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:

Chair   Date

Date

Date

University of California at Berkeley

1992

Visibility Computations
in Densely Occluded Polyhedral Environments

Copyright ©1992
by
Seth Jared Teller

Visibility Computations in Densely Occluded Polyhedral Environments

by

Seth Jared Teller

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

1 Introduction 1
- 1.1 Motivation 1
- 1.2 The Basic Approach 2
  - 1.2.1 Spatial Subdivision 2
  - 1.2.2 Visibility Precomputation 3
  - 1.2.3 Dynamic Phase 5
- 1.3 Theoretical Concerns 6
- 1.4 Engineering Observations and Assumptions 6
- 1.5 Practical Issues 7
- 1.6 Organization 8

2 Previous Work 11
- 2.1 Point Visibility Queries – Visibility as Sorting 11
- 2.2 Region Visibility Queries – Visibility as Light Propagation 13

3 Computational Framework 19
- 3.1 Occluders and Detail Objects 19
- 3.2 Spatial Subdivision 20
  - 3.2.1 Point Location 20
  - 3.2.2 Object Population 21
  - 3.2.3 Neighbor Finding 21
  - 3.2.4 Portal Enumeration 21
  - 3.2.5 Cell Adjacency Graph 22
- 3.3 Static Visibility 22
  - 3.3.1 Cell-to-Cell (Coarse) Visibility 23
  - 3.3.2 Generating Portal Sequences 23
  - 3.3.3 Stab Trees 25
  - 3.3.4 Cell-to-Region (Fine) Visibility 26
  - 3.3.5 Cell-to-Object (Fine) Visibility 27

3.4 Dynamic Visibility: On-Line Culling . . . 30
3.4.1 Observer View Variables . . . 31
3.4.2 Eye-to-Cell Visibility . . . 31
3.4.3 Eye-to-Region Visibility . . . 32
3.4.4 Eye-to-Object Visibility . . . 33

4 Two Dimensional Environments **35**
4.1 Major Occluders and Detail Objects . . . 35
4.2 Spatial Subdivision . . . 35
4.2.1 Point Location . . . 37
4.2.2 Object Population . . . 37
4.2.3 Neighbor Finding . . . 39
4.2.4 Portal Enumeration . . . 39
4.3 Static Visibility Operations . . . 40
4.3.1 Cell-to-Cell Visibility . . . 40
4.3.2 Cell-to-Region Visibility . . . 41
4.3.3 Cell-to-Object Visibility . . . 43
4.4 Dynamic Visibility Queries . . . 44
4.4.1 Observer View Variables . . . 44
4.4.2 Eye-to-Cell Visibility . . . 44
4.4.3 Eye-to-Region Visibility . . . 46
4.4.4 Eye-to-Object Visibility . . . 47

5 Three-Dimensional Axial Environments **49**
5.1 Major Occluders and Detail Objects . . . 49
5.2 Spatial Subdivision . . . 49
5.2.1 Splitting Criteria . . . 50
5.2.2 Point Location . . . 51
5.2.3 Cell Population . . . 52
5.2.4 Neighbor Finding . . . 52
5.2.5 Portal Enumeration . . . 52
5.3 Static Visibility Operations . . . 54
5.3.1 Cell-to-Cell Visibility . . . 54
5.3.2 Cell-to-Region Visibility . . . 58
5.3.3 Cell-to-Object Visibility . . . 61
5.4 Dynamic Visibility Queries . . . 61
5.4.1 Observer View Variables . . . 61
5.4.2 Eye-to-Cell Visibility . . . 62
5.4.3 Eye-to-Region Visibility . . . 63
5.4.4 Eye-to-Object Visibility . . . 63

6 Line Coordinates **65**
6.1 Plücker Coordinates 65
6.2 Degrees of Freedom in Line Space 67
6.3 Computing the Incident Lines 69
6.4 Application 72

7 Stabbing 3D Portal Sequences **73**
7.1 Stabbing General Portal Sequences 73
7.2 Penumbrae and Antipenumbrae 75
7.2.1 Event Surfaces and Extremal Swaths 77
7.2.2 Boundary and Internal Swaths 78
7.2.3 Two-Dimensional Example 79
7.2.4 Edge-Edge-Edge Swaths 81
7.2.5 Vertex-Edge Swaths 83
7.2.6 The Containment Function 84
7.2.7 Computing the Antipenumbra 86
7.3 Implementation Issues 87
7.3.1 Current Implementation and Test Cases 87
7.3.2 Parametric Swath Representations 90
7.3.3 Implicit Swath Representations 91
7.4 Applications 92
7.4.1 Weak Visibility in Three Dimensions 92
7.4.2 Aspect Graphs 93

8 Three-Dimensional Polyhedral Environments **95**
8.1 Major Occluders and Detail Objects 95
8.2 Spatial Subdivision 95
8.2.1 Splitting Criteria 96
8.2.2 Point Location 96
8.2.3 Object Population 96
8.2.4 Neighbor Finding 97
8.2.5 Portal Enumeration 97
8.3 Static Visibility Operations 98
8.3.1 Cell-to-Cell Visibility 98
8.3.2 Cell-to-Region Visibility 99
8.3.3 Representing the Cell-to-Region Visibility 99
8.3.4 Cell-to-Object Visibility 99
8.3.5 Conservatively Approximating the Antipenumbra 100
8.4 Dynamic Visibility Queries 102
8.4.1 Observer View Variables 102
8.4.2 Eye-to-Cell Visibility 102
8.4.3 Eye-to-Region Visibility 106

8.4.4 Eye-to-Object Visibility . . . 106

9 Results: Soda Hall Data 107
9.1 Geometric Data Set . . . 107
9.1.1 Test Models and Test Walkthrough Path . . . 108
9.2 Implementation . . . 108
9.2.1 Programming Methodology . . . 108
9.2.2 Free-Space . . . 110
9.3 Utility Metrics . . . 112
9.4 Experimental Results . . . 117
9.4.1 Test Walkthrough Path . . . 117
9.4.2 Storage Overhead and Precomputation Times . . . 118
9.4.3 Tabulated Utility Metrics . . . 119
9.4.4 Museum Park . . . 126
9.4.5 General Polyhedral Data . . . 129
9.5 Summary and Reflections . . . 129
9.5.1 ADTs, Invariants, and Witnesses . . . 129
9.5.2 Input Filtering . . . 130
9.5.3 Spatial Subdivision . . . 131
9.5.4 Scaling Effects . . . 132

10 Discussion 133
10.1 Spectrum of Applicability . . . 133
10.2 Algorithmic Complexity . . . 134
10.2.1 Time and Storage Complexities . . . 135
10.3 Frame-to-Frame Coherence . . . 136
10.4 Scaling to Larger Models . . . 136
10.5 Future Directions . . . 138
10.5.1 Practical Spatial Subdivisions . . . 138
10.5.2 High-Order Visibility Effects . . . 138
10.5.3 Occluders and Objects . . . 139
10.5.4 Mirroring and Translucency . . . 139
10.5.5 Visibility Algorithm Efficiency . . . 140
10.5.6 Coherence and Parallelism . . . 140
10.6 Other Applications . . . 140
10.6.1 Global Illumination and Shadow Computations . . . 140
10.6.2 Geometric Queries . . . 141

11 Conclusions 143

List of Figures

1.1 Major occluders (bold) and detail object bounding boxes (squares). 3
1.2 A spatial subdivision. Portals are shown as dashed lines. 3
1.3 The cell-to-cell visibility set of the gray source cell. 4
1.4 The cell-to-object visibility set (filled squares) of the gray source cell. 4
1.5 The eye-to-cell visibility (darkened cells) of the actual observer. 5
1.6 The eye-to-object visibility set (filled squares) of the actual observer. 5

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). 12
2.2 In two dimensions, a pair of occluders can jointly hide points from the light source that neither occluder hides alone. 13
2.3 In three dimensions, three mutually skew occluder edges can generate a quadric shadow boundary. 14
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. 15
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. 16
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. 17
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. 18

3.1 Convex, opaque occluders, in two and three dimensions. 20
3.2 Spatial subdivisions, in two and three dimensions. 21
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. 22
3.4 Some portal sequences that admit sightlines. 24
3.5 Finding sightlines from I. 24

3.6 The stab tree rooted at I. . . . 25
3.7 Distant cells are, in general, only partially visible (gray areas) from the source cell (dark). 26
3.8 Antiumbra and antipenumbra through a series of 2D portals. . . . 27
3.9 Successively narrowing antipenumbrae cast by an area light source in 3D (here, the left-most portal) through the cells of a conforming spatial subdivision. . . . 28
3.10 Cell-to-object visibility (filled squares) for a given source in 2D. . . . 28
3.11 An object or occluder can be backfacing with respect to a *generalized* observer. . . . 29
3.12 A source cell may reach another through several paths. . . . 29
3.13 Observer view variables as view cone (2D) and view frustum (3D). . . . 31
3.14 Eye-to-cell and eye-to-region visibility sets for an actual observer. . . . 32
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. . . . 33

4.1 A linear-size constrained triangulation of the n occluders. Occluders are shown as bold segments, portals as dashed lines. . . . 36
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. . . . 37
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. . . . 38
4.4 Occluders intersecting the cell boundary in a point may be ignored during portal enumeration. . . . 39
4.5 Oriented portal sequences, and separable sets $\mathbf{L}$ and $\mathbf{R}$. . . . 40
4.6 Distant cells are, in general, only partially visible from the source. . . . 41
4.7 In two dimensions, a portal sequence admits an "hourglass" of stabbing lines. . . . 42
4.8 The left-hand points form a convex chain. . . . 43
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). . . . 43
4.10 Observer view variables in 2D. . . . 44
4.11 Culling O's stab tree against a view cone C. . . . 45
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. . . . 46
4.13 The view cone during the stab tree DFS. . . . 47
4.14 The 2D eye-to-object visibility computation. . . . 47

5.1 Axial splitting planes are chosen to coincide with portal edges. . . . 51
5.2 Portal enumeration as a set difference of sets of rectangles. . . . 53
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). . . . 53
5.4 Lines in $\mathbf{R}^2$ and their dual representation. . . . 54
5.5 An axial portal sequence and stabbing. . . . 55

5.6 Reducing three-dimensional stabbing to a series of two-dimensional problems. 56
5.7 The structure of the region of feasible lines is independent of P. 57
5.8 An axial portal sequence can be decomposed into three sets of 2D hourglass constraints. 59
5.9 The linearized visibility volumes emanating from an axial source cell. 60
5.10 Three-dimensional observer view variables. 62
5.11 Decomposing axial portals into constituent axial eye-centered constraints. 62
5.12 The four to six faces of an eye-centered bounding box pyramid. 64

6.1 The right-hand rule applied to $side(a, b)$. 66
6.2 Directed lines map to points on, or hyperplanes tangent to, the Plücker surface. 67
6.3 Generically, no real line is incident to five given lines. 68
6.4 Generically, two real lines are incident to four given lines. 68
6.5 Generically, a one-parameter family of lines is incident to three given lines. 69
6.6 The four Π_k determine a line to be intersected with the Plücker quadric. 69
6.7 The two lines incident through four generic lines in 3D. 70

7.1 The stabbing line s must pass to the same side of each e_k. 74
7.2 The 5D point $S = \Pi(s)$ must be on or above each hyperplane h_k. 74
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. 76
7.4 Umbra and penumbra of an occluder (bold) in 2D. 76
7.5 Antiumbra and antipenumbra admitted by a 3D portal sequence. 77
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). 78
7.7 Traces (intersections) of extremal lines and swaths on the Plücker surface in 5D (higher-dimensional faces are not shown). 79
7.8 Extremal swaths arising from a two-dimensional portal sequence. 80
7.9 An internal swath in a two-dimensional portal sequence. 80
7.10 A boundary swath in a two-dimensional portal sequence. 81
7.11 An internal EEE swath (i), viewed along $\mathbf{L}$ (ii) and transverse to $\mathbf{L}$ (iii). The $\mathbf{N}_i$ can be contained, and the swath is therefore internal. 82
7.12 A boundary EEE swath (i), viewed along $\mathbf{L}$ (ii) and transverse to $\mathbf{L}$ (iii). The $\mathbf{N}_i$ cannot be contained, and the swath is therefore a boundary swath. 82
7.13 Boundary (i) and internal (ii) VE swaths. 83
7.14 Directions of containment and non-containment, with moving $\mathbf{N}_c$, for fixed $\mathbf{N}_a$ and $\mathbf{N}_b$. Transition directions are marked. 84
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.) . . . 85
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. 85

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. 87
7.18 The relationship between extremal stabbing lines and boundary swaths. 88
7.19 An observer's view of the light source (i). Crossing an extremal VE (ii), EEE (iii), or degenerate (iv) swath. 88
7.20 An area light source and three portals can yield a disconnected antipenumbra. 89
7.21 A disconnected antipenumbra boundary in 3D is isomorphic to a collection of loops in 5D line space. 90
7.22 Parametrizing VE swaths (i) and EEE swaths (ii). 90
7.23 Three generator lines and part of the induced regulus of incident lines. 91
7.24 The aspect of the light source, as seen through the portal sequence. 93

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. 98
8.2 A convex polyhedron and a regulus can intersect in only three ways. 100
8.3 The plane through edge **e**, separating portals **P** and **L**. 101
8.4 A giftwrap step on edge **e**, from $\mathbf{v}_1$ to $\mathbf{v}_L$. 102
8.5 Three-dimensional observer view variables. 103
8.6 Encountering a portal in the eye-to-cell DFS. 103
8.7 Detecting an impassable portal edge constraint. 104
8.8 Detecting a superfluous portal edge constraint. 104
8.9 Handling degeneracies in the eye-to-cell DFS. 105

9.1 The top two floors and roof of the geometric model. The test path (grey) snakes in and out of the building. 109
9.2 The test path, grey-scaled from low z-values (black) to high z-values (white). 109
9.3 Naive search through free-space causes a combinatorial explosion. Dashed lines are portals; some of the sightlines computed are shown. 110
9.4 A metacell (bold dashed outline), with portal crossover constraints attached to each entry portal. 112
9.5 Encountering a metacell during a constrained DFS. 113
9.6 Detail objects can prohibit inter-portal visibility (constituent cells not shown). 113
9.7 The instrumented culling modes (NC mode not shown). 116
9.8 The ten museums in museum park. 126
9.9 Snapshots of the museum park spatial subdivision. 128
9.10 The final subdivision of museum park. 128

List of Tables

9.1 Object and pixel metrics for the partial model. 119
9.2 Object and pixel metrics for the full model. 120
9.3 Culling and drawing time metrics for the partial model. 120
9.4 Culling and drawing time metrics for the full model. 121
9.5 Average, minimum and maximum object metrics for the partial model. 122
9.6 Average, minimum and maximum object metrics for the full model. 122
9.7 Average, minimum and maximum culling times for the partial model. 123
9.8 Average, minimum and maximum culling times for the full model. 123
9.9 Average, minimum and maximum drawing times for the partial model. 124
9.10 Average, minimum and maximum drawing times for the full model. 124

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. 137

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 [HT92]. 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 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.

[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.

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.

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).

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

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

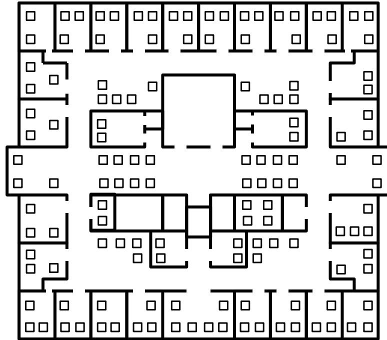

Figure 1.1: Major occluders (bold) and detail object bounding boxes (squares).

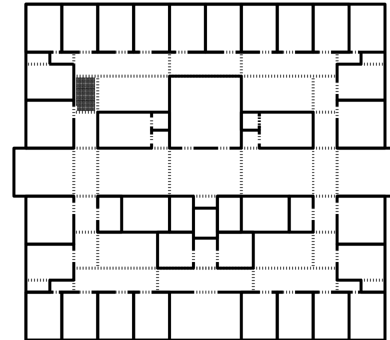

Figure 1.2: A spatial subdivision. Portals are shown as dashed lines.

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.

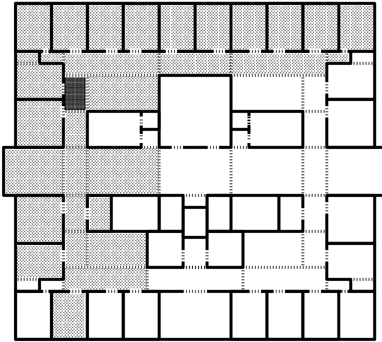

Figure 1.3: The cell-to-cell visibility set of the gray source cell.

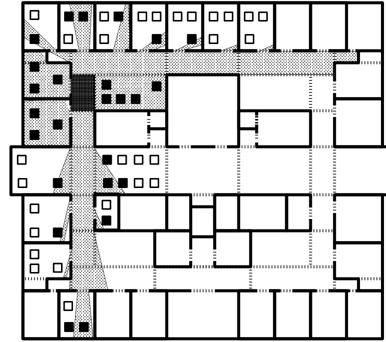

Figure 1.4: The cell-to-object visibility set (filled squares) of the gray source cell.

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 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.

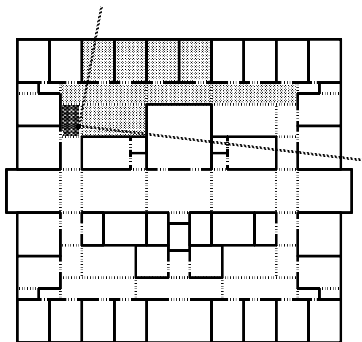

Figure 1.5: The eye-to-cell visibility (darkened cells) of the actual observer.

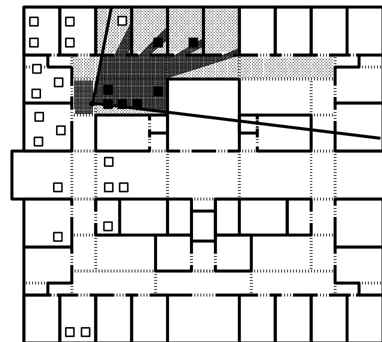

Figure 1.6: The eye-to-object visibility set (filled squares) of the actual observer.

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

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

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.

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).

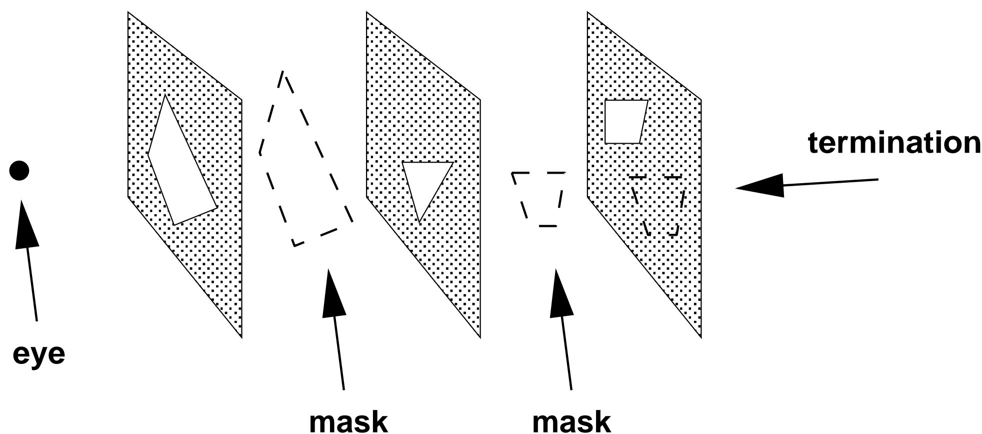

Figure 2.1: Jones' hidden-surface method. The depth-first search of the cell adjacency graph terminates when the current mask has no intersection with the next portal (at right).

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.

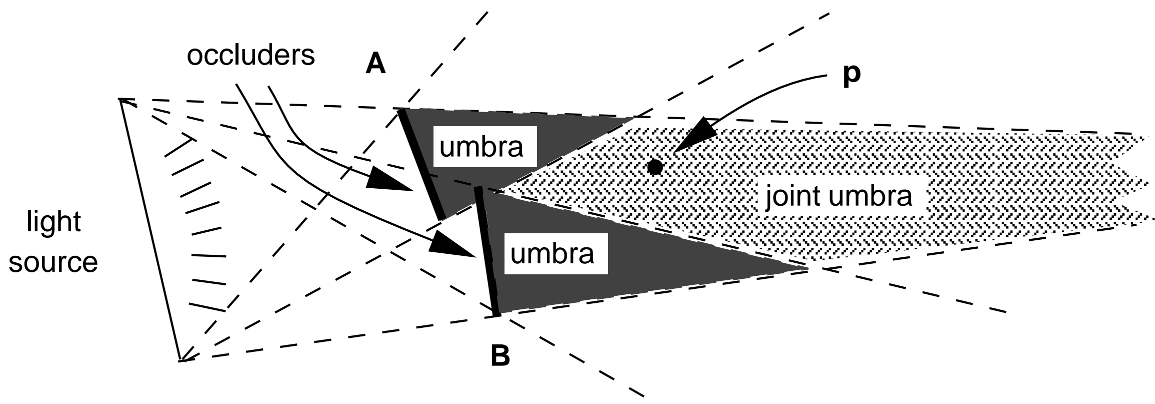

Figure 2.2: In two dimensions, a pair of occluders can jointly hide points from the light source that neither occluder hides alone.

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 light source, yet not in either of the individual occluders' umbrae. It is easy to see that two occluders **A** and **B**

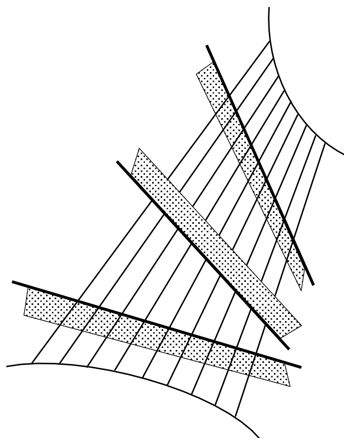

Figure 2.3: In three dimensions, three mutually skew occluder edges can generate a quadric shadow boundary.

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.

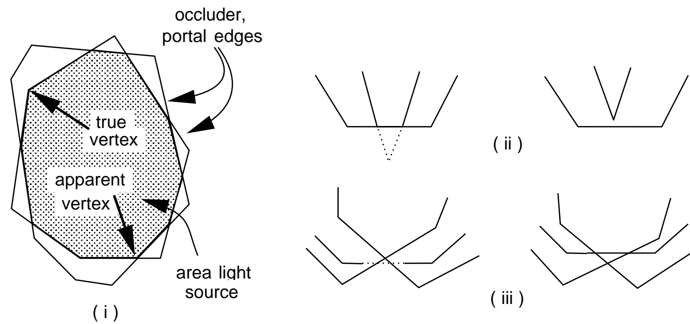

Figure 2.4: An aspect of a polyhedral object in the presence of polyhedral occluders (i). The aspect can change qualitatively in only two fundamental ways, along VE (ii) or EEE (iii) surfaces.

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 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).

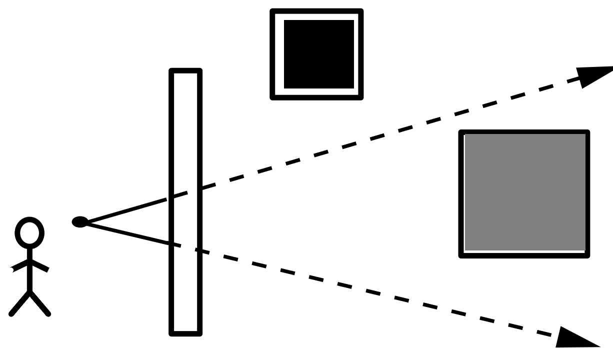

Figure 2.5: Octree-based culling [GBW90]. The contents of the black cell are correctly marked invisible. The contents of the gray cell are marked potentially visible and subsequently rendered, even though the cell is entirely occluded by the foreground rectangle.

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.

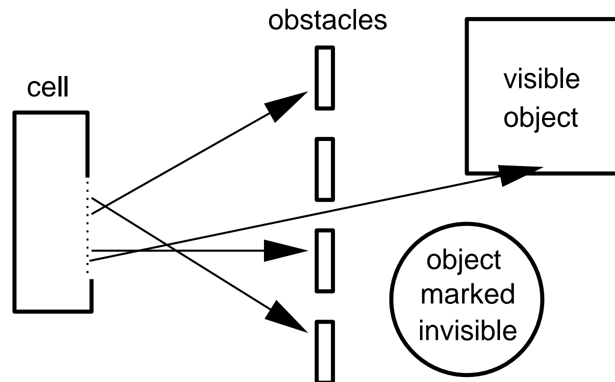

Figure 2.6: Stochastic ray-casting from portals [Air90]. The square object is correctly determined potentially visible. The circular object is not reached by a random ray, and is (incorrectly) determined to be invisible from all points in the source cell.

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].

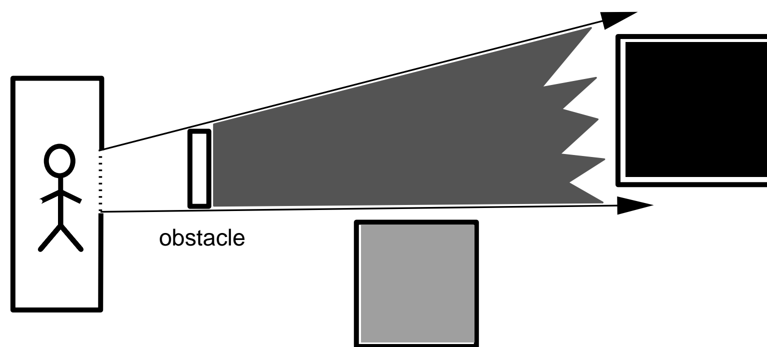

Figure 2.7: Shadow-volume casting [Air90]. Portals are treated as area light sources. Occluders cast shadows which generally remove cell contents or objects from the PVS.

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.

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

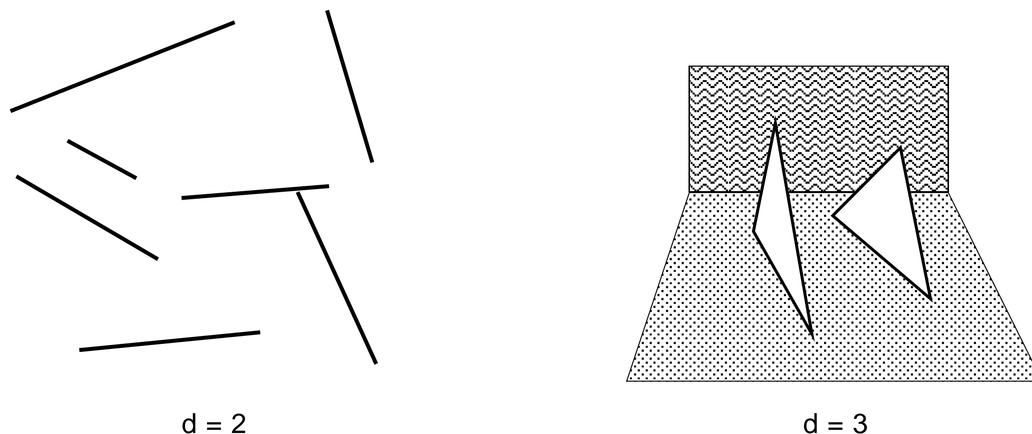

Figure 3.1: Convex, opaque occluders, in two and three dimensions.

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.

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.

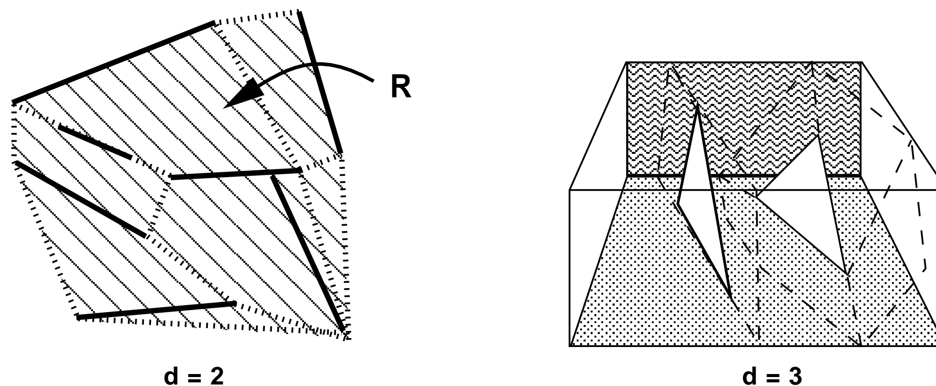

Figure 3.2: Spatial subdivisions, in two and three dimensions.

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.

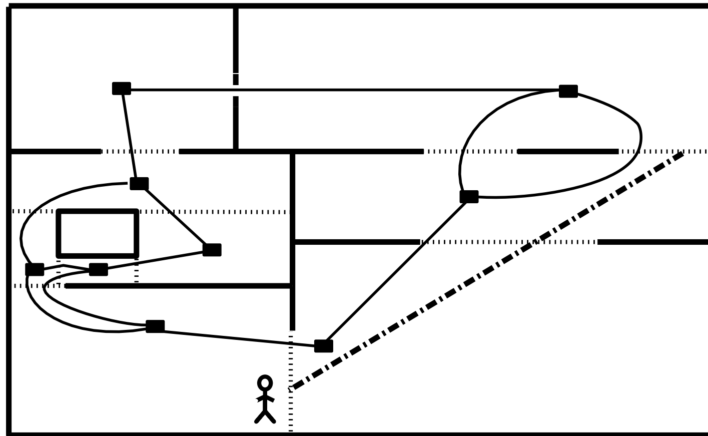

Figure 3.3: A 2D spatial subdivision, and corresponding adjacency graph. An observer is schematically represented at the lower left, and a sightline (broken) stabs a portal sequence of length three.

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 walk-through 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.

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*($\mathbf{P}$) that, given a portal sequence $\mathbf{P}$, determines either a stabbing line for $\mathbf{P}$ 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 //):

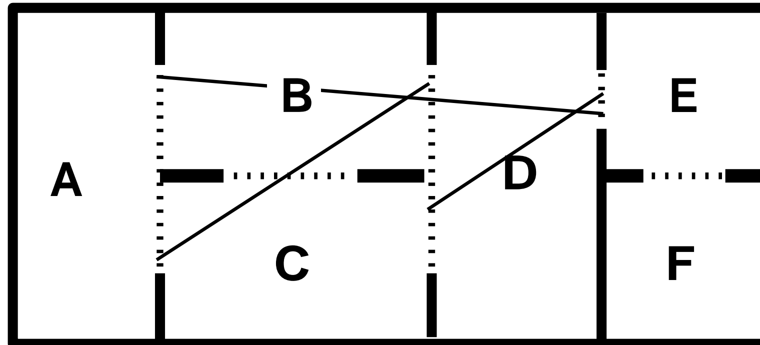

Figure 3.4: Some portal sequences that admit sightlines.

```
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
```

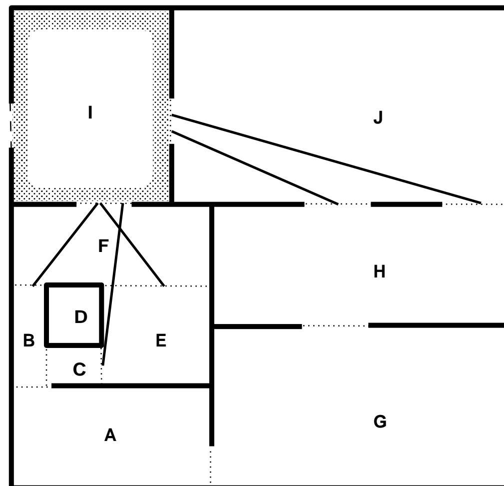

Figure 3.5: Finding sightlines from I.

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

Find_Visible_Cells $(I,\ \mathbf{P} = [\ ],\ \mathcal{V} = \emptyset)$
  Find_Visible_Cells $(F,\ \mathbf{P} = [I/F],\ \mathcal{V} = \{I\})$
    Find_Visible_Cells $(B,\ \mathbf{P} = [I/F,\ F/B],\ \mathcal{V} = \{I, F\})$
    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.

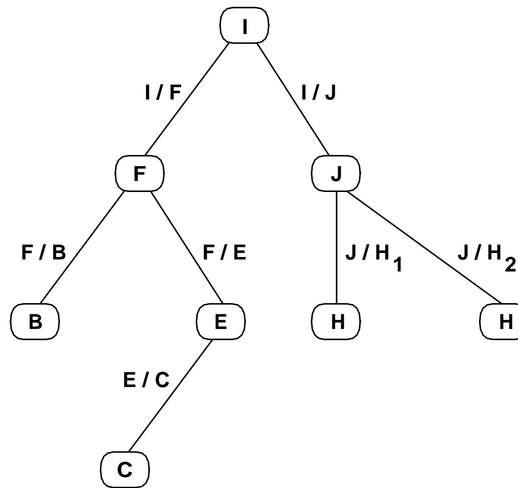

Figure 3.6: The stab tree rooted at I.

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.

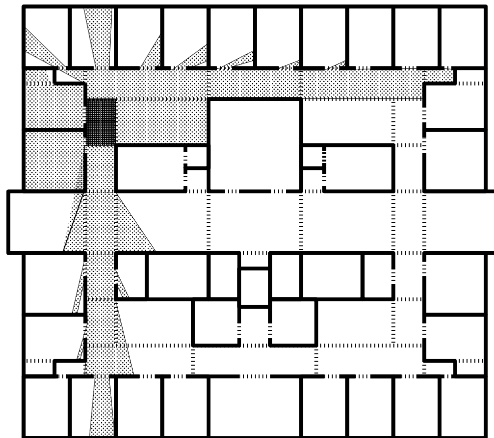

Figure 3.7: Distant cells are, in general, only partially visible (gray areas) from the source cell (dark).

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

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

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 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.

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 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

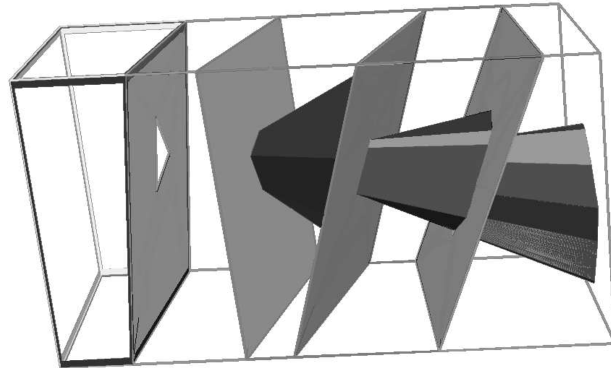

Figure 3.9: Successively narrowing antipenumbrae cast by an area light source in 3D (here, the leftmost portal) through the cells of a conforming spatial subdivision.

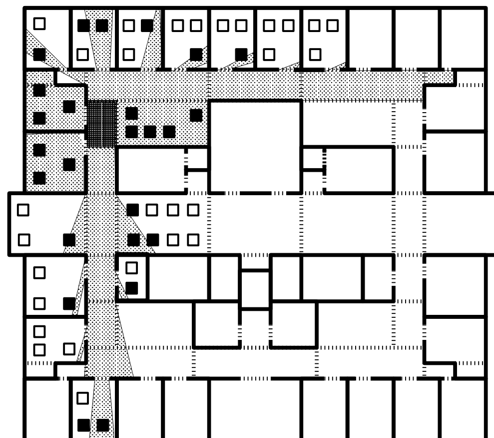

Figure 3.10: Cell-to-object visibility (filled squares) for a given source in 2D.

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.

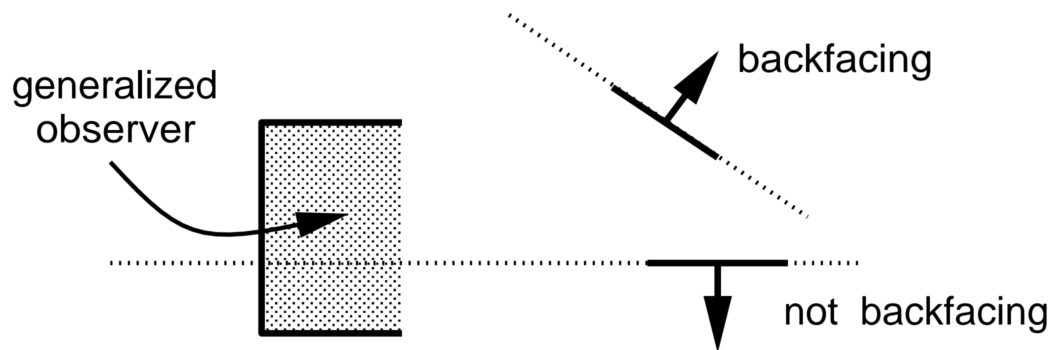

Figure 3.11: An object or occluder can be backfacing with respect to a *generalized* observer.

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.

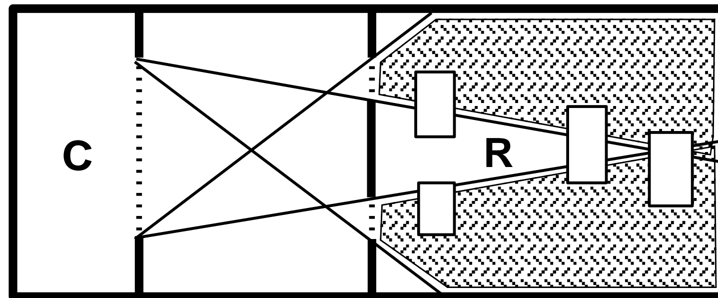

Figure 3.12: A source cell may reach another through several paths.

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.

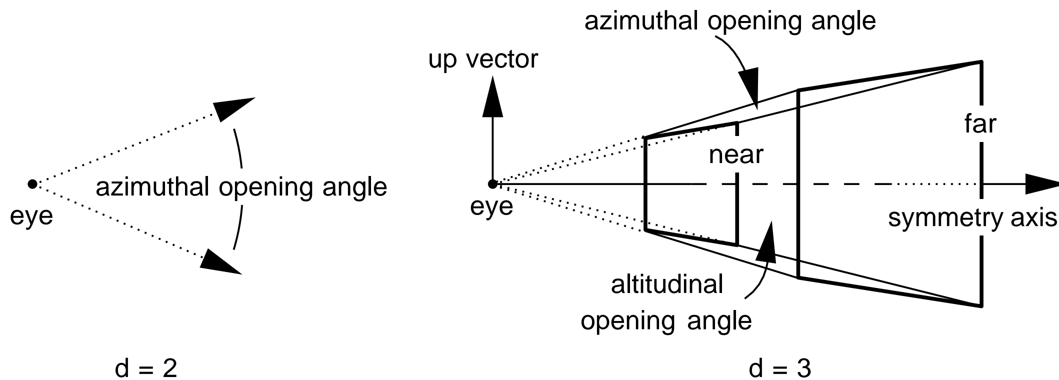

Figure 3.13: Observer view variables as view cone (2D) and view frustum (3D).

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.

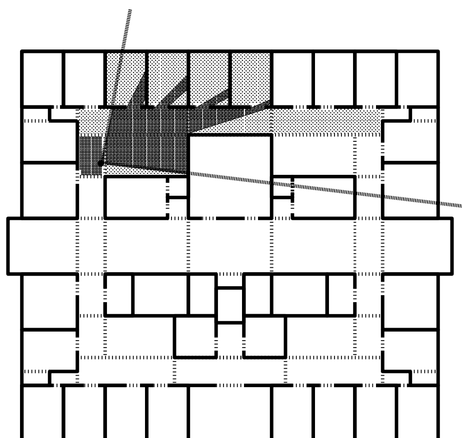

Figure 3.14: Eye-to-cell and eye-to-region visibility sets for an actual observer.

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.

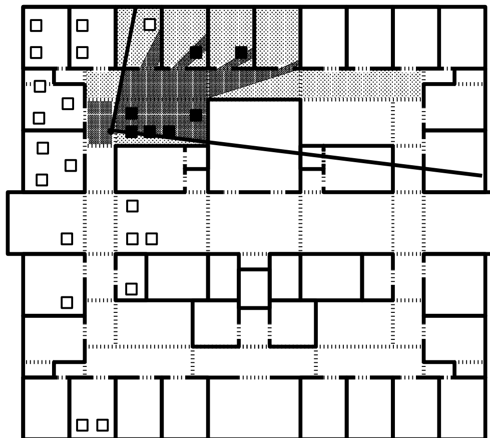

Figure 3.15: Eye-to-cell (light areas), eye-to-region (dark areas), and eye-to-object visibility set (those dark squares incident on the eye-to-region visibility) for an actual observer.

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.

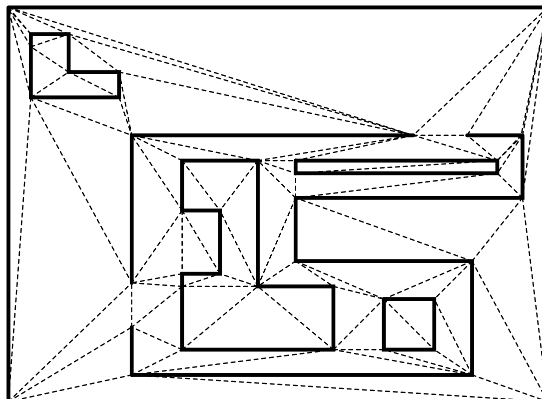

Figure 4.1: A linear-size constrained triangulation of the n occluders. Occluders are shown as bold segments, portals as dashed lines.

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*** $[x_{0,min} \dots x_{0,max}], \dots, [x_{k-1,min} \dots x_{k-1,max}]$. 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 $[x_{s,min} \dots a)$ and $[a \dots x_{s,max})$, 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 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.

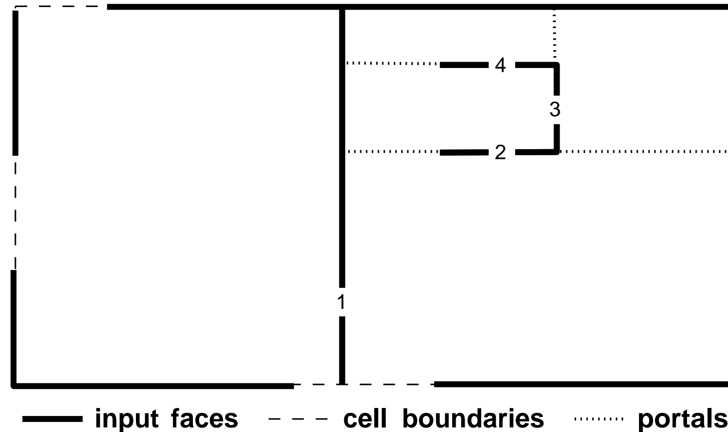

Figure 4.2: The leaf cells of a linear-size k-D tree over n line segments. Split planes are numbered in the order which they were introduced.

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 [PS85]; 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.

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.

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 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, PS85]. 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.

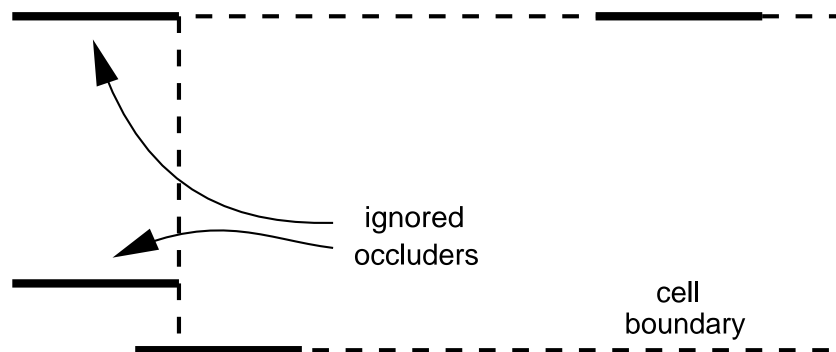

Figure 4.4: Occluders intersecting the cell boundary in a point may be ignored during portal enumeration.

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.

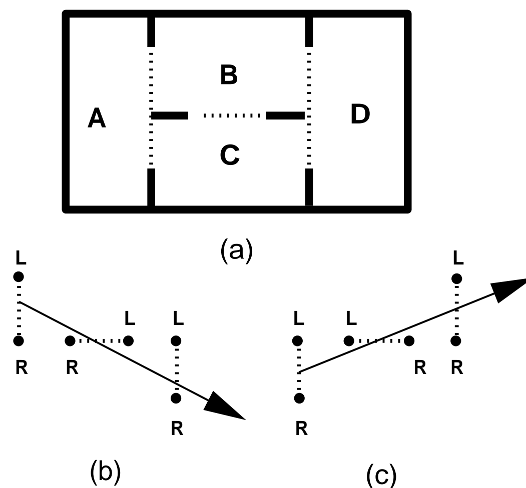

Figure 4.5: Oriented portal sequences, and separable sets **L** and **R**.

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

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

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).

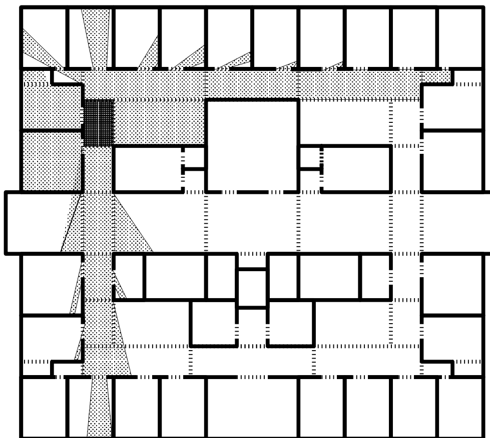

Figure 4.6: Distant cells are, in general, only partially visible from the source.

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, PS85, 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.

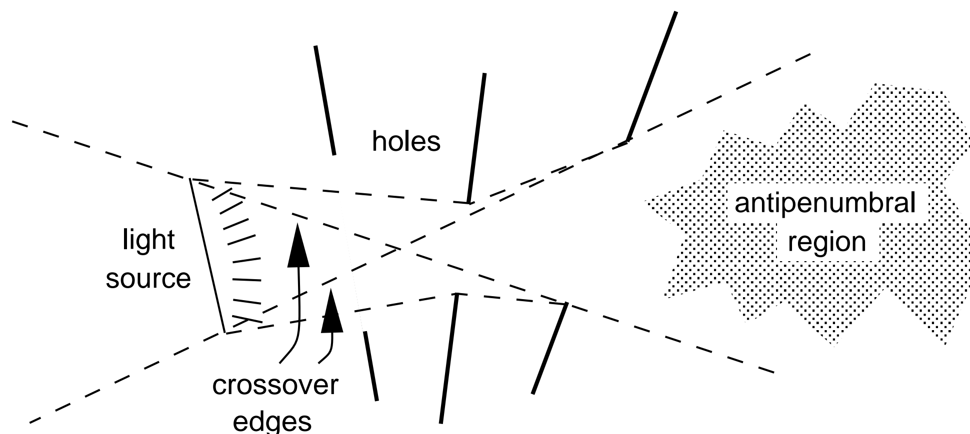

Figure 4.7: In two dimensions, a portal sequence admits an "hourglass" of stabbing lines.

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

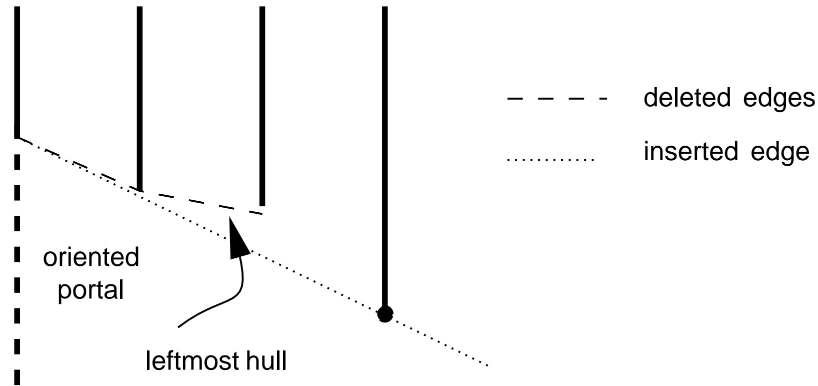

Figure 4.8: The left-hand points form a convex chain.

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).

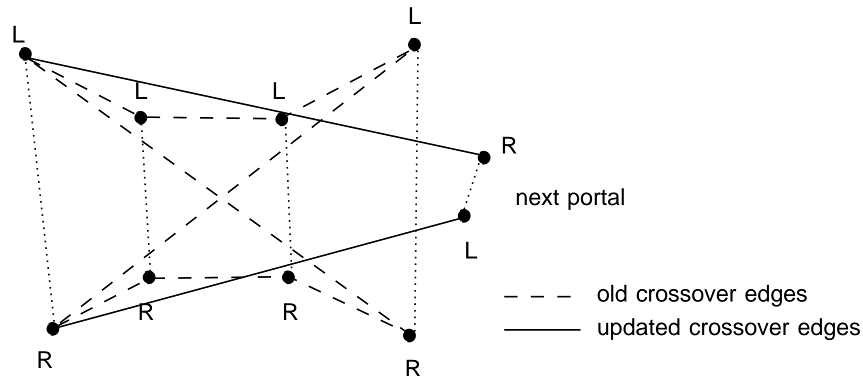

Figure 4.9: A 2D portal sequence terminates if the updated crossover edges do not intersect (above), or if the newly encountered portal does not intersect the active antipenumbral region (not shown).

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.

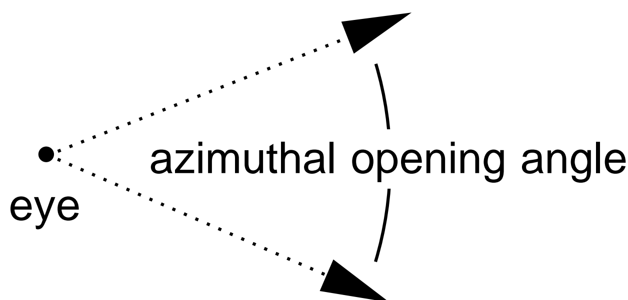

Figure 4.10: Observer view variables in 2D.

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.

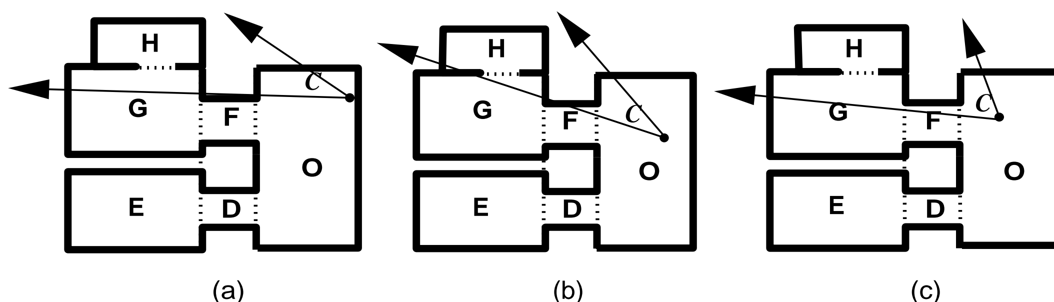

Figure 4.11: Culling O's stab tree against a view cone C.

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 $\mathcal{S}$, subject to the constraint that every cell traversed must intersect the interior of C. This requires time $O(|\mathcal{S}|)$, 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 $\mathcal{S}$ permits an efficient implementation of this sufficient criterion since $\mathcal{S}$ 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.

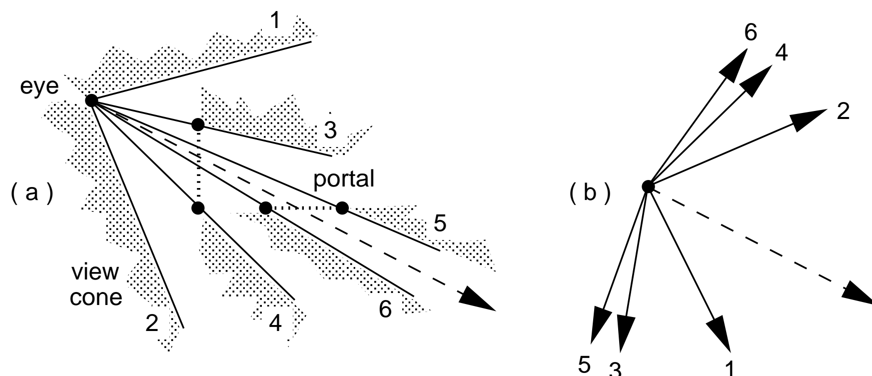

Figure 4.12: The $2m+2$ halfspace normals arising from a portal sequence of length m (a), and the corresponding 2D linear program (b). The dashed arrow is a feasible solution.

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).

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.

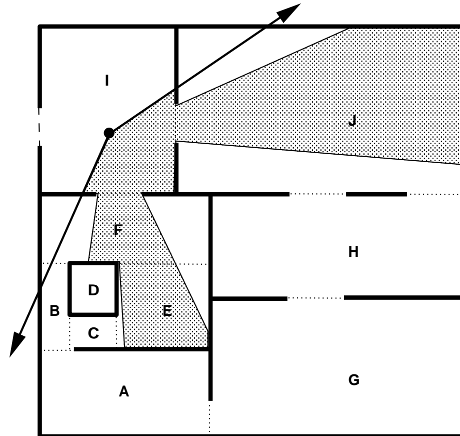

Figure 4.13: The view cone during the stab tree DFS.

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.

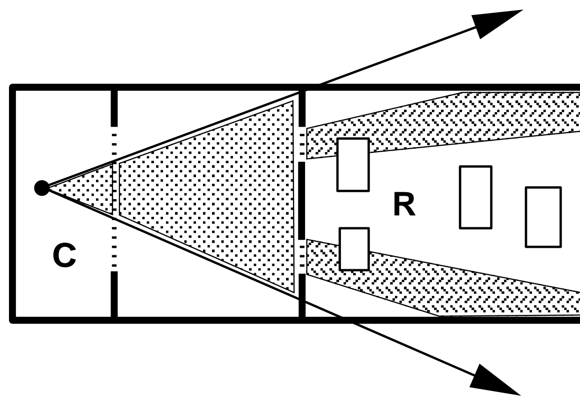

Figure 4.14: The 2D eye-to-object visibility computation.

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.