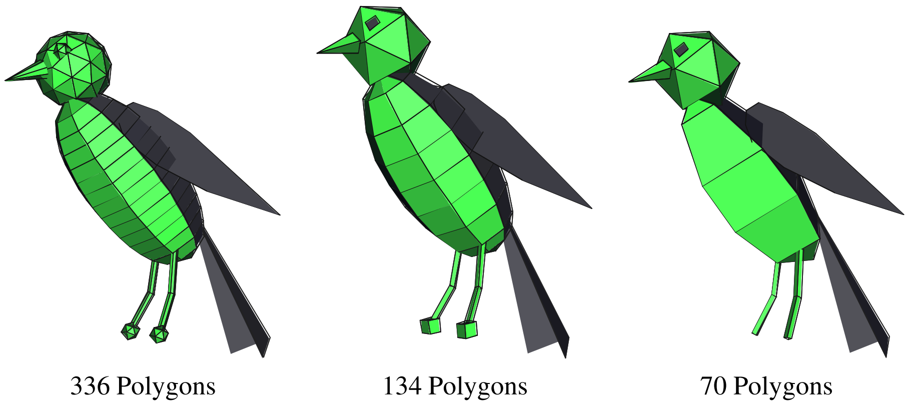


Figure 9.15: Three LODs for a canary.

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

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

9.2.4 Results

Using the techniques described in this section, we built a three dimensional polygonal model of Soda Hall, complete with furniture, textured materials, and multiple LODs for all furnishings. In all, the model contains 2,217,792 polygons, of which 1,418,807 represent objects at their highest LOD. Including the walls, ceilings, and floors of the building, the model contains 14,478 object instances of 8,037 unique object descriptions. It contains 129 unique pieces of furniture, 406 unique materials, and 58 unique textures. The display database for this model requires 21.5MB of storage if object instances reference shared object definitions, and 349.5MB of storage if all object instances are flattened (i.e., a separate copy of the object definition is stored for each instance). Overall statistics regarding the number of objects described at each LOD, and the cumulative number of polygons used to represent them are shown in Table 9.1.

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

Table 9.1: Multi-resolution modeling statistics for Soda Hall.

9.3 Precomputation

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

The steps of the precomputation phase are shown in Figure 9.16. We first perform a *spatial subdivision* in which the building model is partitioned into roughly room-sized *cells*, and a cell adjacency graph and an index of objects spatially incident upon each cell is constructed. We then perform a *visibility precomputation* in which sets of cells and objects visible from each cell are computed. Finally, a radiosity computation can be performed. The results of the precomputation phase are stored in the display database for use during the walkthrough phase.

9.3.1 Spatial Subdivision

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

After subdivision, cell *portals* (i.e., the transparent portions of shared boundaries) are identified and stored with each leaf cell, along with an identifier for the neighboring

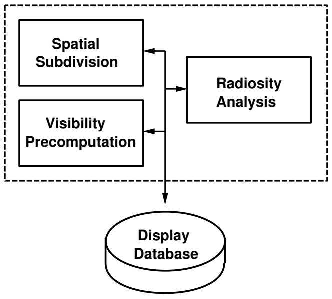


Figure 9.16: Functional steps of the precomputation phase.

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

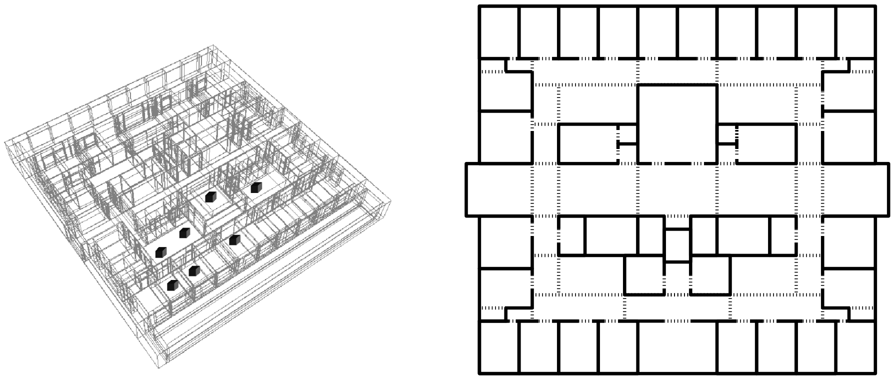
Sets of objects partially and completely inside cell boundaries are also constructed and stored with each cell. The space occupied by each object in the model is classified with respect to each cell in the spatial subdivision by a traversal of the k-D tree. At each node of the tree traversal, the bounding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

The current implementation of the building walkthrough system supports spatial subdivisions containing only axis-aligned, rectangular cell boundaries and portals (i.e., cells are axial three dimensional boxes, and portals are axial two dimensional rectangles). Such a spatial subdivision for the sixth floor of Soda Hall is shown in three dimensions in Figure 9.17a – cell boundaries are shown as gray outlines. Figure 9.17b shows a two dimensional schematic representation of the subdivision, in which opaque cell boundaries are represented by thick, black lines and portals are represented by dashed lines. During spatial subdivision, we precompute and store in the display database for each cell: 1) the portals on its boundaries, 2) the cells sharing its boundaries, and 3) the objects completely or partially inside its boundaries. See [35, 37] for more details.

9.3.2 Visibility Precomputation

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

We observe that a cell's visibility is the region of space to which an unobstructed *sightline* can lead from some point inside the cell. Such a sightline must be disjoint from



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

Figure 9.17: Spatial subdivision of the sixth floor of Soda Hall. The image on the left shows the actual three dimensional subdivision, while the image on the right shows a two dimensional schematic representation.

any opaque cell boundaries, so it must intersect, or *stab*, a portal in order to pass from one cell to the next (Figure 9.18). Sightlines connecting cells that are not immediate neighbors must traverse a *portal sequence*, each member of which lies on the boundary of an intervening cell. Therefore, a cell's visibility is the union of all points in space that can possibly be reached by a sightline that originates inside the cell, and intersects only portals at cell boundaries along the way.

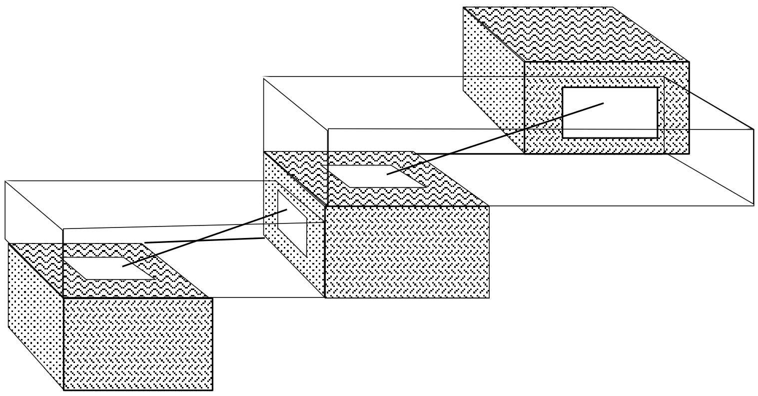


Figure 9.18: A sight-line stabbing a portal sequence.

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

The visible region of cells farther away from C typically narrows as the length of the portal sequence increases. After stepping through n portals, the visible region is a bowtie-shaped bundle of lines that stab every portal of the sequence, and which "fans out" beyond the final portal into an infinite wedge (Figure 9.19). During each iteration of the depth-first search, the visible region of the reached cell is determined by clipping the infinite wedge to the reached cell's boundary. The visibility of the source cell is the union of the visible regions of cells reached during the search (Figure 9.20).

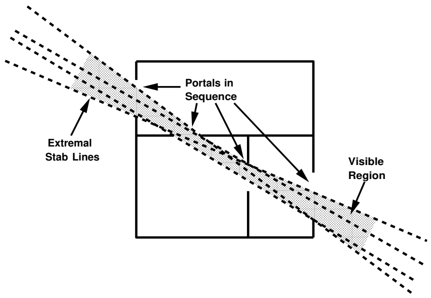


Figure 9.19: Bowtie-shaped region containing sightlines stabbing a portal sequence (stipple gray). Opaque boundaries are shown in solid black. Extremal sightlines are shown as dashed lines.

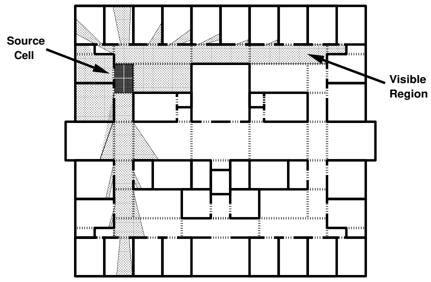
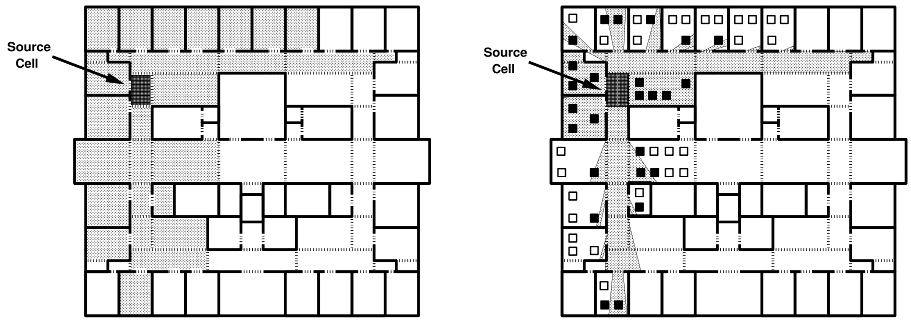


Figure 9.20: Cell visibility (stipple gray) for source cell (dark gray).

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

During the depth-first search for the source cell C, we construct its *cell-to-cell* and *cell-to-object* visibilities – i.e., the sets of cells and objects, respectively, that are potentially visible from C. The cell-to-cell visibility for C is the set of cells reached during the depth-first search originating from C. The *cell-to-object* visibility is the set of objects whose bounding boxes are incident upon some region visible from C. In Figure 9.21, schematic diagrams show the cell-to-cell visibility (stipple gray) and cell-to-object visibility (solid black squares) for a particular source cell (dark gray) in two dimensions. Figure 9.22 shows the cell visibility for a source cell in three dimensions. The visible region is the volume of space enclosed in the polyhedral beams emanating from the source cell; the cell-to-cell visibility is the set of cells outlined; and the cell-to-object visibility (not shown) includes all objects incident upon the brown polyhedral beams.

The *cell-to-cell* and *cell-to-object* visibility sets are stored in the display database



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

Figure 9.21: Two dimensional schematic diagram of a) cell-to-cell visibility (stipple gray), and b) cell-to-object visibility (shown as solid black squares) for a particular source cell (dark gray).

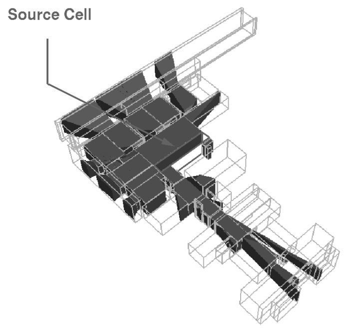


Figure 9.22: Cell visibility in three dimensions. Visible region (beams) and cell-to-cell visibility (outlined) are shown for one source cell.

for each source cell, C, in the form of a *stab list*. A stab list contains an entry for each reached cell, R, consisting of: 1) a reference to R, 2) a set of *halfspaces* bounding the portion of R visible from C, and 3) a set of objects in R visible from C, i.e., that are completely or partially inside the assembled halfspaces. One special case exists: if R is a neighbor of C, all objects are tagged as visible from C without any halfspace or object set computations. During the interactive walkthrough phase, the stab list for the observer's cell is retrieved from the display database and culled dynamically based on the observer's position and view direction to help determine the set of objects to load into memory and to render during each frame (Section 9.4.2).

9.3.3 Results

Mean and maximum precomputation statistics for cells in the spatial subdivision of Soda Hall are shown in Table 9.2. In all, the spatial subdivision contains 5,060 leaf cells, of which 1,889 are possibly inhabitable by an observer – the other 3,171 cells occupy dead space inside the walls and ceilings. Except for a few large cells that have many portals and objects incident upon them (e.g., the cells that span the entire length of the building just outside the windows), the spatial subdivision classifies the object distribution and visibility properties of the building model fairly well.

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

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

9.4 Interactive Walkthrough

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

We use the object hierarchy, spatial subdivision, and results of the visibility precomputation stored in the display database (summarized in Figure 9.23), along with real-time display and database management algorithms to achieve interactive frame rates.

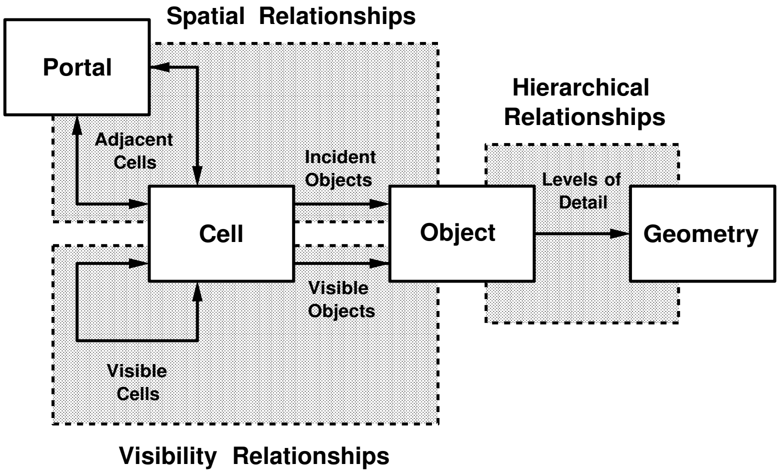


Figure 9.23: Organization of data in the display database.

Execution during the walkthrough phase proceeds as diagrammed in Figure 9.24. In every frame, the system performs seven operations, each of which can run asynchronously in a separate concurrent process in a two-forked pipeline:

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

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.

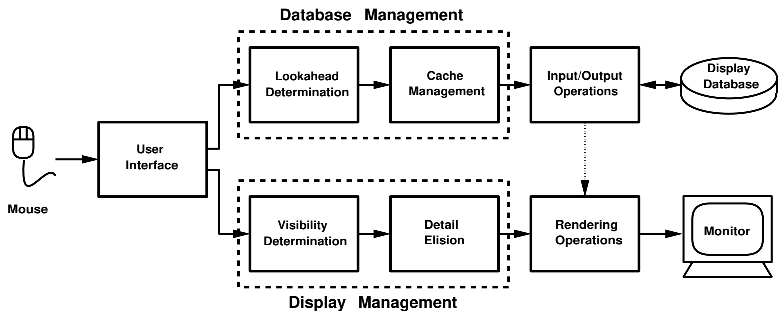


Figure 9.24: Functional operations of the walkthrough phase.

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.

9.4.1 User Interface

During every frame of an interactive walkthrough, the system first determines the simulated observer viewpoint. We represent an observer viewpoint by a *view frustum*, which is specified by an *"eye" position*, a *view direction*, *azimuthal* and *altitudinal half angles*, and an *up vector* (Figure 9.25). This representation is mapped to a perspective viewing transformation [13] for display purposes by setting the center of projection to the observer eye position, the view reference point to the center of the window, and the view plane normal and up vectors to the corresponding frustum parameters (Figure 9.26). The ratio of the tangents of the azimuthal and altitudinal half angles is kept equal to the aspect ratio of the viewport on the graphics workstation display in order to avoid distortion due to anisotropic scaling.

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

The walkthrough program also supports a variety of panels with toggles, buttons, and sliders to control various program options and display modes for visualization of our algorithms (Figure 9.28). For instance, the program allows a user to view the building model from either an "interior view" in which the model is rendered from the simu-

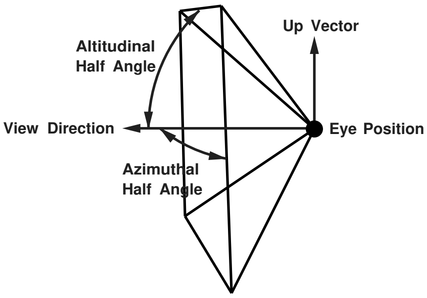


Figure 9.25: View frustum variables.

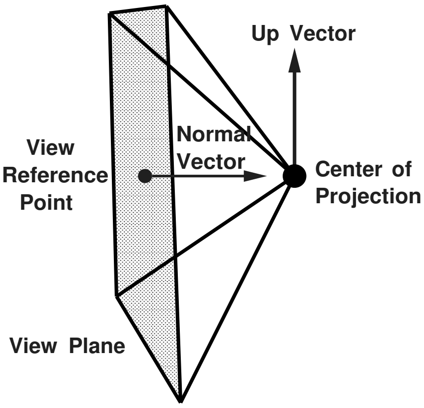
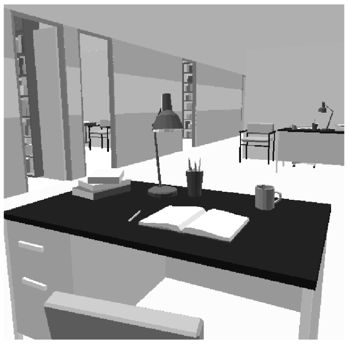
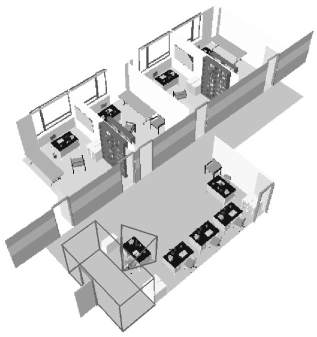


Figure 9.26: Mapping view frustum to perspective projection.

lated observer viewpoint (the default), or from a "bird's eye view" in which the model is rendered as if the user were looking down from the sky (Figure 9.27). In either case, the user can choose to see outlines of cells and objects in various visibility sets, shade objects based on the LOD chosen for display, and/or draw outlines of cells and objects cached in memory. We have found these visualization controls to be essential for debugging and verifying the behavior of complex three dimensional geometric algorithms.



a) Interior View



b) Bird's Eye View

Figure 9.27: Visualization program supports two views.

9.4.2 Display Management

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

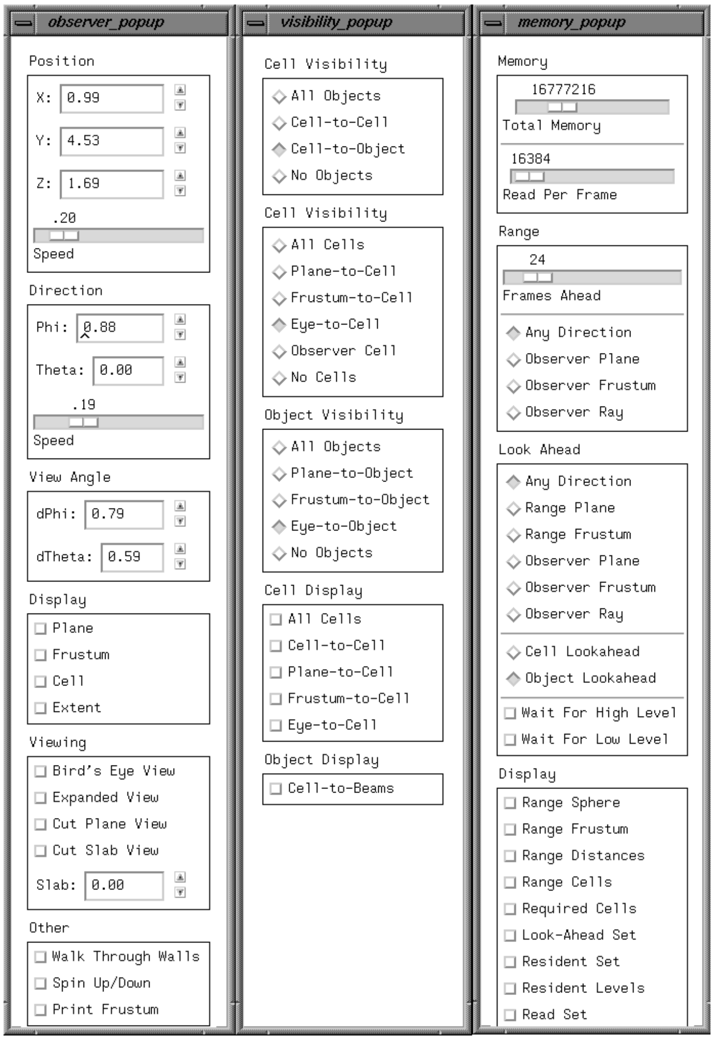


Figure 9.28: Panels used for controlling parameters for a) observer navigation, b) visibility determination, and c) memory management.

server, 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.

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 9.3.2). The difference is that we can compute the visibility for the actual observer viewpoint during the walkthrough phase, whereas we computed visibility for each cell (i.e., a generalized observer free to look in any direction and move to any position within the cell) during the precomputation phase.

Given an observer view frustum, F, we first identify the cell, C, containing the observer position and initialize the *visible region* to be the wedge which is the intersection of the volumes enclosed by F and C. Next, we perform a constrained depth-first traversal of the cell adjacency graph, starting at C and propagating outward through portals to neighboring cells. The region of space visible to a view frustum through a sequence of axial rectangular portals can be well-approximated by a polyhedral wedge with at most ten sides (one for each of four planes bounding the view frustum, plus one for each of six planes derived from portal edges parallel to the x, y, and z axes). For each step through a portal, P, into a reached cell, R, we update the wedge visible through the current portal sequence, W, by intersection with a wedge bounded by planes through the observer eyepoint and edges of the portal (Figure 9.29). If the resulting intersection is empty (i.e., W is disjoint from P, so the new portal sequence does not admit any sightline passing through the observer eye position), that branch of the depth-first search is terminated. Otherwise, the region of R visible to the observer (i.e., the intersection of R and W) is included in the visible region, and the search recurses into neighboring cells. See [35, 37] for more details.

A schematic diagram of the visible region computation is shown in two dimensions in Figure 9.30. The observer view frustum is represented in two dimensions by a wedge outlined by thick black lines, and the visible region is shown in stipple gray.

During the depth-first search, we construct the *eye-to-cell* and *eye-to-object* visibilities for the observer view frustum – i.e., the sets of cells and objects, respectively, that are reached by some sightline that contains the observer view position, lies within the observer view frustum, and does not pierce any opaque cell boundaries. Construction of these sets can be accelerated by taking advantage of the precomputed visibility information stored in the display database since only cells and objects in the precomputed visibility sets of the cell containing the observer must be checked for a feasible sightline in real-time. Note, however, that the real-time visibility computation does not depend on the availability of precomputed visibility information. All objects incident upon cells reached during the real-time visibility search can be checked for a feasible sightline with only slightly extra computation (less than 20%) if a visibility precomputation has not been performed.

Figure 9.31 shows a schematic representation of the eye-to-cell visibility and eye-to-

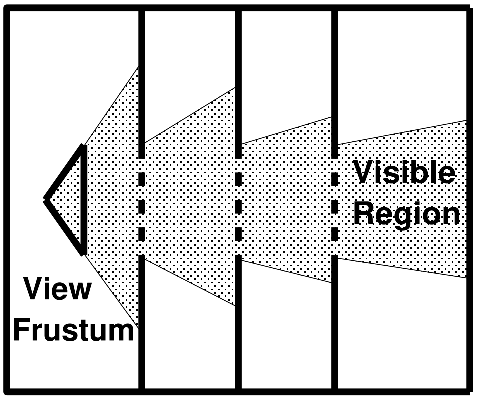


Figure 9.29: Visible region is a wedge that typically narrows as it traverses through more portals.

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 9.32 and 9.33, respectively.

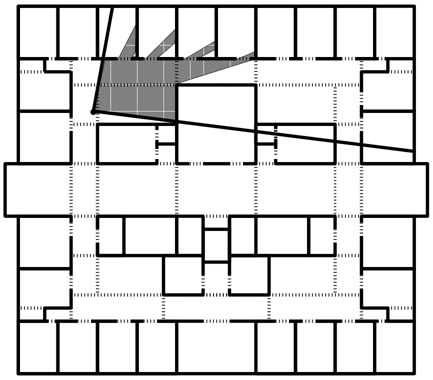


Figure 9.30: Visible region for view frustum in two dimensions.

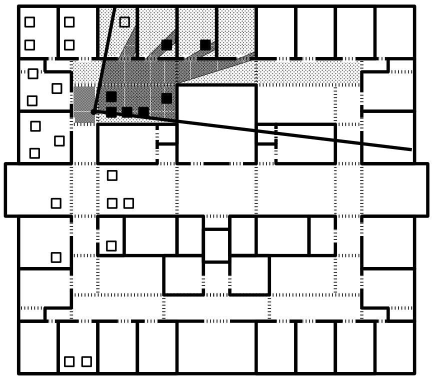


Figure 9.31: Objects in the eye-to-object visibility are shown as by solid squares.

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 reduce rendering times if we render only objects in the eye-to-object visibility during each frame of an interactive walkthrough.

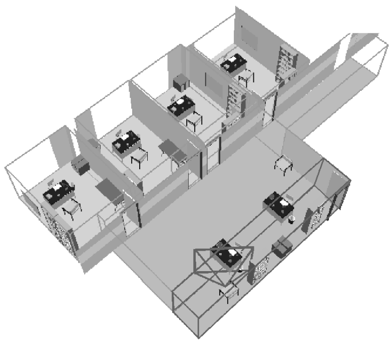


Figure 9.32: All objects incident upon cells in the eye-to-cell visibility.

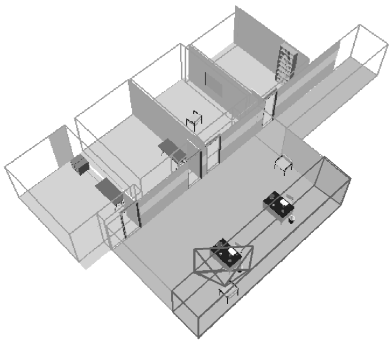


Figure 9.33: Objects in the eye-to-object visibility.

Detail Elision

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

To reduce the number of polygons rendered in each frame, an interactive visualization system can use *detail elision*. If a model can be described by a hierarchical structure of *objects*, each of which is represented at multiple *levels of detail* (LODs) (see Section 9.2.2), simpler representations of an object can be used to improve frame rates and memory utilization during interactive visualization. This technique was first described by Clark [10], and has been used by numerous commercial visualization systems [32]. If different representations for the same object have similar appearances and are blended smoothly, using transparency blending or three dimensional interpolation, transitions between levels of detail are barely noticeable during visualization.

Previously described detail elision techniques [7, 15, 28, 29, 32, 42], use size or distance heuristics (often with feedback control) to choose a LOD for each object rendered. Simpler representations are used for objects that are small or far away from the observer and thus map to very few pixels on the workstation screen. Although size and distance heuristics for LOD selection improve frame rates in many cases, they do not generally produce a *uniform* or *bounded* frame rate, even if the threshold is adjusted dynamically with feedback control. During visualization of very large and discontinuous virtual environments, scene complexity can vary radically between successive frames. For instance, in a building walkthrough, the observer may turn around a corner into a large atrium, or step from an open corridor into a small, enclosed office. In these situations, the number and complexity of the objects visible to the observer may change suddenly. Thus, the size threshold chosen based on the time required to render previous frames is inappropriate, and can result in very poor performance until the system reacts. Overshoot and oscillation can occur as the feedback control system attempts to adjust the size threshold to achieve the target frame rate. These effects can be quite disturbing to a user of an immersive visualization system.

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

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

$$\begin{array}{l} \text{Maximize}: \\ \qquad \sum_S \textit{Benefit}(O, L, R) \\ \text{Subject to:} \\ \qquad \sum_S \textit{Cost}(O, L, R) \leq \textit{Target Frame Time} \end{array} \tag{9.1}$$

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

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

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

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

We use the *Cost* and *Benefit* heuristics to choose a set of object tuples to render each frame by solving the constrained optimization in Equation 9.1. Unfortunately, this constrained optimization problem is NP-complete. It is the Continuous Multiple Choice Knapsack Problem [18, 22], a version of the well-known Knapsack Problem in which elements are partitioned into candidate sets, and at most one element from each candidate set may be placed in the knapsack at once. We have implemented a simple, greedy approximation algorithm for this problem that selects object tuples with the highest *Benefit*/*Cost* [23, 30]. Our implementation finds an approximate solution that is at least half as good as the optimal solution in $O(n \log n)$ for n potentially visible objects. However, since the initial guess for the LOD and rendering algorithm for each object is generated from the previous frame, and there is often a large amount of coherence from frame to frame, the algorithm completes in just a few iterations on average. Moreover, computations are done in parallel with the display of the previous frame on a separate processor in a pipelined architecture and thus do not increase the effective frame rate as long as the time required for computation is not greater than the time required for display. See [16, 17] for more details.

9.4.3 Memory Management

Realistic-looking three dimensional models may be much larger than can fit into main memory. Thus, an interactive walkthrough system must swap portions of the model in and out of memory in real-time as the observer navigates through the model. This is especially difficult since it takes a large amount of time to load data from disk into memory. We must not only store in memory the portion of the model that is visible from the current observer viewpoint, but also we must pre-fetch portions of the model that might become visible in future frames. Consequently, we have implemented a predictive memory management algorithm that forecasts a range of possible observer viewpoints during the next N frames and uses precomputed cell-to-cell and cell-to-object visibility information fetched from the display database to determine a *lookahead set* of objects (i.e., a set of objects that are likely to be visible to the observer during the next N frames). We choose a level of detail at which to store each lookahead object in a *memory resident cache*. Simple cache management algorithms determine which objects to load into memory from disk, and which to replace when the cache is full, as the observer moves through the model.

Pre-Fetch Algorithm

An ideal memory management algorithm predicts the observer viewpoint for each future frame perfectly. It can then use the visibility determination and detail elision algorithms described in Sections 9.4.2 and 9.4.2 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.

In order to pre-fetch objects into memory before they are rendered, we must determine in advance which objects are likely to become visible to the observer. In each frame of an interactive walkthrough, we compute a set of *range cells*, R, that are likely to contain the observer eye position during the next N frames by performing a shortest path search of the cell adjacency graph. The search, implemented using Dijkstra's method [12], adds cells to the range set ordered by the minimum number of frames before the observer can enter the cell. Maximum positional and rotational velocities and constraints preventing observers from walking directly through solid walls are used to determine the number of frames required for the observer to enter a cell. Figure 9.34 shows an example shortest path search result. Each cell is labeled by the minimum number of frames before it can contain the observer, assuming the observer is constrained to the maximum positional and rotational velocities, and cannot walk through walls. For $N = 4$, range cells are highlighted in cross-hatch.

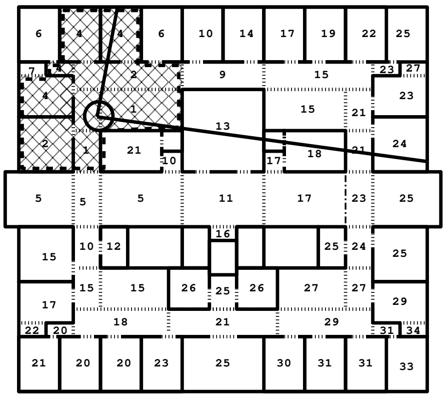


Figure 9.34: The observer range cells (cross-hatch) contain all observer view positions possible during the upcoming $N = 4$ frames. Each cell is labeled by the number of frames before the observer can be resident in it.

We use precomputed cell-to-cell and cell-to-object visibility information for the range cells fetched from the display database to compute a set of *lookahead objects* that are likely to be visible to the observer during the next N frames. When a new range cell is discovered during the shortest path search, we add all potentially visible objects incident upon the cell to the lookahead set. Figure 9.35 shows an example computation of the lookahead set of objects. Each cell is labeled by the minimum number of frames before it can become visible to a cell in the observer range set. For $N = 4$, cells in the range set are highlighted again in cross-hatch, and cells containing objects in the lookahead set are highlighted in stipple gray.

Figure 9.35: The lookahead cells (stipple gray) contain all objects that can be visible to the observer during the upcoming N frames. Each cell is labeled by the number of frames before it can become visible to the observer.

As each object is added to the lookahead set, we mark and claim memory for all LODs for the object that can possibly be rendered during the next N frames. We use a size threshold for static detail elision, along with precomputed information regarding which objects can be drawn at a given LOD for an observer inside a particular cell, to choose a maximum LOD at which to store each potentially visible object. The effect is that objects near the observer range are stored in memory up to higher LODs than ones further away, as shown in Figure 9.36.

The shortest path search for range cells and lookahead objects terminates when either: 1) there are no cells remaining that can contain the observer during the next N frames, or 2) all available memory has been claimed (as long as all objects visible from the current observer viewpoint are in the lookahead set). In either case, the set of lookahead objects is certainly a superset of the objects visible from the current observer viewpoint, as well as a good estimate of the objects that are most likely to be rendered in upcoming frames. See [17] for more details.

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

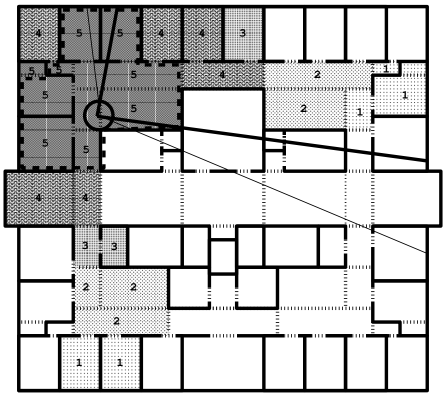


Figure 9.36: Lookahead objects are stored in memory only up to the LOD at which they can possibly be rendered during the next N frames. Each cell is labeled and shaded by the maximum level of detail any object incident upon it is stored in memory – darker shades of gray represent higher levels of detail.

memory (i.e., the size of the physical memory of the workstation minus the amount reserved for the spatial subdivision and precomputed visibility information).

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

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

Figure 9.37 shows results of the cache management algorithm for a particular observer path. Each cell is labeled by the number of frames since objects incident upon it were included in the lookahead set. The shade of each cell indicates whether or not it contains objects in the memory resident cache (stipple gray), read set (SE/NW-hatch), or release set (SW/NE-hatch).

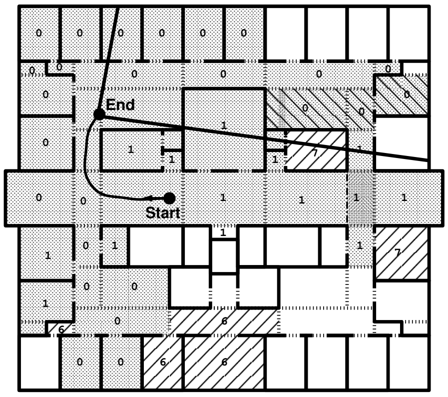


Figure 9.37: Cache management algorithm results. Each cell is labeled by the number of frames since objects incident upon it were included in the lookahead set. Shading for each cell indicates whether or not it contains objects in the memory resident cache (stipple gray), read set (left-hatch), and resident set (right-hatch).

Asynchronous Pre-Fetch

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

In our first implementation, the walkthrough system stalled when it found that an object to be rendered had not yet been loaded into memory at the appropriate LOD. It simply waited until the appropriate LOD for an object was loaded into memory, and then it continued rendering. Needless to say, this behavior was extremely bothersome.

At times, the system would stall for several seconds waiting for a particular object geometry that was rendered for only a few frames.

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

9.5 Results

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

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

In both experiments, we used the observer path shown in Figure 9.38. It contains over 9,000 observer viewpoints, and visits the top four floors of Soda Hall.

9.5.1 Display Management

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

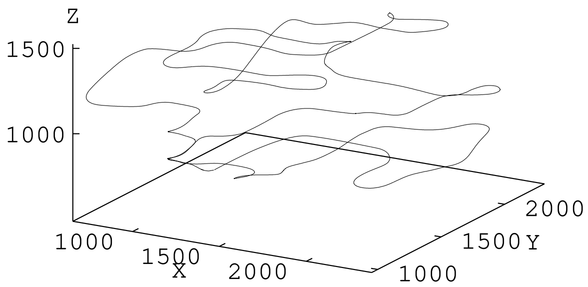


Figure 9.38: Test observer path through the top four floors of Soda Hall.

- **Entire Model:** Renders every object in the model at its highest level of detail – i.e., without any visibility determination or detail elision.

- **Visibility:** Renders each object in the eye-to-object visibility of observer viewpoint at its highest level of detail.

- **Detail Elision:** Render each object in the eye-to-object visibility of observer viewpoint with the level of detail chosen by the optimization detail elision algorithm to bound the frame rate at a minimum of fifteen frames per second.

During each test, we measured the compute time (i.e., the real-time visibility determination and detail elision compute times), the rendering time, and the frame time (i.e., total time between successive frames), as well as the numbers of cells, objects, and polygons rendered in each frame. Mean and maximum statistics for all observer viewpoints along the test walkthrough path are shown for each combination of visibility determination and detail elision algorithms in Tables 9.3–9.4. Figure 9.39 contains a plot of the frame time for each observer viewpoint along the test path during the test using both visibility and detail elision.

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

During the *Visibility* test, in which we used the eye-to-object real-time visibility determination algorithm without detail elision, we rendered 5,002 polygons (0.35% of the model) on average, and the mean frame time was 0.067 seconds (14.9 frames per second). However, the maximum frame time was still quite large using only visibility determination because there are some viewpoints along the test path at which the scene visible to the observer is very complex (469 objects containing 45,828 polygons at the highest level of detail). At these viewpoints, the rendering time increased to 0.557 seconds (1.8 frames per second).

During the *Detail Elision* test, in which we used both the eye-to-object visibility algorithm and the optimization detail elision algorithm to choose an appropriate LOD for

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

Table 9.3: Mean and maximum compute time, rendering time, and frame time statistics collected during display management tests.

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

Table 9.4: Mean and maximum numbers of cells, objects, and polygons rendered during display management tests.

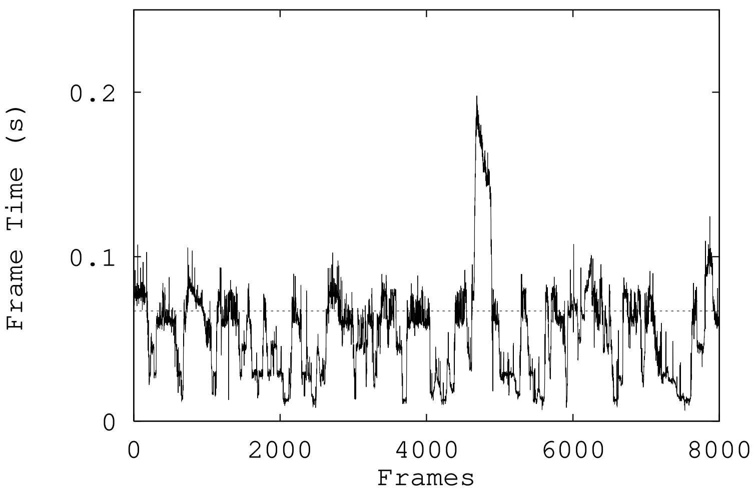


Figure 9.39: Frame time for each observer viewpoint along the entire test observer path during the test using both visibility determination and detail elision.

each potentially visible object dynamically, we rendered only 2,534 polygons (0.18% of the model) and generated an image every 0.047 seconds (21.3 frames per second) on average. The maximum rendering time was 0.083 seconds, and the the maximum response time was 0.274 seconds – short enough for the system to maintain a highly interactive feel. A plot of rendering times measured during the test with detail elision are shown in Figure 9.41. Rendering time is bounded by the detail elision algorithm and the observer step size is adjusted by the previous frame time resulting in a smooth, interactive walkthrough with constant velocity. Although these bounded rendering times were achieved by rendering simpler representations for some objects during some frames, the visual effects of detail elision were barely noticeable (see Figure 9.40 for an example).

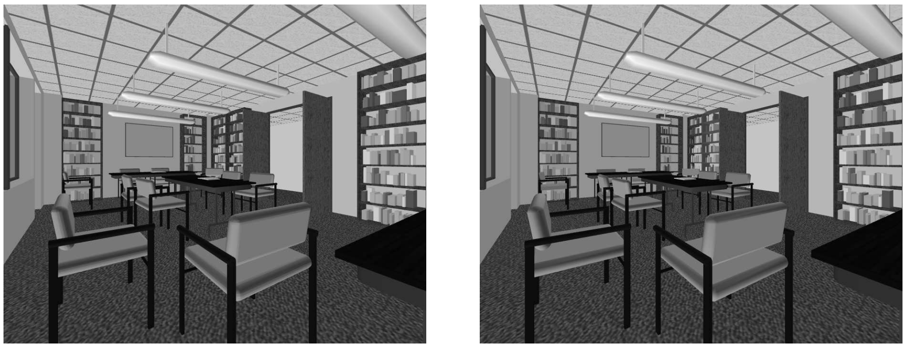


Figure 9.40: Images of library generated using (left) no detail elision (19,821 polygons), and (right) using the *Optimization* detail elision algorithm with a target frame time of 0.05 seconds (3,568 polygons).

There is one peak in the frame time during the detail elision test (near frame 4700 in Figure 9.39). This peak occurred because the computation time was larger than the rendering time during this portion of the walkthrough test. Figure 9.42 shows the measured compute time, and rendering time for each frame along the portion of the walkthrough path for which computation was most expensive. Since the walkthrough system was configured as a two process concurrent pipeline in this test, with visibility determination and detail elision computations performed in one process and rendering in another, the effective frame time was closely correlated with the maximum of the compute time and the rendering time. The process performing visibility and detail elision computations slowed to 0.199 seconds per frame in the worst case, and caused the effective frame rate to slow to 5 frames per second for a short period, even though the rendering time was still less than 0.8 seconds. However, note that the frame times would have been far greater if either the visibility determination or detail elision computation were not performed. There is currently no adaptive control over the time required for computation in our system. In a future version, perhaps computations can be optimized, executed with more parallelism, or be made adaptive so that they never exceed the target frame time.

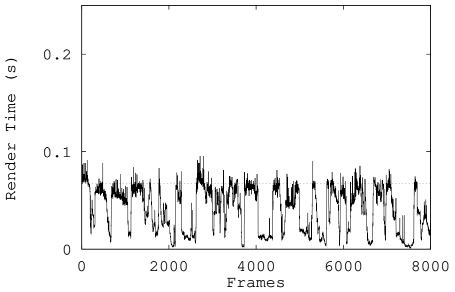


Figure 9.41: Rendering times during *Detail Elision* test for each observer viewpoint along the test observer path.

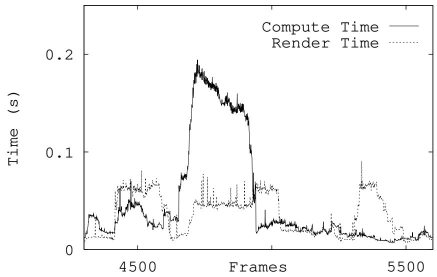


Figure 9.42: Compute time and rendering time during *Detail Elision* test for each observer viewpoint along the portion of test observer path where computation was most expensive.

9.5.2 Memory Management

In the second experiment, we tested the cumulative effectiveness of the display and memory management algorithms described in the previous sections. We used the same display management algorithms as were used in the test labeled *Detail Elision* in the previous test. However, in this experiment, we used the flattened model of Soda Hall (349.5MB of storage) and the predictive memory management algorithm described in Section 9.4.3 to swap objects in and out of memory in real-time as the observer moved along the test walkthrough path. To test the algorithms to their fullest, we artificially limited the size of the memory resident cache to 16MB.

In addition to the statistics measured during the display management experiment, we gathered statistics regarding the the swap time (i.e., the time required to load objects into memory from disk), and the sizes of resident, lookahead, rendered, and read sets. Mean and maximum statistics for all observer viewpoints along the test walkthrough path are shown for tests with and without memory management in Table 9.5. Table 9.6 contains mean and maximum statistics regarding the lookahead sets collected during the memory management test. Figure 9.43 contains a plot of the frame time for each observer viewpoint along the test path during the test using memory management.

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

Of the 349.5MB of data in the entire model, only 382KB of data was required to describe the polygons rendered during each frame on average. The lookahead algorithm was able to determine a relatively small set of object descriptions to store in memory (9.916MB on average) by predicting which object descriptions were most likely to be

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

Table 9.5: Mean and maximum compute time, rendering time, swap time, and frame time statistics collected during tests with and without memory management.

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

Table 9.6: Mean and maximum set statistics collected during memory management tests.

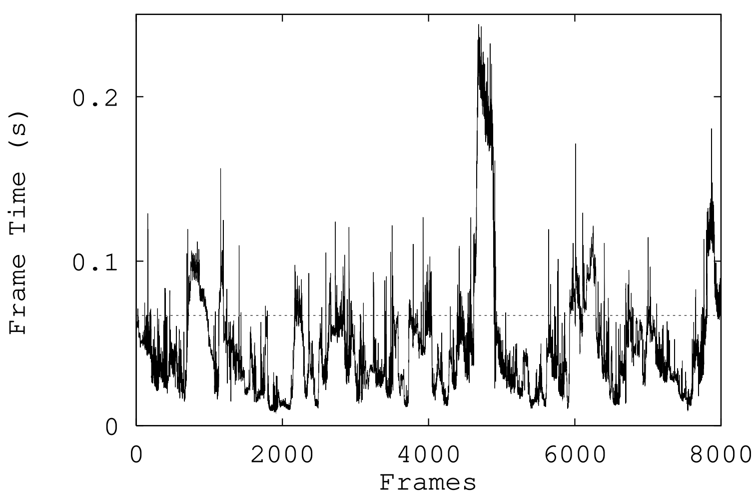


Figure 9.43: Frame time for each observer viewpoint along the entire test observer path during the test with memory management.

rendered during upcoming frames. The lookahead computation required very little extra time (0.002 seconds per frame on average), and executed in a separate asynchronous process, and thus had very little impact on the effective frame time.

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

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

9.6 Conclusion

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

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

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

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

We have found it possible to swap objects in and out of memory as a simulated observer walks through a model much larger than physical memory. We use an asynchronous 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 frames. An approximate least-recently used algorithm determines which objects to replace in the memory resident cache as new objects are added.

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

9.7 Acknowledgements

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

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Lecture F

Wrap-up and Future Directions

Eric L. Brechner
The Boeing Company
P.O. Box 24346, M.S. 7L-43, Seattle, WA 98124-0346
brechner@boeing.com[1]

Section 10

Slides

[1]Eric Brechner is now at Microsoft Corporation, One Microsoft Way, Redmond, WA 98052-6399, *ericbrec@microsoft.com*

The Boeing Company
Research and Technology

Wrap−up and Future Directions

Eric L. Brechner
May 12, 1995

What we think we know

The Boeing Company
Research and Technology

- Control of movement through a scene
- Maintenance of near−constant frame rates
- Choice of appropriate level of detail
- Optimization of graphics performance for drawn objects
- View frustum culling
- Basic techniques for database pre−fetching

What we are still investigating

The Boeing Company
Research and Technology

- Occlusion culling (visibility)
- Object simplification
- The impact of different culling and simplification techniques on database management

What are the limits?

The Boeing Company
Research and Technology

- A VR version of the Boeing 777 running at 60 frames per second would require handling over 30 billion polygons per second.
- Although you can never predict what the future may hold, it's a safe bet the operative word will be "MORE."
- Is there a limit? What about an infinite database?
- Considering an infinite database, like the one we view through our eyes, can point to key issues and possible solutions.

Four key concepts

The Boeing Company
Research and Technology

1. Displaying an infinite database is not as bad as it sounds – both the output device (the screen) and the receiving device (your eye) have limited resolution.
2. The key to displaying an infinite database is to never draw anything in it, at least not exactly.
3. The approximation you use for your data should primarily be concerned with the light emitted by the objects.
4. The approximation should be different when viewed on a scale of meters, as opposed to light–years, as opposed to angstroms.

Choosing an approximation

The Boeing Company
Research and Technology

1. It should draw quickly on current hardware.
2. It should have simple computations for occlusion, ray and object intersections, containment, and extreme values.
3. It should approximate the light emanating from a volume of space, as opposed to a surface.
4. It should be exact to the original in the limit.

Convex polyhedra

The Boeing Company
Research and Technology

1. They are made of polygons (fast to draw on current hardware).
2. They have all kinds of great properties for fast computation of goodies (many based on the Simplex method).
3. They are solids.
4. They represent a linear approximation of the original, which can be exact in the limit for piecewise smooth volumes.

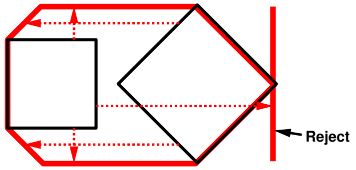
The full approximation

The Boeing Company
Research and Technology

- Create convex polyhedra from your initial input geometry using either 3D convex decomposition or 3D mesh generation.
- Using some standard clustering algorithm, build a bounding volume hierarchy where each convex polyhedron in the hierarchy contains its children.
- Compute the cluster polyhedra by sliding boundary faces in the cluster until they bound the entire cluster, removing redundant faces.

How a cluster is approximated

The Boeing Company
Research and Technology



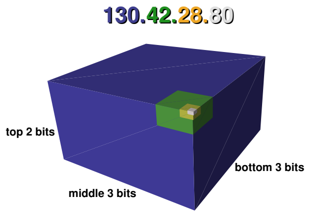
The ultimate 3D Web browser

The Boeing Company
Research and Technology

- **Create a 3D environment where the geometry represents locations on the Web.**
- **Use the Internet address for geometric location.**
- **Use a special VRML file stored at the site for geometry or use a default 3D icon based on the domain name.**
- **Place the contents of a site inside its geometry laid out based on a 3D directory tree structure, a nested directory tree structure, or a 3D network based on internal site links.**
- **Depict documents as bulletin boards or book–like objects.**

More on location

The Boeing Company
Research and Technology



Traveling through the Web

The Boeing Company
Research and Technology

- **Fly freely through space.**
- **Move along links depicted as highways.**
- **Jump across links.**
- **Take along your friends, depicted as personalized VRML figures.**

Do it today using the technology in this course!

Section 11

Wrap-up and Future Directions

As with most interesting problems, the walkthrough problem has some areas that are better understood than others. Although new advances are always possible, the areas of controlling movement through a scene, maintaining near-constant frame rates, choosing an appropriate level of detail, optimizing graphics performance for drawn objects, view frustum culling, and the basic techniques for database pre-fetching are well understood and have generally available algorithms that address them.

However, many aspects of occlusion culling (visibility), object simplification, and the impact of different culling and simplification techniques on database management are still undergoing a great deal of research. Much of this research is concerned with handling databases with several orders of magnitude more data then those in current use. How much data can we expect the walkthrough system of the future to handle? Is there a breaking point?

11.1 The infinite database problem

What if we consider the problem of displaying a database with an infinite amount of data? Could we achieve interactive rates? There is reason to believe we could.

11.1.1 Four key concepts

There are four key concepts to solving the infinite database problem:

1. Although there is an infinite amount of input data, there is a finite amount of output data.
2. If your database is infinite, you cannot draw anything in it exactly.
3. The approximation you use for your data should approximate the light emitted by the objects, not the objects themselves.
4. The approximation should be a function of scale.

First, even with an infinite database of polygons, the output is still going to be a screen with limited resolution. A typical 1280x1024 screen with 24 bits of color only holds about 4MB of data per frame, about 9MB if you include a 24 bit Z-buffer and an 8 bit alpha channel. So no matter how big your data starts out, it ends up being a size that most systems can process with ease. Even if you had an infinite resolution display device, your eyes still have limited resolution and bandwidth. Thus there is some hope that if you can pick out the right 4MB for each frame, you can walk through an infinite database.

Second, assuming the database has a somewhat uniform density, that is, there are no objects made of only a finite number of polygons, then it is not possible to draw all the polygons in any one object. If you could, then you could draw the entire database exactly since each object is no less infinite than the database itself. Instead, some approximation to each object must be drawn; no object in the database can be drawn exactly.

Third, for display purposes, the approximation of an object must only look like the object, since it cannot possibly be the object. How an object looks, excluding any occluding objects, is entirely dependent on the light emitted by the object. Therefore, the approximation of an object must be primarily concerned with approximating the light emitted by that object. Approximating the shape of the object is important as well to account for light occlusion (shadows) and object occlusion, but this is secondary to the impact of the visible light which the object reflects, transmits, or creates.

Finally, the approximation should be different when viewed on a scale of meters, as opposed to light-years, as opposed to angstroms. This usually implies that the approximation is hierarchical in nature. That could mean a hierarchy of wavelet transforms, of bounding regions, of topological simplifications, of texture maps, of fractal definitions, etc. The important point is that the appropriate approximation is chosen for the scale being displayed. This is very similar to choosing the level of detail, except that you first choose the appropriate approximation before choosing the level of detail at which to draw that approximation. For some approximations, there is only one level of detail for any given scale, but that may not be true of all types of approximations.

11.1.2 Choosing the right approximation

Given the large variety of approximations available, how do you choose the right one to best describe the light emanating from an object? Arguably, the choice of approximation should be based on the following set of criteria:

1. it should be able to be drawn quickly on current hardware;
2. it should have simple computations for occlusion, ray and object intersections, containment, and extreme values;
3. it should approximate the light emanating from a volume of space, as opposed to the light emanating from a surface;
4. it should approach the original data exactly as the scale goes to zero.

The first item is just a practicality. The purpose of the approximation is to reduce the amount of time needed to draw a complex scene. Therefore, you would certainly want your approximation to draw quickly. For most current hardware, that means the approximation should be representable as polygons. Naturally, if you can acquire special purpose hardware to draw some other form of approximation quickly, then this is not a real constraint.

The second item is also practical, but does not necessarily involve hardware. Computing occlusion, ray and object intersections, containment, and extreme values are very common operations in walkthrough. For high frame rates, it will be necessary to be able to do these computations quickly for your approximations.

The third item is more subtle. An infinite geometric database will likely contain an infinite number of objects. Even approximating the entire surface of an object with a single surface approximation will leave you with an infinite number of approximations. Clearly groups of objects will need to be approximated. But as soon as you consider a group of objects, you are approximating a volume of space, not a surface. By choosing an approximation that represents volumes, you have some hope of reducing the problem.

The final item refers to the consistency of the approximation. Because you can never draw the exact contents of the database, but you can get as close as you wish to that data, it follows that as you approach the limit of scale, the approximation should approach the exact data. If your approximation does not converge on the true data, then there is no hope of freely moving through different length scales and seeing consistent results.

11.1.3 A possible approximation choice

One approximation that conforms to the above criteria is a convex polyhedron. Its boundary is made of polygons so it is fast to draw. Because it is convex the various common computations can be done quickly, many using the Simplex Method.[1] It is a solid, so it can be thought of as approximating a volume. And finally, if used as a piecewise linear approximation to a volume function, it should be able to approximate piecewise smooth volumes to arbitrary accuracy.

The approximation hierarchy would be a bounding volume hierarchy where each convex polyhedron in the hierarchy contains its children. The hierarchy could be built using any robust clustering algorithm, starting with some initial set of polyhedra. The initial set of polyhedra could be computed from 3D convex decomposition or 3D mesh generation of some initial input geometry. Finally, the cluster polyhedral approximations could be generated by sliding the boundary faces of polyhedra in the cluster until they contain all the points in the cluster, then removing redundant faces (see figure 11.1).

The reason why you might choose to slide boundary faces in the cluster out until they bound the entire cluster, instead of simply computing a convex hull, is that you want to approximate the light emanating from the cluster, not just the shape. The light characteristics are in large measure derived from the normals of the faces, therefore, you would want to preserve those normals. By sliding the existing boundary faces, you

[1] George B. Dantzig, *Linear Programming and Extensions*. Princeton, New Jersey: Princeton University Press, 1963.

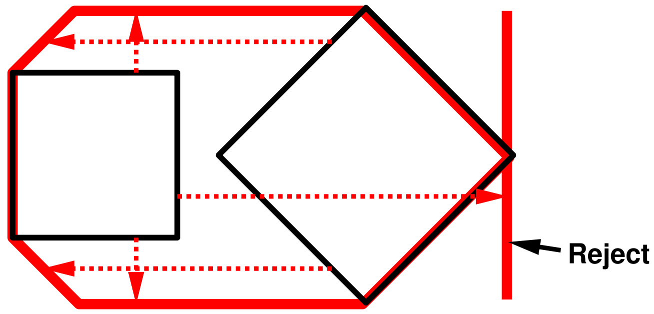


Figure 11.1: Approximating a cluster of polygons.

accomplish just that.

This approach is under current investigation at Boeing for use in producing a full airplane walkthrough and immersive virtual reality environment. Initial results are promising, but several issues remain unresolved. In addition, more accurate approximations of the light emitted by a cluster in all directions are still being investigated.

11.2 The ultimate 3D Web browser

One possible application for an infinite database walkthrough system is for visually traversing the World Wide Web.[2] An algorithm could be devised for locating Web sites in the virtual environment based on Internet address (e.g., figure 11.2). The geometry for the Web site could be determined by a default based on the domain name or a special VRML[3] file stored at the Web site. The contents of a Web site could be placed inside its geometric representation and laid out based on a 3D directory tree structure, a nested directory tree structure, or a 3D network based on the internal site links. Documents themselves would be displayed on bulletin boards or book-like objects.

To travel through the environment, you could either fly freely through space or move along links depicted as highways using fast jumps when desired. Browsers could communicate with each other to allow multiple participants to browse together, seeing each other depicted as either some default geometry or as a personalized VRML figure.

The largest problem in producing such a browser would be the pre-fetching of data, which can be quite slow. However, techniques currently used in existing 2D browsers, like placeholders, could be applied to 3D browsers as well.

There is no apparent reason why a 3D Web browser could not be written to run on mid-range graphics workstations today, using existing technology. Much of the technology necessary has been collected here in these notes.

[2] *http://www.w3.org/hypertext/WWW/TheProject.html*

[3] Virtual Reality Modeling Language (*http://www.eit.com/vrml/*)

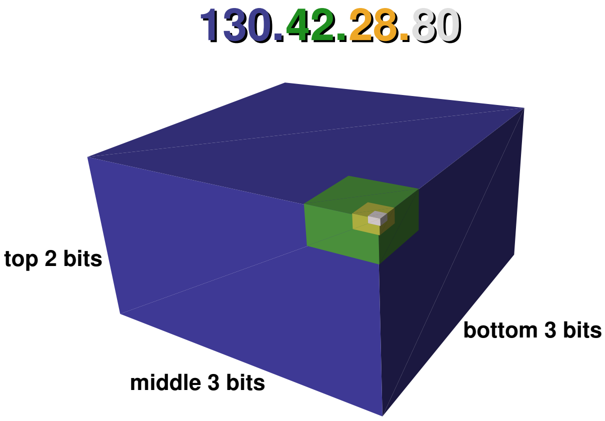


Figure 11.2: A possible location scheme based on Internet address.