Visibility Computations
in Densely Occluded Polyhedral Environments

by

Seth Jared Teller

B.A. (Wesleyan University) 1985
M.S. (University of California at Berkeley) 1990

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

Committee in charge:

Professor Carlo H. Séquin, Chair
Professor Raimund Seidel
Professor Jean Pierre Protzen

1992

The dissertation of Seth Jared Teller is approved:

Carlo H. Séquin Oct. 20, 92
Chair Date

Kevin A. Seidel 92/10/24
Date

10/21/92
Date

University of California at Berkeley

1992

Visibility Computations

in Densely Occluded Polyhedral Environments

by

Seth Jared Teller

Visibility Computations in Densely Occluded Polyhedral Environments

by

Seth Jared Teller

Carlo H. Séquin

Carlo H. Séquin
Thesis Chair

Abstract

This thesis investigates the extent to which precomputation and storage of visibility information can be utilized to accelerate on-line culling and rendering during an interactive visual simulation of a densely occluded geometric model.

Architectural walkthroughs and other visual simulation applications demand enormously powerful graphics hardware to achieve interactive frame rates. Standard computer graphics rendering schemes waste much computational effort processing objects that are not visible to the simulated observer. An alternative is to precompute *superset visibility information* about the model, by determining what portions of the model will definitely be invisible for an observer in certain locations. This information can then be used during the simulation phase to dramatically reduce the number of model entities that must be processed during each frame time.

The *visibility precomputation* phase first *subdivides* the model into *cells* by partitioning the space embedding the model along the planes of large opaque polygonal *occluders*, such as walls, floors, and ceilings. The remainder of the geometric data, for example furniture and wall trim, are considered to be non-occluding *detail objects*. For each cell, a coarse visibility determination is first made as to what other cells might be visible from it. The detail objects are then inserted into the subdivision, and a finer-grain visibility determination is made for these objects and stored with each cell.

The *on-line culling* phase dynamically tracks the position and field of view of the simulated observer through the cells of the spatial subdivision. The precomputed visibility information is subjected to further on-line culling operations that use the observer's exact position and field of view. The resulting reduced set of objects is issued to graphics hardware, where a discrete depth-buffer solves the hidden-surface problem in screen space.

The visibility framework is defined generally in terms of *conforming* spatial subdivisions that support a small number of abstract operations. All visibility determinations are proven to produce a *superset* of the objects actually visible to the observer. This is crucial, since omitting any visible object would cause an erroneous display. The generally small set of invisible objects produced by the on-line culling operation is then removed by the graphics rendering hardware.

We implemented these abstract notions for several interesting and realistic input classes, i.e., axial and non-axial scenes in two and three dimensions. We evaluated the usefulness of the precomputation and culling scheme using objective metrics of culling effectiveness, pixel depth complexity, and on-line culling and rendering time. The test data was a complex, three-dimensional architectural model comprising ten thousand detail objects and almost three-quarters of a million polygons. On-line frame times decreased from about ten seconds for the unprocessed model, to a tenth of a second, thus accelerating frame rates by a factor of about one hundred.

In memory of Carolyn Hayes and Professor René de Vogelaere

Contents

List of Figures

List of Tables

Acknowledgments

My advisor at Berkeley, Carlo Séquin, and my mentor at Silicon Graphics, Jim Winget, have made my experience as a graduate student thoroughly enjoyable. Carlo's inexhaustible curiosity and energy have been a great inspiration throughout my time in Berkeley. Jim's constant encouragement and positive challenges have spurred my work when it otherwise might have lagged or charted an easier course. Jim also saw to it that I was treated as "family" at Silicon Graphics, for which I am very grateful.

Raimund Seidel shared many pearls of geometric wisdom with me, starting with his wonderful course on computational geometry. Although I did enter Berkeley with the express intention of studying computer graphics, it was Raimund's skill, encouragement, and infectious enthusiasm that spurred my interest in computational geometry and eventually led to this thesis combining work in both fields. Raimund also became a member of my thesis committee, contributing many helpful comments even while he was on sabbatical.

Several other professors, at Berkeley and elsewhere, shared their time and experience with me on too many occasions to list. Jean Pierre Protzen, Dean of the College of Environmental Design, sat on my thesis committee. The late René de Vogelaere taught me some crucially important classical geometry, and managed to teach me not only facts and theorems but some new ways to think about them, which I have since found useful on a daily basis. Jim Demmel seemed always to be available for questions on the details of implementing numerical algorithms, and indeed, his advice made possible a robust implementation of the line-stabbing algorithm in Chapter 6. Vel Kahan willingly educated me about matters ranging far wider than numerical computation, and calmed my apprehension about re-entering Evans hall after the Loma Prieta earthquake. Pat Hanrahan and Leo Guibas were unceasingly generous with their time and encouragement.

Berkeley is a great resource not only of faculty but of students. I was fortunate to know many of these students both as friends and colleagues. Michael Hohmeyer was always willing to listen to my half-baked ideas. Much of the prose in §5.3.1 was co-authored with him in [HT91]. He has also been a constant and valuable friend in too many other ways to mention. Henry Moreton showed me the ropes as a first-year graduate student. Eric Enderton and I forged a bond while completing our Master's degrees. Since then, he has often called from work (i.e., the real world) late in the evening to point out that neither of us had yet eaten dinner, and to suggest a remedy. Ziv Gigus has been a good friend and a caring and constructive critic.

Berkeley has a great graphics gang. Tom Funkhouser, Delnaz Khorramabadi, Ajay Sreekanth,

Dan Rice, Thurman Brown, Laura Downs, Rick Braumoeller, Maryann Simmons, and Priscilla Shih, have all been friends and enthusiastic contributors to the walkthrough project.

The theory students have also been valuable to know (not least because they have nice offices with windows and comfortable couches). Jim Ruppert willingly batted around research ideas and gently educated me about theoretical issues in computational geometry. Nina Amenta constantly exhorted me to keep in mind the value of science. Dana Randall shared her insight both inside and outside of school. Ashu Rege and Will Evans clarified several technical points, and were also good to have (respectively) on the softball diamond and the ultimate frisbee field.

Randi Weinstein taught me about marine biology and tide-pooling. Annalisa Rava, at times Wesleyan cohort, Berkeley housemate, and Santa Cruz denizen, kept me in touch with a world removed from graduate school. Oliver Grillmeyer and I shared many late-night adventures. Finally, it was my great fortune while at Berkeley to witness the start of a social institution, the Hillegass House. Steve Lucco, Ken Shirriff, John Hartman, Ramon Caceres, and Will Evans were overwhelming in their generosity of spirit, and in their unmatched hospitality.

Silicon Graphics has been a tremendously important part of my time here. My first summer internship at SGI quickly led to friendship with Efi Fogel. Paul Haeberli often helped me with figure and video preparation; he is also the creator of Figure 6.7. Melissa Anderson made me feel remembered and included. Forest Baskett generously supported my doctoral research with both materiél and frequent encouragement. John Airey, Mark Segal, and Derrick Burns were each role models in their distinct style.

It is difficult to describe to anyone unfamiliar with the arduous bureaucracy of Berkeley how essential it is to have the assistance of capable friends. In this I most gratefully thank Kathryn Crabtree, Liza Gabato, Terry Lessard-Smith, Bob Miller, Jean Root, and Teddy Diaz, all of whom tackled several hopelessly snarled situations on my behalf.

Finally, I am grateful to my family: to my grandmother, to my parents, to my brothers Adam and David, to my cousin Joshua (also a student here), and to all of the rest of the clan, for their love and encouragement.

Chapter 1

Introduction

1.1 Motivation

Suppose one wishes to simulate, on a generic graphics workstation, the visual experience of navigating through a complex synthetic environment, for example a large building. This simulation is to be both realistic and fast; that is, successive scenes rendered on the workstation monitor should reflect a consistent, physically sensible representation of the environment, and should appear in rapid succession, ten or more times per second.

Both of these goals are at odds with the basic fact that, if these synthetic environments are to be interesting and complex, then the representational *model* of the environment will consequently comprise a very large number of individual elements, and require a large amount of storage. Such models are so complex that they cannot fit in the workstation's memory, and have so many individual pieces that they cannot be rendered at sufficiently fast frame rates[1].

However, many models (e.g., architectural models) are *densely occluded*; that is, only a small portion of the model is *visible* from the point of view of an observer inside. Clearly, when simulating the inside observer's point of view, only this latter portion need be drawn in order to produce the correct scene. Any invisible elements would be, by definition, eventually clipped away or obscured. If this visible portion could be rapidly identified, the need to process every element of the model every frame would be obviated.

Given that environmental simulation requires substantial computation and rendering resources, it is worthwhile to investigate the extent to which expenditure of storage and precomputation can

[1]This statement can reasonably be made in general, since, no matter what memory or rendering resources are available, growing user expectations will lead to geometric models whose complexity outstrips these resources.

accelerate visual simulation rates. This thesis introduces general, robust, and effective techniques to precompute *superset* visibility information, that is, to expend computational resources before the simulation phase in order to accelerate the determination of more detailed visibility information during the simulation phase.

Precomputing the *exact* set of visible elements for every viewpoint and view direction in and around a three-dimensional model is a task both conceptually and computationally daunting. Widely available graphics hardware effectively solves the "hidden surface" problem [Ake89, KV90]. That is, given a viewpoint, a field of view, and a collection of opaque elements, the hardware resolves any occlusion among the elements in a discretized space and displays them in perspective from the specified viewpoint[2]. Given such graphics hardware, we can therefore make the critical observation that *from any viewpoint, rendering any superset of the elements visible from that viewpoint will produce a correct scene.* The approach developed in this thesis relies on this observation, in that it computes only a conservative superset of visible objects.

1.2 The Basic Approach

We show that superset visibility determination is a considerably more tractable problem than an exact visibility computation. We introduce an *abstract* computational model consisting of *input* and two computational phases. The input is a collection of major occluders and some set of detail objects, each associated with a spatial extent (i.e., a bounding box). The first phase, *visibility preprocessing, spatially subdivides* the geometric model into chunks typically separated by occluders. Each chunk is then treated as a *virtual light source*; any entities reached by a light ray are potentially visible to an observer in the chunk; entities not reached by light are definitely not visible to the observer. In this way, coarse and fine visibility information is *precomputed* among the chunks, and the spatial subdivision is *annotated* with this information. The second, *dynamic*, phase tracks a moving observer inside the model and quickly determines a superset of the model elements visible to that observer by further *on-line culling* of the precomputed, annotated visibility data. Here, "on-line" means that the culling operation must generate a set of visible entities in a *frame time* of one hundred milliseconds or less, as each actual observer position is registered.

[2]We assume that rendering artifacts due to hardware sampling are unimportant.

1.2.1 Spatial Subdivision

We assume that the model representation (and therefore the creator of the model) distinguishes between occluders and objects (Figure 1.1), and that all occluders for the geometric model in consideration can be simultaneously memory-resident. (In practice, abandoning this assumption requires sophisticated virtual-memory techniques, but no conceptual changes at the level of the visibility computations.) The space occupied by the model is then partitioned, via a *spatial subdivision*, into *cells* that are of limited extent compared to the entire model. The subdivision terminates when every major occluder lies on the boundary of one or more spatial cells (Figure 1.2).

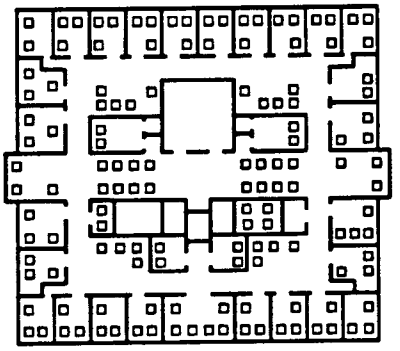


Figure 1.1: Major occluders (bold) and detail object bounding boxes (squares).

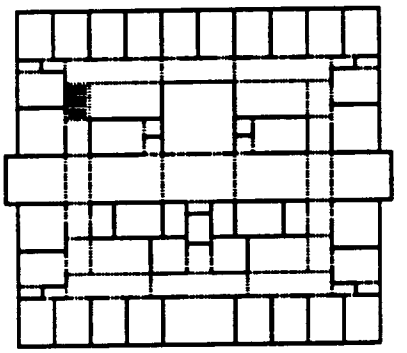


Figure 1.2: A spatial subdivision. Portals are shown as dashed lines.

A *portal enumeration* stage exploits the fact that subdivision planes (lines, in 2D) are induced along the major occluders present in the model. Each plane contributes to the planar boundary of one or more spatial cells; each such boundary may be partially or completely obscured by occluders. The set complement of each boundary and its coaffine occluders is then computed, and *portals* are explicitly constructed wherever any cell shares a transparent boundary with an immediate neighbor cell. Each portal stores an identifier for the cell to which the portal leads. We say that a cell complex consisting of convex cells and explicit portals is a *conforming* spatial subdivision.

Both the subdividing process and the resulting subdivision can be treated as an abstraction, subject to a few reasonable requirements about data organization. Broadly, the purpose of spatial subdivision is to partition the model into "chunks" which can then be examined individually to determine their occlusion properties. This chunking is advantageous for two reasons. In the large, spatial subdivision makes manageable the amount of data to be considered at one time, and imposes a global sorting and partitioning on the model data. In the small, the subdivision imposes a spatial hierarchy on the components of the model; determinations made about chunks can be applied to each

chunk component in an efficient manner. In practice, chunking makes sense for real architectural models because it makes explicit the intuitive difference between large-scale or "structural" model elements (i.e., occluding walls, ceilings, floors) and small-scale or "detail" model elements (i.e., phones, cups, lamps). Chunking also puts explicit partitions between regions that are intuitively distinct; for example, between rooms, and between separate floors of a building.

1.2.2 Visibility Precomputation

Intuitively, visual interaction between cells will in general be limited, since intervening occluders will tend to obscure the space in one cell from the viewpoint of any observer in the space in another cell. The transparent portions of shared cell boundaries are portals (cf. Figure 1.2). We define a *generalized observer* as an observer constrained to a given cell (the *source* cell), but free to move anywhere inside this cell and to look in any direction (one source cell is shown in grey in Figure 1.2). A generalized observer may see into a neighbor cell only through a portal; and into a more distant cell only through a *portal sequence*. These sequences typically impose significant constraints upon the generalized observer's visibility, preventing the observer, for example, from seeing the entirety of any cell reached through a general portal sequence.

The occluders and subdivision uniquely determine a *cell-to-cell visibility* relation, in which two cells are linked only if there exists a *sightline*, or line segment disjoint from any occluders, connecting the cell boundaries (Figure 1.3). Thus, the conforming spatial subdivision gives a local character to the problem, reducing it to one of computing the (typically limited) interactions of each small portion of the model with only those elements that possibly visually interact with the portion. We say that this visibility information is *static*, since it has no dependence on time or on the precise position of the observer.

Each detail object is assumed to have an associated bounding representation (e.g., an axial bounding box). After spatial subdivision, the subdivision cells are *populated* with detail objects, by associating with each cell those objects whose bounding representations are spatially incident upon the cell. Detail objects, which typically constitute the great majority of model data, are spatially associated with cells, and visibility information associated with cells, as a *local* operation. As in cell-to-cell annotation, *cell-to-object visibility* is established when sightlines are found to exist between generalized observers and detail object bounding boxes in distant cells (Figure 1.4). This is again static information, depending only on the positions of occluders, object bounding volumes, and the particular spatial subdivision. The cell-to-object annotation follows spatial subdivision and

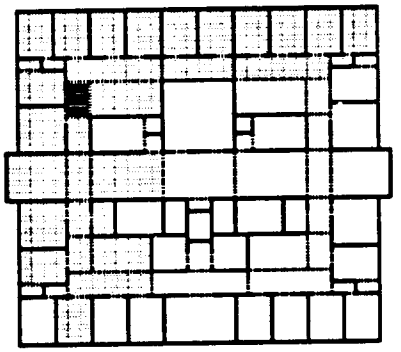


Figure 1.3: The cell-to-cell visibility set of the gray source cell.

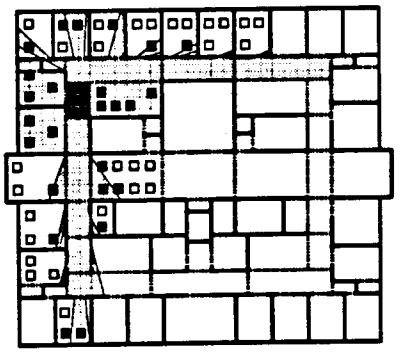


Figure 1.4: The cell-to-object visibility set (filled squares) of the gray source cell.

cell-to-object visibility determination, and completes the preprocessing phase.

Visibility precomputation can itself be considered in two stages, *gross* and *fine* culling. Gross culling simply derives a single bit of information about each cell pair in the model: whether the two cells are mutually visible, or equivalently, whether a generalized observer in one cell can see some point in the other. Fine culling, on the other hand, establishes a more complex relationship among cells and objects; for example, that an observer in a particular cell can or cannot potentially see a particular object, polygon, or even point in another cell. Clearly gross and fine culling can be structured *hierarchically*; if the generalized observer cannot see into a particular cell, there is no need to examine the objects in that cell from the point of view of the generalized observer. Gross and fine static culling are done for each cell in the model, so that any subsequent dynamic observer position can be processed correctly.

1.2.3 Dynamic Phase

The preprocessing phase posited a *generalized observer* whose position is never precisely known. In contrast, the *dynamic phase* tracks the instantaneous position of an *actual observer*, i.e., one with a known position and field of view. In this phase, scenes are continuously synthesized from the virtual point of view of the actual observer and displayed at interactive frame rates (typically ten to thirty times each second), using a graphics workstation. Static visibility information is exploited in the dynamic phase. An *on-line culling* operation using the actual observer's position and field of view, and the annotated spatial subdivision, efficiently determines a superset of that portion of the model that must be rendered to guarantee the correctness of the synthesized scene.

The dynamic phase can itself be partitioned into three stages: *point-location*, *on-line culling*,

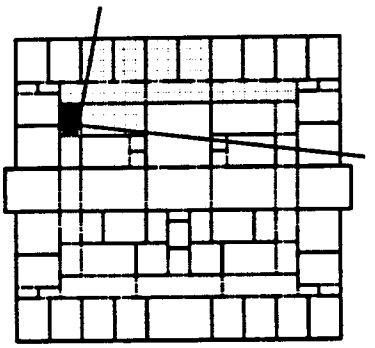


Figure 1.5: The eye-to-cell visibility (darkened cells) of the actual observer.

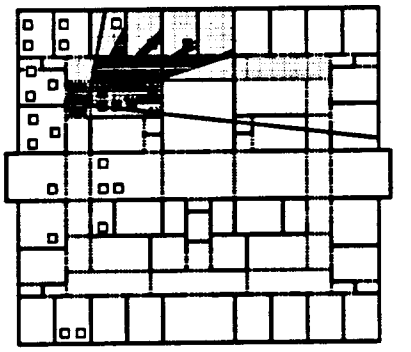


Figure 1.6: The eye-to-object visibility set (filled squares) of the actual observer.

and *rendering*. Point-location is the determination of the spatial cell enclosing the observer's position. The high coherence of walkthrough paths implies that most point-location queries yield the same result as the previous query. On-line culling involves the retrieval of the static visibility data for the cell containing the observer, and selection from among this superset data using precise knowledge of the observer's position and field of view. Finally, the rendering component is simply the dispatch of any potentially visible objects (e.g., polygons) to the graphics hardware for display. The goal, of course, is to produce a sufficiently small upper bound on visibility that the resulting set of polygons can be rapidly displayed, and to compute this set in time comparable to that required to display a frame.

First, the *eye-to-cell visibility* set contains a superset of those cells visible to the observer (Figure 1.5) and is clearly a *subset* of the source's cell-to-cell visibility. Next, the *eye-to-object visibility* set contains a superset of those objects, in the previously determined cells, to which a sightline exists from the eye (Figure 1.6).

As we will show, the abstract visibility operations we define are valid over *any* conforming spatial subdivision. However, the operations will be efficient and practical only when the spatial subdivision has reasonable storage complexity with respect to the number of occluders. We will seek to construct such subdivisions with various dimension- and occluder-dependent *splitting criteria*, defined individually in later sections. The overall goal of these criteria is to produce cells whose boundaries are mostly opaque, so that visibility is highly constrained for an observer in such cells. A secondary goal is to produce cells that are not too large, and that have aspect ratios reasonably close to one.

1.3 Theoretical Concerns

This thesis addresses both the *theoretical* issue of determining superset visibility, and the *engineering* task of using such techniques to achieve working visual simulations of realistic architectural models. We show that our theoretical approach is provably correct, in that it never misclassifies a truly visible object[3], regardless of the type of geometric model.

Superset visibility determination is sufficient as a culling process for visual simulation. The theoretical and engineering challenge is to produce superset visibility bounds that can be computed efficiently, and are usefully *tight*; i.e., not much bigger than the set of truly visible objects. We therefore require objective metrics with which to evaluate the effectiveness of the precomputation and on-line culling scheme. We have measured both the spatial and computational aspects of these techniques. For instance, one measure of the expense of visibility preprocessing is the storage cost incurred, expressed as overhead on the storage required for the geometric model alone. Another measure analyzes the culling effectiveness of the static and dynamic phases; what fraction of the model is deemed potentially visible, on average, to an observer following a real path through the model, and what percentage *survives*, i.e., contributes to the rendered image? Of course, these measures are dependent on the actual geometric model, and in some cases on the path of the simulated observer. Another measure analyzes, in a real engineering walkthrough system, the overall rendering speedup attributable to the use of these visibility techniques. These metrics are discussed further in Chapter 9.

1.4 Engineering Observations and Assumptions

This thesis is concerned with environments for which visibility preprocessing is promising. We have observed that the visibility within many typical architectural environments is substantially limited; i.e., that from most points inside, only a small portion of the environment is visible. In the case of geometric models of such environments, our algorithms make the critical assumption that the number of polygons truly visible from most points in the model is about the number of polygons that may be rendered at interactive frame rates on a state-of-the-art graphics workstation. That is, even though the model may be enormously complex (containing, say, a million polygons), only a small fraction of these (say, ten thousand) need to be rendered for most viewpoints. We say that models with this property are *densely occluded*, and argue that, to be simulated smoothly, the

[3]By "truly visible," we mean that a line segment from the eye to some point on the polygon intersects no other polygon.

model must have a roughly constant visual complexity, commensurate with the speed of available hardware.

Another observation is that architectural environments are typically comprised of two kinds of entities: large, simple, structural elements (i.e., walls, floors, beams, and ceilings) that generally cause substantial occlusion, and small, complex things (i.e., desks, chairs, and clutter) that generally do not occlude much from a wide range of viewpoints. We call the first type of entity *major occluders* or simply *occluders*, and the second type *detail objects* or simply *objects*. We assume that occluders and objects are distinguishable from each other in the input to the visibility algorithms. This makes sense, since an efficient modeling system would represent occluders and objects distinctly (for example, by representing walls with a few large polygons rather than hundreds of small ones). Therefore, it is worthwhile to segregate occluders and detail objects, and consider the occluders' effects upon visibility before the effects due to detail objects, since (to first approximation) these latter effects will typically be subtler and more costly to compute .

Detail objects are generally complex geometric assemblages that obscure little from most vantage points. For this reason, we treat detail objects as entirely non-occluding. As we shall show, this assumption will never force the visibility computations to produce incorrect (i.e., subset) query results; however, it may slightly increase the size of the potentially visible polygon sets for any particular region of viewpoints. For other environments, this assumption is a bad one, and most occlusion will in fact arise from the combined effects of many detail objects, e.g., the leaves in a forest.

The distinction between occluders and objects is not well-defined, nor do we attempt to make it so. A refrigerator, for example, may occlude very much for nearby viewpoints, but most small motions of the observer can change the set of occluded entities substantially. A wall, on the other hand, typically meets the floor, ceiling, and other walls along shared edges; an observer might have to move a considerable distance to substantively change the occlusion experienced due to a particular wall. We show that, when occluders decrease in size or number, our algorithms degrade gracefully to perform efficient, purely spatial culling. Moreover, our algorithms are independent of object complexity in that they representing objects only by bounding volumes that can be subjected to various geometric culling operations. Our visual simulation implementation does represent detail objects at several levels of complexity, in order that they be drawn more efficiently when their area contribution to the rendered image is small [FST92]. However, since the bounding volumes do not depend on object complexity, we consider the notion of levels of detail to be independent of the issues of visibility computation discussed in this thesis.

1.5 Practical Issues

In practice, there are several other issues that must be addressed by a walkthrough system. Foremost of these is *robustness*; the geometric algorithms used must perform correctly, even for the sometimes highly-degenerate input encountered in the real world. Fortunately, the fact that computations are of *superset* visibility information makes the engineering task easier. When boundary cases occur for which determining the potential visibility of an entity is numerically difficult (for example, an object seen only through an epsilon-wide slit), our visibility algorithms consistently choose "false-positive" outcomes over "false-negative." That is, only a small penalty is incurred if an entity is misclassified as visible: it is rendered, and painted away by depth-buffering hardware. On the other hand, misclassifying an object as invisible may incur a large penalty: the user of the walkthrough simulation is presented with an incorrect scene, without the misclassified entity. Such false-negative errors detract from realism and visual coherence, and our implementation strives to avoid them. We also briefly discuss in Chapter 9 a programming and visualization technique that facilitated the development of very robust implementations of geometric algorithms.

A second issue is that all rendering is *discretized* to a collection of pixels on a readily available graphics workstation. Display-device resolution is limited, and (at least at present) much less than the resolution of human vision. Level-of-detail analysis is desirable to avoid the display of tiny or far-away objects with unwarranted detail, since most of this detail will map to only a few pixels on the screen. The visibility module should therefore support queries usable to bound the level of detail at which an object need be displayed. One such query might, for example, return the largest solid angle an object can subtend when viewed from any point in a (spatially disjoint) cell.

Finally, practical visual simulation systems must support *prediction* of dynamic storage requirements [FST92]. Most interesting and realistic models will be too large to fit at once into the main memory of a typical workstation. But any data, to be displayed, must be memory-resident. The visibility module queries are helpful in predicting the memory demands that will occur in rendering the future views of a moving observer. For example, given bounds on the observer's position and velocity over some time interval, a neighborhood containing all possible positions of the observer in that interval can be computed. The regions visible from this neighborhood would implicate the objects that might have to become memory-resident in order to render all objects visible to the observer during the specified time interval.

1.6 Organization

The thesis is organized as follows. After a review of some relevant prior work (Chapter 2), we introduce an abstract framework in which storage and precomputation can be expended to accelerate subsequent on-line visibility determinations (Chapter 3). We then reify the framework, with specific, dimensionally-dependent techniques of ordering spatial data, precomputing visibility, and on-line culling. Specific techniques are discussed for three interesting classes of input occluders: axial (i.e., axis-aligned) and generally-oriented 2D line segments (Chapter 4); axial 3D rectangles (Chapter 5); and generally-oriented convex polygons in 3D (Chapter 8). The general 3D case is sufficiently complex to warrant a special formalism to deal with the geometry of stabbing lines through arbitrarily oriented polygons. We therefore introduce Plücker coordinates in Chapter 6 as a convenient method of manipulating skew lines in 3D. Then, in Chapter 7, we present an algorithm that computes the boundaries of regions in the model that are illuminated by area light sources.

We have developed novel computational geometry algorithms that are practically realizable (i.e., implementable), robust, and usefully applicable to real data. In practice, we have observed that many architectural models are predominantly composed of axial rectangles. For this class of occluders we show that the visibility techniques are practical, and we describe our implementation of a system for visual simulation of building walkthroughs. This system achieves significant rendering speedups over existing methods when applied to very complex three-dimensional geometric models (Chapter 9). Our test case has been the geometric model of a planned computer science building at Berkeley, a seven-floor structure with an atrium, terraced balconies, scores of hallways, hundreds of rooms, thousands of textures, ten thousand detail objects, and three-quarters of a million polygons [Kho91, FST92]. We also describe a research implementation of the visibility techniques for general polyhedral environments.

Spatial subdivisions, visibility computations, and their applications to real problems in computational geometry and computer graphics are rich, fascinating areas of study. The final component of this thesis is an evaluation of our contribution to these efforts, and several indications of fruitful directions that related research may assume in the future.

Specifically, §10.5 discusses open issues concerning the time- and storage-complexity of spatial subdivision construction and visibility algorithms, and the prospect of generalizing these algorithms to capture very fine-grain visibility effects. Several unresolved questions regarding coherence and parallelism are discussed. Lastly, §10.6 sketches applications of these subdivision and visibility techniques to problems such as meshing and form-factor computation in radiosity, and rendering

environments containing shadowed, reflective, and translucent surfaces.

Chapter 2

Previous Work

There is a tremendous amount of literature germane to visibility determination. In order to reduce the amount of material surveyed, we consider an existing work relevant only if it substantively meets at least one of the following criteria, or is otherwise pedagogically or historically important:

One. *The work must address visibility determination from a region, such as a line segment, area, or volume.* This criterion excludes works that, for example, address solving the hidden-surface problem from a single specified point.

Two. *The work must perform some sort of accelerating precomputation,* that is, expend computational and storage resources before any determination of point visibility. This criterion excludes algorithms that, for example, must examine every polygon in a scene to describe each rendered frame.

Three. *The work must describe visibility queries that compute areas or volumes, or collections of polygons.* This criterion excludes most works that, for example, discuss acceleration schemes for ray-casting queries that return a point on the first object hit.

2.1 Point Visibility Queries – Visibility as Sorting

One of the earliest visibility algorithms was due to Jones, who in 1971 described a visibility scheme based on spatial subdivision [Jon71]. After spatially subdividing a model, by hand, into convex cells, a point-visibility query is made by projecting cell openings, or *portals*, onto the view plane and proceeding recursively through the spatial subdivision adjacency graph. As each new portal is encountered, it is intersected with the "mask" or aggregate convex region currently visible (Figure 2.1). If the intersection is empty, the active branch of the search terminates. Otherwise, the

contents of the current cell are clipped to the active mask and drawn, and the search proceeds with the new more restricted mask. Thus Jones' approach also solves the hidden-line, hidden-surface problem for a polyhedral model, with the restriction that *every* face in the model be assigned to the boundary of some cell.

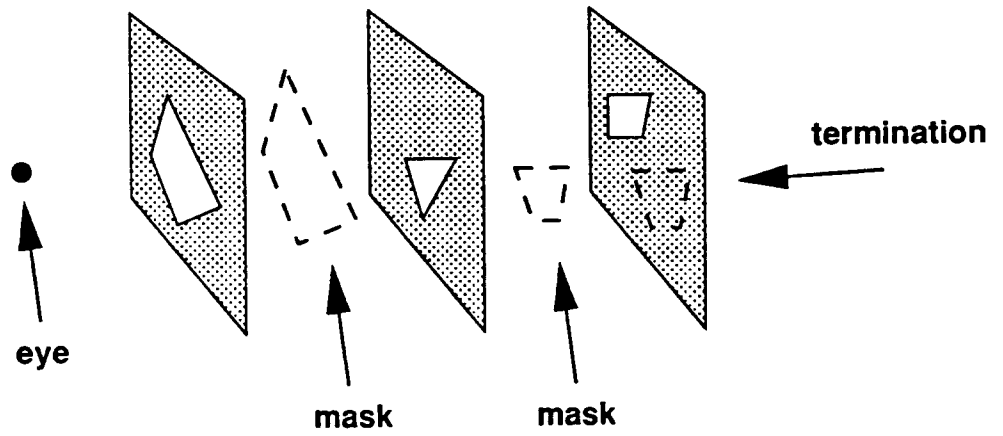


Figure 2.1: Jones' hidden-surface method. The depth-first search of the cell adjacency graph terminates when the current mask has no intersection with the next portal (at right).

This approach is roughly equivalent to our dynamic *eye-to-cell* visibility computation. Our method is different, and more efficient, in several significant ways (to be discussed in §8.4).

The binary space partition (BSP) tree data structure [FKN80] obviates the hidden surface computation by producing a back-to-front ordering of polygons from any query viewpoint. This technique has the disadvantage that every polygon must lie in a splitting plane and, for an n-polygon scene, the splitting operations required to construct the BSP tree generate $O(n^2)$ new polygons in the worst case [PY90]. Moreover, all polygons in the scene must be explicitly processed to generate each rendered frame.

An algorithm based on linear separability of *clusters* of occluders precomputes "face priorities" within clusters, and dynamically determines "cluster priorities" using a BSP-tree like decision tree and linear separators for the clusters [SBGS69]. The face priorities within a cluster are shown to be independent of viewpoint, after backface removal. Thus, after preprocessing, dynamic cluster-priority determination is sufficient to output polygons in appropriate painting order. This algorithm capitalizes on coherence of faces within clusters; however, it cannot handle interpenetrating (i.e., linearly inseparable) clusters and expends storage to determine face orderings, which are less important in these days of ubiquitous hardware hidden surface removal.

Another early hidden surface removal algorithm uses spatial subdivision in z (distance from the observer) to accelerate clipping computations [WA77]. Polygons are sorted into slabs of fixed depth range, and the contents of each slab are sorted into depth order and subjected to a hidden-surface computation based on generalized polygon clipping, under orthographic projection. The resulting polygon masks are then combined to form a final view. This method touches every polygon to generate each rendered frame, and moreover must solve the generalized polygon clipping problem, which is difficult to do robustly [SSS74].

Fixed-grid and octree spatial subdivisions [FI85, Gla84], and subdivision techniques based on ray directionality [AK87, Arv88], accelerate ray-traced rendering by efficiently answering queries about rays propagating through ordered sets of parallelepipedal cells. The advantage of these schemes is that they proceed "forward" along the query ray, terminating at the first polygon hit. Thus, a large parallel query engine might serve, for example, as a means to solve the visible-surface problem independently at each pixel. These ray-based techniques are not yet efficient enough to generate rendered frames at interactive rates.

2.2 Region Visibility Queries – Visibility as Light Propagation

The polygon mask, BSP tree, ray-propagation, and depth-range subdivision-based visibility schemes support only *point* visibility queries; that is, they effectively compute ordered sets of polygons or polygon fragments visible from a point, after preprocessing. (In the mask and BSP algorithms, the ordered set produced is the same for any observer position within a cell.) Since the set of viewpoints from which queries will be made is generally not known in advance, it is useful to compute visible sets from *regions* of points (e.g., line segments, areas, or volumes), where the visibility from a region is simply the union of all points or objects visible from any point in the region. We employ the notion of the generalized observer as a virtual light source to compute this union set efficiently.

In two dimensions, describing shadows cast by an area light source demands that interactions between pairs of occluders be examined. That is, two occluders can jointly hide a point from the light source (i.e., cast it in umbra) when neither occluder alone hides the point. Figure 2.2 depicts a single lineal light source and two line-segment occluders **A** and **B** in two dimensions. The dotted lines demarcate the umbrae of the individual occluders. The point **p** in the figure is hidden from

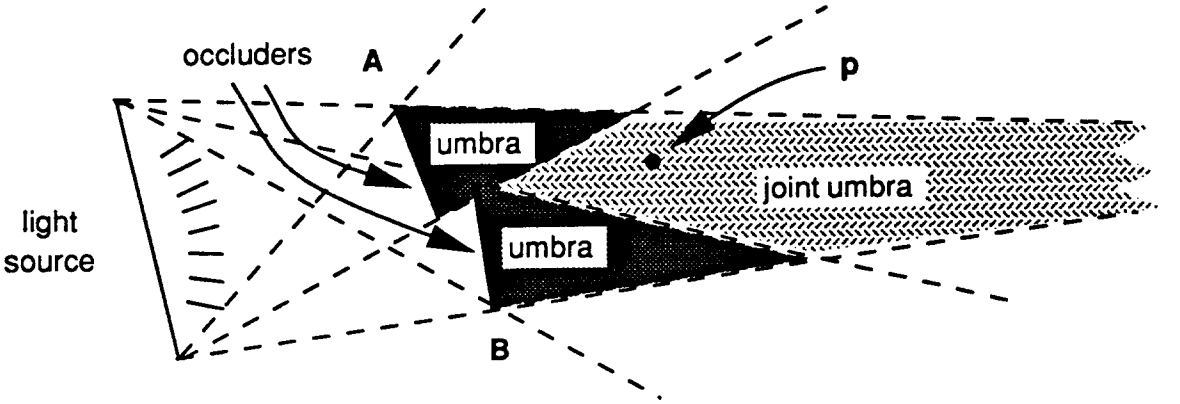


Figure 2.2: In two dimensions, a pair of occluders can jointly hide points from the light source that neither occluder hides alone.

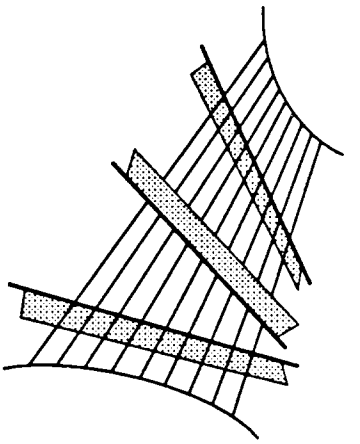


Figure 2.3: In three dimensions, three mutually skew occluder edges can generate a quadric shadow boundary.

light source, yet not in either of the individual occluders' umbrae. It is easy to see that two occluders **A** and **B** interact only if **A** is in **B**'s penumbra (i.e., if some points on **A** see only some of the light source, due to occlusion by **B**), or vice-versa. In two dimensions, the umbra and penumbra have piecewise-linear boundaries and can be computed straightforwardly.

Analogous pairwise and threeway interactions arise in three dimensions, yielding a non-linear result. In the presence of two or more polygonal occluders, computing the volume illuminated by an area light source involves *reguli*, ruled quadric surfaces of negative Gaussian curvature [Som59], whose three generator lines arise from non-adjacent occluder or light source edges (Figure 2.3). Reguli were first used in the context of occlusion for the *aspect graph* computation, which catalogues all qualitatively distinct line-drawing views of a polyhedral object under orthographic or perspective projection [Kv79, PD90].

Imagine viewing a polyhedral object from a sphere- or cube-shaped *view surface* surrounding the object. A particular *aspect*, or line drawing of the object's edges with hidden lines removed, corresponds to each point on the view surface (where the view direction is chosen so as to intersect some fixed point inside the object). Since each intervening occluder edge "clips" a halfplane away from the visible portion of the object, the observer in general sees a polygonal region of the object (Figure 2.4-i depicts one such region, shown as convex for simplicity). The region edges arise directly from occluder edges. The region vertices arise either from occluder vertices, or from apparent intersections among non-adjacent occluder edges as seen by the observer.

Most motion on the surface produces only *quantitative* changes in the line drawing, as vertices shift position and edges shorten or lengthen. However, at some critical loci, called *event surfaces* [GCS91], the line drawing, and therefore the visibility of some component of the object, changes *qualitatively*. Generically, this happens in one of two fundamental ways for polyhedral objects [GM90]. Along a VE or *vertex-edge* event surface, an occluder vertex appears (disappears) from the region boundary (Figure 2.4-ii). Along a EEE or *triple-edge* event surface, the observer's line of sight simultaneously intersects three occluder edges, causing the appearance (disappearance) of an apparent boundary vertex (Figure 2.4-iii). These event surfaces partition the view surface into regions of constant aspect, bounded by segments of lines and conics.

At present, the best time bound for computing the aspect graph of a general polyhedral object with n vertices is $O(n^4 \lg n + m \lg m + c_t)$, where m is the number of qualitatively distinct views, at worst $O(n^6)$, and c_t the total number of changes between these views [GCS91].

For a convex lineal or areal source and a *single* convex occluder in 3D, the umbra and its union with the penumbra are convex. Such first-order shadows have been employed to yield

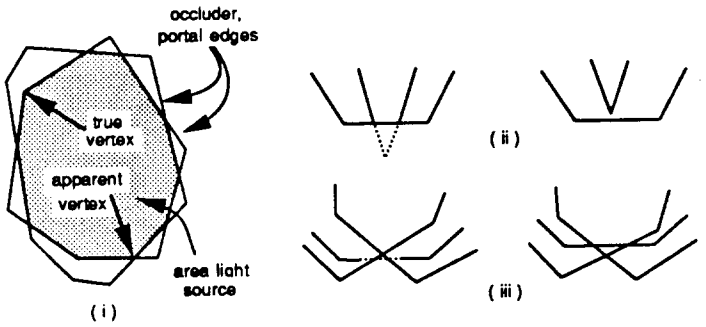


Figure 2.4: An aspect of a polyhedral object in the presence of polyhedral occluders (i). The aspect can change qualitatively in only two fundamental ways, along VE (ii) or EEE (iii) surfaces.

convincing renderings of shadowed scenes [NN83, NN85]. These penumbra algorithms, however, extend to multiple occluders only by effectively approximating the light source as a point or as a set of points. Several algorithms approximating multiple-occluder shadow boundaries have been described [NN85, PW88, CF90, CF92].

For example, in [NN85], the penumbra cast by multiple occluders is approximated by casting each occluder's penumbra individually, then performing polyhedral union and intersection operations on the result. An analogous approach is described in [CF90], where the light source is treated as a discrete set of point sources, and the shadows of collections of occluders are cast and combined. An algorithm proposed in [PW88] replaces the area light source with a point at its center, and describes an error metric that bounds the spatial discrepancy between the computed and true penumbra. This error metric can then be used to control adaptive subdivision of the light source or occluders. Another recent algorithm approximates umbra volumes by constructing "penumbra trees" and "umbra trees"; these are augmented BSP trees whose polyhedral leaf cells bound polygon fragments in partial or complete shadow [CF92].

The notion of *transversals*, or simultaneous intersections by a k-flat, of sets of convex objects has been a popular topic among computational geometry researchers [Ede85, KLZ85, PW89]. Here, we are concerned with transversals of line segments (in 2D) or polygons (in 3D) by $1-$flats, also known as *stabbing* lines. Several researchers have investigated the problem of stabbing a three-dimensional collection of *unoriented* polygons, i.e., polygons that admit a stabbing line in either direction. For a given set of polygons, let e be the total number of edges comprising the set. Avis

and Wenger presented an $O(e^4 \lg e)$ time algorithm to compute stabbing lines [AW87]. McKenna and O'Rourke improved this to $O(e^4\alpha(e))$ time [MO88], where $\alpha(e)$ is the functional inverse of Ackermann's function. If the polygons are triangles, and together comprise g distinct normals, an algorithm due to Pellegrini computes a stabbing line in $O(g^2e^2 \lg e)$ time if one exists [Pel90a]. When g is $O(e)$, this time bound is the same as that due to Avis and Wenger.

A series of algorithms have been proposed that answer questions involving line segment query regions and polyhedral terrains (i.e., height-fields) [DFP+86]. For example, one might ask for the height of the shortest line segment that must be erected over such a terrain so that the top endpoint can see all of its faces. This height is computable in $O(n\lg^2 n)$ time by a simple algorithm [Sha87]. Interestingly, computing the smallest number of points *on* the terrain which together can see all of the terrain is an NP-hard task [CS86]. In any event, the restriction that the input describe a height field is too severe for these techniques to have substantial application here.

Several algorithms generate potentially visible sets (PVS) of polygons with a dynamic query, then solve the hidden-surface problem for this set with hardware, in screen-space. One such approach involves intersecting a view cone with an octree-based spatial subdivision of the input [GBW90]. Although this method provably generates a superset of the visible polygons, it has the undesirable property that it can report as visible an arbitrarily large part of the model when, in fact, only a tiny portion can be seen (Figure 2.5). The algorithm may also have poor average case behavior for scenes with high average depth complexity, i.e., with many viewpoints for which a large number of overlapping polygons paint the same screen pixel(s).

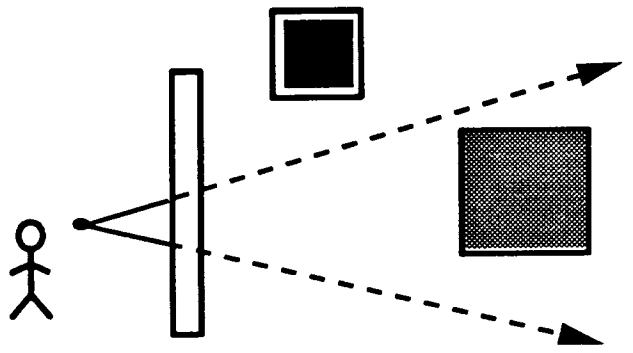


Figure 2.5: Octree-based culling [GBW90]. The contents of the black cell are correctly marked invisible. The contents of the gray cell are marked potentially visible and subsequently rendered, even though the cell is entirely occluded by the foreground rectangle.

Another hardware-based method estimates visibility using *discrete sampling*, after spatial

subdivision and portal-finding. Conceptually, rays are cast outward from a stochastic, finite point set on the boundary of each spatial cell. Polygons hit by the rays are included in the PVS for that cell [Air90]. This approach can *underestimate* the cell's PVS by failing to report some visible polygons (Figure 2.6). Consequently, the algorithm required excessive computation time (several 1989 CPU-months) to produce acceptably tight lower bounds on potentially visible sets.

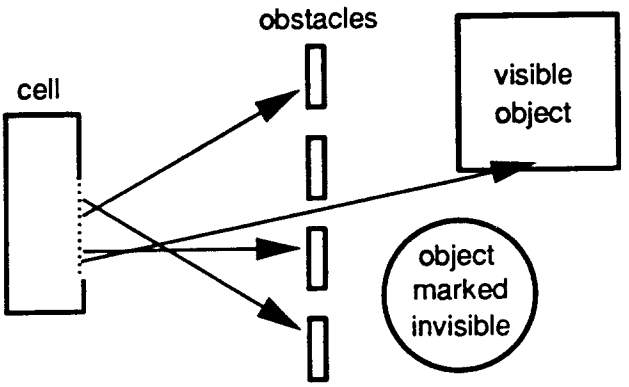


Figure 2.6: Stochastic ray-casting from portals [Air90]. The square object is correctly determined potentially visible. The circular object is not reached by a random ray, and is (incorrectly) determined to be invisible from all points in the source cell.

An object-space overestimation method described in [Air90] finds *portals*, or non-opaque convex regions, in otherwise opaque model elements, and treats them as area light sources (Figure 2.7). Opaque polygons (i.e., occluders) in the model then cause *shadow volumes* to arise with respect to these light sources. Parts of the model inside the combined shadow volumes can be marked invisible for any observer on or behind the originating portal. This method does exploit the hierarchical organization inherent in spatial subdivision, by removing shadowed internal nodes where possible. However, the portal-polygon occlusion algorithm has not found use in practice due to implementation difficulties and high computational complexity [Air90, ARB90].

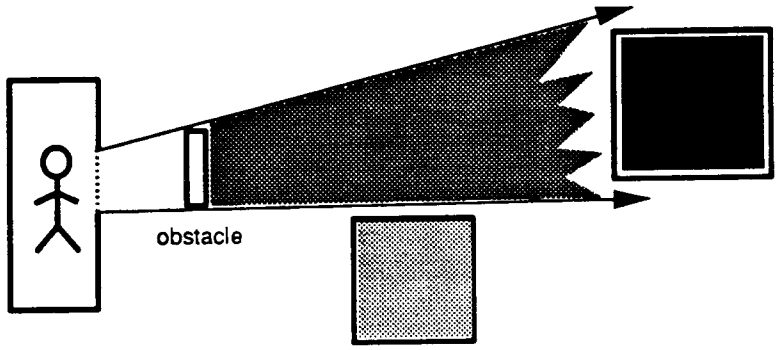


Figure 2.7: Shadow-volume casting [Air90]. Portals are treated as area light sources. Occluders cast shadows which generally remove cell contents or objects from the PVS.

Chapter 3

Computational Framework

This chapter constitutes an overview of the spatial data structures and algorithms that yield a framework for efficient static and dynamic visibility determinations in densely occluded environments. The data structures and algorithms are first presented abstractly, so that their salient features may be elucidated without regard to the dimensionality of the occluders, or any other input attributes. Subsequent chapters reify these data and algorithmic notions for three interesting and common classes of two- and three-dimensional occluders: line segments in the plane, axis-aligned or *axial* rectangles in three dimensions, and generally oriented convex polygons in three dimensions. Each algorithm and data type has been implemented for all three input classes. The second input class, axially aligned 3D occluders, is worthy of special treatment (and later, performance analysis) since it occurs so often for architectural models, and since axial occluders comprised more than 90% the structural features of our test model.

3.1 Occluders and Detail Objects

In d dimensions (here, $d = 2$ or $d = 3$) the *input* to the preprocessing phase is a collection of n $(d-1)$-dimensional opaque, convex *occluders* whose convex hull is a d-dimensional region $\mathbf{R}$. When $d = 2$, for example, the n occluders are coplanar line segments, and $\mathbf{R}$ is a convex polygon. When $d = 3$, the n occluders are convex, planar polygons, and $\mathbf{R}$ is a convex polyhedron (Figure 3.1). We say that a d dimensional occluder is *affine* to the d-flat that embeds it; for example, a line segment is affine to the line of which it is a part, and a planar polygon is affine to the plane of which it is a part. Finally, we say that a planar region is *lineal* if it has zero area, and *superlineal* or *areal* if it has non-zero area.

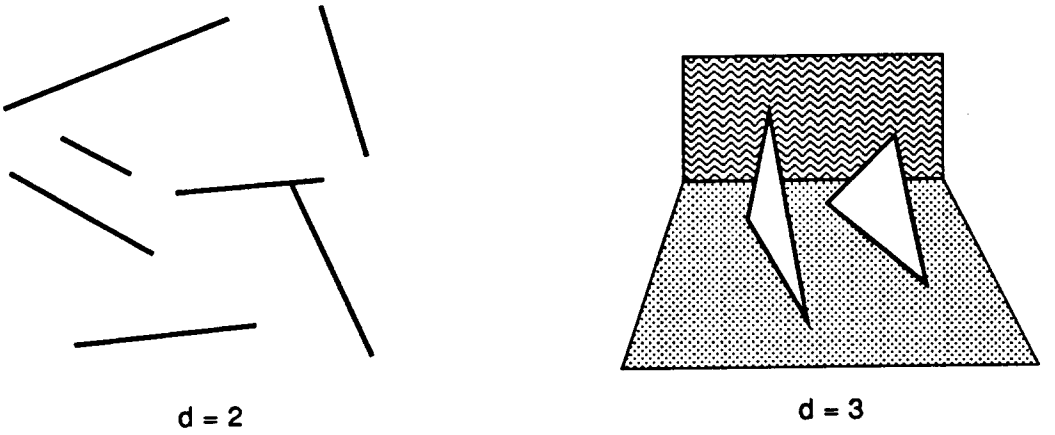


Figure 3.1: Convex, opaque occluders, in two and three dimensions.

The subdivision and visibility algorithms presented here treat major occluders and detail objects differently. Conceptually, the ill-defined question of distinguishing occluders and objects is largely orthogonal to that of visibility determination, since the superset visibility algorithms are provably correct regardless of input. In the real world, occluders and objects may be intermingled (for example, by the modeling system). In this case, heuristics can serve to assign probabilities that particular entities should be treated as occluders or as objects, based on size, genus, connectivity with other entities, and the like. Regardless of how detail objects are determined, we assume that they have associated thereafter a *bounding hierarchy* [Cla76] which can be used to subject the detail object to geometric operations such as culling, without examining all of the entities (for example, polygon vertices) contained within the object.

In what follows, therefore, we make the critical operating assumption that *all detail objects are unoccluding*. Consequently, although objects can be *subjected* to visibility determinations, they themselves do not *contribute* to the determination of visibility for any other object. The term "visibility" henceforth is used to mean the points, regions, objects, or entities that would be seen by a physical observer in an idealized environment in which only occluders can prevent the propagation of light. We show that, in practice, this operating assumption allows the development of time- and storage-efficient algorithms.

3.2 Spatial Subdivision

We define a spatial *cell* in d dimensions as a d-dimensional region with $(d-1)$-dimensional boundaries. We define a *spatial subdivision* or *SSD* as a collection of convex cells intersecting only along their boundaries (Figure 3.2). Recall that a spatial subdivision is *conforming* if it supports the primitive operations of point location, neighbor finding, and portal enumeration.

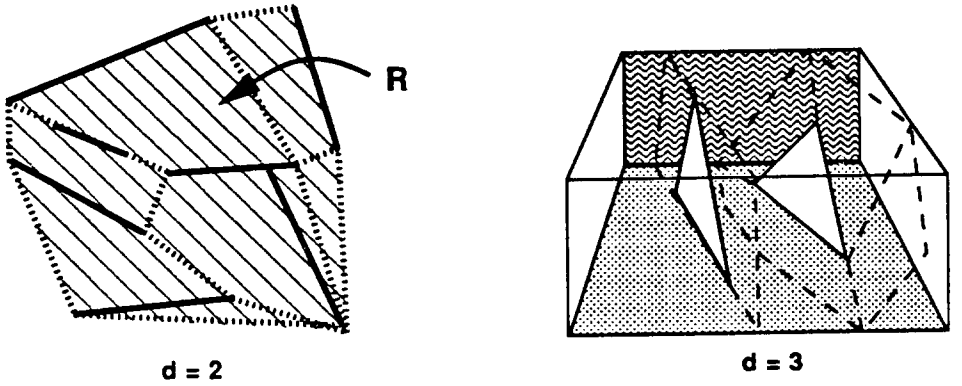


Figure 3.2: Spatial subdivisions, in two and three dimensions.

3.2.1 Point Location

The SSD supports the primitive operation of *point location*; that is, given a d-dimensional query point, the cell containing that point can be determined. We will show that most point location queries are highly *coherent*; that is, their result is highly correlated with that of previous queries. This coherence can be exploited to make point location very fast, in practice.

3.2.2 Object Population

Detail objects can be arranged efficiently, via *population* into the cells of the spatial subdivision. A spatial cell is *populated* with an object if the object, or some simpler *bounding representation* of it, intersects the cell in a d-dimensional region. (Some special instances arise in which a cell need not be populated with a spatially incident object. We review these situations in the sections covering specific input classes.) If a tradeoff arises between the time or storage efficiency of object population and the on-line culling efficiency, we choose the option which favors the efficiency of the on-line operation.

We perform cell population by inspecting each detail object's *bounding representation* for incidence with each spatial cell, and linking the incident objects to cells where appropriate. This can usually be done efficiently. For example, in a hierarchical spatial data structure, population is accomplished recursively by commencing at the root, and descending to spatial children only if the current spatial cell intersects the object bounding box. If a leaf cell is reached by the recursion, the object is associated with that cell.

3.2.3 Neighbor Finding

A spatial subdivision cell's *neighbors* are those other cells which intersect the given cell in a $(d-1)$-dimensional region. Typically, neighbor information is maintained as an invariant during construction of the spatial subdivision. Alternatively, neighbor information can be recovered as a postprocessing stage. Another option is to compute neighbor information *lazily*, i.e., as needed.

3.2.4 Portal Enumeration

Two cells in d dimensions are neighbors if they intersect in a $(d-1)$-dimensional region. An occluder and a cell are *incident* if they intersect in a $(d-1)$-dimensional region. Consider some convex $(d-1)$-dimensional boundary of a cell. We define the cell *egress* on this boundary to be the (not necessarily convex) set difference between the boundary and any occluders coaffine with the boundary. In other words, the egress is the "transparent" portion of the cell boundary. The egress may be incident on several spatially adjacent cells, or on none (if the boundary face is also a face of **R**).

For any cell, *portals* are defined as the elements of any convex decomposition of the cell egress through any particular cell boundary face. Cell portals are shown as dotted line segments (for $d = 2$) and dash-outlined regions (for $d = 3$) in Figure 3.2. The SSD must support *portal enumeration*; that is, given any cell and the set of occluders incident on the cell, the egresses and portals for that cell, along with the incident cells to which they lead, must be determinable. Again, portal information may be maintained as an invariant, constructed in a post-subdivision stage, or generated lazily.

3.2.5 Cell Adjacency Graph

Enumerating the portals of a spatial subdivision amounts to constructing an *adjacency graph* whose vertices correspond to the cells of the subdivision, and whose edges correspond to its portals

(Figure 3.3). This is a particularly useful view of the data structure, since, as we show, the static and dynamic visibility operations we present can be cast abstractly, and implemented, as *constrained traversals* of the adjacency graph.

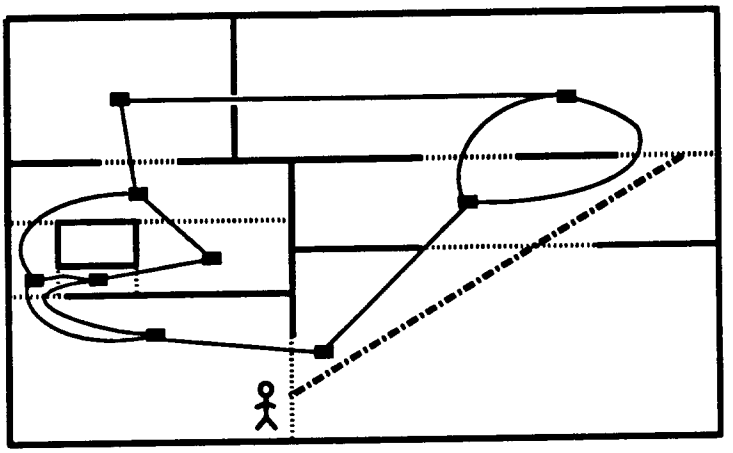


Figure 3.3: A 2D spatial subdivision, and corresponding adjacency graph. An observer is schematically represented at the lower left, and a sightline (broken) stabs a portal sequence of length three.

3.3 Static Visibility

The abstract notions of major occluders, detail objects, and conforming subdivisions (including spatial cells, point location, egress and portal enumeration, and the cell adjacency graph) have been defined. Given any conforming subdivision, several useful abstract visibility queries can be formulated. This section describes the queries independently of the input class or details of the subdivision, assuming only a few geometric predicates. Both the abstract operations and the necessary predicates will be made concrete in subsequent chapters.

3.3.1 Cell-to-Cell (Coarse) Visibility

Recall the definition of a *generalized observer* (§1.2): an observer constrained to a given cell (the *source* cell), but free to move anywhere inside this cell and to look in any direction. Generalized observers are posited during preprocessing to compute upper bounds on visible sets of cells, occluders, and objects. These upper bounds are valid for entire regions, namely, the cells of the subdivision. During the walkthrough or simulation phase, the observer view variables are

known more precisely, producing an *actual observer*. Generally, as knowledge of view variables increases in precision, more discriminating visibility queries become possible.

Once the spatial subdivision has been constructed, we compute cell-to-cell visibility information about the leaf cells by determining cells between which an unobstructed sightline exists. A generalized observer may see into a neighbor cell only through a portal, and into a more distant cell only through a *portal sequence*; i.e., an ordered list of portals such that each consecutive pair of portals lead into and out of the same cell. For any spatial subdivision, it is natural to characterize the set of cells visible to a generalized observer in a source cell. We call these cells the *cell-to-cell visibility* set associated with the source. Since a sightline must be disjoint from any occluders and thus must intersect, or *stab*, a portal in order to pass from one cell to the next, it is sufficient to find a *stabbing line* through a particular portal sequence (i.e., a line that intersects all portals of the sequence) to establish visibility between two cells. For, if some observer could see from a point in the *interior* of one cell to a point in the interior of another, a sightline must exist connecting the boundaries of the source and reached cells (cf. Figure 3.3).

Thus, the problem of finding sightlines between cell interiors reduces to finding sightlines through portal sequences of increasing length. Consequently, a crucial abstract visibility operation is to determine a stabbing line, given a portal sequence, or to determine that no such stabbing line exists. This is done by searching for paths in the adjacency graph that admit sightlines emanating from a given cell boundary.

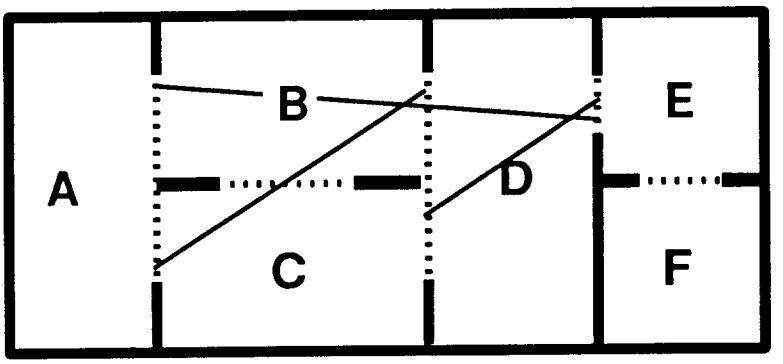


Figure 3.4: Some portal sequences that admit sightlines.

We say that a portal sequence *admits* a sightline if there exists a line that stabs every portal of the sequence. Figure 3.4 depicts six cells A, B, C, D, E, and F. The portal sequence [A/B, B/D, D/E] admits a sightline, where P/Q denotes a portal from cell P to cell Q. Similarly, the portal sequences [A/C, C/B, B/D] and [C/D, D/E] admit sightlines. Thus, A, B, C, D, and E

are mutually visible. In contrast, no portal sequence starting at A admits a sightline reaching F, so cells A and F are not mutually visible.

3.3.2 Generating Portal Sequences

Assume the existence of a predicate *Stabbing_Line*(P) that, given a portal sequence P, determines either a stabbing line for P or determines that no such stabbing line exists. All cells visible from a source cell C can then be found with the recursive procedure (comments are marked with //):

```
Find_Visible_Cells (cell C, portal sequence P, visible cell set V)
    V = V ∪ C                          // note C visible
    for each portal p of C             // N is the cell to which p leads
        P' = P concatenate p           // extend candidate portal sequence
        if Stabbing_Line (P') exists then
            Find_Visible_Cells (N, P', V)   // recur through N
```

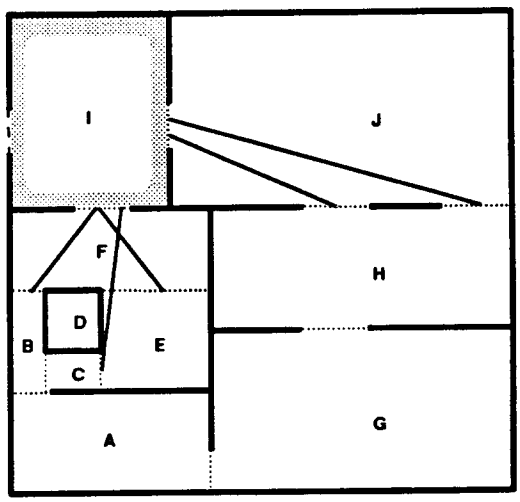


Figure 3.5: Finding sightlines from I.

Figure 3.5 depicts a spatial subdivision and the result of invoking the procedure *Find_Visible_Cells* (cell I, P = empty, $\mathcal{V} = \emptyset$). The invocation stack can be schematically represented as

```
Find_Visible_Cells (I, P = [ ], V = ∅)
  Find_Visible_Cells (F, P = [I/F], V = {I})
    Find_Visible_Cells (B, P = [I/F, F/B], V = {I, F})
```

Find_Visible_Cells (E, $\mathbf{P} = [I/F, F/E]$, $\mathcal{V} = \{I, F, B\}$)
Find_Visible_Cells (C, $\mathbf{P} = [I/F, F/E, E/C]$, $\mathcal{V} = \{I, F, B, E\}$)
Find_Visible_Cells (J, $\mathbf{P} = [I/J]$, $\mathcal{V} = \{I, F, B, E, C\}$)
Find_Visible_Cells (H, $\mathbf{P} = [I/J, J/H_1]$, $\mathcal{V} = \{I, F, B, E, C, J\}$)
Find_Visible_Cells (H, $\mathbf{P} = [I/J, J/H_2]$, $\mathcal{V} = \{I, F, B, E, C, J, H\}$)

The last line shows that I's computed cell-to-cell visibility $\mathcal{V}$ is $\{I, F, B, E, C, J, H\}$.

3.3.3 Stab Trees

The recursive nature of *Find_Visible_Cells()* suggests an efficient data structure: the *stab tree* (Figure 3.6). Each *vertex* of the stab tree corresponds to a cell visible from the source cell (cell I in Figure 3.5). Each *edge* of the stab tree corresponds to a portal stabbed as part of a portal sequence originating on a boundary of the source cell. The stab tree is a tree, since it is isomorphic to the terminating call graph of *Find_Visible_Cells()* above. However, since leaf cells are included in the stab tree once for each distinct portal sequence reaching them, there is in general no one-to-one correspondence between stab tree edges and adjacency graph edges (portals), or between stab tree vertices and adjacency graph vertices (cells). During the preprocessing phase, a stab tree is computed and stored with each leaf cell of the spatial subdivision; the cell-to-cell visibility is explicitly recoverable as the set of stab tree vertices.

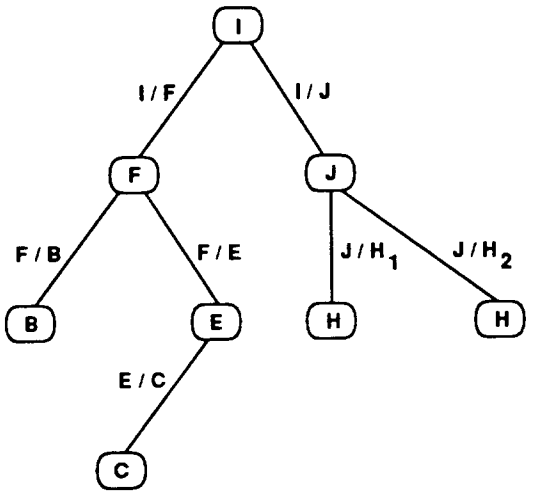


Figure 3.6: The stab tree rooted at I.

To find sightlines, we must generate candidate portal sequences, and identify those sequences that admit sightlines. We find candidate portal sequences with a *constrained graph traversal* on

the cell adjacency graph. Two cells P and Q are *neighbors* in this graph if their shared boundary is not completely opaque. Each convex, connected, non-opaque region of this shared boundary is a portal from P to Q. Given any starting cell C for which we wish to compute visible cells, a recursive depth-first search (DFS) of C's neighbors, rooted at C, produces candidate portal sequences. Searching proceeds incrementally; when a candidate portal sequence no longer admits a sightline, the depth-first search on that portal sequence terminates. As each cell is encountered by the DFS, a stab tree edge is constructed for the traversed portal, and a stab tree vertex corresponding to the reached cell.

Later, we show how to determine the existence of such sightlines for interesting classes of portal sequences in two and three dimensions. Computing this cell-to-cell relation over all cells constitutes *coarse* visibility, or cell-to-cell visibility, determination.

3.3.4 Cell-to-Region (Fine) Visibility

A generalized observer in a given source cell can, by moving to each point in the source cell, see the entirety of the cell. Since all cells are by definition convex, the generalized observer can by assuming positions on the cell portals see all points in the interior of the source cell's immediate neighbor cells. Cells farther away (i.e., reachable only through portal sequences of length greater than one), however, are in general only partially visible to the observer, due to occlusion by the edges of intervening portals (Figure 3.7)[1]. We define the source's *cell-to-region* visibility as the region visible to a generalized observer in the source.

The cell-to-region computation is a *fine*-grained visibility determination, operating on collections of points rather than on cells or objects. Analogously to the stabbing computation, cell-to-region visibility can be cast as a constrained DFS on the cell adjacency graph, along with an augmentation of the abstract stabbing line algorithm *Stabbing_Line*(P). If this procedure succeeds in computing a stabbing line for a particular portal sequence, another abstract procedure *Cell_To_Region*(P) then computes the region in the reached cell visible to a generalized observer situated on the first portal of the sequence P. We call this region the *antipenumbra* of the first portal in the sequence (Figures 3.8 and 3.9) since it can be thought of as the volume illuminated by an extended light source (the first portal) shining through a series of convex holes (the remaining portals). The details of this antipenumbra computation are highly dependent on the specific types of portals encountered, and are deferred to the sections pertaining to concrete visibility algorithms

[1] More precisely, the occlusion is due to the *complements* of the portals; that is, the opaque material abutting the portals.

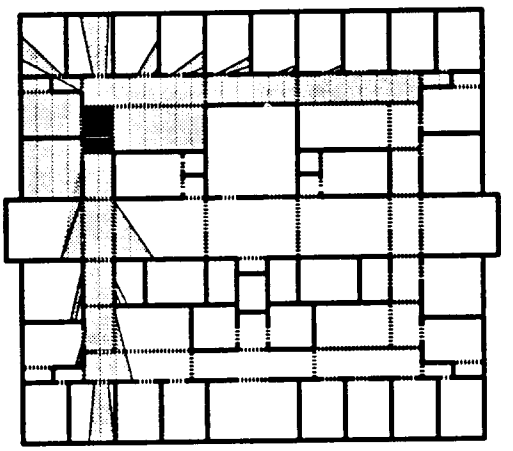


Figure 3.7: Distant cells are, in general, only partially visible (gray areas) from the source cell (dark).

for each input class.

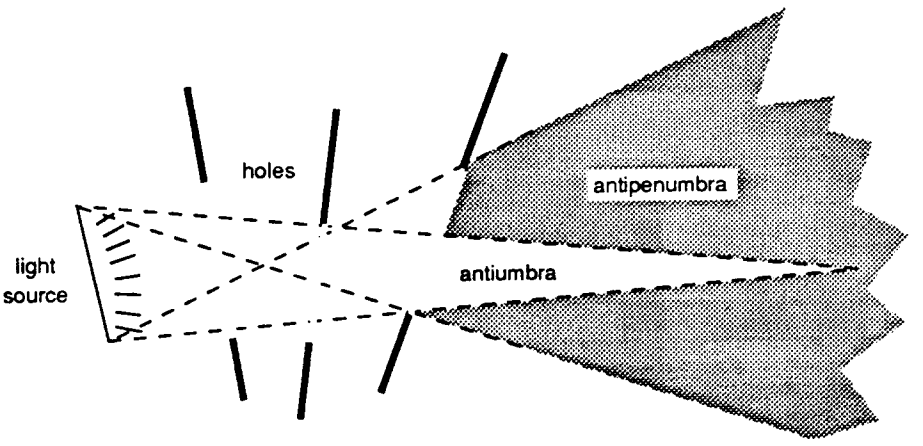


Figure 3.8: Antiumbra and antipenumbra through a series of 2D portals.

An abstract routine *Antipenumbra*(P) computes this antipenumbral volume, where P is a portal sequence, and the first portal of this sequence is a virtual light source. *Antipenumbra*(P) is invoked on a portal sequence P only when *Stabbing_Line* (P) returns successfully, i.e., only when the antipenumbra is known not to be empty.

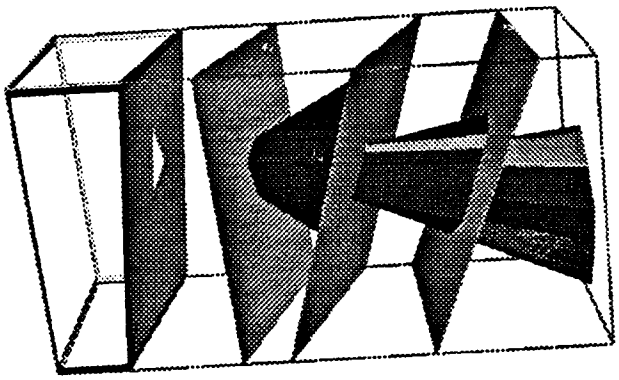


Figure 3.9: Successively narrowing antipenumbrae cast by an area light source in 3D (here, the leftmost portal) through the cells of a conforming spatial subdivision.

3.3.5 Cell-to-Object (Fine) Visibility

It may be storage-intensive to determine and store all cell-visible regions. Instead, we note that an object can only be visible from a given source cell if it is has some intersection with the cell-to-region visibility region of the source in that cell. We call the set of such objects the source's *cell-to-object* visibility (Figure 3.10). Abstractly, the cell-to-object computation amounts

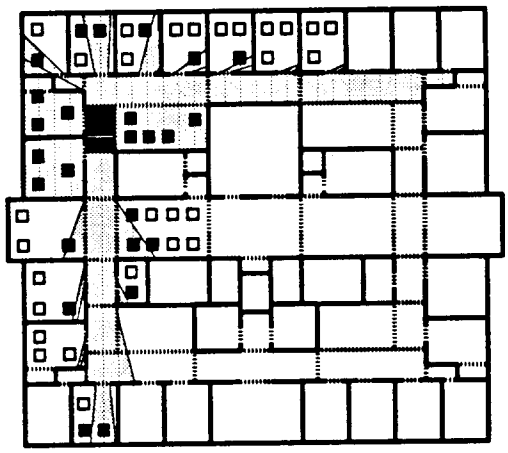


Figure 3.10: Cell-to-object visibility (filled squares) for a given source in 2D.

to a further augmentation of the stabbing line algorithm; once a cell is reached via a particular

portal sequence, and the antipenumbra of the first portal in the sequence is computed, we must be able to determine whether this antipenumbra intersects an object or its bounding representation. Cell-to-object visibiility is another example of fine-grained visibility computation. (In practice, it turns out that this two-step approach of constructing the antipenumbra explicitly, then testing for object intersections, is more efficient than computing individual portal sequence stabbing lines for each object. This phenomenon is discussed with the individual concrete algorithms.) Note that once an object is determined visible via *some* path from the source cell, it need not be tested again, regardless of any other paths reaching cells populated with the object.

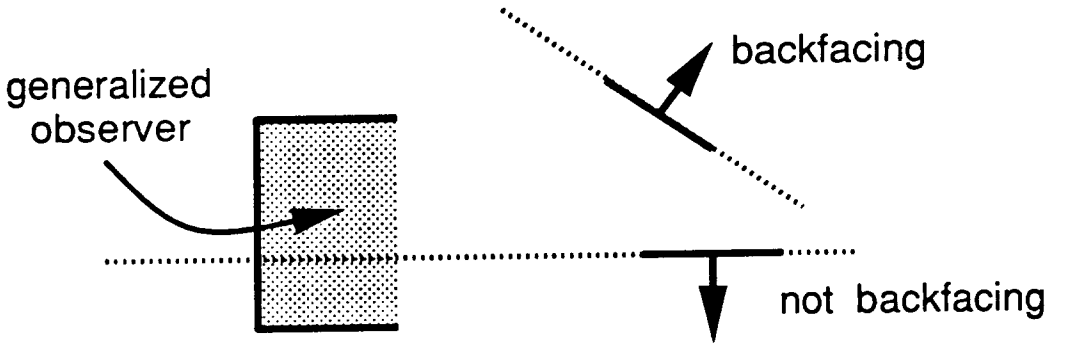


Figure 3.11: An object or occluder can be backfacing with respect to a *generalized* observer.

Note that, as in cell population, a storage optimization can be applied during cell-to-object visibility determination (Figure 3.11). We say that a $(d-1)$-dimensional object (occluder) is *backfacing* with respect to the generalized observer if the generalized observer lies entirely within the closed negative halfspace of the object (occluder). Backfacing objects (occluders) should not be included in the source's cell-to-object visibility, since they cannot be frontfacing for any actual observer in the source cell.

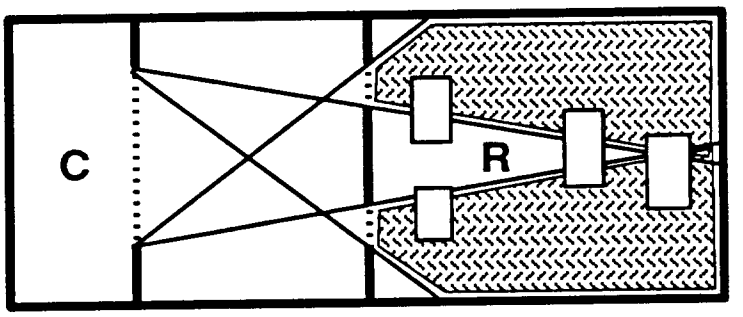


Figure 3.12: A source cell may reach another through several paths.

Finally, formulating the cell-to-object set amounts to *compressing* the stab tree in the following sense. Suppose a generalized observer in a source cell C reaches a particular cell R through several portal sequences. Rather than store the fully elaborated stab tree including several instances of R, the *aggregate* set of objects found visible in R from C can be associated with the single pair $\{C, R\}$, as an augmentation of the source's cell-to-cell visibility set.

3.4 Dynamic Visibility: On-Line Culling

The occluders are assumed to be in fixed position. Consequently, in any spatial subdivision over these occluders, the cell-based visibility relationships are *static*; that is, they depend neither on the progression of time nor on the position or field of view of the simulated observer.

The static nature of these computations limits the achievable tightness of the upper bound on visibility for most regions. That is, since the static visibility computation must allow for all possible viewing configurations attainable by the observer, it generally overestimates the model components visible to any *particular* observer. During the walkthrough phase or interactive simulation of the environment, the instantaneous variables describing the observer's position, view direction, velocity, etc., are known. This more exact knowledge of the observer permits a more discriminating *dynamic* culling operation.

Dynamic culling can itself be coarse or fine-grained. A coarse-grained cull computes the set of cells visible to an actual observer. A finer-grained cull computes the visible subregions (areas or volumes) in each of the cells. Finally, an eye-to-object cull identifies those objects inside the visible cells that are visible to the observer, i.e., reached by a sightline emanating from the eye.

In the on-line culling phase, an algorithmic module computes answers to queries concerning cells and objects potentially visible to a simulated observer with a specified position and field of view. The objects identified by the query are then issued to graphics hardware for hidden-surface removal and rendering.

The on-line culling part of the visibility module supports four queries: two *superset* queries and two *exact* queries. Below, we use the term "entity" to mean anything with a definite spatial extent that can be subjected to geometric culling operations. Typically entities are planar polygons, or perhaps assemblages of polygons. We use the term "region" to mean any 2D area or 3D volume.

The two superset queries are superset region and point visibility queries. The **superset region visibility** query computes a superset of those entities visible from any point in a specified region. The **superset point visibility** query computes a superset of those entities visible from a specified

point, and is therefore a special case of the superset region query.

If the geometric model is sufficiently small (i.e., simple), or if the graphics workstation is sufficiently fast, then the superset queries can be trivially implemented, simply by constructing each query to return the entire model. This clearly would suffice to achieve reasonable frame rates on the graphics workstation, since rendering the entire model at interactive rates would be feasible. In this sense, the class of "interesting" models is comprised of those models for which these trivial queries are insufficient to achieve rapid frame rates.

On the other hand, given robust implementations of just the two superset queries, a functioning walkthrough system can be built for the largest class of model. That is, given both queries, a visual simulation can be effected of a model that is both too large to completely reside in memory, and that contains too many entities (polygons) to be drawn in a single frame time. In the abstract framework presented here, effectively handling this class of model demands both the region and point superset queries.

The two exact queries are exact region and point visibility queries.

The **exact region visibility** query computes exactly those entities, or fragments of entities, visible from any point in a specified region. This is equivalent to the computational-geometric notion of *weak visibility* from the region. We outline a global solution of weak-visibility in Chapter 10.

The **exact point visibility** query computes exactly those entities, or fragments of entities, visible from a specified point. The exact point visibility query effects *hidden-surface removal* from a point, a classic computer graphics problem. We do not further consider this query.

3.4.1 Observer View Variables

We formalize the notion of a simulated observer in terms of the observer's *view variables*, which are necessary and sufficient to determine the subset of the model which must be rendered in order to provide a correctly displayed scene for the observer (Figure 3.13). These variables are dimension-dependent, and describe the momentary position and field of view of the actual observer. The simulation might also maintain time derivatives of the view variables, in order to construct envelopes for the variables that are valid for time intervals extending into the future.

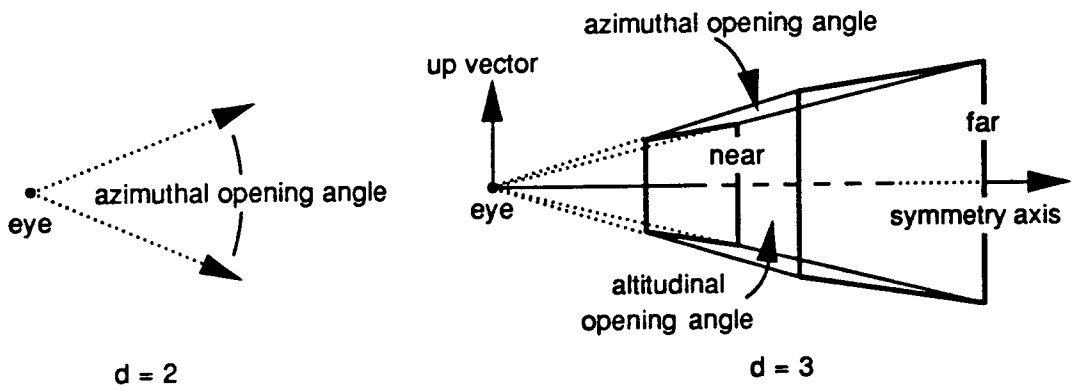


Figure 3.13: Observer view variables as view cone (2D) and view frustum (3D).

3.4.2 Eye-to-Cell Visibility

In contrast to the generalized observer, an *actual* observer is one whose view variables are known precisely. During the on-line culling phase, we can exploit static visibility computations to efficiently compute a superset of the cells and objects visible to the simulated actual observer. The eye-to-cell visibility set is simply the set of cells potentially visible to that observer. Successively tighter (i.e., smaller) upper bounds on this set can be constructed, at progressively greater computational expense.

During the on-line culling phase, the observer is at a known point and has vision limited to a *view cone* or *view frustum* emanating from this point. In two dimensions, the cone can be defined by a view direction and angular field of view; in three dimensions, the frustum can be defined by the left, right, top, bottom, and perhaps near and far planes.

During the simulation phase, it is desirable to have an efficient query that, given the observer's view variables, generates a set of polygons comprising a usefully tight upper bound on the set of visible polygons. We define the observer's *eye-to-cell visibility* set (cells E, F, I, and J in Figure 3.14) as those cells reachable by some ray that contains the eye and that lies inside the view frustum (ignoring occlusion caused by detail objects). Clearly the eye-to-cell visibility set is a subset of the source's cell-to-cell visibility. We show how the eye-to-cell visibility set may be efficiently computed via a traversal of the stab tree or, somewhat less efficiently, of the cell adjacency graph. The eye-to-cell computation is *exact* in the absence of occlusion due to detail objects.

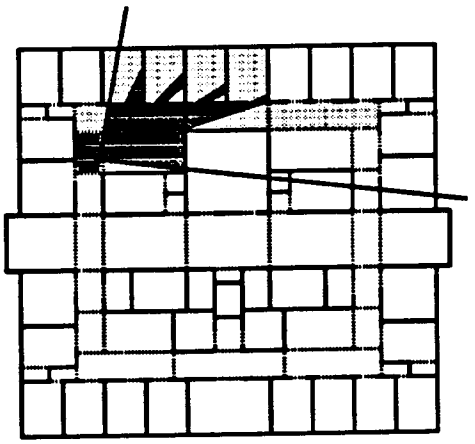


Figure 3.14: Eye-to-cell and eye-to-region visibility sets for an actual observer.

3.4.3 Eye-to-Region Visibility

Since each portal implies a set of constraints, the depth-first-search of the stab tree can be framed as a constrained graph traversal, augmented by an action associated with each successful traversal of a stab tree edge (i.e., portal). To compute eye-to-region visibility, the visible volume is initialized to the entire view frustum. Each traversed portal then "clips away" some portion of this volume in a dimension-dependent fashion. Finally, the remaining pyramidal, eye-centered region is intersected with the reached cell to produce the visible region in the reached cell due to the active path through the stab tree (Figure 3.14). The complexity of this region is shown to be dependent on the input class. In the simplest (2D) case, the visible region is of constant complexity, and can be described by a minimum and maximum angle. In 3D the visible region can have complexity as great as the total number of edges in all the portals along the path reaching the cell.

3.4.4 Eye-to-Object Visibility

Often we wish to identify the potentially visible objects in each reached cell. As in the static case, a fine-grain eye-based object cull can be formulated. We define the *eye-to-object* visibility set (Figure 3.15) as those objects to which a ray exists through the eye and inside the observer frustum (again ignoring occlusion due to detail objects). Clearly this set of objects must be a subset of those in the cell-to-object visibility for the observer cell, and must be incident on those cells in the eye-to-cell visibility set. We illustrate two useful computations that construct the eye-to-object visibility set. One efficiently computes an upper bound on this set; another is more expensive

computationally, but computes a generally tighter (i.e., smaller) eye-to-object set.

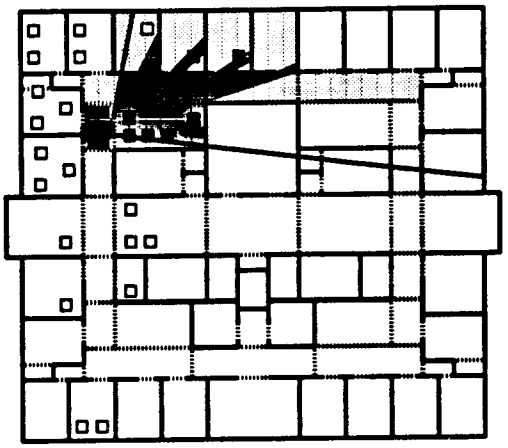


Figure 3.15: Eye-to-cell (light areas), eye-to-region (dark areas), and eye-to-object visibility set (those dark squares incident on the eye-to-region visibility) for an actual observer.

We show in §8.4 that, regardless of input class, casting visible object determination as an *existence* problem, i.e., a question of the existence of a single sightline, results in a substantial speedup over the efficiency of visible region determination. In practice the eye-to-object visibility is more often desired than the eye-to-region visibility (although the latter is useful for purposes of algorithm verification and visualization).

Chapter 4

Two Dimensional Environments

This chapter describes algorithms for each of the abstract visibility operations introduced in Chapter 3. Here, we specify each concrete operation in two dimensions, where occluders are line segments. The discussion of two-dimensional environments is a worthwhile pedagogical introduction to concrete notions of spatial subdivision, visibility precomputation, and on-line culling queries for two-dimensional occluders. Moreover, the 2D techniques are quite efficient and useful in 3D environments that are describable by "floorplans" – e.g., single-floor structures with few internal openings other than doorways [Bel92].

Readers familiar with 2D data structures and visibility computations may wish to skip to Chapter 5.

4.1 Major Occluders and Detail Objects

Occluders are generally-oriented or axis-aligned line segments, described by pairs of endpoints in the plane. Occluders are assumed to intersect only at their endpoints. Objects, for the purpose of visibility determination, are simply bounding boxes (e.g., rectangles).

4.2 Spatial Subdivision

In 2D, the n occluders are finite-length line segments, and we desire a spatial subdivision whose cells are convex polygons, and whose portals are line segments. This can be done in one of two ways. If the line segments are generally oriented line segments (Figure 4.1), any triangulation respecting the line seqments can be used. By Euler's relation, this triangulation contains $O(n)$

triangles. The triangulation consists only of edges which are either occluders, or connect two vertices of distinct occluders. Each cell in the corresponding spatial subdivision is a triangle, and there are at most $O(n)$ lineal portals, each of which can be found and stored in constant time. The well-known constrained Delaunay triangulation can be constructed in $O(n \lg n)$ time [Sei90a]. However, we use a less-efficient greedy triangulation algorithm that simply starts with the input line segments and inserts non-intersecting edges until the result is a triangulation.

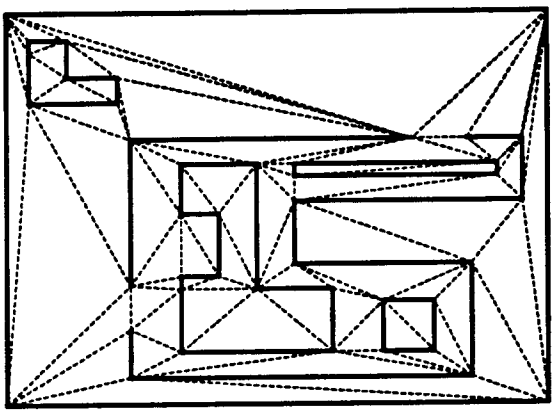


Figure 4.1: A linear-size constrained triangulation of the n occluders. Occluders are shown as bold segments, portals as dashed lines.

In contrast, when the occluders are *axial* (i.e., parallel to the x or y axis), We perform the spatial subdivision using a BSP tree [FKN80] whose splitting planes contain the occluders. For the special case of axial data, the BSP tree becomes an instance of a k-D tree [Ben75] with $k = 2$. Every node of a k-D tree is associated with a spatial cell bounded by k *extents* $[x_{0,min} \cdots x_{0,max}], \ldots, [x_{k-1,min} \cdots x_{k-1,max}]$. This closed definition of cells allows them to intersect along their boundaries. If a k-D node is not a leaf, it has a *split dimension* s such that $0 \leq s < k$; a *split abscissa* a such that $x_{s,min} < a < x_{s,max}$; and *low* and *high child* nodes with extents equivalent to that of the parent in every dimension except $k = s$, for which the extents are $[x_{s,min} \ldots a)$ and $[a \ldots x_{s,max})$, respectively. Thus the cells of the resulting spatial subdivision are axial rectangles, and portals are axial line segments (Figure 4.2).

A minimal-size k-D tree with $O(n)$ cells over n occluders can be constructed in $O(n \lg n)$ time, with worst-case total storage complexity linear in n [PY89]. A balanced k-D tree supports logarithmic-time point location and linear-time neighbor queries.

We construct a non-minimal k-D tree incrementally by greedy selection of an occluder along

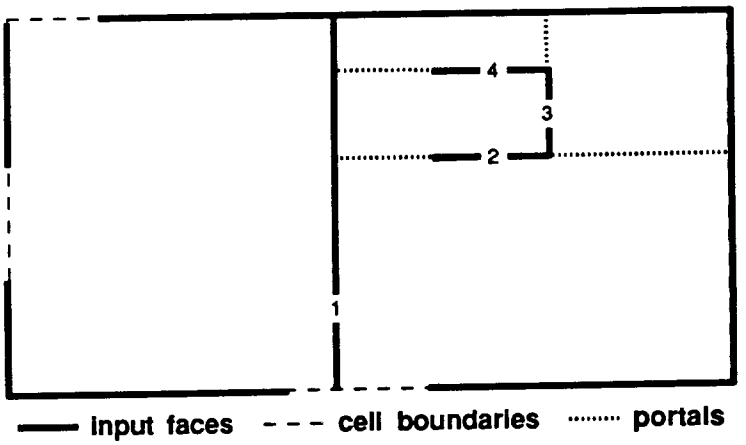


Figure 4.2: The leaf cells of a linear-size k-D tree over n line segments. Split planes are numbered in the order which they were introduced.

which to introduce a splitting plane at each leaf, until no more candidates exist (i.e., until every occluder lies on the boundary of one or more leaves). Under our splitting criteria, the resulting tree is not in general balanced or of linear size, but the creation and search time can not be worse than linear in the number of cells. We have observed the storage requirements of the tree to be reasonable in practice, even for complex environments.

4.2.1 Point Location

Given a constrained Delaunay triangulation over n input segments, a logarithmic-depth search structure can be constructed for the triangulation in $O(n \lg n)$ time [SP85]; however, a linear-time (brute force) search is sufficient for our purposes, due to the high coherence of point-location queries in a real application.

Point location in a k-D tree is straightforward. Recursively, the query point is compared to the current node. If the point is not inside the extent, a special value is returned denoting this. Otherwise, if the node is a leaf node, it is returned as the containing cell. Otherwise, the point is compared to the split abscissa for the cell, and the appropriate subtree of the cell is searched. This procedure is invoked on the root node.

Often, subsequent queries will be coherent, i.e., produce an answer highly related to that of the preceding query. An efficient implementation could exploit this fact by, for example, caching the most recently computed containing cell and examining it for incidence with the query point. Only if this incidence check fails would a new search be performed. More sophisticated schemes might

reuse the last search sequence, or use portals to search the adjacency graph from the cell satisfying the previous point location query.

4.2.2 Object Population

Once the SSD has been generated, the subdivision cells must be *populated* with detail objects. We assume that every object has associated with it an axial bounding box (i.e., a rectangle). Suppose there are n input line segments and m objects to populate. For both 2D input classes, general line segments and axial line segments, object population can be efficiently implemented.

For generally oriented line segments, cell population reduces to determining the intersection of a rectangle with a triangular spatial cell. This can be done in constant time per cell, and a naive algorithm therefore inserts all objects in at most $O(mn)$ time. A more sophisticated approach would be to build an efficient search structure for the triangulation (see, e.g., [Kir83, EGS86, ST86]), permitting $O(\lg n)$-time point location, and constant-time incidence testing of the object bounding box with each intermediate level of the search structure. If an object can impinge on $O(n)$ cells, this improvement yields no asymptotic speedup. If an object can impinge on no more than k connected cells, then the time to insert each object can be no more than $O(k \lg n)$, and the total time for inserting all objects improves to $O(mk \lg n)$. Finally, if neighbor information is available for each cell, then each object can be inserted in $O(\lg n + k)$ time by finding the cell enclosing one point of the object, then "walking" the object boundary while simultaneously traversing and populating any implicated cells.

For axial line segments, population of each leaf cell again reduces to the constant-time task of intersecting an axial bounding box with an axial cell extent. Assuming a linear-size, log-depth spatial subdivision and a maximum of k object-cell incidences per object, the time complexity of object insertion is again $O(mk \lg n)$. If the number of cells is not known to be $O(n)$, as is the case for our heuristic splitting operation, then we must introduce the quantities c, the number of cells in the spatial subdivision, and d, the maximum length of any path from leaf to root. Then the worst-case object insertion time again becomes $O(cm)$, or $O(kdm)$ since each object requires at most kd time to be inserted in at most k incident leaf cells.

There are two storage optimizations that apply to cell population (Figure 4.3). The cell should not be populated with 2D (i.e., areal) detail objects that have only a *point* or zero-D intersection with the source cell, since the object will only be visible if the observer sees into a neighbor of the source intersecting the object in a 1D or 2D region. Second, if 1D (i.e., lineal) objects have

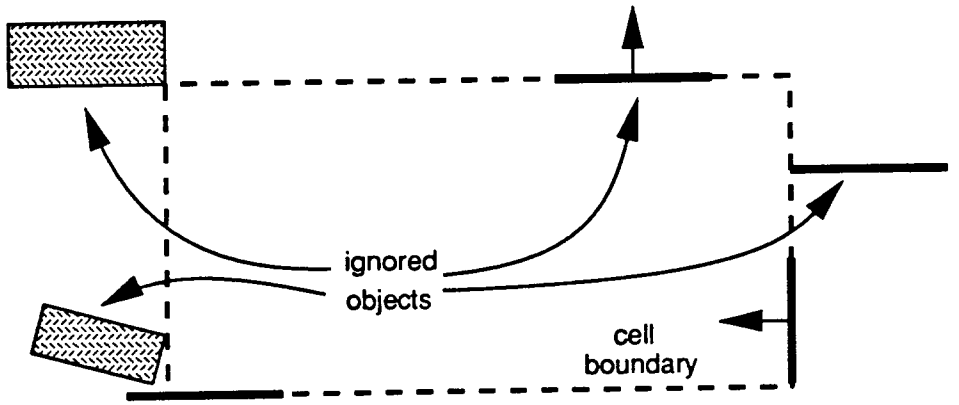


Figure 4.3: One- or two-dimensional detail objects intersecting the cell in a point, or 1D backfacing detail objects, may be ignored during cell population.

associated normal orientations, the cell should not be populated with lineal objects (occluders) that are backfacing with respect to the generalized observer, since these objects (occluders) will be backfacing for all actual observer viewpoints in the cell.

4.2.3 Neighbor Finding

Neighbor finding information in triangulations can be maintained as invariants during construction, or recovered after construction [GS85, SP85]. In k-D trees, neighbor information is easily generated via ascent and descent operators.

4.2.4 Portal Enumeration

Neighbor finding costs, on average, logarithmic time per cell, or $O(n \lg n)$ time overall. When 2D k-D trees are used, a simple post-processing path ascends and descends the tree from each leaf cell, establishing its spatially adjacent neighbors. Although some cells may have $O(n)$ such neighbors, the Euler relation dictates that the planar cell adjacency graph have a number of edges (i.e., neighbor relations) linear in the number of nodes (spatial cells); thus, neighbor finding will on average require constant time per cell, and $O(c)$ time overall (where c is the number of cells).

Note that for any class of 2D input, a cell whose boundary has only a *point* (i.e., zero-dimensional) intersection with an occluder need not take this occluder into account during portal enumeration, as the occluder can have no effect on the visibility of the generalized observer in the cell (Figure 4.4). Note that, unlike the detail object case, both backfacing and frontfacing 1D incident occluders are useful, and should not be ignored during portal enumeration.

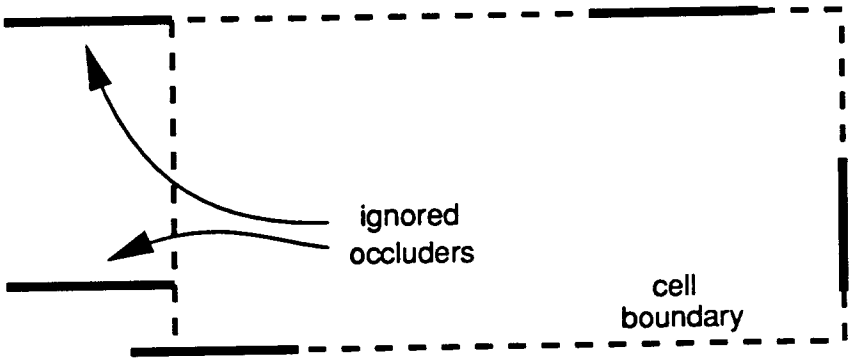


Figure 4.4: Occluders intersecting the cell boundary in a point may be ignored during portal enumeration.

For either 2D input class, portal enumeration is a straightforward operation, equivalent to computing set differences among collections of collinear line segments (the occluders) with the cell boundary (a line segment). Importantly, cell portals are *oriented* by the direction in which they must be crossed during traversal of the adjacency graph. This orientation is crucial to the development of efficient algorithms for finding stabbing lines, constructing static culling regions, and performing on-line culling, as we shall show.

4.3 Static Visibility Operations

This section describes concrete implementations of the abstract algorithms *Stabbing_Line()* and *Find_Visible_Cells()* introduced in Chapter 3, for 2D occluders.

4.3.1 Cell-to-Cell Visibility

Once the 2D spatial subdivision has been constructed, we compute *cell-to-cell visibility* information about the leaf cells by determining cells between which an unobstructed *sightline* exists. This sightline must intersect, or *stab*, a portal in order to pass from one cell to the next. If there exists a sightline through two points in two cells' interiors, there must be a sightline intersecting a portal from each cell. Thus, in 2D, the problem of finding sightlines between cell *areas* reduces to finding sightlines between line segments on cell *boundaries*.

Finding Sightlines Through Portal Sequences

Given the abstract depth-first-search algorithm described in §3.3.3, we must make concrete the notion of constrained search through a two-dimensional planar subdivision.

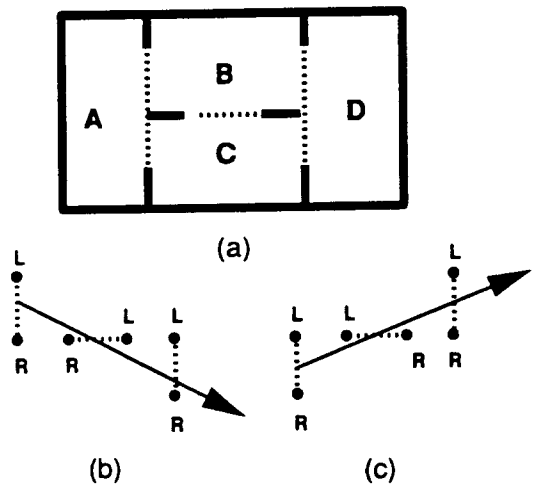


Figure 4.5: Oriented portal sequences, and separable sets **L** and **R**.

The fact that portal sequences arise from directed paths in the subdivision adjacency graph allows us to *orient* each portal in the sequence and compute sightlines efficiently. As the DFS encounters each portal, it places the portal endpoints in a set **L** or **R**, according to the portal's orientation (Figure 4.5). A sightline can stab this portal sequence *if and only if the point sets* **L** *and* **R** *are linearly separable*; that is, iff there exists a line S such that

$$\begin{aligned} S \cdot L \geq 0, \quad & \forall L \in \mathbf{L} \\ S \cdot R \leq 0, \quad & \forall R \in \mathbf{R}, \end{aligned} \tag{4.1}$$

where the notation $S \cdot P$ indicates that the signed distance between the point P and the oriented line S should be computed.

For a 2D portal sequence of length m, this is a 2D linear programming problem of $2m$ constraints. Both deterministic [Meg83] and randomized [Sei90b] algorithms exist to solve this linear program (i.e., find a line stabbing the portal sequence) in linear time; that is, time $O(m)$. If no such stabbing line exists, the algorithms report this fact.

4.3.2 Cell-to-Region Visibility

This section describes a concrete implementation of the abstract algorithm *Cell_To_Region()* introduced in Chapter 3, for 2D occluders.

The depth-first search of the adjacency graph "reaches" a cell each time a portal sequence is successfully stabbed with *Stabbing_Line()*; the computed stabbing line is a "witness" to the fact that an observer can see from some point on a portal of the source cell to a point on a portal of the reached cell. For general portal sequences of more than one portal, however, the reached cell is not visible in its entirety to the generalized observer (Figure 4.6).

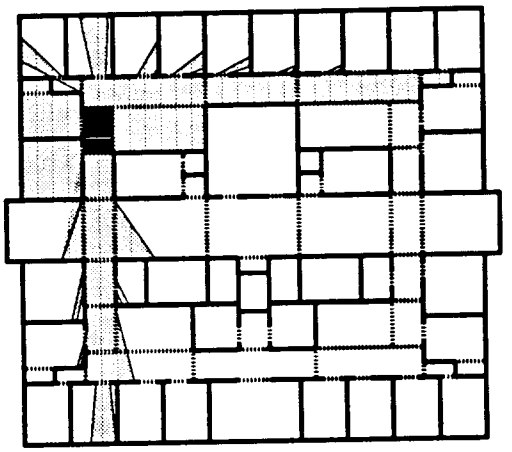


Figure 4.6: Distant cells are, in general, only partially visible from the source.

Conceptually, the portal is treated as a light source, and a description is computed of the light emanating from the portal that propagates into the remainder of the spatial subdivision. Each portal illuminates some volume in its positive halfspace (the halfspace not containing the cell from which the portal exits); the union of all such volumes for all portals on the boundary of a given source cell is the area potentially visible to an observer in the source, or the *cell-to-region* visibility for that source (Figure 4.6).

Computing the region illuminated by, or visible to an observer on, a specified portal is an interesting computational geometry problem in its own right. This antipenumbra computation is related to the 2D and 3D *weak visibility* problem in computational geometry [O'R87, SP85, SS90], and to *shadow casting* from area light sources in 3D, an often-studied problem in computer graphics [NN83, PW88, Air90, CF92].

Two-Dimensional Hourglasses

The bundle of lines stabbing a sequence of line segments in the plane is known as a *bowtie* or *hourglass* [Her87] due to its characteristic shape of an upper and lower convex hull and two crossover edges that connect the hulls (Figure 4.7). The antipenumbral region beyond the last portal is the area enclosed by the positive halfspaces of the crossover edges, oriented so as to contain the illuminated portion of the final portal in the sequence.

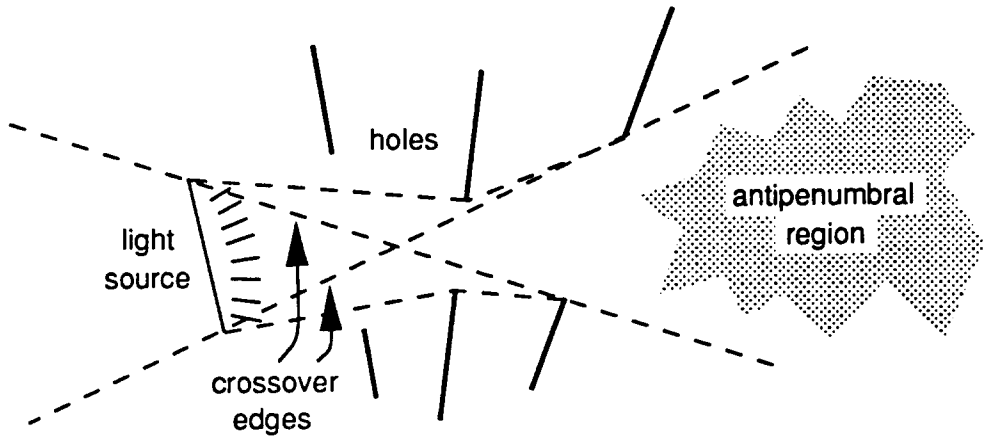


Figure 4.7: In two dimensions, a portal sequence admits an "hourglass" of stabbing lines.

In two dimensions, computing the antipenumbra of an oriented portal sequence of length n can be done straightforwardly in $O(n)$ time. To show this, we demonstrate how a portal can be added to an existing sequence of length $n-1$, and the antipenumbra correctly modified, in constant time per portal. Imagine traversing a portal and emerging in the cell to which the portal leads; the traversal unambiguously determines "left" and "right" portal vertices. The hourglass region then consists of paired hulls, which in the context of oriented portals can be called "leftmost" and "rightmost" hulls.

Clearly, adding a new lefthand or righthand point to either hull may require $n-1$ steps, since all existing hourglass edges may be destroyed by the appearance of the new point (Figure 4.8). However, each hourglass edge may be inserted and or deleted at most once over n portal additions, so the total number of insertions and deletions is n, and the average time per portal insertion is constant. Once the leftmost and rightmost hulls have been built, the crossover edges are found between the first and last vertices of the two hulls in constant time. Finally, the antipenumbra is found to be empty when a left-hand vertex is right of the right-hand crossover edge, a right-hand vertex is left of the left-hand crossover edge, or when the updated crossover edges do not intersect (Figure 4.9).

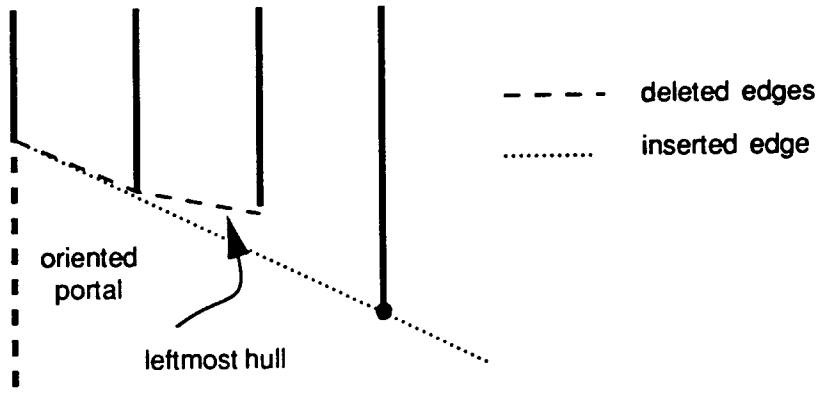


Figure 4.8: The left-hand points form a convex chain.

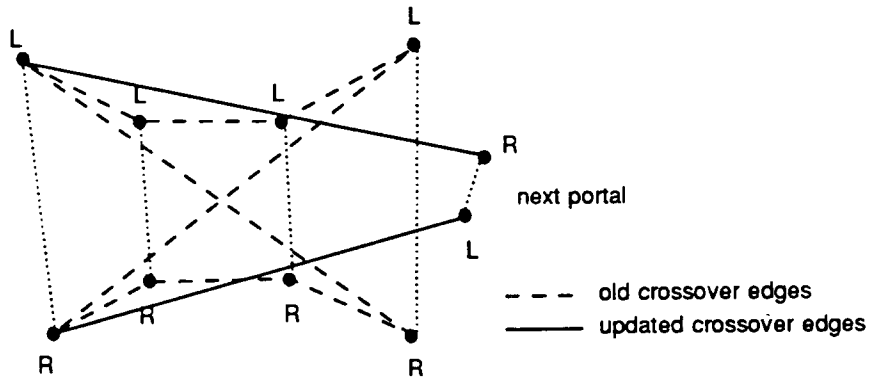


Figure 4.9: A 2D portal sequence terminates if the updated crossover edges do not intersect (above), or if the newly encountered portal does not intersect the active antipenumbral region (not shown).

4.3.3 Cell-to-Object Visibility

Once a cell has been reached through a particular portal sequence, and the antipenumbra of the zeroth portal with respect to the reached cell has been computed, determining the cell-to-object visibility amounts to intersecting each object in the reached cell with the antipenumbral region. In two dimensions, the antipenumbra in any reached cell is a convex polygon of constant complexity; intersecting this with the bounding box of any object therefore requires constant time. Thus, for a particular portal sequence reaching a cell containing m objects, this sequence's contribution to the source's cell-to-object visibility set can be determined in $O(m)$ time. This cost is incurred for each portal sequence reaching the cell from the source. Alternatively, the edges of the object's bounding box can be considered as portals, and the existence of a sightline through the active portal sequence and the object bounding box can be sought. This is typically less efficient computationally, since the per-object complexity of the sightline search is linear in the length of the portal sequence reaching the cell, rather than constant time.

4.4 Dynamic Visibility Queries

4.4.1 Observer View Variables

In two dimensions, the observer's *view variables* are simply the observer location (a 2D point), and bounds on the azimuthal extent of the observer's 2D view cone, represented as the intersection of two halfspaces.

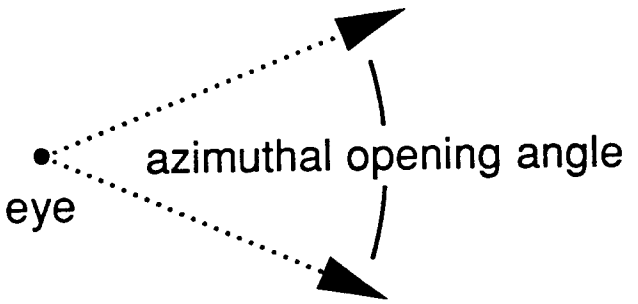


Figure 4.10: Observer view variables in 2D.

4.4.2 Eye-to-Cell Visibility

During the query phase, the observer is at a known point and has vision limited to a *view cone* emanating from this point. Recall that the *eye-to-cell* visibility is the set of cells partially or completely visible to an observer with a specified view cone (Figure 4.11). We present a series of culling methods based on constrained traversal of the stab tree. This series of methods illustrates how the expenditure of increasing amounts of computational effort permits the computation of successively tighter upper bounds on potentially visible cell sets. Each culling method extends directly to three dimensions under a straightforward generalization of the geometric operations involved (e.g., halfspace construction, linear programming). In later chapters, we make these generalizations algorithmically explicit.

Eye-to-Cell Culling Methods

Let O be the cell containing the observer, C the view cone, $\mathcal{S}$ the stab tree rooted at O, and $\mathcal{V}$ the set of cells visible from O (i.e., $\{O, D, E, F, G, H\}$ in Figure 4.11). We compute the observer's eye-to-cell visibility by *culling* $\mathcal{S}$ and $\mathcal{V}$ against C. We discuss several methods of performing this cull, in order of increasing effectiveness and computational complexity. All but the last method yield an overestimation of the eye-to-cell visibility; that is, they can fail to remove a cell from $\mathcal{V}$ for which no sightline exists in C. The last method computes the exact eye-to-cell visibility set.

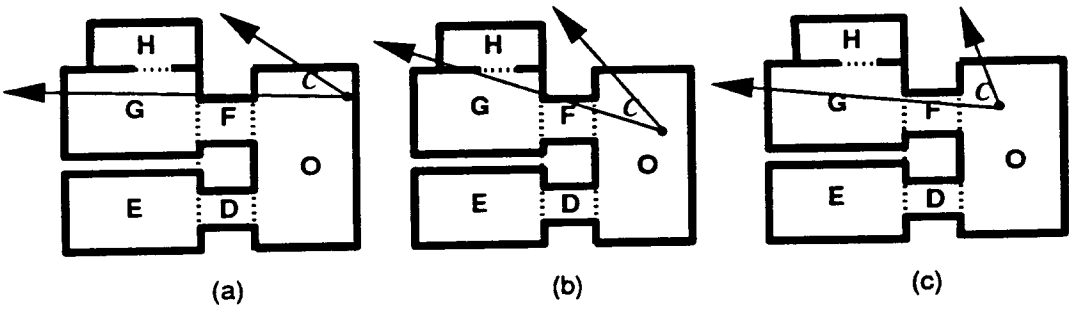


Figure 4.11: Culling O's stab tree against a view cone C.

Disjoint cell. The simplest cull removes from $\mathcal{V}$ those cells that are disjoint from C; for example, cells D, E and F in Figure 4.11-*a*. This can be done in $O(|\mathcal{V}|)$ time, but does not remove all invisible cells. Cell G in Figure 4.11-*a* has a non-empty intersection with C, but is not visible; any sightline to it must traverse the cell F, which *is* disjoint from C. More generally, in the cell adjacency graph, the visible cells must form a single *connected component*, each cell of which has

a non-empty intersection with C. This connected component must also, of course, contain the cell O.

Connected component. Thus, a more effective cull employs a depth-first search from O in S, subject to the constraint that every cell traversed must intersect the interior of C. This requires time $O(|S|)$, and removes cell G in Figure 4.11-*a*. However, it fails to remove G in Figure 4.11-*b*, even though G is invisible from the observer (because all sightlines in C from the observer to G must traverse some opaque input face).

Incident portals. The culling method can be refined further by searching only through cells reachable via *portals* that intersect C's interior. Figure 4.11-*c* shows that this is still not sufficient to obtain an accurate list of visible cells; cell H passes this test, but is not visible in C, since no sightline from the observer can stab the three portals necessary to reach H.

Exact eye-to-cell. The critical observation is that for a cell to be visible, some portal sequence to that cell must admit a sightline that lies *inside* C and *contains the viewpoint.* Retaining the stab tree S permits an efficient implementation of this sufficient criterion since S stores with O every portal sequence originating at O. Suppose the portal sequence to some cell has length m. This sequence implies $2m$ linear constraints on any stabbing line. To these we add two linear constraints, ensuring that the stabbing *ray* lie inside the two halfspaces whose intersection defines C. That is, a stabbing line exists if and only if some vector, anchored at the eye, can have a nonnegative dot product with each of the $2m + 2$ halfspace normals (Figure 4.12). The resulting 2D linear program of $2m + 2$ constraints can be solved in time $O(m)$, i.e., $O(|\mathcal{V}|)$ for each portal sequence.

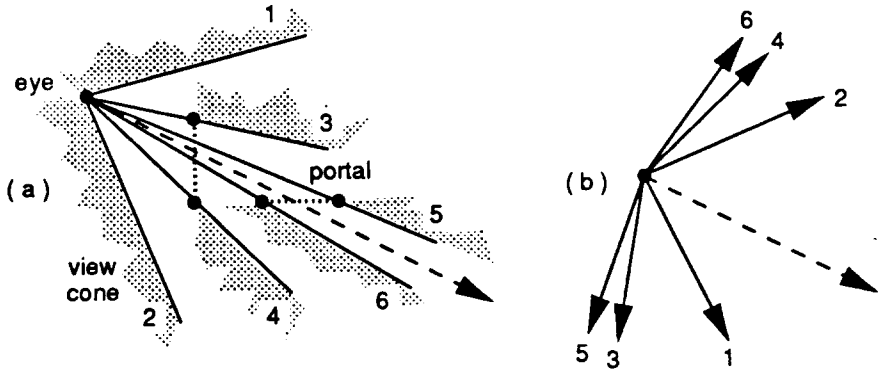


Figure 4.12: The $2m + 2$ halfspace normals arising from a portal sequence of length m (a), and the corresponding 2D linear program (b). The dashed arrow is a feasible solution.

This final refinement of the culling algorithm computes exact eye-to-cell visibility. Figure 4.11-

c shows that the cull removes H from the observer's eye-to-cell visibility since the portal sequence $[O/F, F/G, G/H]$ does not admit a sightline through the viewpoint. This linear programming formulation is not optimal in two dimensions, however. The next section describes a procedure which requires constant time per portal, computes exact eye-to-cell visibility, and computes eye-to-region visibility at constant added cost per encountered cell.

4.4.3 Eye-to-Region Visibility

During the walkthrough phase, the visible region can readily be computed from the stored stab tree. The visible area in any cell is the intersection of that (convex) cell with one or more (convex) wedges emanating from the observer's position (Figure 4.13).

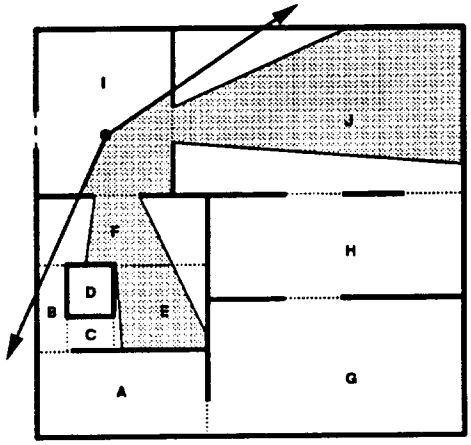


Figure 4.13: The view cone during the stab tree DFS.

The two-dimensional eye-to-region visibility query can be implemented as an action to be applied whenever the eye-based cull reaches a cell through a specified portal. After the view wedge is suitably narrowed by the portal (requiring constant time), the resulting infinite wedge is intersected with the reached cell. The union of all such areas for each path reaching a cell constitutes the source's eye-to-region visibility in that cell. Note that the current portal sequence terminates when the current wedge has no intersection with the newly encountered portal.

4.4.4 Eye-to-Object Visibility

The eye-to-object set comprises those objects potentially visible to an observer. Objects must satisfy many constraints to be visible: 1) they must be located in a cell visible to the source; 2)

they must lie in the source's cell-to-object visibility set; 3) they must lie in a cell in the source's eye-to-cell visibility set; 4) they must intersect the interior of the observer's view cone; and 5) there must exist a sightline between the eye and the object, stabbing each portal in a sequence reaching the cell containing the object. The first three constraints are implemented via a straightforward marking algorithm on the cell adjacency graph.

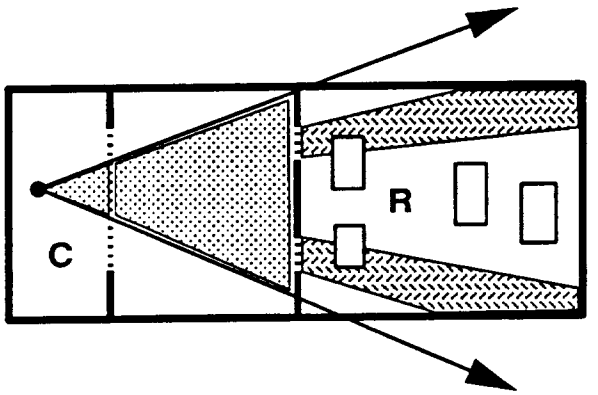


Figure 4.14: The 2D eye-to-object visibility computation.

To see how the final two constraints may be checked, again conceptualize the observer as a light source; this time, as a "flashlight beam" emitting light from a point only in a specific range of directions (the interior of the view cone). The eye-to-region DFS maintains the wedge of light that survives through each encountered portal (Figure 4.14). This wedge can be stored in constant space, using a leftmost and rightmost oriented line, and can be updated in constant time per portal. The wedge "illuminates" an area of constant complexity in every reached cell– namely, the intersection of the current wedge with the cell. The object bounding boxes incident on this illuminated area can be found in constant time per object with a straightforward 2D intersection computation.

Chapter 5

Three-Dimensional Axial Environments

This chapter describes concrete algorithms for each of the abstract visibility operations introduced in Chapter 3, for three dimensional polyhedral input, where major occluders are *axial rectangles*; that is, rectangles parallel to a principal plane, and with edges parallel to the principal axes. For this class of input, which occurs very frequently in architectural models, the visibility data structures, static operations and dynamic queries can be implemented with low storage and time complexity.

5.1 Major Occluders and Detail Objects

Occluders are axial rectangles in 3D, and can be represented by a three-valued integer describing the normal axis, a single floating-point value for the offset of the face along this axis, and two floating-point coordinate values for the minimum and maximum extent of the occluder in each of the two dimensions perpendicular to the axis. Detail objects are simply axial bounding boxes (typically, detail objects correspond to complex rendered geometry, but the bounding boxes are all that concern us for the purposes of visibility computation).

5.2 Spatial Subdivision

As in the 2D axial case, we use a k-D tree [Ben75] for subdivision, with $k = 3$. The essential details of definition are identical to those for the two-dimensional case in §4.2, and are omitted here. Intuitively, the 3D k-D tree is a recursively nested collection of axial parallelepipeds. The root node is a parallelepiped enclosing all data of interest. A node is either a leaf, or it has an axial splitting

plane and two child nodes which are themselves axial parallelepipeds.

Splitting planes are introduced along the major opaque elements in the model, namely the walls, floors, and ceilings. Determining a minimum-size parallelepipedal decomposition of such an environment is NP-hard [Com90], so we attempt to find a small, but not optimally small, subdivision. The subdivision splitting planes are chosen from among the embedding planes of major occluders, using a heuristic that takes both the size and the interaction among occluders into account. This heuristic involves several parameters whose values determine the relative probabilities that particular occluders will contribute as splitting planes at any given tree depth. Their values are unimportant theoretically, since the superset visibility algorithms are provably correct regardless of the spatial subdivision, and important practically only to the extent that they affect storage and query-time efficiency. (In practice, since we are subdividing architectural models, we first split along horizontal planes to separate individual floors, then introduce rounds of x, y, and z splitting to partition rooms from eachother and from dropped ceilings.)

5.2.1 Splitting Criteria

The occluders in a cell to be split are first partitioned into x-, y-, and z-axial sets, and then sorted by increasing axial offset. Note that there may be multiple occluders coexisting at a single offset. If they overlap, this is reported as an input (i.e., modeling) error, and the program continues; otherwise, their areas are summed and associated with the offset. A candidate offset is then chosen by computing the percentage of cross-sectional cell area obscured (i.e., the summed area of each occluder at this offset, clipped to the cell boundaries). Computing obscuration as a cross- sectional area favors large occluders in early splitting steps, then smaller occluders as smaller cells are split. Any offset that is obscured by more than some threshold is stored as a candidate for splitting. Offsets containing the entire cross section of the cell achieve maximum 100 % obscuration and may be split upon immediately. Ties are broken by choosing the offset closest to the median offset of all other axial occluders of the same type as the candidate, to keep the splitting tree reasonably balanced. Ties among offsets that achieve less than maximum obscuration are similarly broken.

Once this obscuration-based splitting completes, the obscuration criterion is deactivated and a new round of splitting based on "cleaving" is done. An occluder A is said to *cleave* another occluder B if the plane of A partitions B into two 2D pieces. A is a minimally-cleaving occluder if, among all candidate occluders in a given cell, A cleaves no more other occluders than any other candidate. The second splitting round subdivides along minimally-cleaving occluders, again breaking ties with

a median bias, until all occluders have been used.

The splitting rounds are biased to produce good subdivisions for architectural models. Early splitting stages search for z offsets with high obscuration and low cleaving; these typically correspond to floors, ceilings, and other structural boundaries between floors of the model. After partitioning the model horizontally at these offsets, the splitting stages are unbiased, and are as likely to split on small horizontal features (i.e., window sills) as on vertical ones (i.e., door jambs). This produces subdivisions that reflect the architectural features of our test data, without giving up occlusion effects due to the occluders deferred until later splitting rounds; every occluder is eventually split upon and affects portal enumeration.

After these splitting rounds, the *cell volume* and *cell aspect* rounds commence. The cell volume round subdivides those cells that are larger than some specified fraction of the total model volume. The cell aspect round subdivides those cells whose aspect, or ratio of maximum to minimum linear dimension, is greater than a threshold. Note that some splitting method is required that synthesizes its own subdivision planes in "freespace," since after the first two rounds, no occluders intersect the interior of any cell. In both the cell volume and cell aspect rounds, splitting is done along portal boundaries; i.e., split planes are chosen so as to coincide with some (axial) portal edge (Figure 5.1). Among candidate offsets arising from portal edges, the offset that will result in the least-maximum aspect among the resulting children cells is chosen. If no portal edge is suitable, the cell is split in half along its maximum dimension.

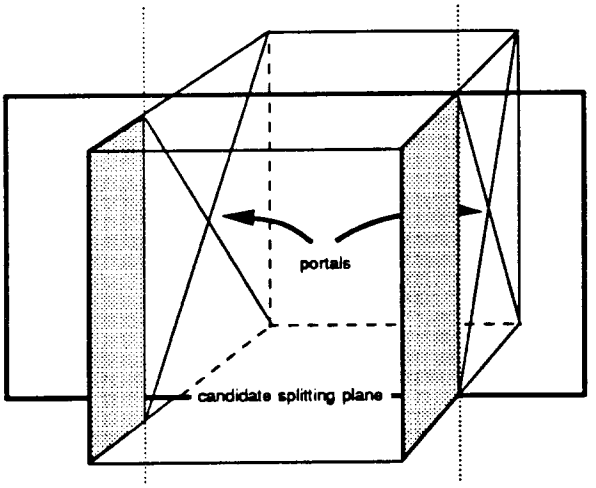


Figure 5.1: Axial splitting planes are chosen to coincide with portal edges.

All of these splitting criteria can be efficiently evaluated for axial input. At the cost of an

initial $O(n \lg n)$ sort, the input polygons can be segregated by normal axis, and sorted by their axial offsets. At each candidate node, the split dimension and abscissa can be determined in time $O(f)$ at each split, where f is the number of faces stored with the node.

The subdivision terminates when three conditions are met: 1) all sufficiently large occluders are coplanar with an axial boundary face of at least one subdivision leaf cell; 2) all cell volumes are sufficiently small; and 3) all cell aspect ratios are sufficiently close to one. We have found that this subdivision procedure yields a tree whose cell structure reflects the "rooms" of our architectural model, and which finds subtle but important visibility-reducing features, such as door frames, window sills, and half-height work area partitions.

5.2.2 Point Location

Point location in 3D k-D trees is a straightforward recursive procedure analogous to that of §4.2.1. The *per-node* search time is constant; a point can be checked for inclusion in a parallelepiped in $O(1)$ time. The check requires six floating-point comparisons, since each coordinate of the 3D point must be compared for inclusion within a one-dimensional range.

5.2.3 Cell Population

Object insertion amounts to a straightforward generalization of point location: the query object is itself an extent, rather than a point; and the result of the query can be a set of leaf cells, rather than a single cell, to which the extent is incident. Again, insertion begins by checking the object bounding box and the k-D tree root node for incidence: a constant-time intersection of three 1D intervals. The recursion then commences as with point location; at the bottom of the recursion, the object is attached to each leaf cell to which its bounding box is determined incident. The algorithmic cost of insertion is therefore $O(c)$, where c is the total number of leaf nodes in the k-D tree that are populated by the object. If a more fine-grained detailed object description is available (for example, a list of its convex hull vertices), these may also be checked against the leaf cell extent before population, at the cost of incurring computation times proportional to object complexity. Finally, as in the 2D case, an incremental algorithm could insert any point on the object, then walk the object boundary, traversing cell boundary faces and populating cells as they are encountered.

5.2.4 Neighbor Finding

Determining the spatial neighbors of a given source cell in a 3D k-D tree involves ascending the tree to find the nodes representing the split planes that induced each source cell boundary face. The tree is then descended until all lateral relatives of the source that share one of its boundary faces are collected. For efficiency, a lazily-constructed neighbor list can be attached to each face of the source cell.

5.2.5 Portal Enumeration

Once the neighbors across a particular source cell boundary face have been found, portal enumeration can proceed. The spatial data structure stores with each cell face a list of incident occluders. The *egress* for this face is the set difference between the axial rectangle of the boundary face, and the incident occluders, themselves axial rectangles (Figure 5.2). Given the axial nature of

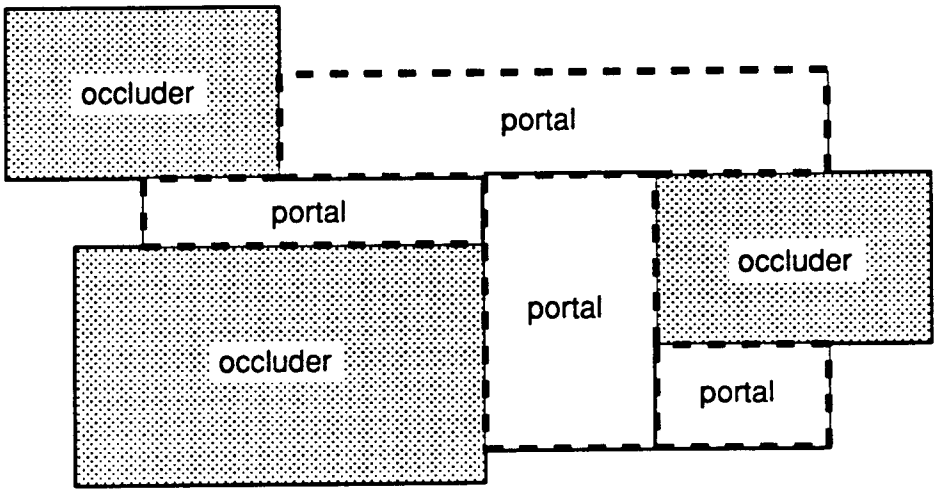


Figure 5.2: Portal enumeration as a set difference of sets of rectangles.

the data, this set difference can be performed efficiently and robustly. Our implementation simply initializes a rectangle list to a single element (the boundary face), then subtracts each occluder extent in turn from the set union of the rectangles on the list. The unoccluded portion of the cell boundary is maintained as a rectangular decomposition throughout the subtraction. The primitive operation subtracts one rectangle from another in constant time, producing at most a constant number of new rectangles (Figure 5.3). When each of the occluders has been subtracted, the remaining rectangles comprise a rectangular decomposition of the boundary face egress. This decomposition is then

compared to the neighbor list for the boundary face, and each rectangle is clipped to any incident faces from the neighbor cell. Finally, a postprocessing pass coalesces adjacent rectangles that lead to the same neighbor cell, and a portal is created for each resulting rectangle/neighbor pair. If this merging generates non-convex portals, the merge is suppressed. The result is a portal list comprised solely of rectangles.

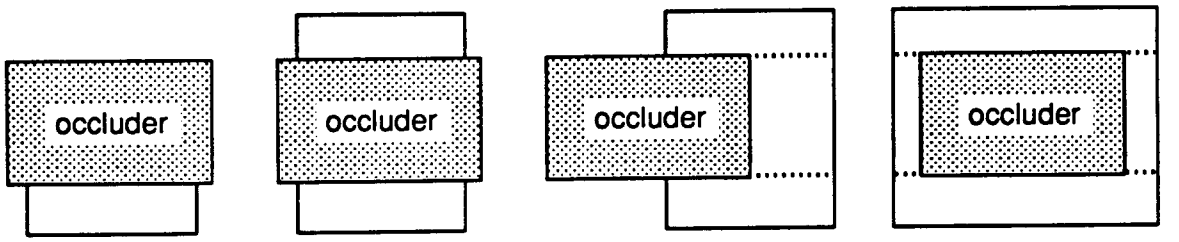


Figure 5.3: The primitive operation subtracts one rectangle from another. Assuming the two rectangles intersect, the result must be zero (not shown), one, two, three, or four new rectangles (left to right, above).

5.3 Static Visibility Operations

Restricting portals to be axial rectangles yields static visibility algorithms that are substantially faster than those for the general case (§8.3). Determining a stabbing line through n axial portals can be done in $O(n)$ time [Ame92, Meg91], whereas the fastest known algorithm for stabbing general portals requires $O(n^2)$ time. Determining a linear bound on the antipenumbra through n axial portals requires $O(n \lg n)$ time, compared to $O(e^2)$ time for general portals, where e is the total number of edges comprising the portal sequence. Finally, computing the cell-to-object visibility can be done in $O(1)$ time per object for axial portal sequences, whereas the general case requires $O(e)$ time per object.

5.3.1 Cell-to-Cell Visibility

Establishing cell-to-cell visibility requires a stabbing line algorithm for axial portal sequences. We have developed and implemented an $O(n \lg n)$ time algorithm that determines a stabbing line through a collection of n axial rectangles, or determines that no such stabbing line exists.

Before considering the problem in three dimensions, we consider the two dimensional analog. In the plane, we are looking for an oriented line that intersects each of n vertical or horizontal line

segments. In addition, we are given the direction in which the stabbing line must traverse the line segments: either bottom to top, or top to bottom for the horizontal line segments, and either left to right, or right to left for the vertical line segments. This is equivalent to the following problem. Given two sets of points, $\{p_i\}$ and $\{q_i\}$, find a line that passes above the $\{p_i\}$ and below the $\{q_i\}$. Such a line is called a feasible line.

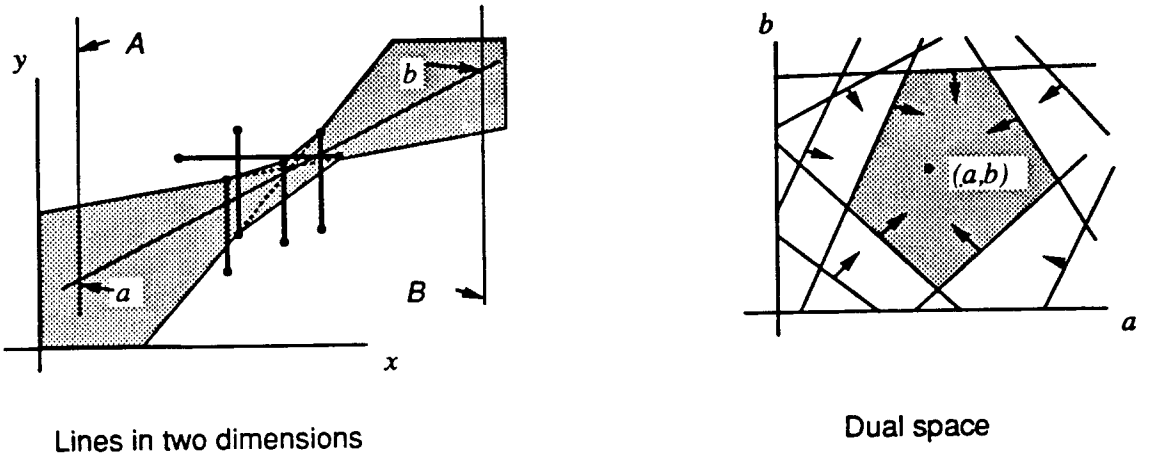


Figure 5.4: Lines in $\mathbf{R}^2$ and their dual representation.

While the set of feasible lines appears as an hourglass-shaped shaded region, as depicted in Figure 5.4, in a dual space that suitably parametrizes all lines, the set of feasible lines is a convex polygon. To see this, parametrize the feasible lines by their intercepts, a and b, with two vertical lines, A and B, respectively (Figure 5.4). Each line in (x, y) space corresponds to a point in (a, b) space. The set of lines in (x, y) space intersecting a point in (x, y) space corresponds to a line in (a, b) space. The set of lines passing above a point corresponds to a half plane in (a, b) space, as indicated by the arrows in Figure 5.4. The set of lines that pass above the $\{p_i\}$ and below the $\{q_i\}$ is then the intersection of these halfspaces, a convex polygon. Each vertex of this polygon corresponds to a line in (x, y) space. Since there are $2n$ points in (x, y) space there are $2n$ half planes in (a, b) space. Thus, the polygon has at most $2n$ vertices.

Therefore, in two dimensions, a feasible line can be found in linear time using a linear programming algorithm [Meg83, Sei90b].

Stabbing Isothetic Rectangles in R^3

In three dimensions we wish to solve the following problem: given a set S of n oriented, axial rectangles in R^3, determine if there is a line that intersects all of them (Figure 5.5). We first present an $O(n^2)$ algorithm and then show how various improvements can be made to it so that it runs in $O(n \lg n)$ time. We begin by making the following observation: If there is any feasible line, then there must be a feasible line intersecting four rectangle edges. This is the case since four scalar degrees of freedom describe a line in 3D. Let X, Y, and Z be the set of x-aligned, y-aligned, and z-aligned edges, respectively. Since each edge belongs to one of X, Y, or Z, then if there is any feasible line, there must be a feasible line intersecting two edges from the same set. This is the key to exploiting the fact that the edges come from three sets of parallel edges.

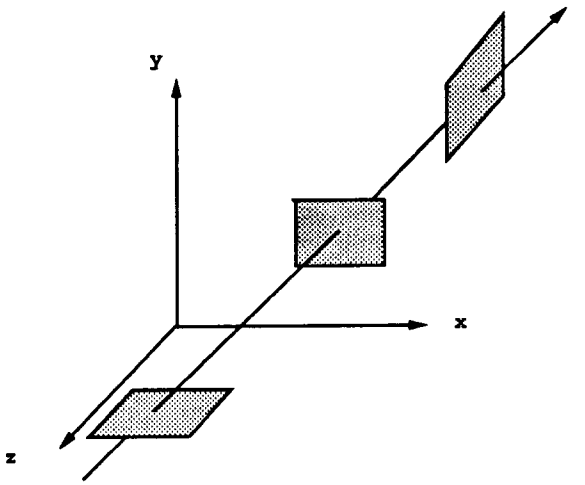


Figure 5.5: An axial portal sequence and stabbing.

A priori, we should consider all pairs of x-aligned edges, all pairs of y-aligned edges, and all pairs of z-aligned edges. However, if there are m x-aligned edges there will be at most m pairs defining planes that contain *any* feasible lines (Figure 5.6). Consider a line L in the plane defined by the x-aligned edge pair A and B. A plane Q perpendicular to the x axis is shown, along with the intersections of Q with the x-aligned edges. Also shown is l, the projection of L onto Q. Note that L will be feasible with respect to the x-aligned edges if and only if l is feasible with respect to the intersection points on Q. If we can determine the pairs of intersection points that define feasible lines in Q, then we have also determined the pairs of x-aligned edges that are feasible with respect to the other x-aligned edges. This is exactly the two-dimensional problem of the previous section. By explicitly constructing the polygon shown in Figure 5.4, only a small list of pairs of x-aligned

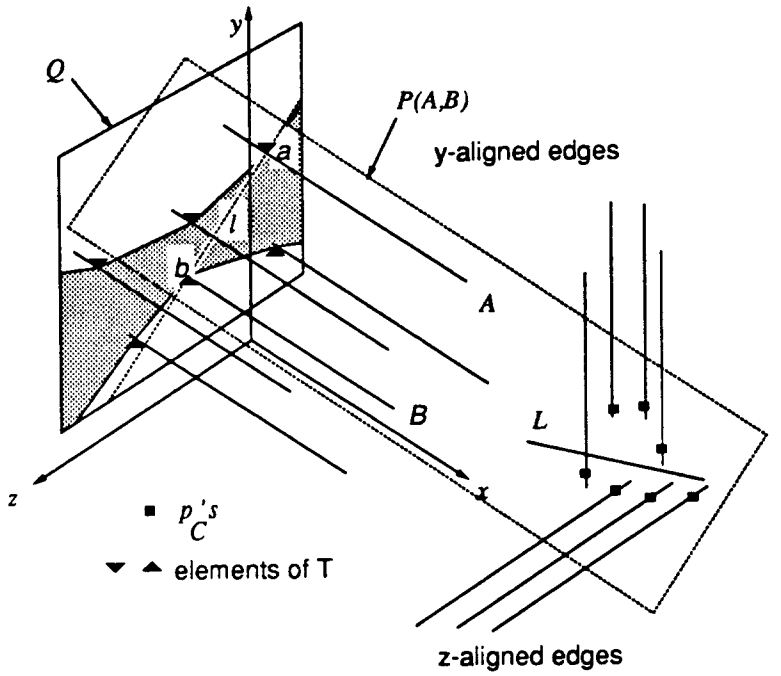


Figure 5.6: Reducing three-dimensional stabbing to a series of two-dimensional problems.

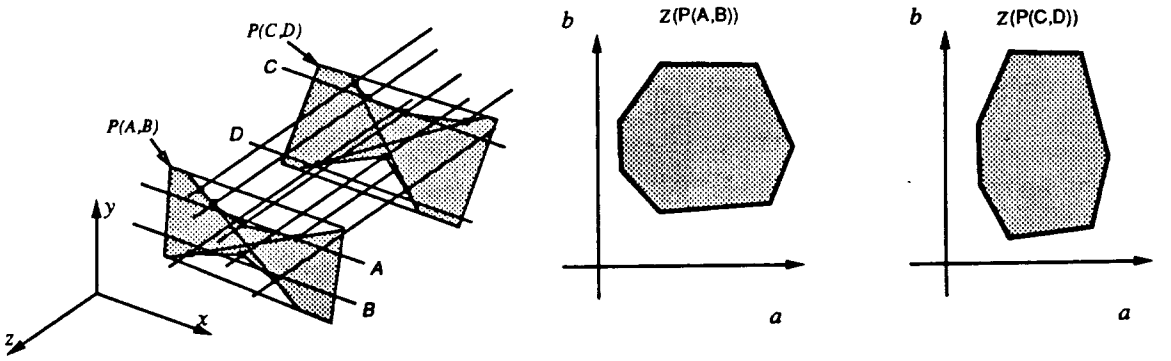


Figure 5.7: The structure of the region of feasible lines is independent of P.

edges must be consider. In fact, the number of pairs will be no greater than the number of x-aligned edges. Clearly, this holds in the y and z directions as well.

We can devise an $O(n^2)$ algorithm as follows: For each of the directions x, y, and z, construct a plane Q perpendicular to the current direction. Construct the intersection points of Q with the lines parallel to the current direction. Construct the convex hull in the dual space of the lines in Q feasible with respect to the points on Q. Each vertex of this convex hull corresponds to a line l in Q. For each l, construct the plane P parallel to the current direction and containing l. Calculate the intersection points of the lines from the other two directions with the plane P. Solve the resulting two-dimensional stabbing problem as a linear programming problem. In pseudo code:

Stabbing_Line_1 (X, Y, Z)

```
(1)   S = X ∪ Y ∪ Z
(2)   for G = X, Y, Z
(3)       F = S \ G
(4)       let Q be a plane perpendicular to the lines in G
(5)       let T be the intersections of G with Q
(6)       let H be the convex hull (in the dual space) of the
              lines in Q feasible with respect to T
(7)       for each corner l ∈ H
(8)           determine the plane P defined by l and the
                  direction of the lines in G.
(9)           for each C ∈ F
(10)              let p_C be the point where C intersects P
(11)          endfor
(12)          use a linear programming algorithm to determine if
                  there exists an L in P satisfying the p_C's.
(13)          if there exists an L return L
(14)      endfor
(15)  endfor
(16)  return infeasible
```

This algorithm can be improved to run in $O(n \lg n)$ time. Consider again the case of pairs of x-aligned edges ($G = X$ on line 2). Let Y'_P be the intersection of the lines in Y with the plane P,

and similarly define Z'_P. Let $\mathcal{Y}(P)$ be the convex hull of the duals of the lines in P feasible with respect to Y'_P, and similarly define $\mathcal{Z}(P)$. In the inner loop we are testing $|X|$ times whether $\mathcal{Y}(P)$ and $\mathcal{Z}(P)$ intersect. Each test takes $O(|Y|+|Z|)$ time if we use a linear programming algorithm as in **Stabbing_Line_1()**. Coordinatize the points in P by their x and z coordinates. In this coordinate system $\mathcal{Y}(P)$ is independent of P; it is constant. Let a typical line l_i in Z be defined by $y = y_i$ and $x = x_i$. If the equation of P is $by + cz + d = 0$, then the coordinates of the intersections of the l_i with P are $(x_i, -(c/b) - (a/b)y_i)$. Thus the various polygons $\mathcal{Z}(P)$ parametrized by P are all affine transformations of one another. Consequently, the *structure* of $\mathcal{Y}(P)$ and $\mathcal{Z}(P)$ do not depend on P. As depicted in Figure 5.7, $\mathcal{Z}(P)$ is merely stretched or shrunk in the z direction.

In [DS89] an $O(\lg\ n)$ algorithm is described for testing whether convex polygons comprising n edges in total intersect. This algorithm requires $O(n\ \lg\ n)$ time to build a query structure. Since the structure of $\mathcal{Y}(P)$ and $\mathcal{Z}(P)$ do not change as P is changed, we can build the query structure once and reuse it for each of the $|X|$ planes P. Thus, we can compute whether $\mathcal{Y}(P)$ intersects $\mathcal{Z}(P)$ for a particular P in $O(\lg(|\mathcal{Y}|+|\mathcal{Z}|))$ time. The resulting $O(n\ \lg\ n\ )$ algorithm is as follows:

Stabbing_Line_2 (X, Y, Z)

(1) *Let* $\mathcal{S}[0] = \mathcal{X}()$ *(the logarithmic query structure for x)*

(2) *Let* $\mathcal{S}[1] = \mathcal{Y}()$ *(the logarithmic query structure for y)*

(3) *Let* $\mathcal{S}[2] = \mathcal{Z}()$ *(the logarithmic query structure for z)*

(5) **for** $i = 0, 1, 2$

(4) **for** *each vertex L in* $\mathcal{S}[i]$ *(L is a line)*

(6) *determine the plane P defined by L and the direction of the lines in* $\mathcal{S}[i]$*;*

(7) **if** $\mathcal{S}[(i+1) \bmod 3](P)$ *intersects* $\mathcal{S}[(i+2) \bmod 3](P)$ **return** *the line corresponding to the point of intersection*

(8) **endfor**

(9) **endfor**

(10) **return** *infeasible*

Finally, we note that two $O(n)$ algorithms have recently been proposed to find stabbing lines through isothetic rectangles [Ame92, Meg91]. We have implemented [Meg91], but because of the rather large constants it entails we found it no faster in practice than our $O(n \lg n)$ algorithm, for typically occurring portal sequences (with length up to about forty).

5.3.2 Cell-to-Region Visibility

Casting the sightline search as a graph traversal yields a simple method for computing the partially visible portion of each reached cell. The traversal *orients* each portal encountered, since the portal is crossed in a known direction. Axial portals in three dimensions can be analogously oriented, then decomposed into at most three pairs of hourglass constraints, one from each collection of portal edges parallel to the x, y, and z axes.

Figure 5.8 depicts one such portal sequence in 3D, and the x axis-parallel and z axis-parallel constraints arising from its decomposition. These have been projected into "lefthand" and "right-hand" points onto x = constant and z = constant planes, respectively. (The y constraints and projections are not shown.) These lines induce an *hourglass*-shaped 3D volume that bounds all lines stabbing the sequence. When this volume is cut at the entrance portal to the reached cell, a half-infinite volume that we call a *visibility funnel* is produced. This funnel is, in general, bounded by quadric surfaces (§8.3); however, we can efficiently compute a constant-complexity polyhedral volume enclosing it simply by intersecting three constant-complexity polyhedral wedges, one from each of the axis-parallel decompositions. The polyhedral wedges are comprised of planes constructed from the interior hourglass edges for each of the x, y, and z constraint sets (cf. Figure 4.7). Since each of these hourglasses bounds the set of lines for its associated constraints, the aggregate set of planes must bound all sightlines emanating from the original portal. We call the intersection of this linearized funnel with the reached cell the source's *linearized visibility volume* in the reached cell. Later, we show how to compute the source's exact (i.e., quadratically bounded) visibility volume for any sequence of convex portals in three dimensions.

Suppose the stab-tree computation reaches a cell via some portal sequence of length n. If $n = 1$, the reached cell is entirely visible to the generalized observer. Otherwise, we decompose the sequence into three sets of at most n axial lines each. From each set, the two bounding planes can be constructed in $O(n)$ time. We then compute, in constant time, the common intersection of the bounding plane halfspaces with the parallelepipedal extent of the reached cell, using a 3D convex hull algorithm. The resulting volume contains all lines stabbing the portal sequence, and intersects all objects visible from the source cell through the portal sequence. Figure 5.9 depicts a source cell and the linearized visibility volumes constructed within each reached cell, one for each occurrence of the cell in the source's stab tree.

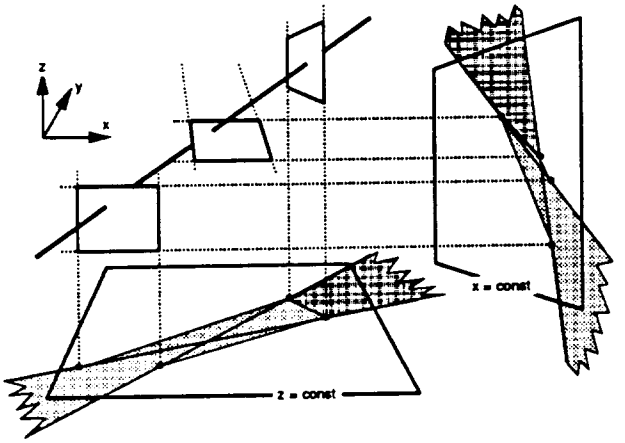


Figure 5.8: An axial portal sequence can be decomposed into three sets of 2D hourglass constraints.

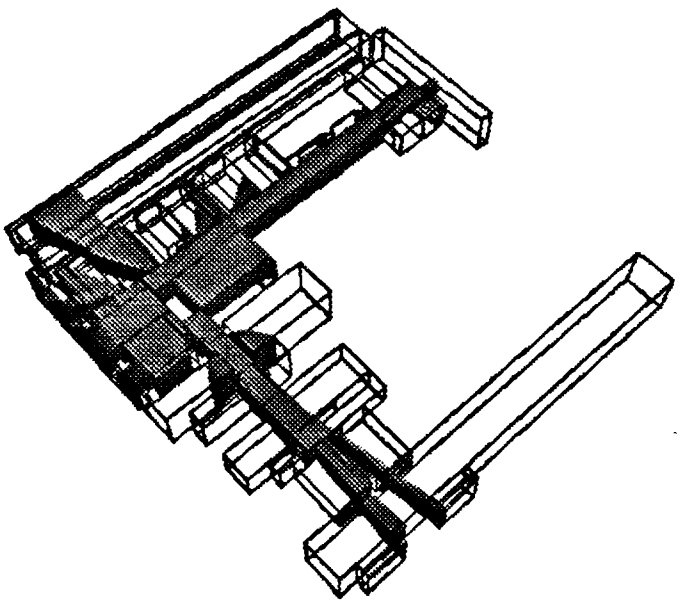


Figure 5.9: The linearized visibility volumes emanating from an axial source cell.

5.3.3 Cell-to-Object Visibility

The cell-to-object visibility can also be cast as an action to be applied whenever the sightline DFS reaches a cell via a particular portal sequence. The DFS maintains three active hourglasses, each of which induces two planes beyond the final portal in the sequence (cf. Figure 5.8), for a total of six planes. Each of the planes can be oriented so that its positive halfspace includes the bundle of lines stabbing the portal sequence (clipped to rays emanating from the final portal). The common interior of these positive halfspaces can be checked for incidence with each detail object bounding box in constant time, since the associated linear program has at most eighteen constraints: six from the hourglass extrema, six from each axial object bounding box, and six from the axial cell bounding box. Note that the cell bounding planes are only superfluous when the object is completely contained inside the reached cell. If the object is only partially incident on the reached cell, these six halfspaces must be included in the linear program; otherwise, the ray bundle and the object bounding box might have a computed intersection outside of the reached cell.

These constraints are cast as a three-dimensional linear program as follows. Suppose there are six hourglass planes B_k, six planes O_k whose positive halfspaces intersect in the object bounding box, and (possibly) six planes whose positive halfspaces intersect in the axial cell. There are at least twelve and at most eighteen total planes h_k. The bounding box is incident on the interior of the hourglasses if there exists a point $\mathbf{p} = (p_x, p_y, p_z, 1)$ such that

$$\mathbf{h}_k \cdot \mathbf{p} \geq 0, \qquad \text{for all } k, \tag{5.1}$$

where $\mathbf{h}_k \cdot \mathbf{p}$ denotes the signed distance of the point $\mathbf{p}$ from the plane $\mathbf{h}_k$. This is a three-dimensional linear programming problem, solvable in time linear in the number of constraints (here, at most 18). We shall see that the linear program is analogous, but two-dimensional, when the observer position is known.

5.4 Dynamic Visibility Queries

As in the static phase, axial portals yield on-line culling algorithms that are significantly faster than those for the general case (§8.4). In particular, during the eye-to-cell, eye-to-region, and eye-to-object computations, each newly encountered portal and object bounding box can be checked for an eye-centered sightline in constant time, and each potentially visible volume computed in constant time, rather than time proportional to the length and edge complexity of the portal sequence reaching the cell.

5.4.1 Observer View Variables

In three dimensions, the observer's view frustum can be entirely defined by a view direction; azimuthal and altitudinal half-angles; and an "up-vector" that specifies the remaining "twist" degree of freedom. In practice, the frustum is represented as four implicit plane equations whose positive halfspaces intersect in the view volume; this form is particularly well-suited to linear programming formulations. In general, the planes bounding the frustum are not spanned by axial lines and the eye, since they may assume general values as functions of the input half-angles and the up-vector.

Near and far planes bounding the frustum may also be specified; in what follows, we assume that the near plane is at the eye, and the far plane is at infinity.

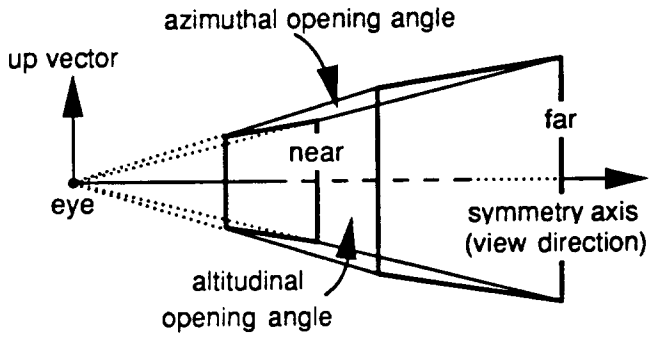


Figure 5.10: Three-dimensional observer view variables.

5.4.2 Eye-to-Cell Visibility

For a cell to be visible, some portal sequence to that cell must admit a sightline that lies inside the view frustum and contains the eyepoint. Retaining the stab tree permits an efficient implementation of this criterion. During the stab tree DFS rooted at the source cell, each encountered portal is again decomposed into its constituent axial constraints (Figure 5.11). Projecting the set of feasible lines onto each principal plane yields a half-infinite "wedge" of lines, beginning at the plane of the final portal.

Two extremal lines are maintained on each of the three principal planes. As each portal is encountered, it is decomposed, and the set of extremal lines is suitably narrowed in the appropriate axial plane, using the 2D method described in §4.4.2. If the extremal lines exclude the portal entirely, the DFS is immediately terminated. Otherwise, the three resulting pairs of lines together imply six constraint planes in three-space. These six planes are combined with the four planes defining the

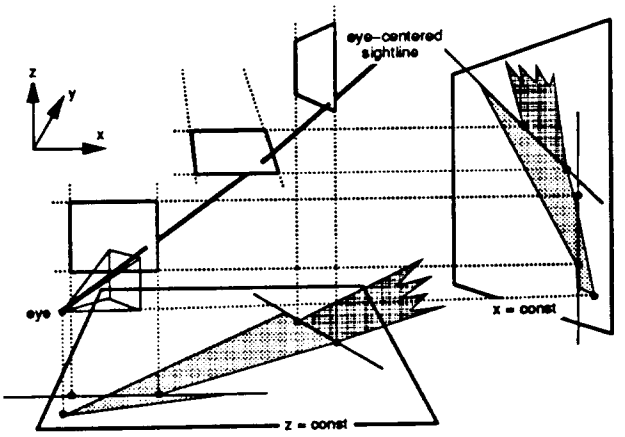


Figure 5.11: Decomposing axial portals into constituent axial eye-centered constraints.

view frustum to produce at most ten linear constraints on the existence of a stabbing line through the eye. These constraints can be cast as a linear program of three-coefficient constraints, as in §8.4.2. We can therefore examine the ten linear constraints for a feasible solution (a stabbing line through the eye and the portal sequence) in constant time. If the linear program fails to find a stabbing line through the eye, the most recent portal is impassable, and the active branch of the DFS terminates.

The depth-first nature of the search ensures that portal sequences are assembled incrementally. Further, since each newly encountered portal can be examined for a solution in constant time, a portal sequence of length n can be checked for sightlines in $O(n)$ time.

5.4.3 Eye-to-Region Visibility

The eye-to-region computation in axial environments is accomplished by augmenting the eye-to-cell DFS. When the DFS succeeds in stabbing a portal with an eye-based sightline, the active constraint set is exactly the four frustum planes, plus at most six planes due to any active axial portal edges. Thus, the potentially visible volume in the reached cell is bounded by at most sixteen planes: four from the frustum, six from the portal sequence, and six from the parallelepipedal cell. A 3D convex hull algorithm computes the convex polyhedron that is the common interior of all sixteen halfspaces in constant time.

5.4.4 Eye-to-Object Visibility

Eye-to-object visibility can also be formulated as a DFS augmentation. Surprisingly, examining each object for an eye-centered stabbing line can be done with at most ten halfspaces, in constant

time. This is due to the fact that the object bounding box is axial, and induces at most six silhouette planes containing the eye (Figure 5.12). The set of silhouette planes is assembled via a table-lookup on the $26 = 3^3 - 1$ possible positions of the eye with respect to the three infinite axial slabs whose intersection is the box. Each of the three pairs of silhouette planes can be compared to the three

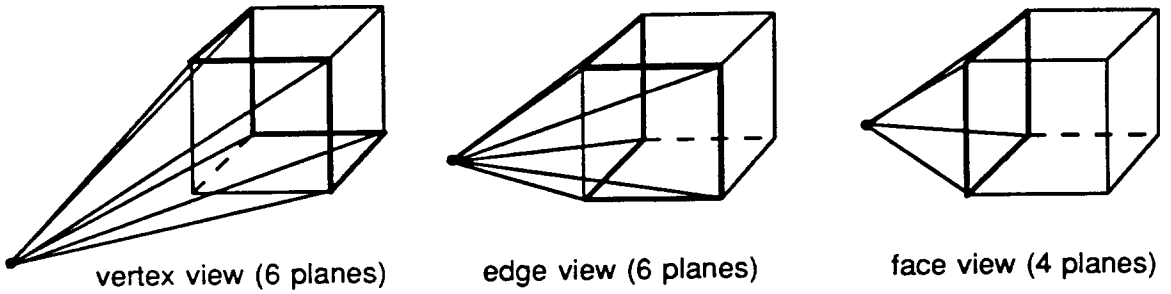


Figure 5.12: The four to six faces of an eye-centered bounding box pyramid.

pairs of portal-induced constraint planes, just as if the silhouette planes arose as portal boundaries. The result is a two-dimensional linear program with at most ten normal constraints (analogous to that of Figure 4.12, §4.4.2), solvable in constant time. As in the cell-to-object computation, if the object tested is not entirely contained in the reached cell, six more constraints must be added to the linear program to ensure that the computed sightline intersects the object bounding box inside the cell.

Analogously to the cell-to-object case, there is an alternative method for computing eye-to-object visibility. Rather than performing an explicit sightline test per-object, we can construct a constant-size description of the eye-to-region visibility (a polyhedron) in the reached cell, and compare objects to the interior of this polyhedron. This construction is attractive in that it discards irrelevant halfspace constraints. However, we do not use this method for three reasons. First, the polyhedron construction is a 3D operation and is hard to implement both efficiently and robustly. Second, the resulting polyhedron/bounding box intersection check requires a 3D linear program, rather than 2D as in the sightline check, and the linear programming algorithm we use has $O(d!n)$ time complexity; the result is that the linear program slows down by a factor of three in the worst case (for which no superfluous constraints are eliminated). Finally, since the sightline check is constant-time in the first place, we can improve its efficiency by at most a constant factor.

Chapter 6

Line Coordinates

The primitive operations of stabbing portal sequences and computing antipenumbrae of portal sequences in three dimensions require a substantial digression into the behavior of lines in space, involving new coordinate systems and high-dimensional geometric objects and algorithms. We begin by reviewing Plücker coordinates [Som59, Sto89], a useful tool for manipulating generally-oriented lines in three dimensions (in our algorithms, portal edges and stabbing lines).

6.1 Plücker Coordinates

We use the Plücker coordinatization [Som59, Sto89] of directed lines in three dimensions. Any ordered pair of distinct points $p = (p_x, p_y, p_z)$ and $q = (q_x, q_y, q_z)$ defines a directed line ℓ in three dimensions. This line corresponds to a projective six-tuple $\Pi_\ell = (\pi_{\ell 0}, \pi_{\ell 1}, \pi_{\ell 2}, \pi_{\ell 3}, \pi_{\ell 4}, \pi_{\ell 5})$, each component of which is the determinant of a 2×2 minor of the matrix

$$\begin{pmatrix} p_x & p_y & p_z & 1 \\ q_x & q_y & q_z & 1 \end{pmatrix}. \tag{6.1}$$

We use the following convention dictating the correspondence between the minors of Equation 6.1 and the $\pi_{\ell i}$:

$$\begin{aligned} \pi_{\ell 0} &= p_x q_y - q_x p_y \\ \pi_{\ell 1} &= p_x q_z - q_x p_z \\ \pi_{\ell 2} &= p_x \quad - q_x \\ \pi_{\ell 3} &= p_y q_z - q_y p_z \end{aligned}$$

$$\pi_{\ell 4} = p_z - q_z$$
$$\pi_{\ell 5} = q_y - p_y$$

(this order was adopted in [Pel90b] to produce positive signs in some useful identities involving Plücker coordinates).

If a and b are two directed lines, and Π_a, Π_b their corresponding Plücker *duals*, a relation *side*(a, b) can be defined as the permuted inner product

$$\Pi_a \odot \Pi_b = \pi_{a0}\pi_{b4} + \pi_{a1}\pi_{b5} + \pi_{a2}\pi_{b3} + \pi_{a4}\pi_{b0} + \pi_{a5}\pi_{b1} + \pi_{a3}\pi_{b2}. \quad (6.2)$$

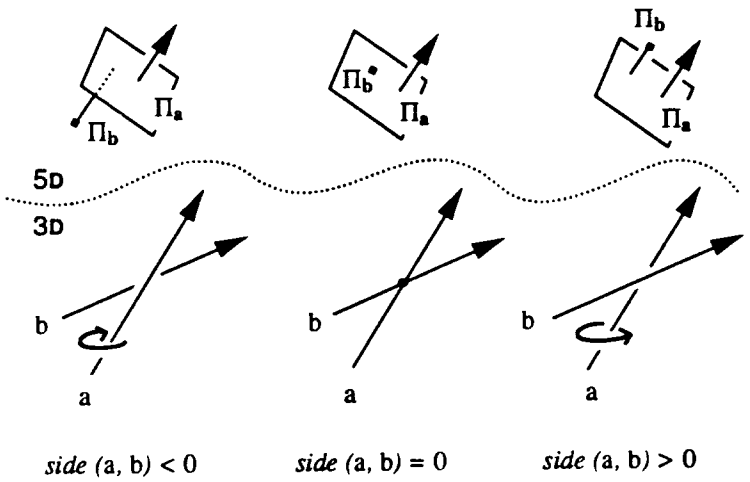


Figure 6.1: The right-hand rule applied to *side*(a, b).

This sidedness relation can be interpreted geometrically with the right-hand rule (Figure 6.1): if the thumb of one's right hand is directed along a, then *side*(a, b) is positive (negative) if b goes by a with (against) one's fingers. If lines a and b are coplanar (i.e., intersect or are parallel), *side*(a, b) is zero. Alternatively, Equation 6.2 can be considered to compute (a positive multiple of) the signed volume of the tetrahedron formed by the four endpoints of the two specified line segments. When the line segments intersect or are parallel, the tetrahedron has zero volume.

The six-tuple Π_ℓ can be treated either as a homogeneous point in 5D, or (after suitable permutation via Equation 6.2) as the coefficients of a five-dimensional hyperplane. The advantage of transforming lines to Plücker coordinates is that detecting incidence of lines in 3D becomes equivalent to computing the inner product of a 5D homogeneous point (the dual of one line) with a 5D hyperplane (the dual of the other).

Plücker coordinates simplify computations on lines by mapping them to points and hyperplanes, which are familiar objects. However, although every directed line in 3D maps to a point in Plücker coordinates, not every point in Plücker coordinates corresponds to a *real line*. Only those points Π satisfying the quadratic relation

$$\Pi \odot \Pi = 0 \tag{6.3}$$

correspond to real lines in 3D. All other points correspond to *imaginary lines* (i.e., lines with complex coefficients).

The six Plücker coordinates of a real line are not independent. First, since they describe a projective space, they are distinct only to within a scale factor. Second, they must satisfy Equation 6.3. Thus, Plücker coordinates describe a four-parameter space. This confirms basic intuition: one could describe all lines in three-space in terms of, for example, their intercepts on two standard planes.

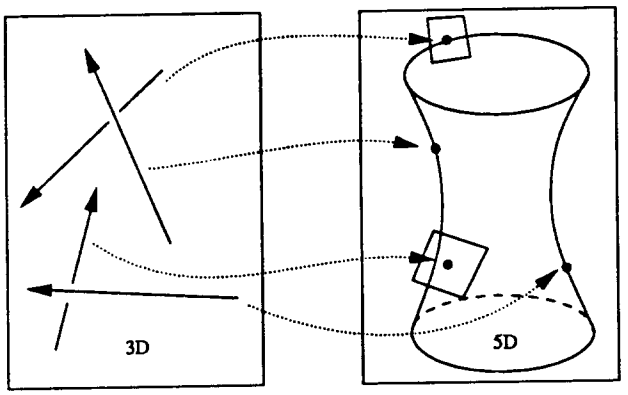


Figure 6.2: Directed lines map to points on, or hyperplanes tangent to, the Plücker surface.

The set of points in 5D satisfying Equation 6.3 is a quadric surface called the *Plücker surface* [Som59]. One might visualize this set as a four-dimensional ruled surface embedded in five dimensions, analogous to a quadric hyperboloid of one sheet embedded in three-space (Figure 6.2).

Henceforth, we use the notation $\Pi : \ell \rightarrow \Pi(\ell)$ to denote the map Π that takes a directed line ℓ to the Plücker point $\Pi(\ell)$, and the notation $\mathcal{L} : \Pi \rightarrow \mathcal{L}(\Pi)$ to denote the map that takes any point Π on the Plücker surface and constructs the corresponding real directed line $\mathcal{L}(\Pi)$. Any plane or hyperplane h is *oriented* in the sense points to one side of the hyperplane have positive signed distance from h, points to the other side have negative signed distance from h, and points on the hyperplane have distance zero from h. We denote the closed nonnegative halfspace of h as h^+, and say that a point in this halfspace is *on* or *above* h.

6.2 Degrees of Freedom in Line Space

Referring solely to the dimensionality of line space and to Equation 6.3, we can make several useful arguments about the degrees of freedom (DOFs) inherent in the specification and manipulation of three-dimensional lines.

Suppose five lines are specified in 3D (Figure 6.3), and we wish to determine whether there exists a sixth line incident on the given five. This question can be answered generically with a DOF argument. Starting with five degrees (the dimensionality of the problem), each of the five specified lines induces a hyperplane in Plücker space, and removes a DOF, leaving zero degrees of freedom, i.e., a point. The Plücker quadric removes one more degree, resulting in -1 degrees of freedom, an overspecified problem that generally has no solution. In other words, the five hyperplanes determine a unique point, which almost always lies off of the Plücker surface. In general, therefore, the 3D solution lines have complex coefficients.

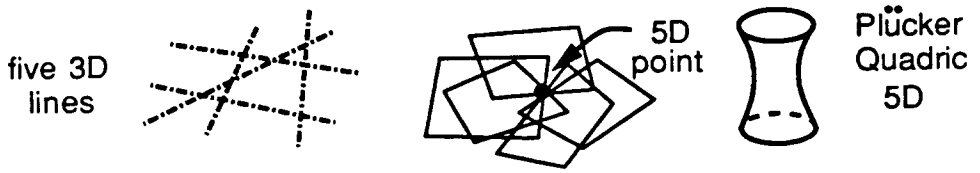


Figure 6.3: Generically, no real line is incident to five given lines.

A similar analysis applies to the case of *four* lines. Given four lines, we wish to determine the set of lines incident on the four. An analogous DOF argument consumes four each for the planes, leaving a single DOF, or a line in 5D. This line generally intersects the Plücker quadric in two places, yielding two points (Figure 6.4). The points lie on the surface, by construction, yielding two real solutions; that is, generically, two lines through the four input lines.

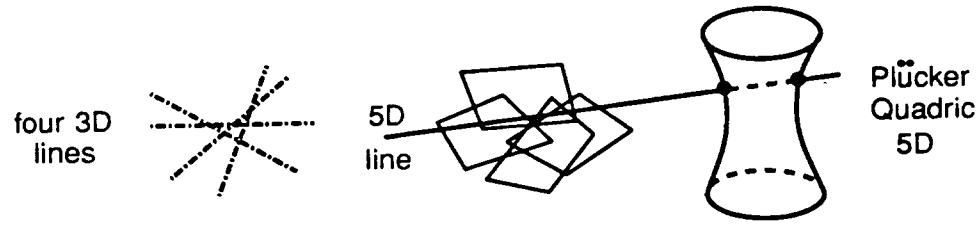


Figure 6.4: Generically, two real lines are incident to four given lines.

The surprising result is that, through four lines in three dimensions, there generally exist *two* other lines. This single fact captures the special behavior of lines in three dimensions, and the

profound difference between their essentially quadratic nature and the essentially linear nature of points and planes.

Finally, given three lines, we can describe the family of lines incident on the specified lines. The classical result is that these lines form a ruled surface in 3D [Som59]. In five dimensions, the DOF argument starts with five DOFs, and removes three, one for each line The resulting two DOFs span a 2D-plane in 5D (Figure 6.5). The intersection of a plane and a quadric is a conic. The conic is a 1-parameter curve in 5D, isomorphic to the continuous set of lines comprising the ruled surface in 3D.

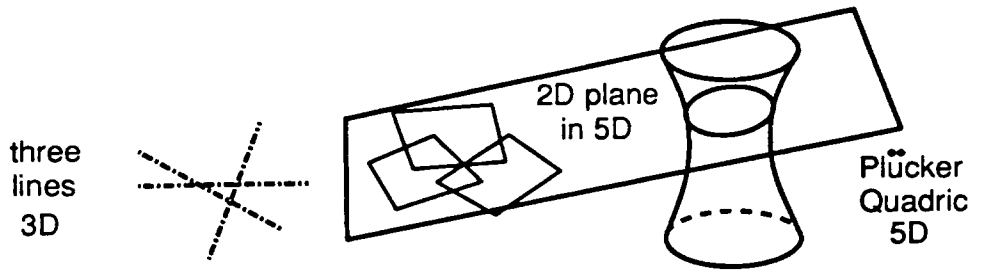


Figure 6.5: Generically, a one-parameter family of lines is incident to three given lines.

6.3 Computing the Incident Lines

Suppose we are given four lines $l_k, 0 \leq k \leq 3$ in 3D, and wish to compute all further lines that are *incident* on, or intersect, the l_k. By the sidedness relation (Equation 6.2), we wish to find all lines s such that side$(s, l_k) = 0$ for all k. Each line l_k, under the Plücker mapping, is mapped to a hyperplane Π_k in P^5. Four such hyperplanes intersect in a *line* L in 5D. In Plücker coordinates, L contains the images under of all lines, real or imaginary, incident on the four l_k. To find the *real* incident lines in 3D, we must intersect L with the Plücker quadric (Figure 6.6). As in three space, a line-quadric intersection may contain 0, 1, 2, or (since the quadric is ruled) an infinite number of points.

Thus the incidence computation has two parts. The first is an intersection of four hyperplanes Π_k, to form a line (the *null space*) of the Π_k. The second is an intersection of this line with a quadric surface to produce a discrete result. We have implemented this computation in the C language using a FORTRAN singular value decomposition package from Netlib [DG87]. Figure 6.7 depicts the algorithm applied to four generic lines. Note that the input lines (thick) are mutually skew, and that each of the two solution lines (thin) pierces the input lines in a distinct order.

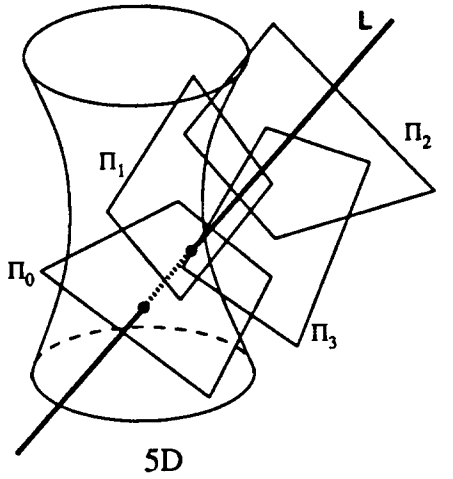


Figure 6.6: The four Π_k determine a line to be intersected with the Plücker quadric.

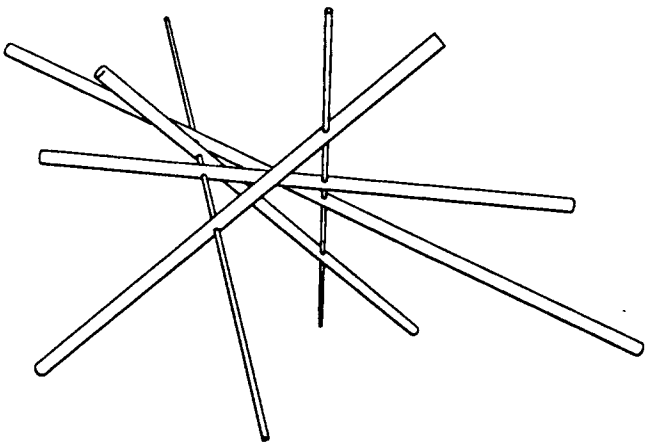


Figure 6.7: The two lines incident through four generic lines in 3D.

The first part of the incidence computation is formulated as a singular value decomposition. Each of the lines l_k corresponds to a six-coefficient hyperplane Π_k under the Plücker mapping. Thus, we must find the null-space of the matrix

$$\mathrm{M} = \begin{pmatrix} \Pi_{04} & \Pi_{05} & \Pi_{03} & \Pi_{00} & \Pi_{01} & \Pi_{02} \\ \Pi_{14} & \Pi_{15} & \Pi_{13} & \Pi_{10} & \Pi_{11} & \Pi_{12} \\ \Pi_{24} & \Pi_{25} & \Pi_{23} & \Pi_{20} & \Pi_{21} & \Pi_{22} \\ \Pi_{34} & \Pi_{35} & \Pi_{33} & \Pi_{30} & \Pi_{31} & \Pi_{32} \end{pmatrix}.$$

By the singular value decomposition theorem [GV89], this 4×6 real matrix can be written as the product of three matrices, $\mathrm{U} \in \mathrm{R}^{4\times4}$, $\Sigma \in \mathrm{R}^{4\times6}$, and $\mathrm{V} \in \mathrm{R}^{6\times6}$, with U and V orthogonal, and Σ zero except along its diagonal:

$$\mathrm{M} = \mathrm{U}\Sigma\mathrm{V}^T = \begin{pmatrix} u_{00} & \cdots & u_{03} \\ \vdots & & \vdots \\ u_{30} & \cdots & u_{33} \end{pmatrix} \begin{pmatrix} \sigma_0 & & & 0 & 0 \\ & \sigma_1 & 0 & 0 & 0 \\ & & \sigma_2 & 0 & 0 \\ & 0 & & \sigma_3 & 0 & 0 \end{pmatrix} \begin{pmatrix} v_{00} & \cdots & \cdots & v_{05} \\ \vdots & & & \vdots \\ & & & \\ \vdots & & & \vdots \\ v_{50} & \cdots & \cdots & v_{55} \end{pmatrix}.$$

The σ_i can be ordered by decreasing magnitude, and comprise the *singular values* of M; the number of non-zero σ_i equals the rank of M. Each zero or elided σ_i corresponds to a row of V; collectively, these rows form the *null space* of M.

If $\sigma_3 \neq 0$ then the null space of M is spanned by the vectors comprising the last two rows of V. Call these rows F and G.

Consider the map $\Lambda : \mathrm{P} \rightarrow \mathrm{P}^5$ defined by

$$\Lambda(t) \equiv \mathrm{F} + t\mathrm{G}.$$

The null space property implies that

$$\Lambda(t) \odot \Pi_k = 0, \qquad 0 \leq k \leq 3, \forall t.$$

Since Λ is injective, it is an isomorphism between P and the set of all lines (real and imaginary) tight on the l_k. Thus there is a one-to-one correspondence between the real lines incident to the l_k and the roots of

$$\Lambda(t) \odot \Lambda(t) = 0.$$

This is a quadratic equation in t:

$$\mathbf{F} \odot \mathbf{F}t^2 + 2\mathbf{F} \odot \mathbf{G}t + \mathbf{G} \odot \mathbf{G} = 0,$$

or

$$at^2 + 2bt + c = 0,$$

where $a = \mathbf{F} \odot \mathbf{F}$, $b = \mathbf{F} \odot \mathbf{G}$, and $c = \mathbf{G} \odot \mathbf{G}$. This is an "even" quadratic with the discriminant $b^2 - ac$, rather than the more familiar $b^2 - 4ac$ [dV91].

If $a^2 + b^2 + c^2 = 0$, all t are solutions, and the null-space line in 5D lies in the (ruled) Plücker quadric. In this case any linear combination of $\mathbf{F}$ and $\mathbf{G}$ corresponds to a real line incident on the l_k.

If $b^2 - ac < 0$ there are no real lines incident on the l_k. If $b^2 - ac = 0$, there is a single line incident on the l_k given by $\mathbf{L}(t), t = \frac{-b}{a}$. If $b^2 - ac > 0$ there are two real lines $\mathbf{L}(t)$ corresponding to

$$t = \frac{-b \pm \sqrt{b^2 - ac}}{a}. \tag{6.4}$$

It may be the case that some of the σ_i are zero. If n of the σ_i are zero, then the set of real and imaginary lines incident on the l_k are spanned by the vectors comprising the last $n + 2$ rows of $\mathbf{V}$. This set of lines can be parametrized by $\mathbf{P}^{n+1}$. The real lines in this set must satisfy the Plücker relationship (Eq. 6.3), inducing a quadratic constraint on $\mathbf{P}^{n+1}$. This is a quadratic equation on the line for $n = 0$; a conic in the plane for $n = 1$; and a quadric surface in projective space for $n \geq 2$. The solution to the quadratic equation can be all of $\mathbf{P}^{n+1}$, empty, reducible or, when $n > 1$, irreducible. If it is reducible, each component can be parametrized by $\mathbf{P}^n$; otherwise it is irreducible, and the entire set of lines can be parametrized by $\mathbf{P}^n$. In the following we consider the various special cases that arise.

If $\sigma_3 = 0$ and $\sigma_2 \neq 0$ then $\mathbf{M}$ has rank three. That is, only three rows of $\mathbf{M}$ are linearly independent. In this case, the set of lines incident on the l_k are those lines whose Plücker coefficients are orthogonal to the first three rows of $\mathbf{V}$. Thus, by the SVD, the last three rows of $\mathbf{V}$ span the space of lines (real and imaginary) incident on the l_k. Consider the map $\mathbf{\Lambda}(u, v) : \mathbf{P}^2 \rightarrow \mathbf{P}^5$ given by

$$\mathbf{\Lambda}(u, v) = u\mathbf{F} + v\mathbf{G} + \mathbf{H}$$

where $\mathbf{F}$, $\mathbf{G}$ and $\mathbf{H}$ are the last three rows of $\mathbf{V}$. $\mathbf{\Lambda}(u, v)$ parametrizes the real and imaginary incident lines. The real lines incident on the l_k must satisfy

$$\mathbf{\Lambda} \odot \mathbf{\Lambda} = 0.$$

This is a quadratic equation $q(u, v) = 0$ in the variables u and v. If the solution is a pair of lines, the set of lines incident on the l_k comprise two 1-parameter families of lines. Otherwise the conic can be parametrized by a single variable t. Thus if $u(t), v(t)$ satisfy $q(u(t), v(t)) = 0$, the incident lines are given by $\mathcal{L}(u(t)\mathbf{F} + v(t)\mathbf{G} + \mathbf{H})$.

If $\sigma_3 = 0$, $\sigma_2 = 0$, and $\sigma_1 \neq 0$, the set of real and imaginary lines incident on the l_k can be parametrized by

$$\Lambda(u, v, w) = u\mathbf{F} + v\mathbf{G} + w\mathbf{H} + \mathbf{I},$$

where $\mathbf{F}$, $\mathbf{G}$, $\mathbf{H}$ and $\mathbf{I}$ are the last four rows of $\mathbf{V}$. Again, the real lines satisfy a quadratic equation $q(u, v, w) = 0$ in $\mathbf{P}^3$. The zero surface of this equation can be parametrized by the projective plane.

6.4 Application

The Plücker formulation gives us a set of primitives for 3D line manipulation that is analogous to those for manipulating points and planes in 3D. With these primitives, we can straightforwardly represent, for example, bundles of light rays that stab 3D portal sequences and illuminate reached polyhedral cells. This critical visibility operation is the subject of the following chapter.

Chapter 7

Stabbing 3D Portal Sequences

In preparation for the generalization of the subdivision and visibility algorithms to arbitrarily-oriented 3D occluders, we can formulate the following abstract problem. Suppose we wish to describe the set of light rays originating at a convex, polygonal light source and passing through each of a sequence of convex polygonal holes. This set of light rays can be conceptualized in several distinct ways. Foremost, it constitutes the volume potentially visible to an observer situated on the light source (or, in the case of our visibility traversal algorithms, situated on or behind the plane of the first portal in a portal sequence). The ray bundle also characterizes the weak visibility of the light source; i.e., the set of points from which the light source is partially or totally visible, or, conversely, the set of points partial or total illuminated by the light source.

First, it is crucial to determine whether or not the hole sequence *can* be stabbed; i.e., whether there exists a *single* light ray stabbing the hole sequence. Second, once a stabbing line has been established, we wish to characterize the *antipenumbra* of the light source with respect to the hole sequence: the region beyond the plane of the final hole that is illuminated by the light source. Both the stabbing and antipenumbra algorithms exploit the fact that the holes are *oriented*; given a particular hole sequence, each hole can be considered to admit light in only one direction.

7.1 Stabbing General Portal Sequences

In the following analysis, we call the light source and holes the *generator polygons*. In total, these polygons have n directed *generator edges* E_k, $k \in 1, \ldots, n$ (we assume at first that no two edges from different polygons are coplanar). Each edge E_k is a segment of a directed line e_k. Since the polygons are oriented, the e_k can be arranged so that if some directed line s *stabs* (intersects)

each polygon, it must have the same sidedness relation with respect to each e_k (cf. Equation 6.2). That is, any stabbing line s must satisfy (Figure 7.1):

$$side(s, e_k) \geq 0, \qquad k \in 1, \ldots, n.$$

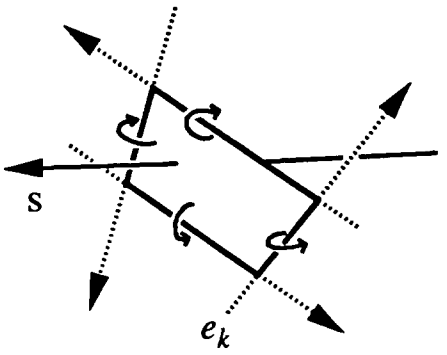


Figure 7.1: The stabbing line s must pass to the same side of each e_k.

Define h_k as the oriented Plücker hyperplane corresponding to the directed line e_k:

$$h_k = \{\mathbf{x} \in \mathbf{P}^5 : \mathbf{x} \odot \Pi_k = 0\}.$$

For any stabbing line s, $side(s, e_k) \geq 0$. That is, $S \odot \Pi_k \geq 0$, where $S = \Pi(s)$, and $\Pi_k = \Pi(e_k)$. The 5D point S must therefore lie on or above each hyperplane h_k (Figure 7.2), and inside or on the boundary of the convex polytope $\bigcap_k h_k^+$.

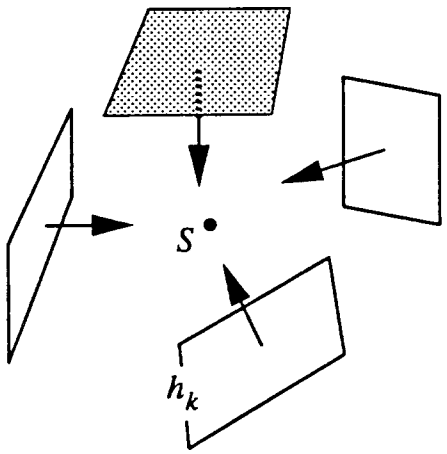


Figure 7.2: The 5D point $S = \Pi(s)$ must be on or above each hyperplane h_k.

The face structure of the polytope $\bigcap_k h_k^+$ has worst-case complexity quadratic in the number of halfspaces defining it [Grü67], and can be computed by a randomized algorithm in optimal $O(n^2)$

expected time [CS89]. We define the *extremal* stabbing lines as those lines incident on four generator edges. Specifying a line in three dimensions requires determining four degrees of freedom. For example, a line could be specified by its two-valued intercepts on two standard planes (in practice, the intercepts may assume infinite values and a six-valued, homogeneous representation is used [Han84, Sto89]). Thus, four lines (i.e., constraints) are necessary to determine a line. Imagine that some stabbing line exists through the portal sequence, and is incident on no portal edges. Intuitively, we can "slide" the line so that it remains a stabbing line, until it becomes incident on a single portal edge. Maintaining the edge incidence, we can again slide the stabbing line until it becomes incident on two portal edges, then three, then four. Since determining the stabbing line requires four degrees of freedom, and each portal edge removes a single degree of freedom, the stabbing line is now uniquely determined by incidence on four portal edges; that is, it is an extremal line. We conclude that if *any* stabbing lines exist through the portal sequence, then at least one such stabbing line must be extremal.

The structure of the polytope $\bigcap_k h_k^+$ yields all extremal stabbing lines [Pel90b, TH92]. Each such line ℓ is incident, in 3D, upon four of the e_k. Consequently, the Plücker point Π_ℓ must lie on four of the hyperplanes h_k in 5D, and must therefore lie on a *1D-face*, or edge, of the polytope $\bigcap_k h_k^+$. Thus, we can find all extremal stabbing lines of a given polygon sequence by examining the edges of the polytope for intersections with the Plücker surface. The extremal stabbing line corresponding to each intersection can be determined in constant time from the four relevant generator edges E_k, using the implemented algorithm of §6.3.

Figure 7.3 depicts the output of an implementation of this algorithm. The input consists of five polygons, with $n = 23$ edges total. The 5D convex polytope $\bigcap_k h_k^+$ has 275 edges, which together yield 40 intersections with the Plücker surface, and thus 40 extremal stabbing lines. All stabbing lines, considered as rays originating at the plane of the final hole, must lie within the antipenumbra of the light source (here, the leftmost polygon in the sequence). Some extremal stabbing lines lie in the interior of the antipenumbra because the edge graph of $\bigcap_k h_k^+$, when "projected" via $\mathcal{L}$ into three-space, overlaps itself.

This section concludes the descriptions of algorithms for stabbing sequences of convex polygonal portals, and computing the antipenumbra of an area light source through such portal sequences.

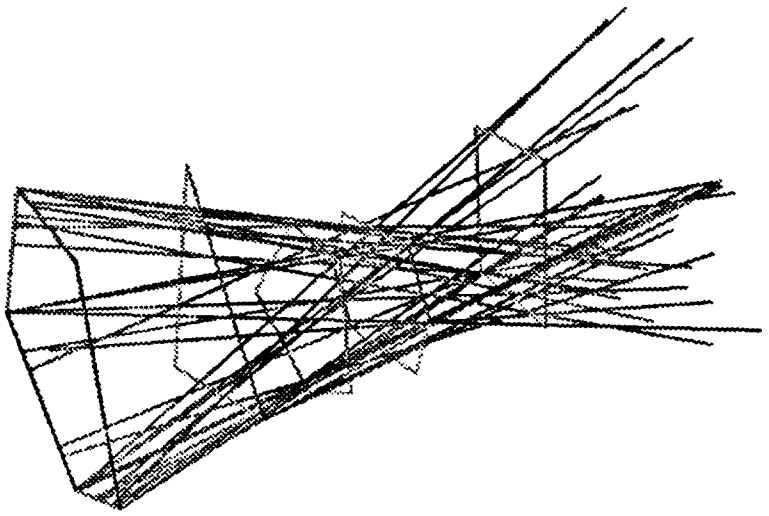


Figure 7.3: The forty extremal stabbing lines of five oriented polygons in 3D. The total edge complexity n is twenty-three. Note the hourglass-shape of the line bundle stabbing the sequence.

7.2 Penumbrae and Antipenumbrae

Suppose an area light source shines past a collection of convex occluders. The occluders cast shadows, and in general attenuate or eliminate the light reaching various regions of space. There is a natural characterization of any point in space in this situation, depending on how much of the light source can be "seen" by the point. Figure 7.4 depicts a two-dimensional example. If the point sees none of the light source (that is, if all lines joining the point and any part of the light source intersect an occluder), the point is said to be in *umbra*. If the point sees some, but not all, of the light source, it is said to be in *penumbra*. Otherwise, the point may see all of the light source.

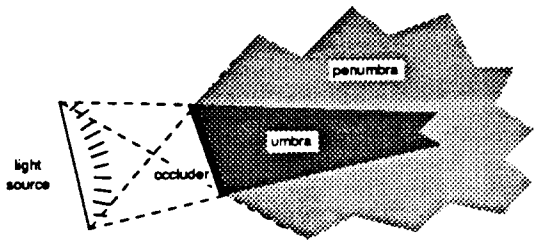


Figure 7.4: Umbra and penumbra of an occluder (bold) in 2D.

Imagine that the occluders are replaced by convex portals in otherwise opaque planes. In a

sense complementary to that above, every point in space can again be naturally characterized. We define the *antiumbra* cast by the light source as that volume from which the entire light source can be seen, and the *antipenumbra* as that volume from which some, but not all, of the light source can be seen (Figure 7.5). For a given light source and set of portals or occluders, the umbra is the spatial complement of the union of antiumbra and antipenumbra; similarly, the antiumbra is the spatial complement of the union of umbra and penumbra. Both the antiumbra and antipenumbra may be empty beyond the plane of the final portal, i.e., they may contain no points.

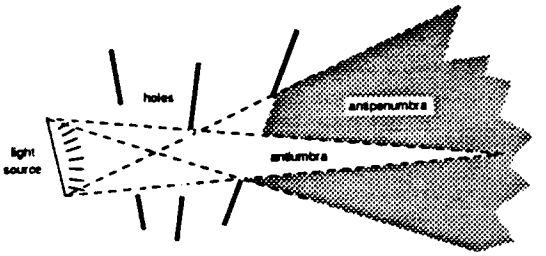


Figure 7.5: Antiumbra and antipenumbra admitted by a 3D portal sequence.

We wish to characterize the volume illuminated by a light source (the first portal in a sequence) within a cell that is reached by a line stabbing the sequence. Characterizing this volume for a sequence of a single portal is trivial: the antipenumbra (illuminated volume) in this case is the entire halfspace beyond the portal, intersected with the reached cell, which is, of course, an immediate neighbor of the source cell. By the convexity of cells, this intersection yields exactly the volume of the reached cell. Computing the antipenumbra of portal sequences with length two or more involves interactions among vertices and edges of different portals and, as we will show, requires both linear and quadratic implicit primitives to correctly describe the illuminated volume.

Recall that portals are oriented; in two dimensions this means each portal has a "left" end and a "right" end, from the point of view of a light ray traversing the portal. In three dimensions, the portal orientation means that the portal polygon, viewed as an ordered sequence of points, appears in clockwise order from the point of view of an observer on the light ray encountering the portal.

7.2.1 Event Surfaces and Extremal Swaths

There are three types of extremal stabbing lines: vertex-vertex, or VV lines; vertex-edge-edge, or VEE lines; and quadruple edge, or 4E lines. Imagine "sliding" an extremal stabbing line (of any type) away from its initial position, by relaxing exactly one of the four edge constraints determining the line (Figure 7.6). The surface, or *swath*, swept out by the sliding line must either be planar (if

the line remains tight on a vertex) or a regulus, whose three generator lines embed edges of three different polygons. In the terminology of [GCS91], swaths are VE or EEE *event surfaces* important in the construction of aspect graphs, since they are loci at which qualitative changes in occlusion occur.

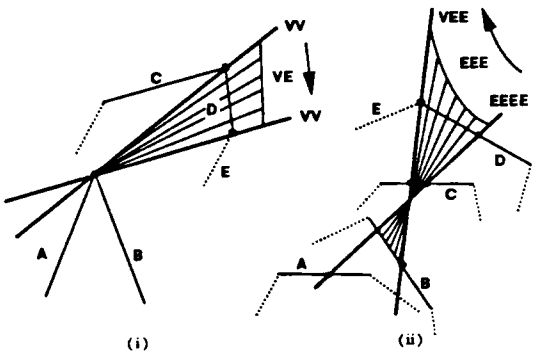


Figure 7.6: Sliding a stabbing line away from various extremal lines in 3D generates a VE planar swath (i) or a EEE quadratic surface (ii).

Figure 7.6-i depicts an extremal VV stabbing line tight on four edges A,B,C and D. Relaxing constraint C yields a VE (planar) swath tight on A, B, and D. Eventually, the sliding line encounters an obstacle (in this case, edge E), and terminates at a VV line tight on A,B,D, and E. Figure 7.6-ii depicts an extremal 4E stabbing line tight on the mutually skew edges A,B,C, and D; relaxing constraint A produces a EEE (regulus) swath tight on B, C, and D. The sliding line eventually encounters edge E and induces an extremal VEE line, terminating the swath.

Consider the same situations in the 5D space generated by the Plücker mapping (Figure 7.7). Extremal lines map to particular points in Plücker coordinates; namely, the intersections of the edges, or $1D$-faces, of the polytope $\bigcap_k h_k^+$ with the Plücker surface. Since swaths are one-parameter line families, they correspond to *curves* in Plücker coordinates. These curves are the *traces* or intersections of the 2D-faces of $\bigcap_k h_k^+$ with the Plücker surface, and are therefore conics (in 5D, a polytope's 2D-faces are planar, and determined by the intersection of some three h_k). We call the 3D swaths corresponding to these 5D conic traces *extremal swaths*. All swaths have three generator lines, and consequently three generator edges, arising from elements of the portal sequence.

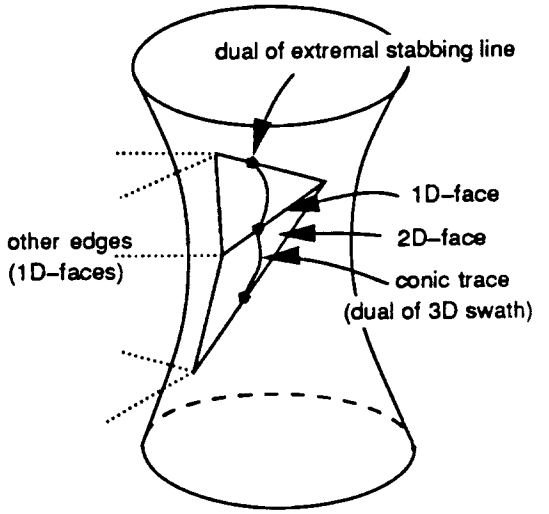


Figure 7.7: Traces (intersections) of extremal lines and swaths on the Plücker surface in 5D (higher-dimensional faces are not shown).

7.2.2 Boundary and Internal Swaths

A 1D line family can be extremal (that is, lie on the boundary of the convex hull) in 5D, yet lie wholly inside the antipenumbra in three-space. Just as there are extremal stabbing lines that lie in the interior of the antipenumbra, so there are extremal swaths in the interior as well. We define a *boundary* swath as an extremal swath that lies on the boundary of the antipenumbra (these are the shaded surfaces in Figure 7.15-ii). All other extremal swaths are *internal*. Examining the 2D-faces of the polytope $\bigcap_k h_k^+$ yields all extremal swaths; however, we must distinguish between boundary and internal swaths. This distinction can be made purely locally; that is, by examining only the swath's three generator edges and, in turn, their generator polygons. From only this constant number of geometric constraints, we show how to determine whether stabbing lines can exist on "both sides" of the swath in question. If so, the swath cannot contribute to the antipenumbra boundary, and is classified as internal.

7.2.3 Two-Dimensional Example

The notions of boundary and internal swaths are most easily illustrated in two dimensions (Figure 7.8). As in §4.3, portals and the light source are line segments. The hourglass of lines that stabs the portal sequence contains an extremal swath (in 2D, a line segment) wherever a stabbing line can be incident on two linear constraints (i.e., portal endpoints). The illuminated volume

beyond the final portal is a convex volume bounded by the two crossover hourglass edges and the final portal itself. The crossover edges are therefore boundary swaths in two dimensions, since they separate the region of partial illumination from that of zero illumination. The remaining extremal edges are valid stabbing lines, by construction, but are internal swaths.

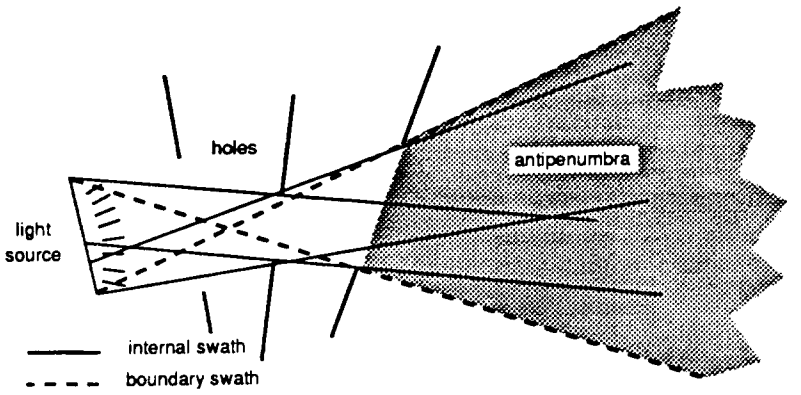


Figure 7.8: Extremal swaths arising from a two-dimensional portal sequence.

To see why this is so, imagine taking hold of a non-crossover hourglass edge **E**, and pivoting it away from each of its defining point constraints **p** and **q**, in turn (Figure 7.9). The portals incident to the constraint vertices **p** and **q** lie on the same side of **E**. Therefore, pivoting on **p**, the stabbing line must move into the interior of the incident portal. Beyond the plane of the final portal, the stabbing line will lie to one side of **E** (the broken lines in the figure). Analogously, pivoting the hourglass edge **E** on the point **q** produces a stabbing line that lies to the other side of **E** beyond the final portal (the dotted lines in the figure). Since **E** admits valid stabbing lines into both its signed halfspaces, it cannot separate an illuminated from a non-illuminated region, and is therefore an internal swath.

Finally, consider a crossover hourglass edge **E** (Figure 7.10). The portals incident to the constraint vertices **p** and **q** lie on opposite sides of **E**. Pivoting **E** on **p** or **q** must move the stabbing line into the interior of the appropriate portal, and yields only lines on one side of **E** (the side containing the antipenumbra). Therefore, **E** is a boundary swath.

7.2.4 Edge-Edge-Edge Swaths

In three dimensions, internal and boundary EEE swaths can be distinguished as follows. Suppose three mutually skew generator edges A, B, and C give rise to an extremal EEE swath. Choose some line L incident on the generators respectively at points **a**, **b**, and **c** (Figure 7.11-i). At these points, erect three vectors N_a, N_b, and N_c, perpendicular both to L and to the relevant

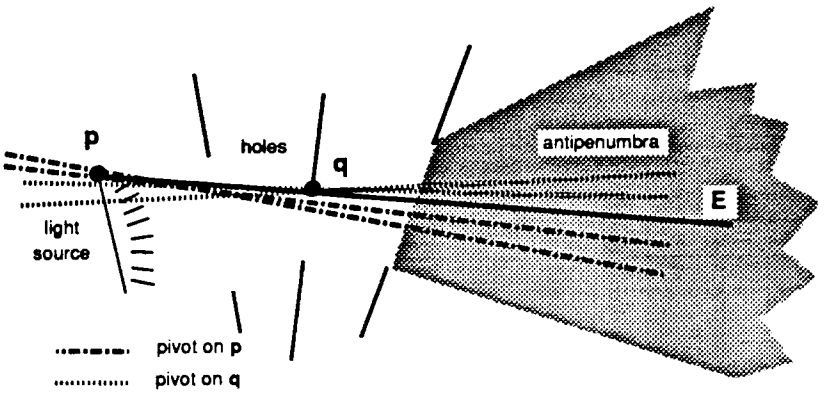


Figure 7.9: An internal swath in a two-dimensional portal sequence.

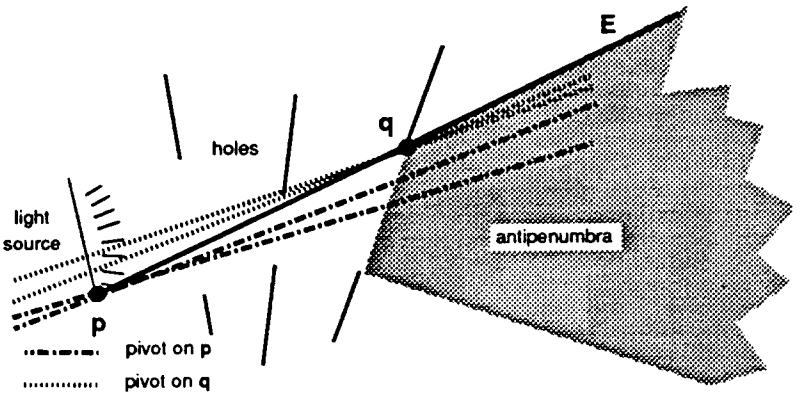


Figure 7.10: A boundary swath in a two-dimensional portal sequence.

generator edge, with their signed directions chosen so as to have a positive dot product with a vector pointing into the interior of the edge's generator portal. (We refer to these vectors collectively as the N_i.)

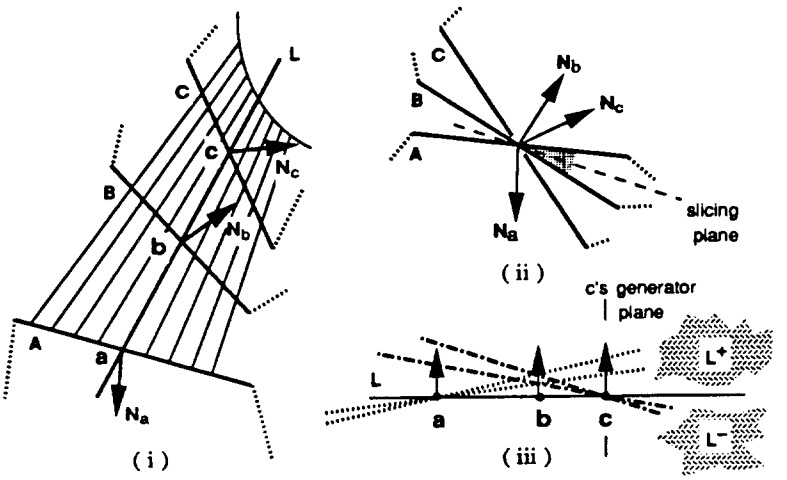


Figure 7.11: An internal EEE swath (i), viewed along L (ii) and transverse to L (iii). The N_i can be contained, and the swath is therefore internal.

We say that three coplanar vectors can be *contained* if there exists a vector whose dot product with all three vectors is strictly positive. We claim that a swath is internal if and only if its corresponding N_i can be contained.

Suppose that the N_i can be contained (as in Figure 7.11). Consider any vector having a positive dot product with the N_i (such as one pointing into the gray region of Figure 7.11-ii). Next, "slice" the configuration with a plane containing both L and this vector, and view the swath in this plane (Figure 7.11-iii).

The trace (intersection) of the swath on this plane is simply the stabbing line L, which partitions the slicing plane into two regions L^+ and L^- beyond the plane of C's generator polygon (Figure 7.11-iii). We can move L infinitesimally by keeping it tight on, say, point a. The interiors of the other two portals allow L to pivot in only one direction, thus generating stabbing lines into L^+ (dotted). Analogously, pivoting about point c generates stabbing lines into L^- (dashed). Since, by construction, the swath admits stabbing lines on both sides, it cannot be a boundary swath.

In contrast, the N_i of Figure 7.12-i cannot be contained. Thus, any vector (including one chosen from the gray region of Figure 7.12-ii) will have a negative dot product with at least one, and at most two, of the N_i. Suppose it has a negative dot product with N_c. Slice the configuration

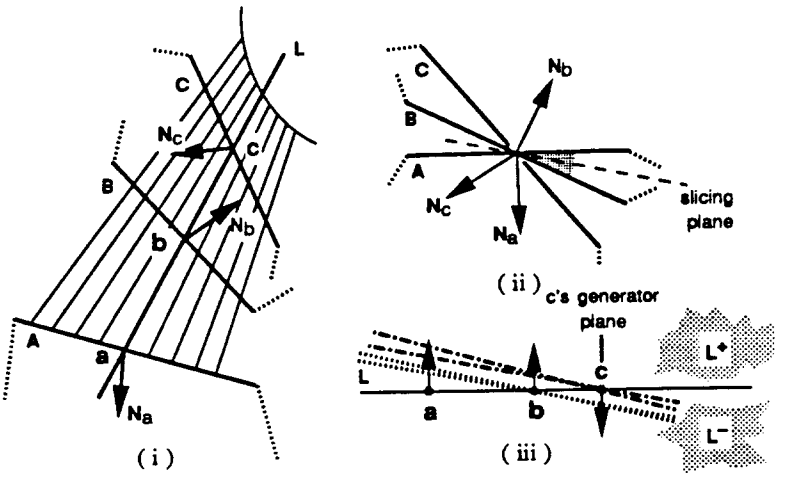


Figure 7.12: A boundary EEE swath (i), viewed along L (ii) and transverse to L (iii). The N_i cannot be contained, and the swath is therefore a boundary swath.

with a plane containing both L and one such vector (Figure 7.12-iii). Pivoting on point b generates stabbing lines into L^- (dotted), as does pivoting on point c (dashed). The configuration does not admit any stabbing lines into L^+. Since this is true for any choice of slicing plane and (as we will show) for any choice of L, we conclude that the swath is a boundary swath; i.e., it separates a region of zero illumination from a region of partial illumination.

7.2.5 Vertex-Edge Swaths

Suppose the swath in question has type VE. This is just a special case of the EEE construction, in which two of the three edges involved intersect. This case is worthy of separate discussion because it shows the simple behavior of the containment function as EEE skew edges "collapse" to some intersection point. This degenerate behavior also has implications in the sharpness of the shadow boundary along the swath, although these are not yet fully understood.

Label the two generator edges defining the VE swath vertex v as A and B, the remaining edge C (not in the plane of A and B), the two relevant generator polygons P and Q, and the plane through C and v as S (Figure 7.13). Orient S so that Q (C's generator polygon) is above it (i.e., in S^+); this is always possible, since Q is convex. Plane S divides the space beyond the plane of Q into two regions S^+ and S^-. If stabbing lines can exist in only one of these regions, the swath is a boundary swath. This occurs if and only if S is a *separating plane* of P and Q; that is, if and only if polygon P is entirely below S.

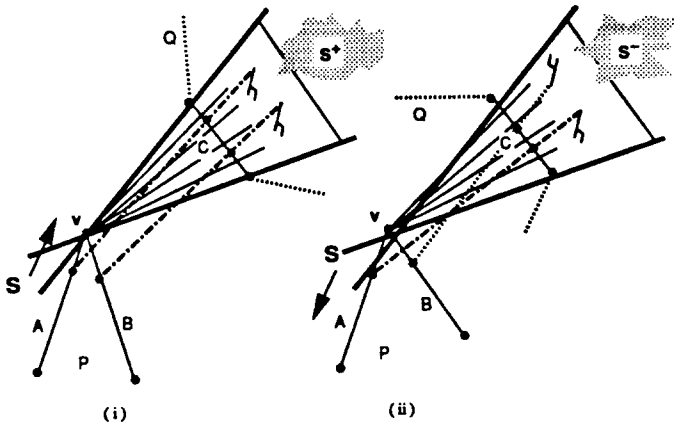


Figure 7.13: Boundary (i) and internal (ii) VE swaths.

Suppose S separates polygons P and Q. Imagine choosing some stabbing line from the swath, and moving it so that it comes free from the swath vertex v, but remains tight on edge C (and remains a valid, though non-extremal, stabbing line). If S separates P and Q, the moving line's intersection with P can only lie below S (Figure 7.13-i); thus, beyond the plane of Q, all such stabbing lines must intersect S^+. Similarly, should the stabbing line move away from C and into Q's interior, while remaining tight on v, it can only intersect S^+. The swath admits only stabbing lines in S^+, and is therefore a boundary swath.

Suppose, in contrast, that the plane S does not separate P and Q, i.e., that one or both of the edges A and B lie in S^+ (Figure 7.13-ii). Again move a swath line so that it comes free from the swath vertex v, but stays tight on edge C. If the line (dashed) moves along A above (say) plane S, it will intersect the region S^- beyond the plane of Q. Similarly, should edge B lie below S, motion along B produces stabbing lines (dotted) in S^+. Otherwise, if both A and B lie above S, lines tight on v and intersecting Q's interior reach S^+. The swath admits stabbing lines into both S^- and S^+, and therefore cannot be a boundary swath.

Note that the three-vector construction for EEE swaths is applicable in this (degenerate) setting as well, since S is a separating plane if and only if the normals erected along A,B, and C cannot be contained.

7.2.6 The Containment Function

The containment function is a criterion for distinguishing between boundary and internal swaths. However, it applies only along a single stabbing line, not over an entire swath. Fortunately,

evaluating the containment function anywhere along a swath produces the same result, since the basic configuration of the normals erected at the constraining portal edges remains the same. To see why this is so, consider any configuration of three coplanar vectors N_a, N_b, and N_c. Suppose that the configuration changes continuously from containable to non-containable (e.g., by rotation of vector N_c), as in Figure 7.14. At either moment of transition (marked with dotted lines in the figure), N_c and one of N_a or N_b must be antiparallel.

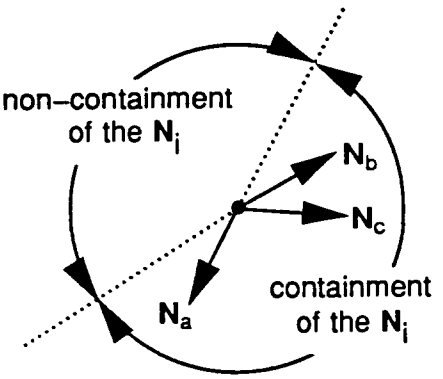


Figure 7.14: Directions of containment and non-containment, with moving N_c, for fixed N_a and N_b. Transition directions are marked.

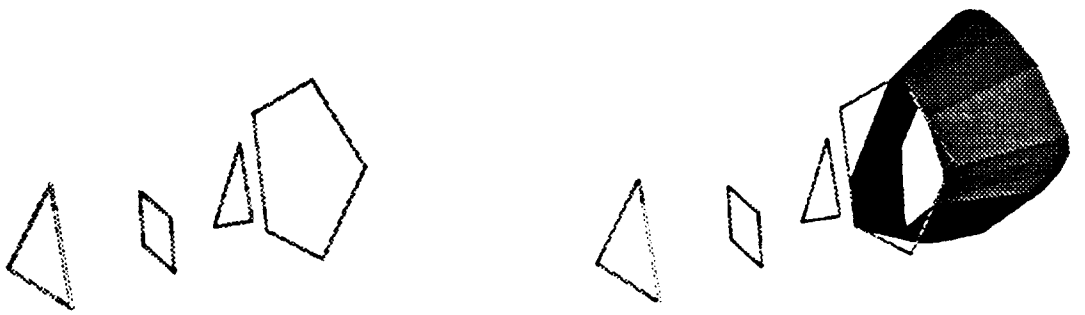


Figure 7.15: (i) The antipenumbra cast by a triangular light source through three convex portals ($n = 15$); (ii) VE boundary swaths (dark), EEE boundary swaths (light). (Figure 7.16 depicts the traces of the boundary swaths on a plane beyond that of the final hole.)

For this to occur in the EEE swath construction (Figure 7.12) two of the three generator edges must be coplanar. Similarly, in the VE swath construction (Figure 7.13), edge C and one of the edges A or B must be coplanar. But the generator edges are *fixed*; only the sliding line varies.

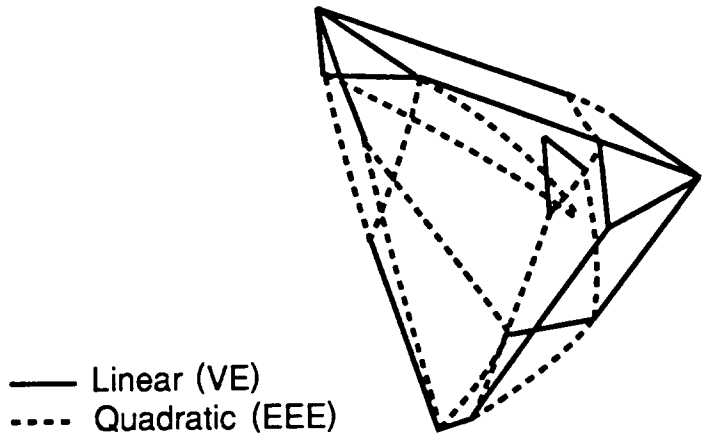


Figure 7.16: The internal swaths induced by the portal sequence of Figure 7.15, intersected with a plane beyond that of the final portal to yield linear and conic traces.

Thus, even though sliding an extremal line along a swath generates a continuously changing vector configuration, the *containment* of the vectors is a constant function. If either of the EEE or VE degenerate cases occurs in practice, the containment function is indeterminate. We detect this while examining the polytope face structure, and use special-case processing to generate the correct swath.

7.2.7 Computing the Antipenumbra

The computational machinery necessary to determine the antipenumbra is now complete. We make the following assumptions: the input is a list of m oriented polygons, $P_1 \ldots P_m$, given as linked lists of edges, the total number of edges being n. The first polygon P_1 is the light source, and all others are holes. The polygons are disjoint, and ordered in the sense that the negative halfspace determined by the plane of P_i contains all polygons P_j, $i < j \leq m$ (thus, an observer looking along a stabbing line from the light source would see the vertices of each polygon arranged in counterclockwise order).

The algorithm dualizes each directed input edge to a hyperplane in Plücker coordinates, then computes the common intersection of the resulting halfspaces, a 5D convex polytope. If there is no such intersection, or if the polytope has no intersection with the Plücker surface, the antipenumbra is empty. Otherwise, the face structure of the polytope [Grü67, Bau72] is searched for traces of boundary swaths resulting from intersections of its $1D$-faces (edges) and 2D-faces (generically, triangles) with the Plücker surface (cf. Figure 7.7). These traces are linked into loops, each corresponding to a connected component of the antipenumbral boundary. There may be multiple

loops since the intersection of the polytope boundary with the non-planar Plücker surface may have several components.

Each loop consists of intersections of polytope edges and of triangles with the Plücker surface, in alternation (Figure 7.17). Each edge intersection (a point in 5D) is the dual of an extremal stabbing line in 3D; each triangle intersection (a conic in 5D) is the dual of an extremal swath in 3D. Incidence of an edge and triangle on the polytope implies adjacency of the corresponding line and swath in 3D; stepping across a shared edge from one triangle to another on the polytope is equivalent to stepping across a shared extremal stabbing line (a "seam") between two extremal swaths in three-space. Thus the algorithm can "walk" from 2D-face to 2D-face on the polytope's surface, crossing the $1\,D$-face incident to both at each step.

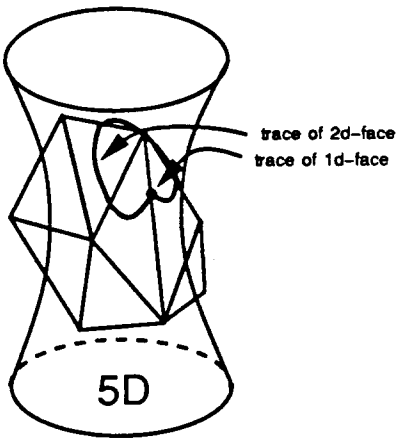


Figure 7.17: Loops in 5D line space. Each piecewise-conic path in 5D is isomorphic to the boundary of a connected component of the antipenumbra. One such loop is shown.

Each conic encountered in 5D implicates three generator edges in 3D, which are examined for containment in constant time (note that the containment axis, line L in Figure 7.12, can be taken as the dual of any intersection of the triangle's edge with the Plücker quadric). Conics whose dual edges are containable are duals of internal swaths; otherwise, they are duals of boundary swaths. As the face structure of $\bigcap_k h_k^+$ is traversed and boundary swaths are identified, they are output. When a loop is completed, the search is continued at an unexamined edge of the polytope, if any. Each closed loop found in this manner in 5D is the dual (under the Plücker mapping) of the boundary of one connected component of the antipenumbra in three-space.

The algorithm can be described in pseudocode as:

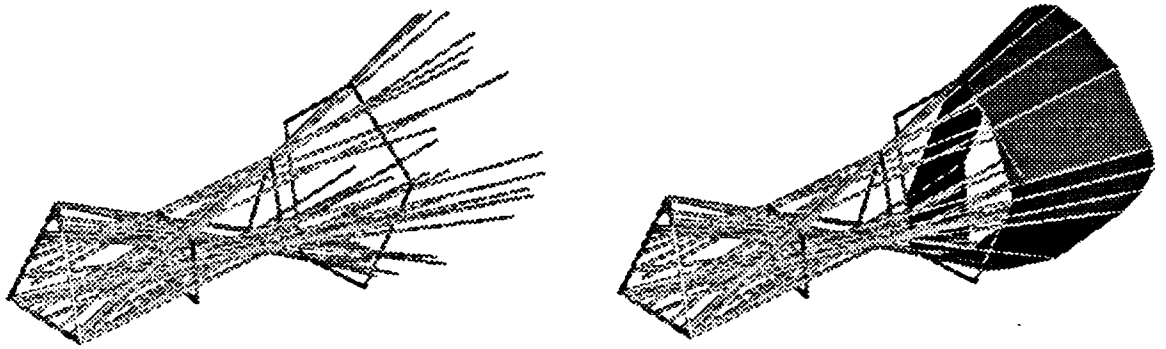
input directed edges E_k from polygons $P_1 \ldots P_m$

convert directed edges to directed lines e_k
transform e_k to Plücker halfspaces $h_k = \Pi(e_k)$
compute 5D convex polytope $\bigcap_k h_k^+$ as winged-edge data structure
identify 2D-face intersections of $\bigcap_k h_k^+$ with Plücker surface
classify resulting extremal swaths as boundary or internal
    (i.e., mark 2D-faces in 5D with containment function in 3D)
for each connected component of the antipenumbra found by DFS
    traverse boundary 2D-faces of $\bigcap_k h_k^+$
        map each trace found to a 3D boundary swath
    output swath loop as piecewise quadratic surface

7.3 Implementation Issues

7.3.1 Current Implementation and Test Cases

We have implemented the antipenumbra algorithm in C and FORTRAN-77 on a 20-MIP Silicon Graphics superworkstation. To compute the convex hull of n hyperplanes in 5D, we used a d-dimensional Delaunay simplicialization algorithm implemented by Allan Wilks at AT&T Bell Labs and Allen McIntosh at Bellcore [WMB92], and a d-dimensional linear programming algorithm [Sei90b] implemented by Michael Hohmeyer at U.C. Berkeley.



(i) The 39 extremal stabbing lines from Figure 7.15.

(ii) Some extremal stabbing lines from "seams" on the antipenumbra boundary.

Figure 7.18: The relationship between extremal stabbing lines and boundary swaths.

Figure 7.15-i depicts a set of four input polygons with $n = 15$, and the leftmost polygon acting as a light source. The antipenumbra computation took about 2 CPU seconds. The polytope $\bigcap_k h_k^+$,

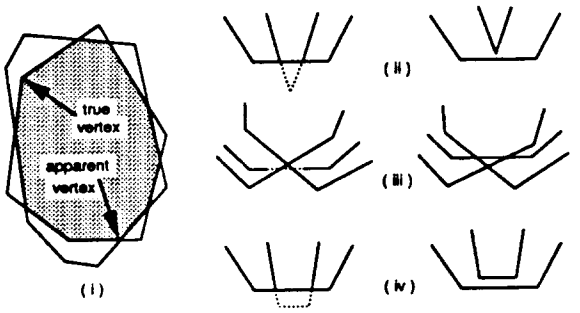
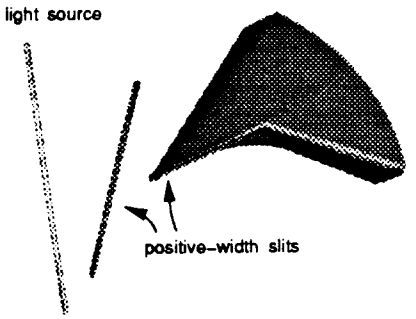
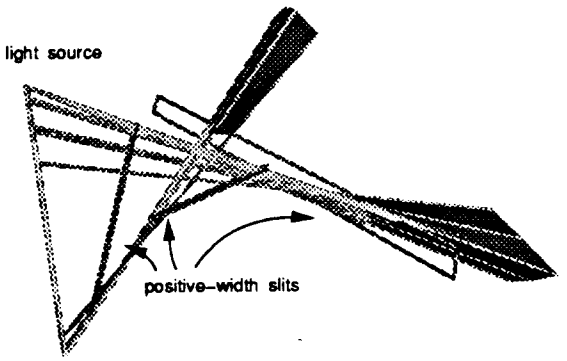


Figure 7.19: An observer's view of the light source (i). Crossing an extremal VE (ii), EEE (iii), or degenerate (iv) swath.



(i) The antipenumbra cast by a light shining through two slits.



(ii) The third slit "clips" the antipenumbra, yielding two pieces.

Figure 7.20: An area light source and three portals can yield a disconnected antipenumbra.

formed from 15 halfspaces in 5D, has 80 facets, 186 2D-faces (triangles), and 200 1D-faces (edges). Of the 186 triangles, 78 induce dual traces on the Plücker surface, generating 78 extremal swaths in 3D. Of these 78 swaths, 62 are internal; these swaths are shown projected to linear and conic traces in Figure 7.16. The remaining 16 swaths are boundary swaths (Figure 7.15-ii); of these, 10 are planar VE swaths (light), and 6 are quadric EEE swaths (dark). Of the 200 edges, 39 intersect the Plücker surface to yield extremal VV, VEE, or 4E stabbing lines (Figure 7.18-i). Note that 16 of the 39 extremal stabbing lines form "seams" of the antipenumbral boundary, as they demarcate junctions between adjacent VE and/or EEE swaths (Figure 7.18-ii).

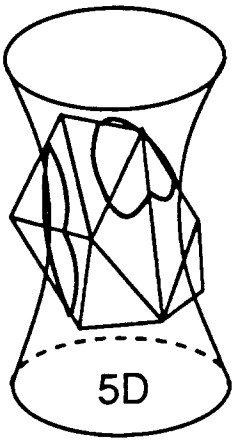


Figure 7.21: A disconnected antipenumbra boundary in 3D is isomorphic to a collection of loops in 5D line space.

Surprisingly, an area light source and as few as three holes can produce a disconnected antipenumbra. First, a light source and two holes, all thin rectangular slits, are arranged so as to admit a fattened regulus of antipenumbral light (Figure 7.20-i). A third slit then partitions the fattened regulus into two disconnected components (Figure 7.20-ii). There are consequently two loops on the trace of the polytope with the Plücker surface (Figure 7.21).

The coordinate entities necessary and sufficient for specifying swaths in the antipenumbra algorithm above (a vertex and an edge for a VE swath; three edges for a EEE swath) are not necessarily the most useful representation for swaths in other contexts. A *parametric* representation for swaths would be useful, for example, for drawing them or for continuously indexing the lines that comprise these (ruled) surfaces. On the other hand, an *implicit* representation for swaths partitions space into three regions, above, below, and on the swath, in the same manner as a hyperplane equation, and would be useful for inside-outside tests.

7.3.2 Parametric Swath Representations

Given the vertex and edge that span a VE swath, computing a parametric representation of the swath is straightforward. Then the one-parameter family of lines comprising a VE swath with swath vertex **v** and edge through vertices **a** and **b** can be generated by interpolating along the segment **ab** to give the point **p**, and producing the line through **v** and **p** (Figure 7.22-(i)).

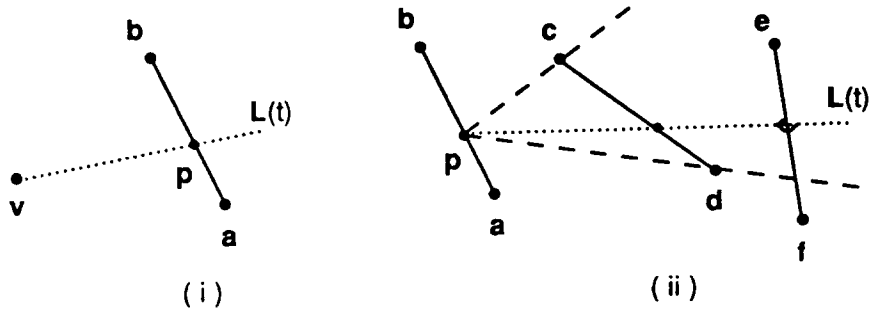


Figure 7.22: Parametrizing VE swaths (i) and EEE swaths (ii).

Parametrizing the one-parameter family of lines comprising a EEE swath is slightly more involved. Suppose the three generator edges for the swath are defined by vertices **a** and **b**, **c** and **d**, and **e** and **f**, respectively. The one-parameter line family is generated via a line **L** that intersects the segment **ab** in the interpolated point **p**; lies in the plane of **p**, **c** and **d**; and intersects the line through **e** and **f** (Figure 7.22-(ii)).

7.3.3 Implicit Swath Representations

Implicitizing a VE swath amounts to determining the plane spanned by the swath vertex and the generator edge. Implicitizing a EEE swath is more involved. Three lines are the "generators" for a regulus (Figure 7.23). If line segments $\overline{\mathbf{pq}}$, $\overline{\mathbf{rs}}$, and $\overline{\mathbf{tu}}$ define the three lines, the implicit equation of the regulus through the lines is [Som59]:

$$|\mathbf{pqrx}|\,|\mathbf{stux}| - |\mathbf{pqsx}|\,|\mathbf{rtux}| = 0 \tag{7.1}$$

where $\mathbf{x} = (X, Y, Z, 1)$ is the unknown point, and the expression $|\mathbf{abcd}|$ denotes the determinant of a 4×4 matrix.

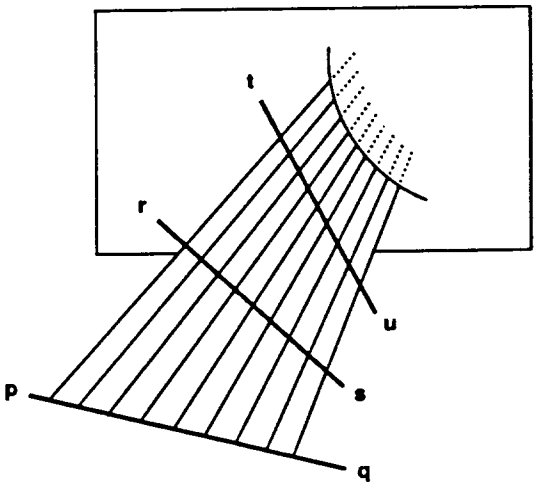


Figure 7.23: Three generator lines and part of the induced regulus of incident lines.

This can be rewritten in a more familiar way as the quadratic form

$$\begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{pmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = 0, \tag{7.2}$$

or

$$Ax^2 + Ey^2 + Hz^2 + 2Bxy + 2Cxy + 2Fyz + 2Dx + 2Gy + 2Iz + J = 0, \tag{7.3}$$

where

$$\begin{aligned}
A &= \alpha_X\beta_X - \gamma_X\delta_X \\
E &= \alpha_Y\beta_Y - \gamma_Y\delta_Y \\
H &= \alpha_Z\beta_Z - \gamma_Z\delta_Z \\
B &= (\gamma_X\delta_Y + \gamma_Y\delta_X - \alpha_X\beta_Y - \alpha_Y\beta_X)/2 \\
C &= (\alpha_X\beta_Z + \alpha_Z\beta_X - \gamma_X\delta_Z - \gamma_Z\delta_X)/2 \\
F &= (\gamma_Y\delta_Z + \gamma_Z\delta_Y - \alpha_Y\beta_Z - \alpha_Z\beta_Y)/2 \\
D &= (\gamma_X\delta_1 + \gamma_1\delta_X - \alpha_X\beta_1 - \alpha_1\beta_X)/2 \\
G &= (\alpha_Y\beta_1 + \alpha_1\beta_Y - \gamma_Y\delta_1 - \gamma_1\delta_Y)/2 \\
I &= (\gamma_Z\delta_1 + \gamma_1\delta_Z - \alpha_Z\beta_1 - \alpha_1\beta_Z)/2 \\
J &= \alpha_1\beta_1 - \gamma_1\delta_1,
\end{aligned}$$

and where the matrices $\alpha, \beta, \gamma, \delta$ correspond, respectively, to the terms in Equation 7.1, and subscription denotes the determinant of the relevant 3×3 minor; e.g.,

$$\alpha_X = \begin{vmatrix} p_y & p_z & 1 \\ q_y & q_z & 1 \\ r_y & r_z & 1 \end{vmatrix}; \beta_Y = \begin{vmatrix} s_x & s_z & 1 \\ t_x & t_z & 1 \\ u_x & u_z & 1 \end{vmatrix}; \gamma_Z = \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ s_x & s_y & 1 \end{vmatrix}; \delta_1 = \begin{vmatrix} r_x & r_y & r_z \\ t_x & t_y & t_z \\ u_x & u_y & u_z \end{vmatrix}.$$

When the three generator lines are not mutually skew, the coefficients degenerate to those of a cylinder or double plane.

7.4 Applications

7.4.1 Weak Visibility in Three Dimensions

The antipenumbra algorithm can be used to solve a previously open problem, that of computing weak and strong visibility among three-dimensional occluders [O'R87]. A polygon Q is said to be *weakly visible* from a point **p** if some, but not all, open segments connecting **p** and a point of Q's interior are disjoint from the interior of any other polygons (i.e., if **p** "sees" part of Q). The polygon Q is *strongly visible* if all such segments are disjoint. Given Q, the weak (strong) visibility problem is to compute those points from which Q is weakly (strongly) visible.

Given a collection of occluders, weak visibility can be established as follows. Subdivide the space of the occluders using a BSP tree, such that *every* occluder contributes a splitting plane. Given a query light source, mark the cells on which it is incident. For each incident cell, prefix all portal sequences emanating from the cell with the query source. The union of the antipenumbra cast through all such portal sequences is exactly set of points weakly visible from (i.e., totally or partially illuminated by) the query light source.

It can easily be shown that, given a particular portal sequence, those points from which a polygon is *strongly* visible through the sequence form a convex polyhedron. This polyhedron can be computed with a giftwrapping algorithm analogous to that of §8.3.5, except that the planes involved are *common*, rather than separating, tangent planes.

7.4.2 Aspect Graphs

Finally, we note that the computation of extremal swaths in the antipenumbra algorithm constitutes a complete *aspect graph* description of the light source (Figure 7.24. Outside the

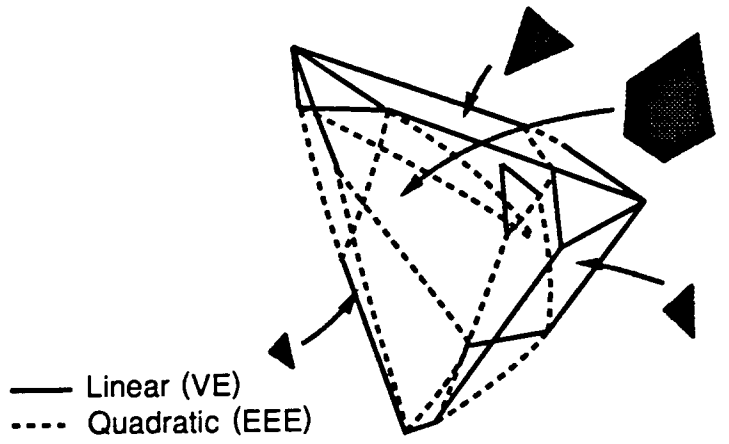


Figure 7.24: The aspect of the light source, as seen through the portal sequence.

antipenumbra, the light source aspect is empty; inside the antipenumbra, event surfaces of the aspect are simply the internal VE and EEE swaths generated by the antipenumbra algorithm. Thus, for the specific case in which the input comprises a portal sequence, the aspect graph computation requires only $O(n^2)$ time, a substantial improvement over the $O(n^6 \lg n)$ general case.

Chapter 8

Three-Dimensional Polyhedral Environments

We have first described visibility computations abstractly, and then concretely for two-dimensional and for axial three-dimensional occluders. There is no conceptual obstacle to extending these techniques to *general* polyhedral environments in three dimensions. Processing such environments in our framework requires 1) a conforming spatial subdivision (including portal enumeration); 2) a stabbing line algorithm; 3) a method of determining and storing cell-to-region and cell-to-object visibility; and 4) a method for determining eye-to-cell and eye-to-object visibility on-line.

The 3D antipenumbra (cell-to-region) visibility computation was presented in the preceding chapter. This chapter describes concrete algorithms for each of the remaining abstract subdivision and visibility operations introduced in Chapter 3, all of which have been implemented. Here, each operation is made concrete for general three dimensional polyhedral environments, in which occluders are arbitrarily oriented convex polygons.

8.1 Major Occluders and Detail Objects

General three-dimensional occluders are oriented convex polygons, typically represented as ordered lists of at least three 3D vertices, with an implied normal arising from the right-hand rule. (In practice, the normal is computed using Newell's algorithm [FvD82].) Detail objects, as in the axial case, are treated by the visibility computations as simply 3D bounding volumes subject to culling operations.

8.2 Spatial Subdivision

In 3D, the n occluders are finite, generally oriented convex polygons, and we desire a spatial subdivision whose cells are convex polyhedra, and whose portals are convex polygons. The BSP tree [FKN80] data is a recursive subdivision of space based on splitting planes, and can be straightforwardly augmented to become a conforming spatial subdivision.

The root node of a BSP tree is all of space (or, in practice, a convex volume containing the region of interest). Each node of a BSP tree is either a *leaf*, or an *internal node* with a *splitting plane* and two child nodes, one corresponding to each halfspace induced by the splitting plane. Each cell of a BSP tree corresponds to the common intersection of all 3D halfspaces encountered while traversing the tree from the root to the cell, and is therefore a convex polyhedron.

8.2.1 Splitting Criteria

Subdividing the BSP tree data structure to achieve a well-balanced, space-efficient resulting tree structure seems inherently difficult [FKN80, Tor90, PY90]. The best current bounds on the time- and space-complexity of tree construction over n polygons are $O(n^3)$ time to construct a tree of size at worst $O(n^2)$. The latter is an optimal bound since there are collections of polygons for which the smallest BSP tree is size $O(n^2)$ [PY90]. This quadratic behavior does tend to be troublesome in practice [Mit92]; with some heuristics, however, we have constructed BSP trees of usable size.

The subdivision rounds for general occluders are analogous to those of §5.2.1, namely maximum obscuration, minimum cleaving, maximum volume, and maximum aspect. In every case, the corresponding operations on general occluders are considerably more expensive. The maximum obscuration computation requires time proportional to the sum of the complexity of the current occluder and the current cell for each candidate occluder, since the cell's cross-section in the plane of the occluder must be computed, and the occluder subtracted from the resulting convex polygon. Similarly, the minimum cleaving round requires time proportional to the square of the number of occluders present in the cell being split, since all pairs of occluders must be subjected to a cleaving test (compared to linear time in the axial case). The maximum volume and maximum aspect tests require time proportional to the complexity of the current cell, rather than constant time as in the axial case.

8.2.2 Point Location

Point location in BSP trees is straightforward, requiring that the query point be checked against the halfspaces whose intersection defines the current node (starting at the root). If the point is outside this polyhedral volume, it is not represented by the BSP tree. Otherwise, if the current node is a leaf node, it is returned as the answer to the point location query. Finally, if the node is internal, the query point determines a positive or negative halfspace of the split plane, and the correct subtree of the current node is recursively searched. As with k-D trees and axial occluders, the time required to point locate in a BSP tree is proportional to the length of the path from the root to the leaf node containing the point.

8.2.3 Object Population

Object population in BSP trees is more complex than for 2D or axial 3D spatial subdivisions. Since each cell is a convex polyhedron, detecting object incidence amounts to deciding whether the intersection of the cell with the object bounding box is non-empty. This is a 3D linear programming problem, and can be solved for a particular cell in $O(s + b)$ time, where s is the number of planar faces bounding the polyhedral cell, and b is the (constant) number of faces comprising the object bounding box. These $s + b$ halfspaces are simply checked for a common intersection; if a more discriminating population method is desired, the convex hull of the detail object's vertices may be precomputed and stored with the detail object, to be intersected with BSP leaf cells as they are encountered and populated.

A storage optimization analogous to that of §4.2.2 applies (cf. Figure 4.3): the cell should not be populated with 3D (i.e., volume) detail objects that have only a zero-D or 1D intersection with the source cell, and, if 2D (i.e., areal) objects have associated normal orientations, the cell should not be populated with backfacing areal objects (occluders), since these objects (occluders) are backfacing for all generalized observer viewpoints.

8.2.4 Neighbor Finding

Neighbor finding in BSP trees can be done robustly in an entirely two-dimensional fashion. Given a particular cell and boundary face, the BSP tree is ascended to find the cell that was split along the plane of the given boundary face. The appropriate subtree of this cell is then searched for all *leaf* cells with a boundary face affine to this plane (there may be no such cells). The matching boundaries of all cells so identified are then subjected to a two-dimensional intersection test on the

shared boundary plane. Only cells with a 2D (i.e., superlineal, or positive area) intersection with the original boundary are marked as neighbors of the given cell.

8.2.5 Portal Enumeration

Whenever a BSP tree cell is split, the split plane gives rise to a boundary face on each child cell, and the occluders affine to that split plane (if any) are attached to both resulting boundary faces. With this operating assumption, the machinery for finding cell egress and portals is straightforward. Egress finding in BSP trees is accomplished by searching for lateral relatives in the tree that share a particular bounding face plane equation with the source cell. If the two cells' bounding faces are coplanar and intersect, the cells are spatial neighbors. Finally, that portion of their shared boundary which is covered by occluders is removed, and any remaining convex portions of the boundary become cell portals (Figure 8.1). In practice, portal enumeration requires a robust package for CSG operations on planar polygons such as that implemented and described in [Air90].

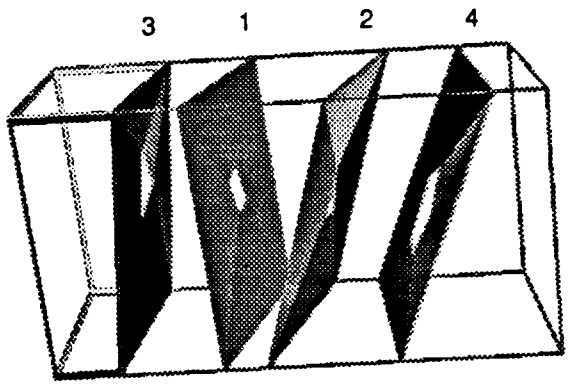


Figure 8.1: Five leaf cells of an augmented BSP tree, with convex portals. The four splitting planes are affine to the four coaffine occluder sets shown; they are numbered by the order in which they occurred.

When spatial cells are general polyhedra, portals are planar non-convex regions formed by (convex) cell boundaries minus unions of opaque occluders on the boundary planes. Non-convex egresses can be handled by partitioning them into convex fragments. Alternatively, any collection of egresses can be coalesced into a single portal by performing a 2D convex hull computation on the egress vertices. This tactic, although increasing portal area above the minimum possible, can

only increase the computed cell-to-cell visibility estimation, ensuring that it remains a superset of the actual visibility; on the other hand, it will reduce the number of portal sequences that must be examined.

Analogously to the two dimensional case of §4.2.4 (cf. Figure 4.4), a cell whose boundary has only a zero-D or 1D (i.e., point or lineal) intersection with an occluder need not take this occluder into account during portal enumeration, as the occluder can have no effect on the visibility of the generalized observer in the cell. Both backfacing and frontfacing 2D incident occluders are relevant, and should not be ignored during portal enumeration.

8.3 Static Visibility Operations

8.3.1 Cell-to-Cell Visibility

Establishing cell-to-cell visibility amounts to determining whether or not a *single* light ray can stab a given portal sequence. The stabbing and cell-to-region algorithms exploit the fact that the portals are *oriented* by the sense in which they are traversed during a directed graph search through the portals of a spatial subdivision. For a particular portal sequence, each portal admit light in only one direction. This is exactly the abstract hole-stabbing problem discussed in Chapter 7.

The abstract cell-to-cell visibility determination is accomplished with a depth-first-search on the cell adjacency graph, and a method for determining stabbing lines. Since algorithms for both the stabbing line and antipenumbrae computations have time complexity $O(n^2)$, incrementally stabbing a portal sequence of edge complexity n (assuming a constant number of edges per portal) requires $O(n^3)$ time. In practice, this seems not to prohibit use of the algorithms on real data sets, for these reasons: 1) most portal sequences are short (less than 10 portals), with four edges per portal, and the algorithm constants are therefore more important than the exponents in the complexity measure; and 2) the implementation of the stabbing line algorithms identifies many "trivial reject" (non-stabbable) inputs in $O(n)$ time due to the rapidity with which linear programming reports the infeasibility of the constraint set.

8.3.2 Cell-to-Region Visibility

We wish to characterize the volume illuminated by a light source (the first portal in a sequence) within a cell that is reached by a line stabbing the sequence. Characterizing this volume for a sequence of a single portal is trivial: the antipenumbra (illuminated volume) in this case is the entire

halfspace beyond the portal, intersected with the reached cell, which is, of course, an immediate neighbor of the source cell. By the convexity of cells, this intersection yields exactly the volume of the reached cell. Computing the antipenumbra of portal sequences with length two or more involves interactions among vertices and edges of different portals and, as we showed in Chapter 7, requires both linear and quadratic implicit primitives to correctly describe the illuminated volume.

8.3.3 Representing the Cell-to-Region Visibility

Cell-to-region visibility for a given source is simply the union of all antipenumbral volumes emanating from portals on the source cell. In fact, this union operation need not be performed explicitly, since the antipenumbra computation is used only to determine whether particular detail objects are visible from the source cell. The union operation can be effectively performed, however, by logically combining the results of inside-outside tests with respect to all antipenumbral volumes emanating from the source.

8.3.4 Cell-to-Object Visibility

Generally, whenever the visibility search reaches a cell via a sequence of two or more portals, not all objects in the reached cell will be visible to an observer in the source cell. Assuming again the availability of an axial bounding box for each detail object, this bounding box must be examined for incidence with the antipenumbral volume in the reached cell. Note that once an object is determined visible via *some* path from the source cell, it need not be tested again, regardless of any other paths reaching cells populated with the object.

We implicitize each swath on the antipenumbral boundary into a plane or quadric equation in the space variables x, y, and z. The bounding box must be incident to the common interior of each implicit surface in order to be visible to the generalized observer in the source cell; computing this incidence function exactly is a quadratic programming problem, both computationally expensive [GJ79, CHG+88] and numerically sensitive. Instead, we again employ the notion of conservative visibility to drastically reduce the complexity and numerical sensitivity of our algorithms. We deem an object potentially visible from the source cell if its bounding box is incident to each induced swath halfspace, considered individually. Computing the inside-outside function for an axial box and a single quadratic surface is straightforward, and eliminates the need for a quadratic programming algorithm.

Computing exact box incidence with the interior halfspace of a VE swath is straightforward.

Computing box incidence with a quadratic halfspace is a more interesting problem. It is an instance of quadratic programming; we wish to determine if the (linear) box interior and the (quadratic) regulus interior have any common volume. Invoking a fully general quadratic programming algorithm is unnecessary, however, given the special nature of the problem. First, we have a constant number of constraints. Second, the quadratic constraint is of a particular class, eliminating a class of intersection events *a priori* that a fully general algorithm would have to treat explicitly.

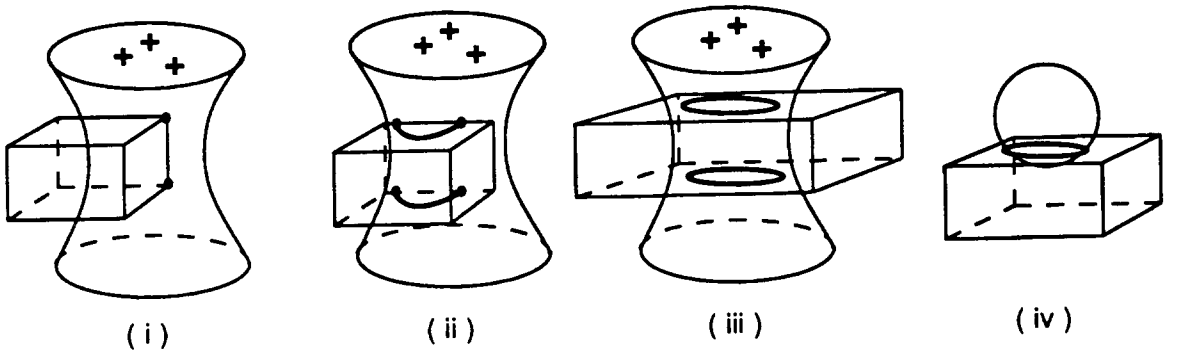


Figure 8.2: A convex polyhedron and a regulus can intersect in only three ways.

Call the halfspace induced by a EEE swath a *EEE halfspace*. There are only three qualitatively distinct ways in which an axial box and a EEE swath may intersect. Suppose the positive halfspace is "inside" the regulus (the regions marked by + in Figure 8.2). First, some or all of the box vertices may lie in the positive halfspace of the regulus (Figure 8.2-i). This may be discovered simply by checking the sign of the implicit quadratic Equations 7.2 or 7.3 with x, y, and z as the coordinates of any box vertex. Second, no box vertices may lie inside, but the regulus may intersect one or more box edges (Figure 8.2-ii). This may be discovered by parametrizing each box edge as a function of a single variable t, substituting into the implicit equation for the regulus, and searching for roots. Finally, the box may have no vertices inside the positive halfspace, and no edge intersections with the regulus (Figure 8.2-iii). In this case, the regulus must be intersected with the plane of each box face, in turn, and the resulting conic examined for an intersection with the rectangular box faces. This intersection reduces to several two-case analyses in two dimensions (each 2D face of the box).

Recall that the EEE swath is a portion of a regulus, and has negative Gaussian curvature. Thus there is one type of intersection that cannot occur: the regulus cannot "dip in" to a face of the box and exit on that same face. This "impossible" situation is illustrated with a sphere in Figure 8.2-iv.

8.3.5 Conservatively Approximating the Antipenumbra

In practice, the antipenumbra and cell-to-object computations are difficult to implement robustly. They depend on high-dimensional convex-hull computations, singular value decompositions of matrices, and robust root finding, all of which have specific and sometimes algorithmically idiosyncratic dependencies on numerical tolerances and geometric degeneracies. Consequently, we have developed an algorithm that bounds the antipenumbra of a light source with a convex polyhedron, using only 3D linear operations such as point-plane and vector-vector inner products. This algorithm is a straightforward assembly of separating tangent planes, which can be thought of as an "internal giftwrap" over the edges of the portal sequence. Asymptotically, this algorithm remains $O(n^2)$ in complexity for general portals with n total edges; however, in practice it runs one to two orders of magnitude faster than the exact antipenumbra algorithm above for portal sequences of a few tens of edges.

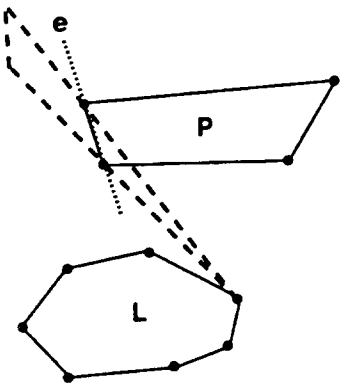


Figure 8.3: The plane through edge **e**, separating portals **P** and **L**.

The linearized antipenumbra algorithm is straightforward. Consider any pair of portals in a sequence, and an edge on one of the portals (Figure 8.3). The edge uniquely defines a *separating plane* that contains each portal in different (closed) halfspaces. This separating plane is spanned by the edge, and some vertex on the other portal. Clearly one halfspace of this plane (the halfspace containing the "later" portal in the sequence) must contain the antipenumbra of any portal sequence containing this portal pair. Consider the at most $O(n^2)$ pairs of portals and edges in a portal sequence of n edges. These $O(n^2)$ pairs induce at most $O(n^2)$ separating planes; each can be oriented in constant time to produce an open halfspace containing the antipenumbra. Finally, the convex hull of the $O(n^2)$ halfspaces can be computed in $O(n^2 \lg n)$ time.

We can improve the algorithm to $O(n^2)$ time by recognizing that, for each *generator edge* of a

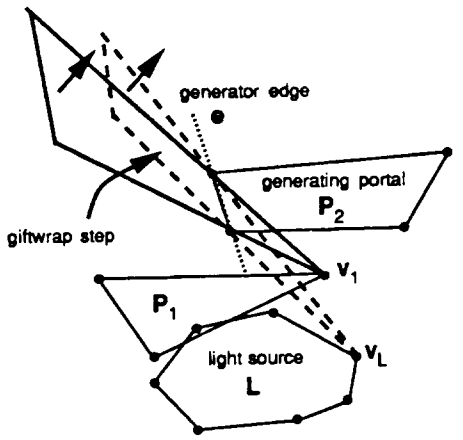


Figure 8.4: A giftwrap step on edge **e**, from $\mathbf{v}_1$ to $\mathbf{v}_L$.

generating portal, at most two separating planes can contribute faces to the linearized antipenumbra boundary (Figure 8.4), since at most one vertex from each halfspace of the generating portal can span a contributing plane with the generator edge. Consider some generator edge **e** on a generating portal $\mathbf{P}_2$, and the portals incident on one halfspace of $\mathbf{P}_2$. Each of these portals has at most one vertex that spans a separating plane with **e** (in the figure, $\mathbf{P}_1$ has vertex $\mathbf{v}_1$ and **L** has vertex $\mathbf{v}_L$). Of these at most $n/3$ vertices, only one will span a plane that contains all of the other vertices and the generating portal in the same halfspace. This single plane is the only one of the $n/3$ candidate planes that can contribute to the boundary of the linearized antipenumbra. All other planes pivoting on this edge will be superfluous. Therefore, over n generator edges there are at most $2n$ boundary planes, and each can be identified in $O(n)$ time. The total time to identify the $2n$ possibly contributing planes is therefore $O(n^2)$, and the time to compute their convex hull is $O(n \lg n)$.

8.4 Dynamic Visibility Queries

The static, generalized visibility operations of the preceding section substantially overestimate the visibility of the actual observer. This section details *dynamic* visibility queries that exploit both the precomputed static visibility information, and the instantaneous knowledge of the observer's view variables, to generate more discriminating visibility information.

8.4.1 Observer View Variables

The observer view variables for dynamic queries are the same as those for the axial 3D case (cf. §5.4.1).

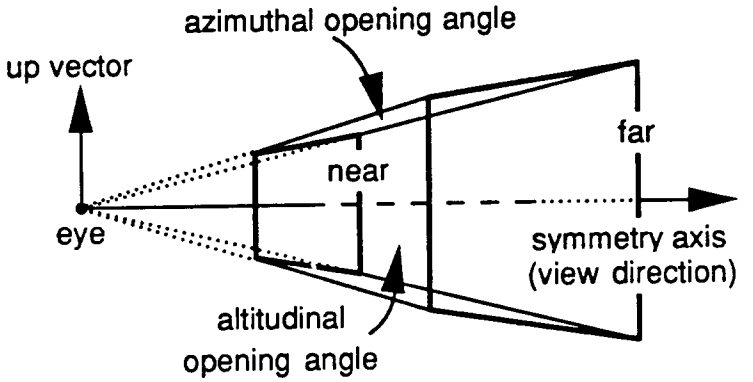


Figure 8.5: Three-dimensional observer view variables.

8.4.2 Eye-to-Cell Visibility

Recall that we have defined the eye-to-cell visibility set as those cells partially or completely visible to an actual observer. The eye-to-cell visibility computation can be cast as a DFS on the stab tree. The volume in the source cell visible to the observer is simply a convex polyhedron, the intersection of the frustum and the source cell. When a portal has been traversed, however, the view of the observer is generally obscured by occluders adjacent to the portal. The eye-to-cell DFS terminates when the observer's view is completely occluded (i.e., there exists no eye-centered stabbing line through the portal sequence).

Computing eye-to-cell visibility consequently reduces to the *existence* problem of determining eye-centered stabbing lines through 3D portal sequences. A list of normal vectors is initialized to the four normals of the frustum bounding planes, each oriented toward the interior of the frustum (Figure 8.6). As each stab tree portal is encountered, it is traversed in a specific direction. Each portal edge is oriented by the traversal; the portal edge and the eye span a plane, oriented so that its positive halfspace contains the interior of the portal. The plane normal is concatenated to a list, and linear programming is used to determine whether there exists a suitable stabbing line, i.e., a vector with a positive inner product with each of the constraint vectors.

These constraints are cast as a two-dimensional linear program as follows. If the k^{th} plane

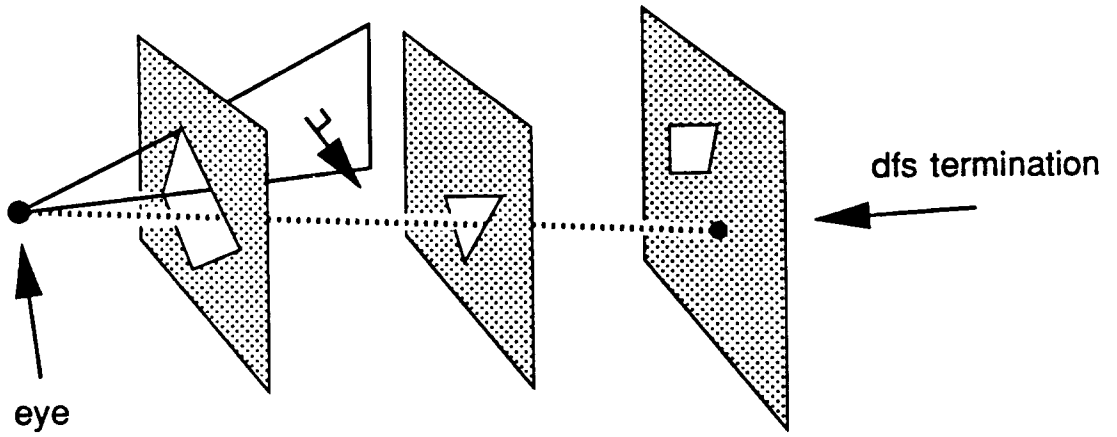


Figure 8.6: Encountering a portal in the eye-to-cell DFS.

equation has normal $\mathbf{n}_k$, any stabbing line through the eye must have a direction vector v such that

$$\mathbf{n}_k \cdot \mathbf{v} \geq 0, \qquad \text{for all } k. \tag{8.1}$$

Note that the linear program is two-dimensional since all of the planes contain the eyepoint; therefore, we need only compute a *vector* that has a non-negative dot product with each of the plane normals. We examine this collection of three-coefficient linear constraints for a feasible solution in linear time using a linear programming algorithm. If the linear program fails to find a stabbing line through the eye, the most recent portal is impassable, the reached cell is not visible through the current portal sequence, and the active branch of the stab tree DFS terminates.

There is an obvious optimization step applicable during the construction of the linear program: each edge of the newly encountered portal spans a plane with the eyepoint, oriented to contain the portal in its nonnegative halfspace. If any *previous* portal in the sequence lies entirely within the *negative* halfspace of this plane, we know immediately that the augmented sequence is impassable (Figure 8.7). On the other hand, if any previous portal in the sequence lies entirely within the non-negative halfspace of this plane, the plane clearly constitutes a superfluous constraint, and should not contribute to the linear program (Figure 8.8). In short, new planes only contribute linear constraints if they *partition every previous portal in the sequence into two areal pieces.*

The depth-first nature of the search ensures that portal sequences are assembled incrementally. Further, each newly encountered portal can be examined for a solution in time linear in the number of portal edges assembled. Thus, a portal sequence with n total edges can be processed in $O(n^2)$ time.

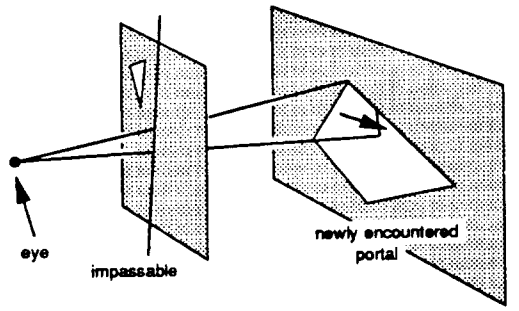


Figure 8.7: Detecting an impassable portal edge constraint.

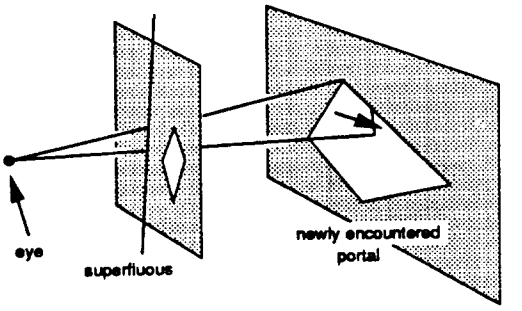


Figure 8.8: Detecting a superfluous portal edge constraint.

Recall Jones' 1971 visibility algorithm [Jon71], which traversed a spatial subdivision adjacency graph, projecting portals onto the view plane and solving the generalized clipping problem for each portal sequence (cf. Figure 2.1). Our algorithm is significantly different from that of Jones. First, we construct the spatial subdivision and portals automatically as a function of the occluders, rather than by hand. Second, we need not ensure that every occluder and detail object polygon end up on the boundary of a cell. Any particular entity can be ignored (treated as invisible); doing so will make our algorithms produce larger potentially visible sets, but not incorrect answers. Third, our off-line (*cell-to-cell*) stabbing line computation typically causes the eye-to-cell visibility computation to terminate sooner than Jones' query, since the eye-to-cell query will not examine portal sequences that are known *a priori* not to admit sightlines. This allows predictive visibility computations, i.e., knowledge that particular cells or objects cannot become visible within specified time intervals. Fourth, our algorithm produces an *unordered* set of potentially visible polygons, and assumes the existence of a depth-buffer to perform hidden-surface elimination. Finally, we use linear programming to compute the *existence* of the intersection of a collection of convex polygons, not the actual intersection (i.e., a convex polygon or the empty set), and therefore compute a superset of the visible objects, not an unnecessarily exact representation of their visible fragments. Consequently, for portal sequences with n edges, our approach has $O(n)$ complexity compared to the $O(n \lg n)$ complexity of Jones' approach. Indeed, since Jones made no distinction between occluders and objects, his algorithm as originally stated would not be applicable to any but the simplest environments in practice. We conclude that the eye-to-cell algorithm presented here is more storage-efficient and substantially faster than that of Jones, in both theory and practice.

Degeneracy

In practice, the observer often ranges so close to the plane of a portal that the planes spanned by the eye and portal edges are ill-determined, or identical to the plane of the portal itself. There is an elegant method of handling this degenerate occurrence. The eye position is compared against the plane equation of the portal. If the eye is closer to the portal than some tolerance, the plane equations are not assembled. Instead, the eye is assumed to lie on the plane of the portal, and it is checked against the two-dimensional portal interior. If it is found to be outside of the portal, the eye is on or very close to an opaque surface, and the DFS is terminated (Figure 8.9-i). If the eye lies in the interior of the portal, exactly one normal is appended to the list of assembled constraints: the normal to the portal plane, oriented so that it contains the cell to which the portal leads (Figure 8.9-ii). The DFS then proceeds normally.

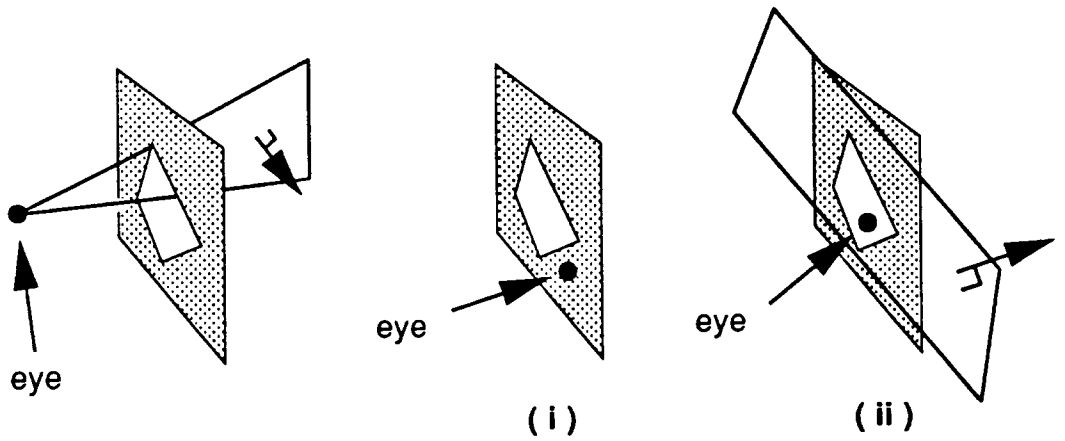


Figure 8.9: Handling degeneracies in the eye-to-cell DFS.

8.4.3 Eye-to-Region Visibility

The frustum is a convex volume; its intersection with any collection of planar halfspaces must therefore be convex. The volume potentially visible to the eye in any cell reached by the eye-to-cell DFS through a particular portal sequence is consequently a convex polyhedron. When a cell is reached, the assembled collection of plane normals is reconverted to plane equations, using the fact that each plane contains the eye point. The plane equations corresponding to the reached cell boundary are added to the list, and a $O(n \lg n)$ time 3D convex hull algorithm is invoked. The resulting convex polyhedron (guaranteed not to be empty by the establishment of a sightline to the

cell) is the eye-to-region visibility due to the current portal sequence, and has complexity at most linear in the number of portal edges in the sequence. The complete eye-to-region visibility in any reached cell is the union of all regions found visible through portal sequences reaching the cell.

8.4.4 Eye-to-Object Visibility

The convex hull computation above is not necessary to compute the eye-to-object visibility; again, the problem can be cast as an existence problem and solved using linear programming. Consider the pyramid with smallest solid angle, that has its apex at the eye and that contains a given axial bounding box. For any eye position outside of the bounding box, the pyramid has either four or six bounding planes (cf. §5.4, Figure 5.12); these are assembled via a table lookup on the position of the eye with respect to the six bounding box planes.

When the eye-to-cell DFS reaches a cell, the list of normals due to the portal sequence is available. Appended to this list, in turn, are the normals of the enclosing pyramid for each detailed object associated with (incident on) the reached cell. The resulting 2D linear program, analogous to that of Equation 8.1, has at most $n + 6$ constraints, and is solvable in $O(n + 6)$ time per object, where n is the total number of edges in the portal sequence reaching the cell.

Again, we briefly consider an alternative method of computing eye-to-object visibility, by constructing the polyhedral eye-to-region visibility explicitly and testing each object boundary against its interior. For general portal sequences, this latter method is aymptotically slower by a factor of $\lg n$, since constructing the convex intersection of n halfspaces requires $O(n \lg n)$ time. We have also observed this method to be slow in practice, due to the difficulty of constructing 3D convex hulls quickly and robustly. In fact, the degeneracy is worse than usual in the cell-to-region computation, since many of the involved planes contain the eyepoint.

Chapter 9

Results: Soda Hall Data

9.1 Geometric Data Set

The test case for these algorithms is a complex geometric model of a planned computer science building. The original floorplan and elevation data were obtained in AutoCAD format [Aut90], but have since undergone substantial, and crucial, alteration [Kho91]. Some alterations were relevant to visibility determination, and these are described in what follows. Many were effected as team efforts by members of the Walkthrough research group.

After the elevation data and floorplan were integrated into a collection of 3D wall, floor, ceiling, door, and window polygons, the data was converted into a convenient format. We chose the Berkeley UNIGRAFIX format [SSW83, SS91], since it is simple, human-readable, and has many associated utilities available in the form of textual filters. Moreover, oriented polygonal faces are central UNIGRAFIX definitional primitives.

The articulated (but unpopulated) geometric model was then subjected to a series of automated *filters* and to some interactive editing, to attain a sensible form [Kho91]. Interpenetrating or overlapping polygons were adjusted or removed. Misoriented polygons (e.g., wall polygons whose face normals pointed into the wall interior, not into the room interior) were reversed. Cracks, or small gaps, in the model were filled by subtly adjusting nearby polygons. This filtering task is formidable, perhaps even ill-defined, for general polyhedral environments. However, it seems tractable for architectural environments which, after all, are intended to represent physically realizable structures.

The filtering task was a practical necessity, rather than a theoretical one, since omitting it would only have increased the running time of the visibility algorithms, without compromising their correctness. In practice, we observed that leaving cracks uncorrected noticeably increased

the time spent in preprocessing, since each uncorrected crack (i.e., portal) became a conduit for many dynamic and on-line visibility tests that otherwise would not have occurred. In the sense of algorithmic correctness, the filtering operation is independent of any abstract notions of visibility determination. We have assumed, throughout, the availability of filtered input to the visibility computation module, and we perform no systematic analysis of the possible time and storage penalties incurred by violations of the filtering goals.

9.1.1 Test Models and Test Walkthrough Path

We extracted two test cases from the geometric model. The "partial" model comprised only the sixth and seventh floors, and the roof, of the building. The "full" model was all five completed floors and the roof; its geometric data required almost three times as much storage as that of the partial model. We constructed the two models in order to analyze the effect of tripling the model data size on visibility storage requirements and computation times.

Our goal in instrumenting the algorithms was to probe their effectiveness under realistic workloads, rather than random or systematic queries. Consequently, we gathered data about the effectiveness of the visibility algorithms by recording a representative "test path" defined during an actual interactive walkthrough session. The test path wanders through the sixth and seventh floors of the model, through the atrium, into and out of several offices, out and in a window, out over the terraces, and through the open-air regions among the columns (Figure 9.2). The path spends considerable time in free-space surrounding the building. These regions challenge the more discriminating culling operations, since almost all portal sequences there admit eye-based sightlines. Consequently, the eye-based DFS spends considerable time traversing empty cells.

9.2 Implementation

9.2.1 Programming Methodology

The visibility algorithms described in this thesis have been implemented in the C language on a Silicon Graphics 340 GTX superworkstation. The implementation mirrors the progression of this thesis; the abstract operations were first implemented for 2D occluders, for axial 3D occluders, and finally for general 3D occluders.

These algorithms exist as part of a large collection of programs that support interactive graphical visualization and manipulation of geometric data. The development of each algorithm was facilitated

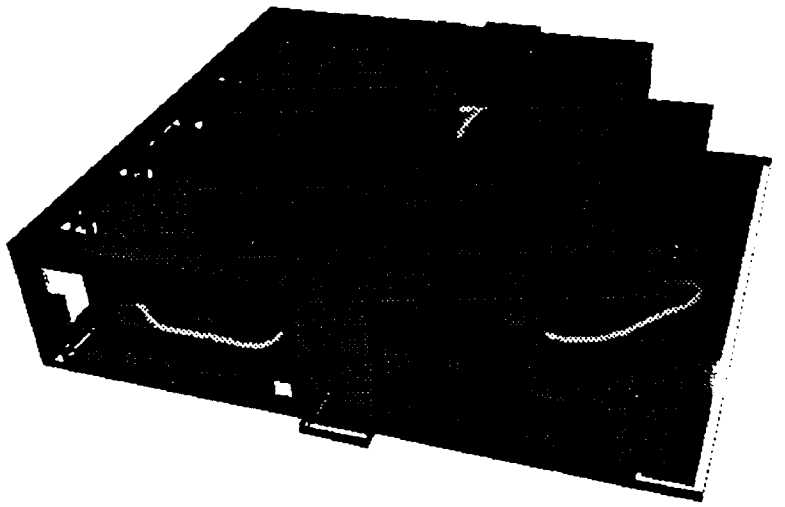


Figure 9.1: The top two floors and roof of the geometric model. The test path (grey) snakes in and out of the building.

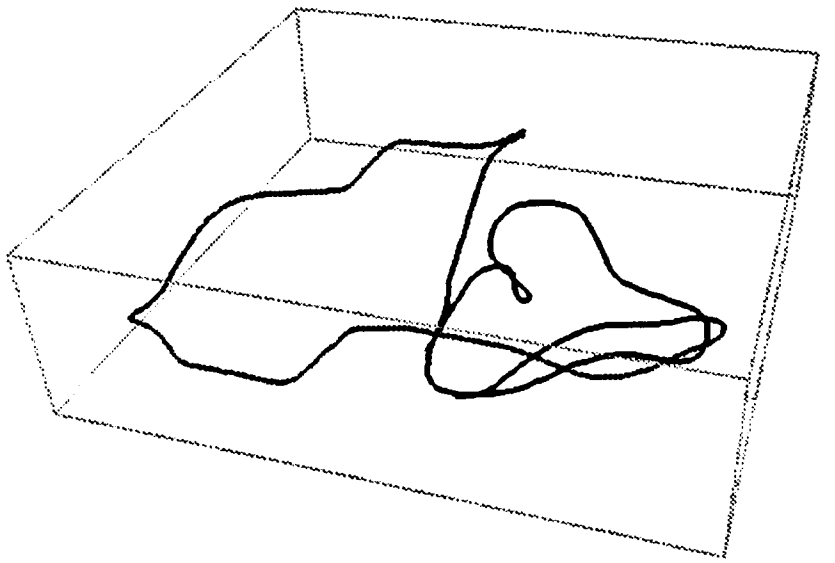


Figure 9.2: The test path, grey-scaled from low z-values (black) to high z-values (white).

by a "visual programming" framework in which visual, selectable instantiations were defined for each geometric data type, in order not only to display the behavior of the data, but to manipulate it directly, for example by use of an interactive input device. This implementation methodology had some surprising and beneficial aspects, and some noteworthy implications.

First, the notion of *displaying* geometric data objects as they are manipulated gives powerful clues as to the correct or incorrect behavior of associated algorithms. We found that if the computed solution "looked wrong," i.e., contradicted intuition, it generally *was* wrong, or at least merited further examination. Second, *manipulating* a geometric algorithm's input with an interactive device is a powerful method for exploring the space of possible inputs. This coverage of, for example, input degeneracies and boundary cases facilitates the development of extremely robust algorithms, because instances of exceptional input are effortlessly generated as test cases. Third, the easy availability of a graphics infrastructure for prototyping algorithms removed much of the onus of implementing complex geometric algorithms.

9.2.2 Free-Space

In practice, our system encountered a difficulty that we termed the "free-space" problem. Define a *free-space* cell as one incident on no major occluders or detail objects; i.e., an empty cell all of whose boundaries are completely transparent. Two factors caused clumps of free-space cells to arise while subdividing the building model: first, non-local splitting caused by the early k-D tree subdivision planes; second, subdivision of large cells and bad-aspect cells (for example, large open areas such the balconies and atrium, or long corridors) to reduce their visibility to and from the remainder of the model.

The free-space problem arises because of conflicting demands on the subdivision method. First, it should exploit the presence of occluders wherever possible (e.g., by splitting along them). Cell boundaries should therefore be obscured by occluders. Second, the subdivision method should produce cells that have usefully small cell-to-cell visibilities, thus highly constraining the generalized observer, even when no occluders are present. This forces some splitting to occur in regions of little "nearby" occlusion.

One important future aspect of this research is the development of improved spatial subdivision techniques for both axial and general three-dimensional data. In the interim, however, we rely on ad-hoc techniques for handling problematic local cell configurations such as clusters of free-space cells.

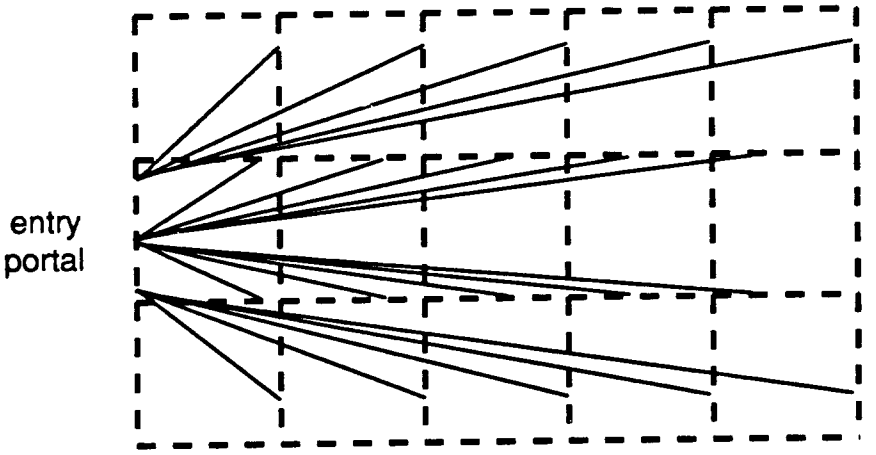


Figure 9.3: Naive search through free-space causes a combinatorial explosion. Dashed lines are portals; some of the sightlines computed are shown.

Clusters of free-space cells, treated naively, slow both the precomputation and query visibility phases. During precomputation, a sightline search encountering the cluster explores all combinatorial paths through it (Figure 9.3), expending time and storage to compute essentially no useful information (since no objects are populated in free-space cells, no cell-to-object visibility determinations can be made). During the query phase, the eye-to-cell search expends computational effort to find eye-based sightlines, and manipulates dynamic stacks of halfspace constraints that need never be applied to discover eye-to-cell visibility. This search is wasteful in that it discovers no visible objects to be drawn; it is necessary, however, because otherwise the traversal algorithms would not be able to proceed beyond free-space cells to more typical cells containing visible objects. Consequently, we have developed a method which searches through a free-space cluster, and emerges from its perimeter, in time comparable to that required for searching a single cell.

Handling Free-Space Cells

We developed a method for handling free-space cells which is a straightforward generalization of the cell and adjacency graph, and of the notion of traversing such a graph. Moreover, in the case of sparsely occluded environments such as terrains, our method has the desirable feature that it degenerates to a purely spatial cull (cf. [GBW90]) – probably as well as one can do in such an environment without a fully general object-space hidden-surface algorithm.

We generalize the notion of a spatial cell and define a *metacell* as any connected cluster of *constituent* cells. The metacell is generally non-convex, and has a portal set containing all

constituent cell portals that lead to a non-constituent cell. Note that this definition prevents a metacell portal from leading into itself or another metacell, a desirable property that simplifies the abstract operations defined over metacells. Point-locating into any constituent cell is equivalent to point-locating into the associated metacell.

The geometric notion of portals is unchanged. However, without cell convexity, it is no longer true that portals in general are mutually visible through their common cell interior (Figure 9.4). Consequently we associate, with each portal *entering* the cell, a list of *portal crossover* constraints which, given the metacell and an entry portal, catalogues those portals through which the metacell may be exited. The crossover constraints can be statically computed by finding all pairs of portals A and B that admit a stabbing line in the interior of the metacell, and can be done with the standard cell-to-cell DFS, constrained to traverse only constituent cell portals. Although this sub-problem may be combinatorially expensive, the advantage of constructing the portal crossover lists is that it need be done only *once* per metacell (e.g., when the cell is first encountered during visibility precomputation), rather than once each time any of the constituent cells is encountered, as in the unmodified scheme. Finally, note that the size of the portal-crossover table can be $O(p^2)$, where p is the number of constituent cell portals in the metacell.

Figure 9.4, depicts a sightline search that enters a metacell via portal A, but that is unable to leave the metacell via portals C or K since, to do so, the sightline would have to reverse itself (i.e., the portal pairs $[A,C]$ and $[A,K]$ do not admit sightlines). The stabbing line search further rules out exit via portals D or J because no sightline entirely within the metacell exists to either of them from A. Thus the portal crossover constraints for portal A comprise the remaining metacell portals, the set B, E, F, G, H, I, L.

Once the notion of a conforming subdivision has been generalized to include metacells, we need only generalize the abstract operation to be performed upon encountering a metacell in the precomputation and query phases. Recall that, in the precomputation phase, the sightline and cell-to-region DFS arrives at a cell with a set of halfspaces encoding the portal constraints encountered so far. When encountering a metacell, rather than recursing and searching through the constituent cells, we merely subject the constituent cell *extents* to the cell-to-object computation as if they were objects, and continue the search using the portal crossover list, as a function of the entry portal (Figure 9.5). (The constituent cells, by definition, contain no occluders or objects to be found visible, and consequently can be ignored for the purposes of computing cell-to-cell or cell-to-object visibility.) Note that without the portal crossover constraints, this algorithm could recurse indefinitely by entering and leaving the same metacell in a single portal sequence.

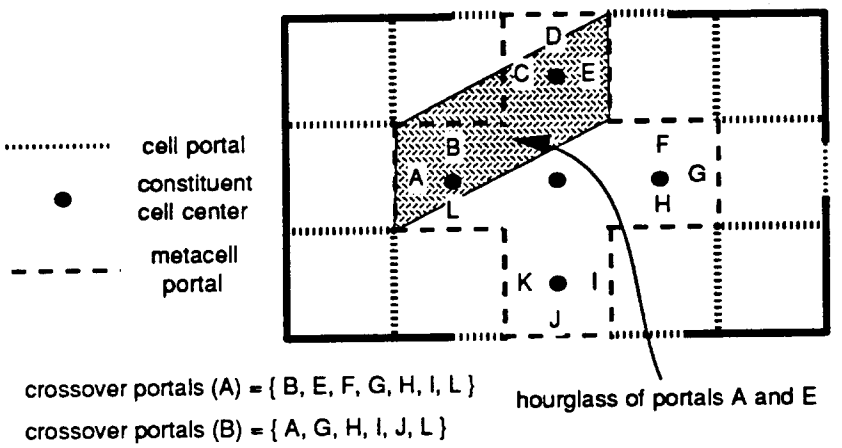


Figure 9.4: A metacell (bold dashed outline), with portal crossover constraints attached to each entry portal.

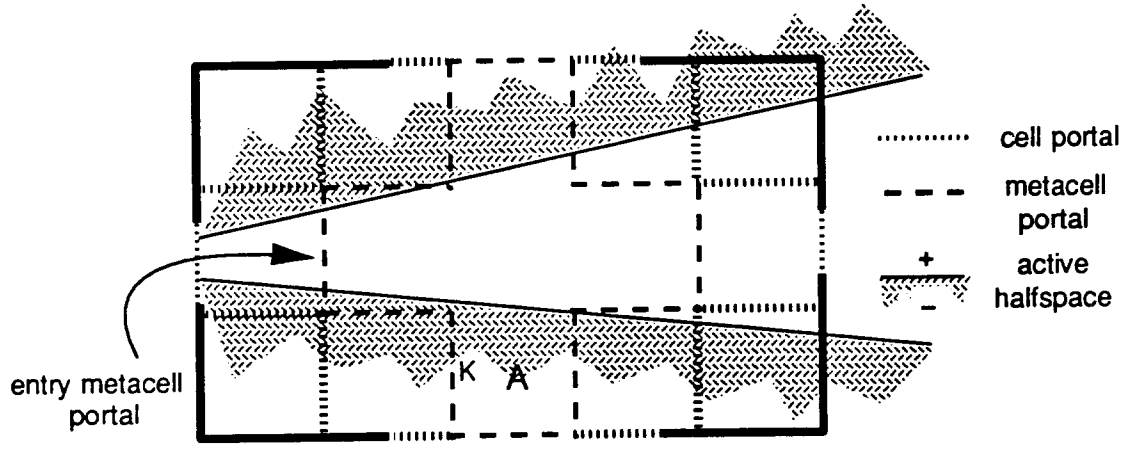


Figure 9.5: Encountering a metacell during a constrained DFS.

Similarly, during the query phase, when the eye-to-cell and eye-to-region DFS arrives at a metacell, it subjects the constituent cells to the active halfspace constraints. Note that reporting the incident cells visible, and subjecting their contents to the eye-to-object test, will generally be unnecessary since the cells have no associated occluders or detail objects. Moreover, the portal crossover list can be used to recurse, through visible portals only, out of the metacell to surrounding cells. For example, in Figure 9.5, the constituent cell A is not visible to the observer, and the crossover list portal K should not be traversed.

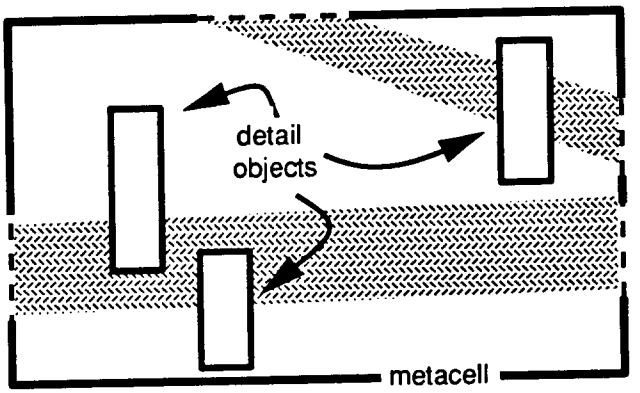


Figure 9.6: Detail objects can prohibit inter-portal visibility (constituent cells not shown).

Note that the portal crossover constraints serve not only to generalize the notion of cells to include non-convexity, but might also be useful in determining the effect of incident detail objects on inter-portal visibility within a given source cell (Figure 9.6). We have not fully explored this possibility.

9.3 Utility Metrics

Several *metrics* serve to evaluate the effectiveness of the visibility preprocessing and on-line culling scheme. These are empirical metrics, in that they describe quantities measured by instrumenting algorithms to record their storage needs, culling and drawing time, output size, visibility overestimation, and depth complexity, in units to be described below.

The *storage overhead* metric quantifies the storage overhead incurred in computing and storing visibility information for a geometric model, expressed both in absolute terms and as a percentage of the storage required to store the model elements (the model "geometry" [FST92]) alone. We do not formulate an analogous *time overhead* metric, since it is difficult if not impossible to objectively

weigh the value of off-line CPU resources expended in light of the fact the on-line simulation may be run indefinitely long for any given populated model with precomputed visibility.

The *culling time* and *drawing time* metrics quantify the time, in milliseconds, required for the culling operation (i.e., producing a collection of potentially visible objects), and the drawing operation (i.e., rendering the collection), as a function of the culling mode. The *frame time speedup* quantifies how much the visibility computation accelerated the rate of delivery of successive images to the framebuffer, compared to rendering using only the graphics hardware and no on-line culling. A frame was *cull-bound* if culling took longer than drawing, and *draw-bound* if drawing took longer than culling; the frame-time speedup is defined as the base (unculled) drawing time divided by the maximum of the culling and drawing times. (In practice, things are more complicated, since the culling and drawing processes may run in parallel and, depending on transport delay, may be almost entirely decoupled [Jon92, FST92].)

For all timing measurements, we simulated a machine with infinite memory (or, equivalently, a perfect database intermediary that incurred no swapping penalty) by "touching" potentially visible objects to bring them in to memory, then performing the culling or drawing operations in a timing loop. We used a hardware clock with 16 μ-second resolution to time operations typically requiring tens of milliseconds. All timing measurements reflect "wallclock time"; that is, we did not instrument CPU usage, but actually measured time-to-completion of the culling and drawing operations as if using a high-accuracy stopwatch. The machine was otherwise quiescent during the timing runs.

The *output size* metric quantifies the efficacy of the preprocessing and on-line culling algorithms in upper-bounding the potentially visible set for generalized and actual observers. This metric can be expressed in units of visible volume, visible objects, or visible polygons. None of these choices is entirely satisfactory. Visible volume units do not account for the fact that the model space is not uniformly populated with detail objects. Visible object units do not account for the differing times required to render different objects. Finally, visible polygon metrics can not encompass the existence of level-of-detail (LOD) algorithms that strive to render far-away and fast-moving objects with fewer polygons than close-by objects presumably visible in greater detail. No level of detail polygon-reduction algorithms [FST92] were used; we worked only in units of objects as defined by the modeling system, and rendered all objects at a constant level of detail (the highest available).

Although LOD algorithms do draw *fewer* polygons for far-away objects, the polygons drawn are typically *larger*, on average. Large polygons can cause the graphics hardware we used to become "fill-limited" [FK91], i.e., exhibit bottlenecks during scan conversion and pixel writes. The

result is that, although LOD modeling might reduce the number of polygons drawn by a factor of five or more, the drawing time typically decreases by only a factor of two to three [FST92]. On the other hand, eschewing LOD analysis can result in wasted detail rendering, needlessly high depth-complexity, and *transform-limited* drawing, i.e., a bottleneck at the top of the graphics pipeline. We have tried to minimize these confounding factors by rendering at a constant level of detail throughout, and by counting rendering in units of objects, assuming the time to render any given object is roughly constant.

The *object survival* and *object overestimation* metrics quantify how much of the graphics hardware capacity is "wasted" by rendering invisible objects (i.e., by classifying invisible objects as potentially visible). We say that an object *survives* in a rendered image if some pixel in the image is painted by some polygon of the object and this pixel is never obscured by a closer polygon. Clearly a culling operation with perfect knowledge would render *only* surviving objects; our culling algorithms generally overestimate by rendering non-surviving objects. One measure of how well the culling performs is the percentage by which object visibility is *overestimated*; i.e., the ratio of the number of objects that are rendered to the number that actually survive in the framebuffer.

The *depth complexity* metric quantifies the reduction in pixel depth complexity, or the number of times each pixel is "painted", due to on-line culling (a depth-buffer test counts as a painting operation). A depth complexity of 1.0 can occur only in an environment containing only a single convex object (with backfaces removed), or when using an "ideal" visibility algorithm with perfect knowledge of the entities and entity fragments visible to the observer (e.g., a screen-space scan-line renderer that writes each pixel exactly once). We ignore the fact that each pixel is painted exactly once while clearing the framebuffer at the start of each frame. This fixes the theoretically optimal depth complexity at one rather than at two (the former choice is more intuitive).

In practice, the lowest-achievable depth complexity is substantively higher than one, since the whole point of depth-buffering is to obviate fragmentation of polygons for rendering; consequently if polygons have to be either drawn, or not drawn, they will overlap substantially in any given (interesting) scene and even exact visibility algorithms (those that render only surviving entities) must produce depth complexities greater than one. Finally, we note that the computed depth complexity may sometimes be less than 1.0, since some scenes (for example, when looking outside of the building) do not paint the entire window.

Analogously, we justify our use of object-based utility metrics, rather than metrics based on polygons, on the fact that the abstract visibility operations don't "know" about polygons – but only about objects. The culling algorithms make binary decisions about objects, determining only

whether to draw them or not. The definition of what constitutes an object is more properly left to the modeling agent than to the visibility scheme.

We quantify depth complexity discretely using the graphics hardware. The rendering engine can be configured to count, in hardware, the number of times each screen pixel is touched during rendering. In Silicon Graphics terminology, depth-buffering is disabled, the framebuffer is cleared to zero, the color of each polygon is set to the integer 1, and the blending function is set to *add* each rendered pixel value to the existing framebuffer value. The final values of each pixel are then summed and divided by the total number of pixels (in our case, 800,000 pixels in a window 1,000 pixels wide by 800 high).

Each of the six metrics – objects drawn, survived, and overestimated, culling and drawing times, and depth complexity – were tabulated (average, minimum, and maximum, where appropriate) for each of seven interesting culling modes. In the following, the "observer cell" is that cell containing the actual observer. The seven culling modes were (Figure 9.7):

1. **No Culling (NC)**: all cells and all objects were drawn (not shown in Figure 9.7).

2. **Spatial Culling (SC)**: cells and objects incident on the view frustum were drawn. We implemented the method of [GBW90] on a conforming subdivision, but did not parallelize the cull as did those researchers.

3. **Cell-to-Cell, Cell-to-Object (CtC,CtO)**: all objects potentially visible from the observer cell were drawn. This metric quantifies how well we can do with no on-line computation.

4. **Frustum-to-Cell, Frustum-to-Object (FtC,FtO)**: all objects incident on the view frustum, and visible from the source cell, were drawn. This mode intersects the observer cell's generalized visibility with the actual (i.e., instantaneous) view frustum.

5. **Eye-to-Cell, Cell-to-Object (EtC,CtO)**: all objects visible from the source cell, and in a cell reached by the eye-to-cell traversal, were drawn. This mode performs sightline computations for cells, and no computation for objects.

6. **Eye-to-Cell, Frustum-to-Object (EtC,FtO)**: all objects visible from the source cell, in a cell reached by the eye-to-cell traversal, and incident on the view frustum, were drawn. This mode performs sightline computations for cells, and frustum-incidence for objects.

7. **Eye-to-Cell, Eye-to-Object (EtC,EtO)**: all objects admitting a sightline from the eye were drawn. This is the "exact" on-line cull described in §5.4.4, in that it performs sightline com-

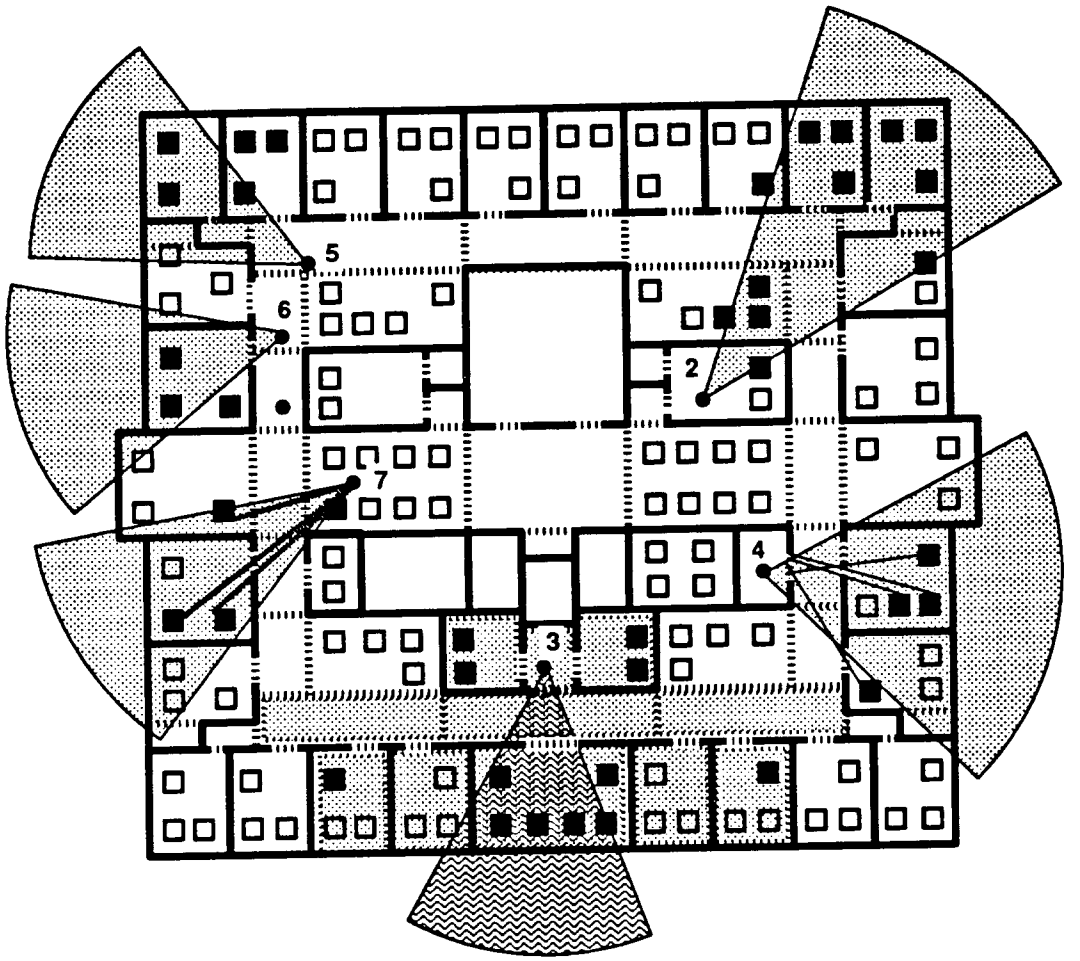


Figure 9.7: The instrumented culling modes (NC mode not shown).

putations for both cells and objects. It is as discriminating as is possible without knowledge of the detail objects' constituent polygons.

Some of the metrics were not tabulated for all culling modes. For example, the object survival metric should be constant regardless of culling mode. We verified that this was the case in practice, and so tabulate it only for the most discriminating culling mode. The culling time for some methods – No Culling and Cell-to-Object – is theoretically zero (in practice, considerable time is required to make the visible sets available to the rendering process [FST92])). The pixel depth complexities for methods differing only by a frustum incidence operation – No Culling vs. Spatial Culling; CtC,CtO vs. FtC,FtO; and EtC,CtO vs EtC,FtO; – are verifiably identical. These simplifications are evident in the tables to follow.

9.4 Experimental Results

9.4.1 Test Walkthrough Path

We sampled every eighth element of a 4,000-frustum recorded walkthrough path through the building model to produce a 500-frustum path (the top two floors and roof of the model, and the test path, are shown in Figure 9.1). (The statistics instrumentations absorbed tens of hours of CPU time, so the unsampled path would have used unreasonable machine resources.) The field of view was constant at 90 degrees wide and 77.3 degrees high (the latter figure was determined by the 5-to-4 aspect ratio of the test window), subtending roughly 15 percent of the observer's view sphere.

We instrumented the culling algorithms to run on both the partial and full models. By constraining the test path to the top portion of the building we were able to utilize it in both models, and compare the resulting storage and time metrics from each run.

9.4.2 Storage Overhead and Precomputation Times

Partial Model

The partial geometric model consisted of the sixth and seventh floors, and roof. In total, this model comprised 4,598 objects and 238,861 polygons (at the highest level of detail), of which 3,871 (about 1.6%), were classified by the modeling system as occluders. Of these, 325 occluders (about 8.4%) were not axial rectangles and were ignored during the spatial subdivision rounds. Thus, more than ninety percent of the occluder-sized polygons in the geometric model supplied suitable

axial splitting planes to the spatial subdivision rounds (this is consistent with the observations of an earlier visibility researcher who found that roughly ninety-five percent of the candidate splitting polygons in his test model were axial [Air92]).

Selecting candidate occluders and subdividing the k-D tree to termination required 55 seconds. Once the subdivision directives were stored, regenerating the k-D tree into 2,163 leaf cells, and enumerating the 11,500 portals of the conforming spatial subdivision, required 6 and 8 seconds, respectively.

Storing the occluders and conforming spatial subdivision required about 3 megabytes (the details of the database format are given in [FST92]). Inserting the detail objects at all levels of detail took about one hour, and increased the size of the model to 62.5 megabytes. Finally, computing visibility information for the model took almost five hours, and increased the model size by about 5.2 megabytes to a total size of 67.7 megabytes. (More than one and half million portal sequences were attempted, with average length of 9.07 portals. The average length of a *failed* portal sequence, i.e., one that did not admit a sightline, was 9.94 portals. The successful and failed portal sequence lengths do not differ by exactly 1.0 because many branches of the DFS terminated not due to impassable portals but to the fact that many cells had only one entrance and no exit, other than the entry portal, which was known *a priori* to be impassable. The reaching portal sequence therefore counted as a success, with no incremental test and failure.) Thus, the storage overhead incurred by visibility computation was less than ten percent of the size of the polygon data in the geometric model (less than fifteen percent if the conforming subdivision is treated as visibility data).

Full Model

The entire geometric model consisted of the third through seventh floors, and roof. In total, this model comprised 13,191 objects and 712,499 polygons (at the highest level of detail), of which 9,699 (about 1.4%), were classified by the modeling system as occluders. Of these, 550 (about 5.7%) were not axial rectangles and were ignored during the spatial subdivision rounds. As in the partial model, more than ninety percent of the occluder-sized polygons in the geometric model supplied suitable axial candidate splitting planes.

Selecting candidate occluders and subdividing the k-D tree to termination required 4 minutes. Regenerating the k-D tree into 5,762 leaf cells, and enumerating the 34,916 portals of the conforming spatial subdivision, required 26 and 21 seconds, respectively,

Storing the occluders and conforming spatial subdivision required about 8.4 megabytes. In-

serting the detail objects at all levels of detail took about four hours, and increased the size of the model to 175.3 megabytes. Finally, computing visibility information for the model leaf cells took almost nineteen hours[1], and increased the model size by about 13.7 megabytes, to a total size of 189.0 megabytes. (We did not compile portal sequence length statistics for this data set.) Thus, the storage overhead incurred by visibility computation was less than 8% of the size of the polygon data in the geometric model (less than 15% if the conforming subdivision is treated as visibility data).

9.4.3 Tabulated Utility Metrics

Tables 9.1 through 9.10 summarize the data for the partial and full models (the tables are arranged in pairs). The most important figures of merit are shown in bold.

Averaged Object and Pixel Depth Metrics

Culling Mode	# Objects Drawn	% Objects Drawn	# Objects Survived	Object Overestimation (%)	Discrete Depth Complexity
None	**4598.0**	100.0%	**40.4**	**11000%**	3.31
Spatial	988.2	21.5%	"	2400%	3.31
CtC,CtO	385.8	8.4%	"	850%	1.78
FtC,FtO	100.0	2.2%	"	150%	1.78
EtC,CtO	87.2	1.9%	"	120%	1.56
EtC,FtO	65.6	1.4%	"	60%	1.56
EtC,EtO	**49.7**	**1.1%**	"	**23%**	1.44

Table 9.1: Object and pixel metrics for the partial model.

The disparity between object overestimation and pixel depth complexity is surprising in both models (Tables 9.1 and 9.2). While the number of objects drawn varied by a factor of more than one hundred, the depth complexity varied by only a factor of about three. Depth complexity grows slowly with the number of objects since, as objects get farther away, they paint fewer screen pixels. The difference between the object counting and depth complexity metrics arises from the fact that the culling operations are best at culling remote objects, but these typically paint relatively few screen pixels. In contrast, nearby objects paint relatively large screen areas but do not interact in the culling operation; i.e., they cannot prevent each other from being rendered, as can occluders.

[1] Static visibility computations were truncated after thirty minutes for free-space cells.

Culling Mode	# Objects Drawn	% Objects Drawn	# Objects Survived	Object Overestimation (%)	Discrete Depth Complexity
None	13191	100 %	**40.4**	32600 %	4.93
Spatial	2433.5	18.5 %	"	5900 %	4.93
CtC,CtO	418.8	3.2 %	"	940 %	1.84
FtC,FtO	105.4	0.80%	"	160 %	1.84
EtC,CtO	90.8	0.69%	"	124 %	1.58
EtC,FtO	66.7	0.51%	"	65 %	1.58
EtC,EtO	49.5	0.38%	"	**22.5%**	1.46

Table 9.2: Object and pixel metrics for the full model.

The number of objects passed by each culling mode, on average, drops monotonically with increasing cull complexity. Note that static (i.e., cell-to-object) alone reduces the amount of visible data to 8.4% of the model, on average; subjecting this generalized visibility to a frustum-incidence operation reduces this amount by almost three-quarters, to 2.2% of the model (in a roughly spherically symmetric model, we expect frustum-incidence to reduce the generalized visibility by the fraction of the unit sphere excluded by the view frustum – in our case, about 85 percent).

Finally, note that the number of surviving objects is identical in the partial and full models (viz. Table 9.1, Table 9.2). This is expected, since object survival should be independent of the spatial subdivision or (superset) culling algorithm used.

Average Object Counts and On-Line Operation Times

Culling Mode	# Objects Passed	% Objects Culled	Cull Time (msec)	Draw Time (msec)	Frame Time Speedup Factor
None	4598	0.0%	n.a.	2805.2	1.0
Spatial	988.2	78.5%	73.1	655.0	4.3
CtC,CtO	385.8	91.6%	n.a.	252.3	11.1
FtC,FtO	100.0	97.8%	6.4	96.4	29.1
EtC,CtO	87.2	98.1%	25.6	90.9	30.9
EtC,FtO	65.6	**98.6%**	**26.1**	**72.9**	38.5
EtC,EtO	49.7	**98.9%**	**75.8**	**55.0**	37.0

Table 9.3: Culling and drawing time metrics for the partial model.

Culling Mode	# Objects Passed	% Objects Culled	Cull Time (msec)	Draw Time (msec)	Frame Time Speedup Factor
None	13191	0 %	n.a.	8900	1.0
Spatial	2433.5	81.6%	192.4	2050	4.3
CtC,CtO	418.8	96.8%	n.a.	316.7	28.1
FtC,FtO	105.4	99.2%	7.3	103.9	85.7
EtC,CtO	90.8	99.3%	33.2	96.9	91.8
EtC,FtO	66.7	99.5%	**33.5**	76.5	**116.3**
EtC,EtO	49.5	99.6%	**86.8**	56.9	**102.5**

Table 9.4: Culling and drawing time metrics for the full model.

Note that, for either model, the two most discriminating modes exhibit culling and drawing times that bracket an intermediate frame rate (in this case, about 50 milliseconds). In the partial model, **EtC,FtO** culling requires only 26 milliseconds but produces a potentially visible object set requiring 73 milliseconds to draw (Table 9.3). The exact **EtC,EtO** cull takes three times as long to complete, but produces a potentially visible set 25 percent smaller (50 vs. 66 objects), reducing the corresponding rendering time by about a quarter to 55 milliseconds (57 milliseconds in the full model[2]). In practice, parallelizing the culling and drawing processes decreases the dependence of frame time on worst-case cull time, but at the cost of increased transport delay.

Average, Minimum and Maximum Object Counts

The purely spatial cull exhibited a wide variance in the number of objects passed for rendering: from one-fifth of one percent of the model to almost seventy-five percent of it (Table 9.5). In practice, the real worst-case frame time for this culling mode is much worse than the six-tenths of a second we quote using the infinite-memory machine model. Since at worst seventy-five percent of the model data (about 45 megabytes) must be read from disk and issued to the graphics pipeline, even (optimistically) assuming a one megabyte per second disk bandwidth, drawing the data returned by the spatial cull would take about forty-five seconds. Drawing all objects in the the partial model (i.e., performing no culling) would take a full minute *per frame*, even under this optimistic bandwidth assumption; drawing the full model in its entirety would require three minutes.

Static culling (**CtC,CtO**) exhibited worst-case maximum visibility of about a fourth of the

[2]Small discrepancies in times between the partial and full models arise from timer aliasing and different caching effectiveness while simulating the two data sets.

Culling Mode	Avg. # (%) Objects Passed	Min # (%) Objects Passed	Max # (%) Objects Passed
None	4598 (100 %)	4598 (100 %)	4598 (100 %)
Spatial	988.2 (21.5%)	9 (0.19 %)	3439 (74.8 %)
CtC,CtO	385.8 (8.4%)	88 (1.9 %)	1130 (24.6 %)
FtC,FtO	100.0 (2.2%)	6 (0.13 %)	455 (9.9 %)
EtC,CtO	87.2 (1.9%)	8 (0.17 %)	298 (6.5 %)
EtC,FtO	65.6 (1.4%)	5 (0.11 %)	218 (4.7 %)
EtC,EtO	49.7 (1.1%)	1 (0.022 %)	177 (3.8 %)

Table 9.5: Average, minimum and maximum object metrics for the partial model.

Culling Mode	Avg. # (%) Objects Passed	Min # (%) Objects Passed	Max # (%) Objects Passed
None	13191 (100 %)	13191 (100 %)	13191 (100 %)
Spatial	2433.5 (18.5 %)	66 (0.5 %)	8278 (62.8%)
CtC,CtO	418.8 (3.2 %)	131 (1.0 %)	1257 (9.5%)
FtC,FtO	105.4 (0.8 %)	6 (0.05 %)	462 (3.5%)
EtC,CtO	90.8 (0.69%)	17 (0.1 %)	313 (2.4%)
EtC,FtO	66.7 (0.51%)	5 (0.04 %)	219 (1.7%)
EtC,EtO	40.1 (0.30%)	1 (0.007%)	136 (1.0%)

Table 9.6: Average, minimum and maximum object metrics for the full model.

partial model, or a tenth of the full model. All of the occlusion-based dynamic culling modes did much better. The least discriminating occlusion-based cull mode **FtC,FtO**, categorized no more than 10% (3.5%, in the full model) of the model objects visible. As the complexity of the dynamic cull mode increases, the maximum number of objects drawn decreases monotonically, as expected.

Average, Minimum and Maximum Cull Times

The minimum cull time roughly the same for each culling mode. In the case of purely spatial culling, this is coincidentally due to the fact that the test path leaves the building and the actual observer looks outside of the model; an observer keeping to the core of the building would experience consistently higher spatial cull times (and consistently lower sightline-based cull times due to the rarity of free-space cells in the core regions).

Culling Mode	Average Cull Time (msec)	Minimum Cull Time (msec)	Maximum Cull Time (msec)
None	n.a.	n.a.	n.a.
Spatial	73.1	4.1	225.4
CtC,CtO	n.a.	n.a.	n.a.
FtC,FtO	6.4	1.6	21.2
EtC,CtO	25.6	2.7	159.0
EtC,FtO	26.1	2.8	157.8
EtC,EtO	75.8	2.8	**499.7**

Table 9.7: Average, minimum and maximum culling times for the partial model.

Maximum culling times (Table 9.7) behave differently than object counts as a function of culling mode. The maximum cull time no longer drops monotonically as culling effectiveness increases, but is minimum at **FtC,FtO** (21 milliseconds, in the partial model) and climbs again as the sightline-based cull is applied first to cells, then the frustum-based cull to objects, then finally the sightline-based cull to both cells and objects. The sightline-based cull exhibited a worst-case computation time of half a second in the partial model, five times slower than the desired 10 Hz frame rate, which occurred in a complex free-space region outside the model. For the full model, the worst-case culling time increased by about 170 milliseconds (34%).

Culling Mode	Average Cull Time (msec)	Minimum Cull Time (msec)	Maximum Cull Time (msec)
None	n.a.	n.a.	n.a.
Spatial	192.4	14.0	555.6
CtC,CtO	n.a.	n.a.	n.a.
FtC,FtO	9.1	2.1	46.7
EtC,CtO	31.0	3.5	429.5
EtC,FtO	30.6	2.7	318.8
EtC,EtO	86.6	2.8	669.2

Table 9.8: Average, minimum and maximum culling times for the full model.

Average, Minimum and Maximum Draw Times

The average and maximum drawing times drop monotonically with increasingly expensive culling modes for both the partial and full model, as expected.

Culling Mode	Average Draw Time (msec)	Minimum Draw Time (msec)	Maximum Draw Time (msec)
None	2805.2	2732.2	3339.0
Spatial	655.0	18.1	2069.5
CtC,CtO	252.3	37.6	957.3
FtC,FtO	96.4	13.5	509.8
EtC,CtO	90.9	13.0	330.3
EtC,FtO	72.9	13.3	247.7
EtC,EtO	55.0	7.9	157.6

Table 9.9: Average, minimum and maximum drawing times for the partial model.

When no culling is employed, the draw time is fairly consistent at just under three seconds for the partial model, and just over nine seconds for the full model. Some variation occurs because, although all of the model data is drawn each frame, the graphics hardware expends differing amounts of time to dispense with polygons that are, for example, outside the view frustum, partially clipped, backfacing, or obscured by nearer polygons, and these types of polygons occur in different relative numbers as the actual observer moves through the model.

If the infinite-memory model is abandoned, drawing either the full model or the entities passed

Culling Mode	Average Draw Time (msec)	Minimum Draw Time (msec)	Maximum Draw Time (msec)
None	9146.4	9042.3	9485.1
Spatial	2052.3	89.1	6253.0
CtC,CtO	316.7	81.7	1138.7
FtC,FtO	103.9	18.7	513.3
EtC,CtO	96.9	14.4	373.4
EtC,FtO	76.5	14.3	266.1
EtC,EtO	56.9	8.1	159.1

Table 9.10: Average, minimum and maximum drawing times for the full model.

by the purely spatial cull would require more than two minutes.

For both models, the spatial culling mode exhibited widely varying draw times (Tables 9.9 and 9.10), spending as much as two to six seconds (on the idealized infinite-memory machine). The dynamic culling modes exhibited draw times in a tighter range, most notably the **EtC,EtO**, which in the worst case passed objects that required about 160 milliseconds to draw in either model. That is, the worst-case drawing time increased by about one-and-a-half milliseconds (1.0 %) under a tripling of the model complexity.

9.4.4 Museum Park

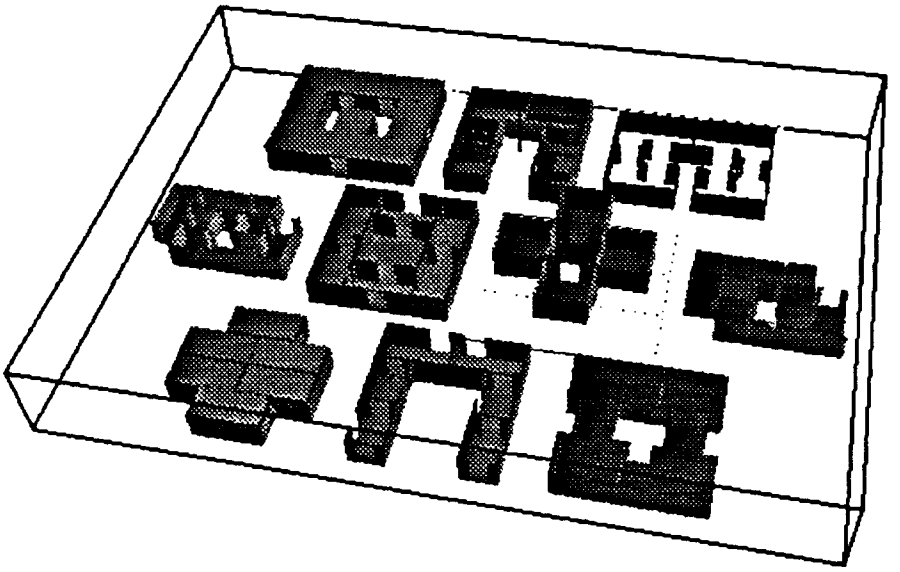
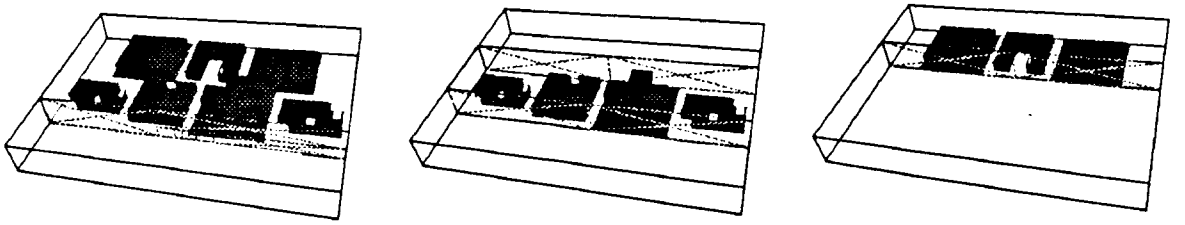


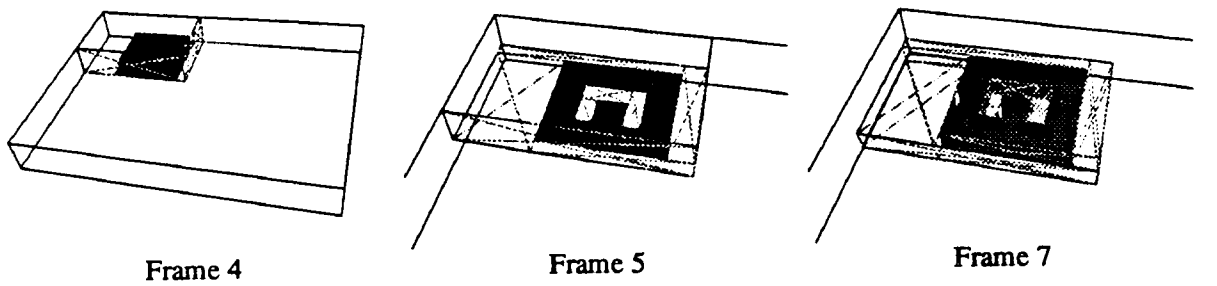
Figure 9.8: The ten museums in museum park.

We briefly tested the spatial subdivision algorithm, which had been devised for building interiors, on a data set that we call "museum park." Ten one- and two-story structures with similar footprints were assembled in a loose array (Figure 9.8), but the individual buildings were not aligned with one another.

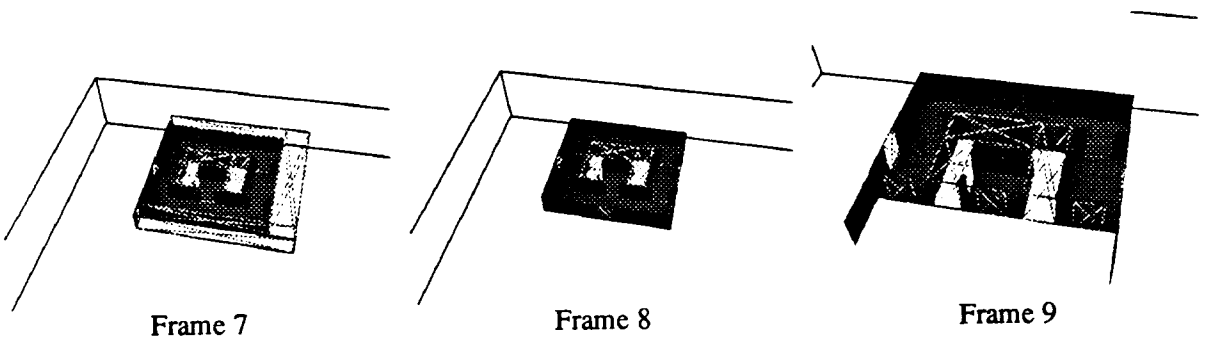


Frame 1 Frame 2 Frame 3

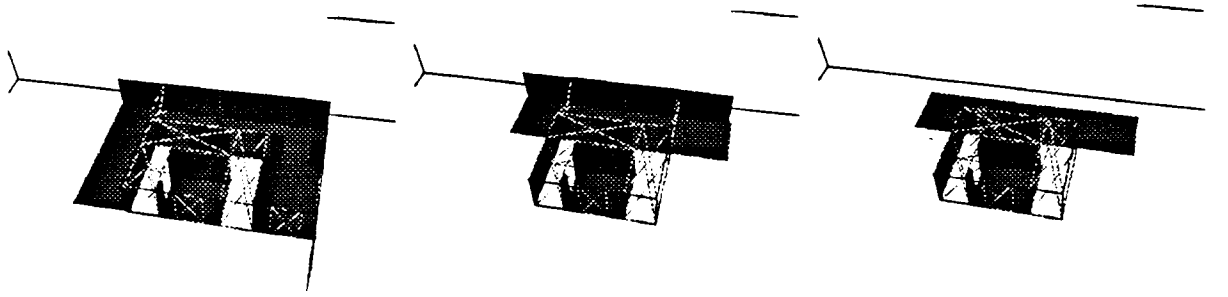
The minimum-cleaving subdivision criterion produced a fairly good spatial subdivision of the museum park. Major splitting planes between the rows of museums were found early, as was the ground-plane on which the museums rested (Frames 1 through 3 of Figure 9.9). After the rows of



Frame 4 Frame 5 Frame 7



Frame 7 Frame 8 Frame 9



Frame 10 Frame 11 Frame 12

Figure 9.9: Snapshots of the museum park spatial subdivision.

museums had been separated, the minimum-cleaving criterion separated each individual museum within rows (the delineation of one such museum is shown in Frames 4 through 8). Finally, the interior of each museum was subdivided as usual (Frames 9 through 12). The splitting process took less than 30 seconds. The final subdivision (Figure 9.10) has substantial free-space splitting, but very few superfluous partition planes inside the structures themselves.

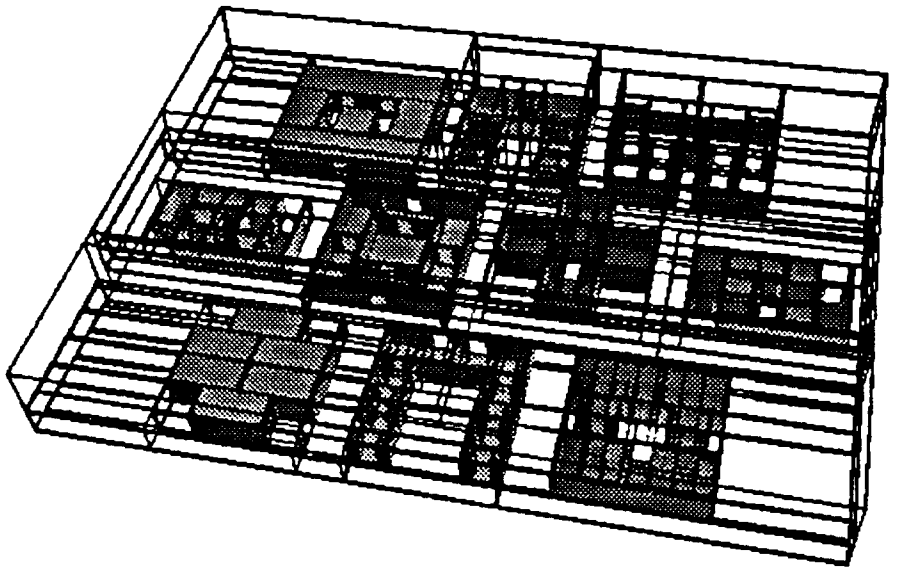


Figure 9.10: The final subdivision of museum park.

9.4.5 General Polyhedral Data

Although the implementations of the conforming subdivision for generally oriented occluders are still in somewhat embryonic stage, we compared subdivision construction and portal enumeration times for identical data. Unfortunately, we have no access to non-axial data sets of complexity comparable to the building model. Consequently, we timed the general BSP-tree based subdivision scheme on the partial geometric model, rotated about two axes to cause all occluders to become non-axial (although they remained rectangular).

Selecting candidate occluders and subdividing the k-D tree to termination required 15 minutes. Once the subdivision directives were stored, regenerating the k-D tree into 1,963 leaf cells, and

enumerating the 9,465 portals of the conforming spatial subdivision, required 284 and 475 seconds, respectively. Storing the occluders and conforming spatial subdivision required about 12 megabytes of memory. We did not compute cell-to-cell and cell-to-object visibility, as the implementation is still somewhat nascent. We can observe that the storage requirements and computation times grew at different rates. The storage required by the conforming subdivision was four times larger than that for the equivalent axial data set. On the other hand, the conforming subdivision construction times was fifty times longer than that for the equivalent axial data set. The considerable computation increase is partially due to substantial "sanity-checking" code that ensured the correctness of the incremental spatial subdivision after each split. Finally, we observe that both the number of cells and number of portals generated are actually slightly *smaller* for the rotated data set.

9.5 Summary and Reflections

One benefit of implementing the subdivision and visibility schemes described in this thesis was the body of practical intuition and engineering experience that resulted. In the following, we briefly describe the non-theoretical, but nonetheless extremely important, issues that arose during the implementation process.

9.5.1 ADTs, Invariants, and Witnesses

First, use of *ADTs* or abstract data types [AHU83], and data and program *invariants* such as Unix C-language "assertions" [Bel79], facilitates the connection between the logical conception and the implementation of an algorithm. An abstract data type encapsulates the definition of, and all operations defined upon, any data object. For example, an ADT for convex polygons might store an ordered list of vertices forming a simple convex planar contour, with an associated normal, plane equation, bounding region, centroid, etc., and provide the operations of creation, deletion, intersection, rendering, etc. This encapsulation hides the details of defining, manipulating, storing, and releasing the data object from conceptually higher-level algorithms that manipulate such data objects as primitive operands.

The C-language `assert()` construct ([Bel79], §3X) provides a very simple and powerful mechanism for gaining confidence in implemented algorithms. The functional statement `assert` (*predicate*) can be embedded anywhere in the program text, where the *predicate* is simply any boolean expression that can be computed at that point in the program. When the assertion

is encountered, the boolean expression is evaluated (this itself may cause function evaluation and other assertions to occur), and if it is found to be false, an error message is generated, an image of the memory state of the running program is stored as a "core" file, and program execution is terminated. For optimization, or when the algorithm is considered iron-clad, all assertions can be deactivated by a recompilation. Even in this deactivated state they serve as logical sidebars to the program's text. We have augmented the `assert()` statement to execute a run-time definable action before terminating execution, so that, for example, troublesome interactively-generated input can be saved for later study in isolation.

Another useful notion is that of a *data invariant* which can be *enforced* when an algorithm receives input. For example, a program that operates only on convex polygons might simply discard any non-convex polygons in its input (and generate a warning message), ensuring that any later computation will operate only on convex polygons.

Finally, a *witness* is a data object that evinces the outcome of a geometric query. This outcome can be a success or failure; for example, a witness to the intersection of a line and a plane is a point lying on both flats (if there is no intersection, the witness might be a polygon edge with the requisite sidedness with respect to the query line, as in Chapter 6). Witnesses can be thought of as short geometric proofs that particular geometric configurations do or do not have specific properties. During implementation, we often relied on witnesses during visual inspection of algorithmic behavior for assurance that our algorithms were functioning correctly. Indeed, the *absence* of a witness was often a powerful short proof of the misbehavior of an algorithm.

9.5.2 Input Filtering

We have found it useful to enforce data invariants on the geometric model for the walkthrough system, since its occluders and objects are derived from a real-world modeling system. Since the architectural model should represent a physically realizable space, it is intuitively clear that the model should have "inhabitable" portions (e.g., common areas, corridors) and "uninhabitable" portions (e.g., the insides of walls, crawl-spaces beneath floors). Moreover, both portions should consistently have non-zero thickness, so that, for example, a single occluder can not serve as a wall to two rooms on opposite sides of the occluder. Ignoring passable elements such as doors and windows, the inhabitable space of the building model should form a single connected component and have a manifold (although perhaps high-genus) boundary.

The manifold nature of the model can be checked by inspecting the *orientation* and *connectivity*

of occluders. Every occluder can be oriented according to the ordering of its vertices. The occluder's front face (i.e., that face on which the vertices appear counter-clockwise) should abut an inhabitable portion of the model, and its back face an uninhabitable portion. The occluder's connectivity is inspected through its shared edges with other occluders; these edges should be oppositely directed, and every edge should be shared by exactly two occluders.

Models satisfying these data invariants prove easier to process both theoretically and in an engineering sense. In particular, it should not be possible to move from an inhabitable region to an uninhabitable one (i.e., inside the wall) without traversing some occluder. The advantage of enforcing this invariant is that the spatial subdivision and adjacency graph construction then "find" the inhabitable portions of the model and store them as connected components. All subsequent visibility computations, defined as traversals of this graph, will then be guaranteed to operate only within inhabitable regions of the model. In other words, this invariant assures that no portals exist between inhabitable and uninhabitable model regions.

The "thickness" of the model has also proven extremely important to the efficiency of the visibility algorithms, in practice: it prevents an observer looking down a long corridor from seeing into far-away offices. Typically, the observer's lines of sight cannot propagate "around" the door frames into the offices (cf., for example, Figure 1.4 in the Introduction).

Polygon orientation proved useful in another sense. The fastest stabbing algorithm for an unordered sequence of general, unoriented polygons having e edges total, requires $O(e^4\alpha(e))$ time [MO88]. When the polygons (i.e., portals) of the sequence are *oriented* to admit stabbing lines in only one direction, the complexity of the stabbing algorithm can be reduced to $O(e^2)$ time (cf. §8.3.1).

9.5.3 Spatial Subdivision

In the presence of general occluders, subdividing three-dimensional space well is an ill-defined and difficult task. Since even the relatively well-posed problem of deciding whether a polyhedron is tetrahedralizable is NP-complete [RS89], it seems plausible that finding "optimal" general subdivisions may be as hard or harder. In the near term, therefore, the most one can practically hope for is "good" subdivisions that exhibit subquadratic size and near-logarithmic depth for practically occurring environments, and which satisfy our occlusion requirements acceptably well.

We have generated useable subdivisions for general occluders using BSP trees, although they

are both slower to construct and search, and more expensive to store, than k-D trees in practice. We have seen that BSP trees may cost as much as $O(n^3)$ time and $O(n^2)$ storage to construct in the worst case [PY90]. Things are somewhat easier when the occluders are axial. It has been shown, for example, that BSP trees over n axial 3D occluders can be constructed with size $O(n^{\frac{3}{2}})$ in optimal $O(n^{\frac{3}{2}})$ time [PY89]. We have not yet implemented this algorithm and so do not know if the subdivisions so generated have reasonable complexity constants, and behave well when subjected to combinatorial visibility algorithms such as those described here.

Another issue concerning subdivisions is that of symmetry between source and reached cells. Our framework uses the same set of cells both as visibility sources (i.e., bounds on a generalized observer) and as reached cells (i.e., bounds on potential visibility of the generalized observer). Perhaps the notions should be decoupled. Thus, a cell could be more highly constrained when acting as a source, but less constrained when traversed by the search algorithms, to reduce the amount of effort spent searching through portals. The concepts of free-space and metacells are essentially ad hoc modifications to the subdivision implementation (but not, substantively, to the abstraction) so that it may operate more asymmetrically in this sense.

The abstraction of a conforming spatial subdivision allows the *annotation* of the cell adjacency graph with *transition* visibility information. Since the cell-to-cell and cell-to-object visibility sets change qualitatively only when a portal is traversed, one might attach to each portal the changes in these visibility sets experienced by a generalized observer crossing the plane of the portal. Such difference lists, of course, would be antisymmetric and functions of the sense in which the portal was traversed (i.e., objects added while traversing the portal from cell A to cell B would be subtracted when traversing it from cell B to cell A). This seems like a worthwhile generalization of the conforming spatial subdivision data structure, but it has not yet been implemented.

The base issue is not one of adjacency graph annotation, since this can be viewed as a compacting postprocess to follow the visibility computations. More importantly, the issue of reusing the visibility information, while it is being computed, is itself a challenging theoretical issue (cf. §10.2). This remains an active area of investigation.

9.5.4 Scaling Effects

We set out to measure the effects of increasing model complexity on three metrics: the storage overhead incurred by precomputing visiblity information; the time required to precompute visibility information; and the cull time and speedup factor during on-line culling, for a fixed path.

We found that tripling the complexity of the geometric model increased the amount of computed visibility information by a factor of roughly two and two thirds. Since the spatial subdivisions were not identical for the two data sets, it is difficult to compare the storage factors with a high degree of confidence. However, we find it encouraging that the increase in visibility storage requirements was about the same as that for geometric data storage.

Visibility computation time scaled less well; tripling the model complexity caused a fourfold increase in visibility precomputation times. This increase had two causes. First, the relative amount of free space in the full model increased, causing greater numbers of expensive cell-to-cell and cell-to-object determinations from empty cells and through clusters of empty cells. Second, the working set and memory requirements of the full model visibility precomputation were greater for the (180 megabyte) full model, and in some cases taxed the (64 megabyte) physical memory capacity of our hardware platform. Excessive swapping activity may have substantially lengthened the observed "wall clock" precomputation times for certain cells with very high visibility.

Finally, the average on-line culling time worsened only slightly then the model tripled in size, from 75.8 to 86.6 milliseconds for an increase of about 14%. During most of the interior portions of the walkthrough path, the culling times were roughly equivalent for the two models. When in the free-space of the full model, however, the cull time increased substantially over the cull time in the same region of the partial model. For example, the longest measured cull time was 500 milliseconds in the partial model, and almost 670 milliseconds in the full model; a difference of 34%.

From these results we can garner two insights. First, we are optimistic that visibility storage needs will indeed scale linearly with geometric data size, for visual models of similar visual complexity. Second, it is clear from the increases in precomputation and on-line cull time that good spatial subdivisions and minimization of problems due to free-space cells are cells must be more thoroughly investigated.

Chapter 10

Discussion

10.1 Spectrum of Applicability

We have classified as "densely occluded" those geometric models for which interior visibility is very limited. Such models occur at one end of a spectrum, which at its other extreme includes "sparsely occluded" models, such as relatively open terrain. The visibility algorithms proposed here would not perform well for sparsely occluded models, because of the predominance of free-space cells (cf. §9.2.2). The cell-to-cell computations would explore (and find sightlines through) all portal sequences, and would find very few restrictions on cell-to-cell, cell-to-region, or cell-to-object visibility.

The spectrum of geometric models implies a corresponding spectrum of visibility, and shadow algorithms. Classical rendering algorithms make the basic assumption that the universe (i.e., geometric model) is fully illuminated, unless it can be established that light is *prevented* from reaching a particular region by an occluder. This is probably the right approach when space is sparsely occluded, for example when the model represents predominantly open terrain, or is a complex localized environment with little gross occlusion, such as a complicated ray-traced scene. Visibility preprocessing makes little sense for this class of model, since computing, storing, retrieving, and reusing coarse visibility information might be computationally harder than simply deriving fine-grain visibility (illumination) information as a function of the instantaneous observer (light source) position.

Light propagation algorithms, such as those presented in this thesis, make the opposite assumption: that *nothing* is visible (illuminated) unless a path can be found to it from the observer (light source). This turns out to be an effective assumption for certain common kinds of scenes, like

architectural models, and leads to dramatic reductions in rendering times. Computing this visibility information requires a few CPU hours on models with complexity corresponding to a design cycle of weeks or months. Storing the visibility information required less than ten percent of the space required for the geometric model data.

Spatial subdivision and hierarchical representation are promising techniques for management of complexity, a pressing problem for many engineering applications such as visual simulation systems. These techniques exploit the *coherence* inherent in spatial data. This coherence is spatial, since nearby entities can be stored near each other, and temporal, since nearby entities are typically referenced (e.g., rendered) near each other in time. Coherence can also be exploited in a host of other ways, for example, while editing or performing global illumination calculations on the model.

The ability to predict visibility-related data demands is a continuing part of an engineering effort to build a working walkthrough system that simulates geometric models several times the size of physical memory [FST92]. We conclude that the tradeoff between expending precomputation and storage, in order to gain predictive power and reduce on-line rendering time, is a worthwhile one for architectural models. Finally, we are optimistic that the techniques presented here will scale to larger environments, since visually complexity seems to grow less than linearly with overall model size. We observed average portal sequence lengths to be roughly constant for a one-story, three-story, and eight-story building model. We expect also that the on-line algorithms will straightforwardly generalize to multiple processors, since they are essentially "read-only" graph traversals.

10.2 Algorithmic Complexity

We have presented the complexity of the stabbing and visible-volume algorithms only in terms of the length of the portal sequence to which they are applied. The question remains as to the length of an average portal sequence, and how many such feasible portal sequences can be expected to emanate from a particular cell, in either two or three dimensions.

Since linear programs are solvable in linear time, *Find_Visible_Cells* adds or rejects each candidate visible cell in time linear in the length of the axial portal sequence reaching that cell. Determining a useful upper bound on the total number of such sequences as a function of the total number of cells visible to a given source cell seems challenging, as this quantity appears to depend on the spatial subdivision in a complicated way. However, for architectural models, we have observed the length of most portal sequences to be a small constant (about ten; cf. §9.4). That is, most cells see only a constant number of other cells (and this constant is small enough to have

practical implications). Were this not so, most of the model would be visible from most vantage points, and visibility preprocessing would be futile. The consequence of the small stabbing length is that visibility storage requirements should be only linear in the number of subdivision cells, since each cell will have only a constant amount of associated information.

The incremental stabbing and antipenumbra algorithms do not fully exploit the coherence and symmetry of the visibility relationship. Visibility is found one cell at a time, and the sightlines and visible regions so generated are meaningful only for the source cell. Later visibility computations on other cells do not "reuse" previously computed sightlines, but instead regenerate them from scratch. To see why reusing sightlines is not easily accomplished, consider a general cell with several portals. Many sightlines traverse this cell, each arriving with a different "history" or portal sequence. Upon encountering a cell, it may be more work for a sightline to check every prior-arriving sightline than it is for the new sightline to simply generate the (typically highly constrained) set of sightlines that can reach the cell's neighbors. It is difficult to see how the constraints due to a reaching portal sequence of arbitrary length may be manipulated efficiently.

For example, a common efficiency technique is to perform algorithmic operations *incrementally*, by considering the input data in turn, and maintaining a correct solution (in our case, a stabbing line and description of the cell-to-region visibility) as each input element (e.g., portal) is encountered. The advantage of incrementality is that we generally do not need to examine and construct the entire portal sequence each time a new portal is encountered.

On the other hand, when used as part of a DFS, incrementality has some important disadvantages. First, we must be able to "undo" the insertion of a portal as efficiently as we can add a portal, since we ascend (i.e., backtrack on) the call stack as often as we descend it. For stabbing lines, this is easy; we merely maintain a stack of (constant size) stabbing lines (constant size) linear constraints and "cut it back" whenever the DFS backs out of a cell. But cell-to-region volumes present a problem; their complexity is linear in the length of the active portal sequence, and their detailed structure (i.e., polyhedral face graph) may involve every edge of the active sequence. Thus, "backing out" of a general portal sequence involves general insertion and deletion operations on convex polyhedra, or (equivalently, but even less efficiently), maintaining a copy of the polyhedron generated for each cell, and discarding it upon ascension of the call stack.

The second disadvantage of incrementality is that the DFS nature of the constrained graph traversal search forces portals to be added in the order in which they are encountered; this destroys the random nature of the linear programs we use, and without randomness the expected time of [Sei90b], for example, increases to $O(n^2)$ from $O(n)$ time. In practice, we do use incrementality

and exploit the fact that, for axial portals, the cell-to-region polyhedra have constant complexity.

The other complexity issue, apart from incrementality, is re-use of portal sequences. Suppose some source cell DFS reaches a cell through a long portal sequence, and finds no further portals to explore. This portal sequence should, in principle, be useable by the neighbors of the source cell for their visibility computations and, when those sequences fail, by their neighbors, until all cells have had visibility computed. Analogously to the incrementality case, we are effectively faced with problem of removing portals from the *beginning*, rather than the end, of the portal sequence, and appropriately propagating all resulting changes in cell-to-region visibility through the resulting sequence. It is difficult to see how to do this robustly and efficiently. Consequently, we are still seeking satisfactory solutions to the portal incrementality and sequence-reuse problems.

Although the visibility computations are not optimal, their results may be compacted by annotating the adjacency graph with visibility information. We call this compacted representation *portal transition data*, since it exploits the fact that qualitative changes in cell-to-cell cell-to-object visibility can only occur at cell boundaries, i.e., along portals. Cell-to-cell and cell-to-object visibility can be encoded as a pair of lists attached to each portal. Each list describes the *changes* in the associated visibility set encountered by a generalized observer traversing the portal. This consists of a set of cells no longer visible from the attained cell (i.e., those cells to be subtracted), and a set of cells newly visible from the attained cell (i.e., those cells to be added).

10.2.1 Time and Storage Complexities

We summarize the complexity results of the subdivision and visibility algorithms described in the preceding text. Foremost, all of our complexity results are highly dependent on the quality of the spatial subdivision produced for the given input, and indeed the extent to which visibility is curtailed inside the environment model. Since our algorithms are conceptually independent of the type of spatial subdivision employed, we have chosen to express algorithm running times as functions of the length of the portal sequences over which the algorithms are run. We reason that the average length of a portal sequence is an inherent attribute of the geometric model, and that the relative goodness or badness of the spatial subdivision should not affect this average length by more than a constant factor. In practice, moreover, for most architectural environments to which these algorithms will be applied, the average stabbable portal sequence length is itself a small constant, perhaps ten or twenty.

There are four critical algorithms whose running times and storage complexities are of theo-

Occluder Class ⇒ Algorithm ⇓	General 2D	Axial 3D	General 3D
Spatial Subdivision	$O(f \lg f)$ time $O(f)$ space	$O(f^{\frac{3}{2}})$ time $O(f^{\frac{3}{2}})$ space in [PY89]	$O(f^3)$ time $O(f^2)$ space in [PY90]
Static Portal Stabbing	$O(n)$	$O(n \lg n)$ $O(n)$ in [Ame92, Meg91]	$O(e^2)$
Static Antipenumbra	$O(n)$	$O(n^2)$ polyhedral bounds in $O(n \lg n)$	$O(e^2)$
Static Object Incidence (per object)	$O(1)$	$O(1)$	$O(b)$ $O(e)$ linearized
On-line Portal Stabbing (per portal)	$O(1)$	$O(1)$	$O(e)$
On-line Object Incidence (per object)	$O(1)$	$O(1)$	$O(e)$

Table 10.1: Summary of algorithm complexities for operations described in this thesis, as functions of: f, the number of occluders; n, the length of an active portal sequence; e, the total number of edges in a 3D portal sequence; and b, which is $O(e^2)$, the worst-case complexity of the 3D antipenumbral boundary.

retical and practical interest here. First, a conforming spatial subdivision must be constructed. We express the construction times and sizes as functions of f, the number of occluders. Next, stabbing, antipenumbra, and on-line queries must be performed. We consider the latter algorithm running times as functions of the portal sequence length n, total edge complexity e, and antipenumbra boundary complexity b (in 3D). First, how efficiently can a *stabbing line* be found through a portal sequence? Second, how efficiently can the *antipenumbra* be cast through a portal sequence, and an object bounding box be examined for inclusion in the antipenumbra? Third, how efficiently can an eye-centered sightline be found through a portal sequence, and an object bounding box be examined for admission of an eye-centered sightline?

The complexity results are presented in the Table 10.1, organized by the three occluder input classes (2D, axial 3D, and general 3D), and by the algorithm of interest. The quantity f is the number of input occluders; $O(b)$ is the complexity of the antipenumbral boundary, which is at present known only to be upper-bounded by $O(e^2)$.

10.3 Frame-to-Frame Coherence

In practice, there is considerable *frame-to-frame* coherence to be exploited in the eye-to-cell visibility computation. During smooth observer motion, the observer's view point will typically spend several frame-times in each cell it encounters. The point-location query will often produce the same result as its previous invocation, and this result (i.e., an identifier for the previously found cell) can be cached. The stab tree for that cell can also be cached as long as the observer remains in the cell. Moreover, the cell adjacency graph allows substantial predictive power over the observer's motion. For instance, an observer exiting a known cell through a portal must emerge in a neighbor of that cell (our implementation disallows "walking through walls" by prohibiting incremental paths that penetrate occluders). A well-designed walkthrough system might prefetch all polygons visible to that cell *before* the observer's arrival, minimizing or eliminating the waiting times associated with typical high-latency mass-storage databases [FST92].

10.4 Scaling to Larger Models

In practice, we have observed that these subdivision and visibility methods scale well with increasing model size. Our test model appeared to have a fairly constant visual complexity, even when increased from a single floor to seven floors. The achieved rendering speedup therefore improved almost linearly as the size of the model increased. Indeed, one could imagine replicating the building model vertically and horizontally several times, without significantly increasing the number of occluders and detail objects visible to an interior observer. This is consistent with the intuitive notion that the internal visual complexity of many architectural environments "tops out" locally, rather than globally. That is, the visual complexity of most architectural models tends to be determined more by local effects (e.g., furnishings, realistic detail, room style) than by global effects (e.g., the total size of the model). For instance, from most points inside a typical apartment or office, an observer can not visually determine whether the apartment is a singleton or part of a block, or whether the office is a single floor or part of a huge skyscraper. Even if this fact could be determined visually (say, by looking out of a window), the remainder of the model would not be visible in its entirety.

Theoretical complexity measures can be reasonably good indicators of how well a given algorithm will scale. However, one must be wary of algorithmic constants, since they are hard to express exactly for non-trivial algorithms. Moreover, any implementation of an algorithm will

be, at some level, highly machine-dependent. In our case, the walkthrough system depends on such quantities as computation, memory, bus, and disk bandwidth, and on such hard-to-quantify graphics characteristics as transformation, clipping, and fill rates [DL90]. Since there is no "typical" rendering load, one must often gather empirical data (as we have done) as one measure of the utility of an implemented algorithm.

Scaling must also be regarded in the context of a practical difficulty, the problem of subdivision in "free space." Our algorithms are posited on the assumption that many occluders exist, and that they are good candidates for splitting space. However, current techniques for subdividing three-dimensional space have "non-local" effects. At early stages of splitting, some occluder must be chosen to split along. Choosing any such occluder may result in the introduction of a subdivision surface (i.e., a splitting plane) in a region of no occlusion, elsewhere in the model. This subdivision surface then later contributes to a large number of cell boundaries, none of which have any coaffine occluders. Consequently, the graph traversal algorithms expend computational resources to traverse these open portals. In §9.2.2, we described a modification to the data structures, precomputation, and query algorithms that overcomes local free-space cell clusters. However, global free-space problems are sure to arise when applying these techniques to, say, an office complex, in which many densely-occluded clusters populate an otherwise sparse space. In these instances, we propose an initial splitting round that attempts merely to separate clusters, rather than to find gross occlusion. This approach has been shown to improve the storage behavior of BSP trees in such environments [Tor90]. We found that the minimum-cleaving splitting criterion, originally formulated for building interiors (§5.2.1), efficiently found cluster separators in our "museum park" test dataset (§9.4.4).

10.5 Future Directions

Practical visibility determinations in three dimensions is by no means a closed subject. Fruitful avenues of investigation abound. Here, we briefly catalogue some areas which seem ripe for further attention.

10.5.1 Practical Spatial Subdivisions

Foremost is the question of efficient and effective spatial subdivision techniques in three dimensions. Managing complex spatial and general data is an important open problem. At present there exist no general, practical schemes for partitioning data in three dimensions in a local fashion

(much as the constrained Delaunay triangulation and Voronoi diagram partition point and segment data in the plane [Sei90a]). Indeed, one has some reason to be pessimistic on this front, since even the task of determining whether or not a polyhedron can be tetrahedralized has been shown to be NP-hard [RS89]. Nevertheless, algorithms that do spatially subdivide real environments continue to emerge (e.g., [CP89, DBS91, MW91, SP91]). Although convexity of cells is a powerful and useful invariant to rely upon, perhaps this may be shown to be unnecessary as more powerful subdivision abstractions emerge (e.g., [Rap91]). Eventually, these schemes will have to handle more general environments as well, including curved surfaces, procedurally generated objects and occluders, and time-dependent entities such as moving objects and multiple visual simulation participants. We are optimistic about the use of cell-based visibility algorithms in large and ever-more variegated architectural models.

10.5.2 High-Order Visibility Effects

In the specific realm of visibility, there are many other interesting open questions. For example, how might the visibility computation influence the subdivision algorithm in order to increase subdivision in the regions where visibility is changing most rapidly? Our scheme approximates "zeroth order" visibility effects by splitting on occluders, "first-order" effects by splitting along portal boundaries (i.e., at occluder edges), and partially recognizes "second-order" and "third-order" effects by identifying antipenumbral boundaries arising from two-way and three-way interactions among occluder (portal) edges. A more complete investigation of these multiple-order visibility effects, as well as effects due to the interaction among objects and occluders, is warranted. Visibility is an inherently *directional* phenomenon, and this too may be incorporated into the preprocessing phase, although angular variables are generally more difficult to manipulate than spatial variables.

The methods described here are particularly appropriate for densely occluded environments, such as many architectural models. However, if the model has many "holes", many distinct portals will be produced along cell boundaries, confounding the cell-to-cell and cell-to-object computations with a combinatorially explosive set of sightlines and halfspaces. This problem can be ameliorated by coalescing portals to allow at most one per cell boundary. Since portals must be convex, this generally results in an aggregate portal bigger than the union of its components, and larger (i.e., less useful) upper bounds on visibility. It would be useful to strike an optimal balance between these competing effects.

It may occur that subdivision on the scene's major structural elements alone does not suffi-

ciently limit cell-to-cell visibility. In this instance, further refinement of the spatial subdivision might help if it indeed reduces visibility, or hurt if it leaves visibility unchanged but increases the combinatorial complexity of finding sightlines. We may avoid this dilemma by exploiting the hierarchical *coherence* inherent in the spatial subdivision and its associated visibility information: after a leaf cell is subdivided, its children can see only a subset of the cells seen by their parent, since no new exterior portals are introduced (and the motion of the child generalized observers is reduced). Thus each child's sightline search is heavily constrained by its parent's portal/visibility list. Typically, the subdivision will further restrict eye-to-cell visibility during the walkthrough phase. Another future direction of this research is the potential of adaptive cell and object subdivision, based on the results of the cell-based (cell, region and object) visibility computations.

10.5.3 Occluders and Objects

We have made an arbitrary distinction between major occluders and detail objects, since conceptually they have different occlusive properties and because, in practice, they were easily distinguishable in our data. However, detail objects and occluders clearly form a continuum, and at some density of objects, sparsity of occluders, or general homogeneity of both, they become indistinguishable. Another worthy area of investigation is the automated classification of model entities as occluders or objects (and the concomitant investigation of visibility effects arising therefrom).

10.5.4 Mirroring and Translucency

Real environments include translucent and reflective surfaces, which themselves affect visibility in complicated ways. Both surface types can be incorporated fairly easily into the visibility framework we describe. Translucent surfaces can act as occluders or portals, depending on the incident angle of light encountering them, their condition (i.e., cleanliness), and the difference in light levels on opposite sides of the glass (the latter is a perceptual effect that may be exploited, for example, to preclude rendering dark surroundings as viewed from inside a bright house).

Mirrors, provided they are accompanied by explicit subdivision boundaries, can be treated as portals that impose a coordinate transform on all visibility searches encountering the mirror. This transform simply reflects the observer, and all active portals, about the plane of the mirror, and continues searching. Note that the portal "handedness" must also be reversed.

10.5.5 Visibility Algorithm Efficiency

One basic thrust of this thesis is the hierarchical chunking of data, and the establishment of visibility links between chunks at different levels of the hierarchy. Practically, the workhorse of this visibility search has been a stabbing-line or light-propagation algorithm that, for various abstractions of occluders and spatial subdivisions, finds straight-line paths or light bundles connecting chunks of the model. For real data, our implementation required six 20-MIP CPU-hours to determine complete cell-to-cell and cell-to-object visibility links for a realistic architectural model. More efficient algorithms, and faster implementations, would allow another dimension of interaction with the model (for example by altering an occluder, or opening or closing a door, and recomputing the affected visibility relationships). These effects can be simulated currently via a "conditional occluder" approach that embeds the conditional nature of a small number of occluders into the cell-based visibility sets. This approach, however, requires foreknowledge of all occluders and objects that may change position over time, and is not viable as a general technique.

10.5.6 Coherence and Parallelism

Both static and dynamic visibility computations are highly coherent. This coherence makes them attractive candidates for parallelization, since 1) visually isolated model regions can have no data interaction in the static phase, and can therefore be isolated in memory, and 2) the eye-based queries amount to read-only tree traversals in the on-line phase, and are straightforwardly parallelizable.

10.6 Other Applications

The subdivision-based visibility techniques described in this thesis can also be applied in other domains. Some of these are briefly discussed.

10.6.1 Global Illumination and Shadow Computations

Existing global illumination algorithms treat visibility as a difficulty to be overcome (typically) by sampling [CG85, HSA91]. However, these computations can be *driven* by visibility determinations in the sense that only visually interacting entities should be allowed to exchange energy directly. Thus, for example, the $O(n^2)$ "for each face, for every other face" construct in [HSA91] could be replaced by "for each face, for every other visible face"; which will typically be

sub-quadratic in occluded environments. Moreover, visually isolated subregions of the model can be solved with independent radiosity computations, to first order.

The antipenumbra computation algorithm of §8.3.2 can be slightly modified to produce all of the VE and EEE critical surfaces induced by a given portal sequence. These critical surfaces encode strong and weak visibility information about source and receiver polygons, and the first- and second-derivative illumination discontinuities that arise due to interactions between two or three occluders. Polygon meshing algorithms that wish to capture these discontinuities as data objects (see, e.g., [BRW89, CF90, Hec91, SLD92]) should find the antipenumbra characterization useful.

10.6.2 Geometric Queries

The subdivision and portal abstractions might be of use to computational geometers, in answering queries about the entities intersected by a given line segment or volume, or about collision-free paths between pairs of points. Similarly, ray-tracing queries typically use sampling to determine the fraction of an area light source visible to a point; analytic visibility-based methods could be used instead.

Chapter 11

Conclusions

We have shown that practical visibility preprocessing and on-line culling is achievable, both theoretically and as a functioning implementation, for an important class of densely-occluded environments. The visibility scheme first subdivides a geometric model into spatial cells, introducing cell boundaries wherever major occluders are present in the model. Next, a coarse visibility determination links cells, and objects within them, that are mutually visible through sightlines, or incident on conservative light bundles.

This coarse visibility determination constitutes a per-cell potentially visible set or PVS of objects as an upper bound on the set of objects visible to any actual observer in a given cell. Thus, a simulated actual observer can be tracked through the spatial cells, while the coarse visibility information stored with each cell is retrieved, and subjected to on-line culling operations. The observer's instantaneous view position and field of view are used to reduce the set of objects potentially visible from *anywhere* in the cell, to those potentially visible from the observer's eyepoint. The resulting subset, typically a small fraction of the model data, is then issued to graphics hardware for rendering and discretized hidden-surface removal.

These subdivision and visibility determination techniques have been implemented for general polyhedral geometric models in three dimensions. Using a furnished model of a planned computer science building as test data, we have achieved static culling rates of over 90% and dynamic culling rates of more than 99%, on average, decreasing the average refresh rates of a simulated walk through the model by a factor of about one hundred. The subdivision and visibility precomputation stages required several hours of compute-time for a five-story building, a reasonable figure given the design time cycle for a project of this size. The results of the visibility computation incurred less than 10% storage overhead (15 megabytes) with respect to the amount of storage required by

the geometric model itself (180 megabytes).

Together, these techniques demonstrate a successful application of computational geometric and computer graphics techniques to the engineering problem of visually simulating a very large geometric model. Due to the hierarchical and local nature of the partitioning and constrained graph traversal computations involved, these techniques should scale effectively to even more complex models. Finally, the visibility computations should have other applications in areas such as global illumination, ray-tracing, and shadow computations.

Bibliography

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[Air90] John M. Airey. *Increasing Update Rates in the Building Walkthrough System with Automatic Model-Space Subdivision and Potentially Visible Set Calculations*. PhD thesis, UNC Chapel Hill, 1990.

[Air92] John Airey. Personal communication, 1992.

[AK87] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics (Proc. SIGGRAPH '87)*, 21(4):55–63, 1987.

[Ake89] Kurt Akeley. The Silicon Graphics 4D/240GTX superworkstation. *IEEE Computer Graphics and Applications*, 9(4):71–83, 1989.

[Ame92] Nina Amenta. Finding a line traversal of axial objects in three dimensions. In *Proc.* 3^{rd} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 66–71, 1992.

[ARB90] John M. Airey, 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):41–50, 1990.

[Arv88] James Arvo. Ray tracing with meta-hierarchies. *SIGGRAPH '88 Course Notes (Introduction to Ray Tracing)*, 1988.

[Aut90] AutoDesk, Inc. Autocad reference manual release 10, 1990.

[AW87] D. Avis and R. Wenger. Algorithms for line traversals in space. In *Proc.* 3^{rd} *Annual Symposium on Computational Geometry*, pages 300–307, 1987.

[Bau72] B. Baumgardt. Winged-edge polyhedron representation. Technical Report No. CS-320, Stanford Artificial Intelligence Report, Computer Science Department, 1972.

[Bel79] Bell Telephone Laboratories, Incorporated. *UNIX User's Manual*, 1979.

[Bel92] Gavin Bell. Personal communication, August 1992.

[Ben75] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[BRW89] Daniel R. Baum, Holly E. Rushmeier, and James M. Winget. Improving radiosity solutions through the use of analytically determined form factors. *Computer Graphics (Proc. SIGGRAPH '89)*, 23(3):325–334, 1989.

[CF90] A.T. Campbell III and Donald S. Fussell. Adaptive mesh generation for global diffuse illumination. *Computer Graphics (Proc. SIGGRAPH '90)*, 24(4):155–164, 1990.

[CF92] Norman Chin and Steven Feiner. Fast object-precision shadow generation for area light sources using BSP trees. In *Proc. 1992 Symposium on Interactive 3D Graphics*, pages 21–30, 1992.

[CG85] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: A radiosity solution for complex environments. *Computer Graphics (Proc. SIGGRAPH '85)*, 19(3):31–40, 1985.

[CHG+88] K. Clarkson, H.Edelsbrunner, L. Guibas, M. Sharir, and E. Welzl. Combinatorial complexity bounds for arrangements of curves and surfaces. *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 568–579, 1988.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *CACM*, 19(10):547–554, 1976.

[Com90] Computer Science 270: Combinatorial Optimization. Class notes. NP-completeness of minimal parallelepipedization, 1990.

[CP89] Bernard Chazelle and Leonidas Palios. Triangulating a nonconvex polytope. In *Proc. 5th Annual ACM Symposium on Computational Geometry*, pages 393–400, 1989.

[CS86] R. Cole and M. Sharir. Visibility problems for polyhedral terrains. Technical Report 92, New York University Courant Institute of Mathematical Sciences, Computer Science Division, 1986.

[CS89] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry II. *Discrete Computational Geometry*, pages 387–421, 1989.

[DBS91] Tamal Dey, Chanderjit Bajaj, and Kokichi Sugihara. On good triangulations in three dimensions. In *Proceedings of 1991 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 431–441, 1991.

[DFP+86] Leila DeFloriani, Bianca Falcidieno, C. Pienovi, D. Allen, and George Nagy. A visibility-based model for terrain features. In *Proc. Int. Symp. on Spatial Data Handling*, July 1986.

[DG87] J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *CACM*, 30(5):403–407, 1987.

[DL90] J. Craig Dunwoody and Mark A. Linton. Tracing interactive 3D graphics programs. *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24(2):155–163, 1990.

[DS89] David P. Dobkin and Diane L. Souvaine. Detecting the intersection of convex objects in the plane. Technical Report No. 89-9, DIMACS, 1989.

[dV91] René de Vogelaere. Personal communication, November 1991.

[Ede85] H. Edelsbrunner. Finding transversals for sets of simple geometric figures. *Theoretical Computer Science*, 35:55–69, 1985.

[EGS86] Herbert Edelsbrunner, Leonidas J. Guibas, and Jorge Stolfi. Optimal point location in monotone subdivisions. *SIAM Journal of Computing*, 15:317–340, 1986.

[FI85] Akira Fujimoto and Kansei Iwata. Accelerated ray tracing. *Computer Graphics: Visual Technology and Art (Proc. Computer Graphics Tokyo '85)*, pages 41–65, 1985.

[FK91] Sharon Fischler and George Kong. *Graphics Tuning, Section 20 of the Power Series Performance Guide, Ed. Veeleen Roufchaie.* Silicon Graphics Computer Systems, 2011 N. Shoreline Boulevard, Mountain View, CA 94039-7311, February 1991.

[FKN80] H. Fuchs, Z. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Computer Graphics (Proc. SIGGRAPH '80)*, 14(3):124–133, 1980.

[FST92] Thomas A. Funkhouser, Carlo H. Séquin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *Proc. 1992 Workshop on Interactive 3D Graphics*, pages 11–20, 1992.

[FvD82] J.D. Foley and A. van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, 1982.

[GBW90] Benjamin Garlick, Daniel R. Baum, and James M. Winget. Interactive viewing of large geometric databases using multiprocessor graphics workstations. *SIGGRAPH '90 Course Notes (Parallel Algorithms and Architectures for 3D Image Generation)*, 1990.

[GCS91] Ziv Gigus, John Canny, and Raimund Seidel. Efficiently computing and representing aspect graphs of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):542–551, 1991.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman, 1979.

[Gla84] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[GM90] Ziv Gigus and Jitendra Malik. Computing the aspect graph for line drawings of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(2):113–122, 1990.

[Grü67] Branko Grünbaum. *Convex Polytopes*. Wiley-Interscience, New York, 1967.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[GV89] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[Han84] Patrick Hanrahan. A homogeneous geometry calculator. Technical Report 3-D No. 7, Computer Graphics Laboratory, New York Institute of Technology, 1984.

[Hec91] Paul S. Heckbert. *Simulating Global Illumination Using Adaptive Meshing*. PhD thesis, Computer Sciences Department, University of California, Berkeley, June 1991.

[Her87] John E. Hershberger. *Efficient Algorithms for Shortest Path and Visibility Problems*. PhD thesis, Stanford University, 1987.

[HSA91] Patrick Hanrahan, David Salzman, and Larry Auperle. A rapid hierarchical radiosity algorithm. *Computer Graphics (Proc. SIGGRAPH '91)*, 25(4):197–206, 1991.

[HT91] Michael E. Hohmeyer 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. Also to appear in *Computational Geometry: Theory and Applications*, 1992.

[Jon71] C.B. Jones. A new approach to the 'hidden line' problem. *The Computer Journal*, 14(3):232–237, 1971.

[Jon92] Michael T. Jones. A high performance visual simulation toolkit. Technical report, Silicon Graphics Computer Systems, Mountain View CA, 94043, March 1992.

[Kho91] Delnaz Khorramabadi. A walk through the planned CS building. Technical Report UCB/CSD 91/652, Computer Science Department, U.C. Berkeley, 1991.

[Kir83] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal of Computing*, 12:28–35, 1983.

[KLZ85] M. Katchalski, T. Lewis, and J. Zaks. Geometric permutations for convex sets. *Discrete Mathematics*, 54:271–284, 1985.

[Kv79] J.J. Koenderink and A.J. van Doorn. The internal representation of solid shape with respect to vision. *Biol. Cybern.*, 32:211–216, 1979.

[KV90] David Kirk and Douglas Voorhies. The rendering architecture of the DN10000VS. *Computer Graphics (Proc. SIGGRAPH '90)*, 24(4):299–307, 1990.

[Meg83] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal Computing*, 12:759–776, 1983.

[Meg91] N. Megiddo. Stabbing isothetic boxes in deterministic linear time. *Personal communication to Nina Amenta*, 1991.

[Mit92] Don Mitchell. On the quadratic behavior of BSP trees for real-world data. *Personal Communication*, 1992.

[MO88] M. McKenna and J. O'Rourke. Arrangements of lines in 3-space: A data structure with applications. In *Proc.* 4^{th} *Annual Symposium on Computational Geometry*, pages 371–380, 1988.

[MW91] Doug Moore and Joe Warren. Bounded aspect ratio triangulation of smooth solids. In *Proceedings of 1991 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 455–464, 1991.

[NN83] Tomoyuki Nishita and Eihachiro Nakamae. Half-tone representation of 3-D objects illuminated by area sources or polyhedron sources. In *Proc. IEEE COMPSAC, 1983*, pages 237–242, 1983.

[NN85] Tomoyuki Nishita and Eihachiro Nakamae. Continuous-tone representation of three-dimensional objects taking account of shadows and interreflection. *Computer Graphics (Proc. SIGGRAPH '85)*, 19(3):23–30, 1985.

[O'R87] Joseph O'Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, 1987.

[PD90] W.H Plantinga and C.R. Dyer. Visibility, occlusion, and the aspect graph. *Int. J. Computer Vision*, 5(2):137–160, 1990.

[Pel90a] Marco Pellegrini. Stabbing and ray-shooting in 3-dimensional space. Technical Report 540; Robotics Report No. 230, New York University Courant Institute of Mathematical Sciences, Computer Science Division, 1990.

[Pel90b] Marco Pellegrini. Stabbing and ray-shooting in 3-dimensional space. In *Proc.* 6^{th} *ACM Symposium on Computational Geometry*, pages 177–186, 1990.

[PW88] Ken Perlin and Xue-Dong Wang. An efficient approximation for penumbra shadow. Technical Report 346, New York University Courant Institute of Mathematical Sciences, Computer Science Division, 1988.

[PW89] R. Pollack and R. Wenger. Necessary and sufficient conditions for hyperplane traversals. In *Proc.* 5^{th} *Annual Symposium on Computational Geometry*, pages 152–155, 1989.

[PY89] M.S. Paterson and F.F. Yao. Optimal binary space partitions for orthogonal objects. In *Proc.* 1^{st} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 100–106, 1989.

[PY90] Michael S. Paterson and F. Frances Yao. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete and Computational Geometry*, 5(5):485–503, 1990.

[Rap91] Ari Rappoport. The n-dimensional extended convex differences tree (ECDT) for representing polyhedra. In *Proceedings of 1991 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 139–147, 1991.

[RS89] Jim Ruppert and Raimund Seidel. On the difficulty of tetrahedralizing 3-dimensional non-convex polyhedra. In *Proc.* 5^{th} *Annual ACM Symposium on Computational Geometry*, pages 380–392, 1989.

[SBGS69] R. A. Schumacker, B. Brand, M Gilliland, and W. Sharp. Study for applying computer-generated images to visual simulation. Technical Report AFHRL TR-69-14, U.S. Air Force Human Resources Laboratory, 1969.

[Sei90a] Raimund Seidel. Constrained Delaunay triangulations and Voronoi diagrams with obstacles. *Extended Abstract*, 1990.

[Sei90b] Raimund Seidel. Linear programming and convex hulls made easy. In *Proc.* 6^{th} *ACM Symposium on Computational Geometry*, pages 211–215, 1990.

[Sha87] Micha Sharir. The shortest watchtower and related problems for polyhedral terrains. Technical Report 334, New York University Courant Institute of Mathematical Sciences, Computer Science Division, 1987.

[SLD92] David Salesin, Dani Lischinski, and Tony DeRose. Reconstructing illumination functions with selected discontinuities. In *Proc.* 3^{rd} *Eurographics Workshop on Rendering*, 1992.

[Som59] D.M.Y. Sommerville. *Analytical Geometry of Three Dimensions*. Cambridge University Press, 1959.

[SP85] Michael Ian Shamos and Franco P. Preparata. *Computational Geometry: an Introduction*. Springer-Verlag, 1985.

[SP91] Nickolas Sapidis and Renato Perruchio. Domain Delaunay tetrahedrization of arbitrarily shaped curved polyhedra defined in a solid modeling system. In *Proceedings of 1991 ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 465–480, 1991.

[SS90] Sack and Suri. An optimal algorithm for detecting weak visibility of a polygon. *IEEE Transactions on Computers*, 0(0):0–0, 1990.

[SS91] Carlo H. Séquin and Kevin P. Smith. Introduction to the Berkeley UNIGRAFIX tools (Version 3.0). Technical Report UCB/CSD 91/606, Computer Science Department, U.C. Berkeley, 1991.

[SSS74] Ivan E. Sutherland, Robert F. Sproull, and Robert A. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974.

[SSW83] Carlo H. Séquin, Mark Segal, and Paul R. Wensley. UNIGRAFIX 2.0 user manual and tutorial. Technical Report UCB/CSD 83/161, Computer Science Department, U.C. Berkeley, 1983.

[ST86] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *CACM*, 29(7):669–679, 1986.

[Sto89] Jorge Stolfi. Primitives for computational geometry. Technical Report 36, DEC SRC, 1989.

[TH92] Seth J. Teller and Michael E. Hohmeyer. Stabbing oriented convex polygons in randomized $O(n^2)$ time. Technical Report UCB/CSD 91/669, Computer Science Department, U.C. Berkeley, 1992.

[Tor90] Enric Torres. Optimization of the binary space partition algorithm (bsp) for the visualization of dynamic scenes. In *Proc. 1st Eurographics Workshop on Rendering*, pages 507–518, 1990.

[WA77] Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. *Computer Graphics (Proc. SIGGRAPH '77)*, 11(2):214–222, 1977.

[WMB92] Allan Wilks, Allen McIntosh, and Richard A. Becker. The data dual. *In preparation*, 1992.