Chapter 5

Three-Dimensional Axial Environments

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

5.1 Major Occluders and Detail Objects

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

5.2 Spatial Subdivision

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

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

5.2.1 Splitting Criteria

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

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

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

After these splitting rounds, the *cell volume* and *cell aspect* rounds commence. The cell volume round subdivides those cells that are larger than some specified fraction of the total model volume.

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

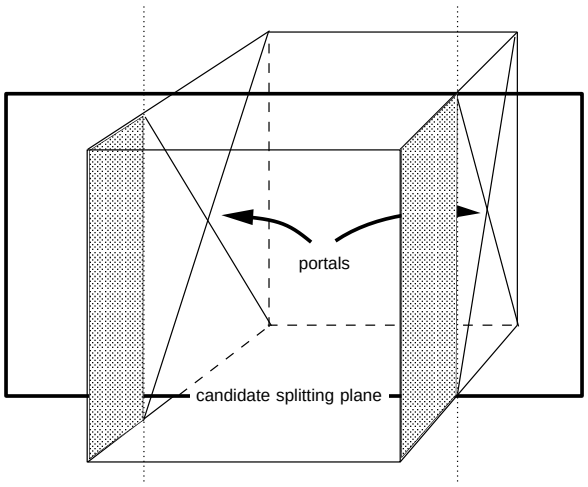


Figure 5.1: Axial splitting planes are chosen to coincide with portal edges.

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

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

5.2.2 Point Location

Point location in 3D k-D trees is a straightforward recursive procedure analogous to that of §4.2.1. The *per-node* search time is constant; a point can be checked for inclusion in a parallelepiped in $O(1)$

time. The check requires six floating-point comparisons, since each coordinate of the 3D point must be compared for inclusion within a one-dimensional range.

5.2.3 Cell Population

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

5.2.4 Neighbor Finding

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

5.2.5 Portal Enumeration

Once the neighbors across a particular source cell boundary face have been found, portal enumeration can proceed. The spatial data structure stores with each cell face a list of incident occluders. The *egress* for this face is the set difference between the axial rectangle of the boundary face, and the incident occluders, themselves axial rectangles (Figure 5.2). Given the axial nature of the data, this set difference can be performed efficiently and robustly. Our implementation simply initializes a rectangle list to a single element (the boundary face), then subtracts each occluder extent in turn from the set union of the rectangles on the list. The unoccluded portion of the cell boundary is maintained as a rectangular decomposition throughout the subtraction. The primitive operation subtracts one rectangle from another in constant time, producing at most a constant number of new rectangles (Figure 5.3). When each of the occluders has been subtracted, the remaining rectangles comprise a rectangular decomposition of the boundary face egress. This decomposition is then compared to the neighbor list for the boundary face, and each rectangle is clipped to any incident faces from the neighbor cell. Finally, a postprocessing pass coalesces adjacent rectangles that lead to the same neighbor cell, and a portal is created for each resulting rectangle/neighbor pair. If this merging generates non-convex portals, the merge is suppressed. The result is a portal list comprised solely of rectangles.

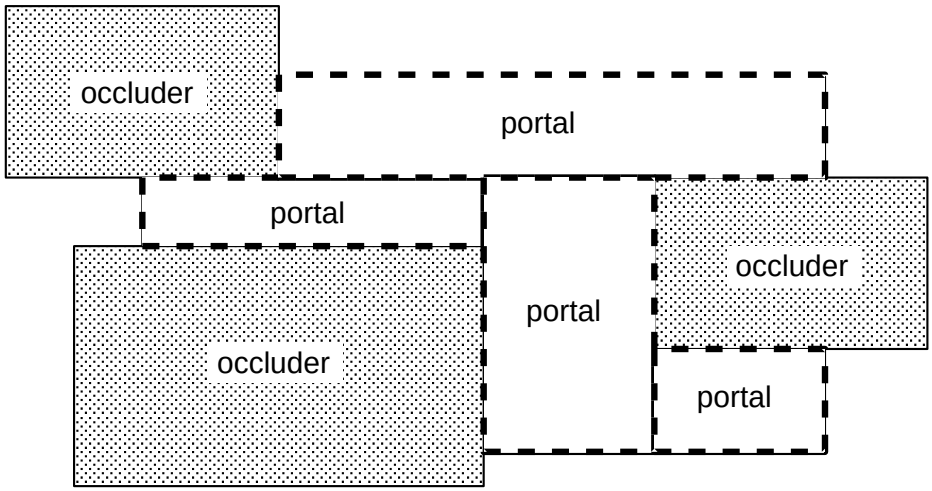


Figure 5.2: Portal enumeration as a set difference of sets of rectangles.

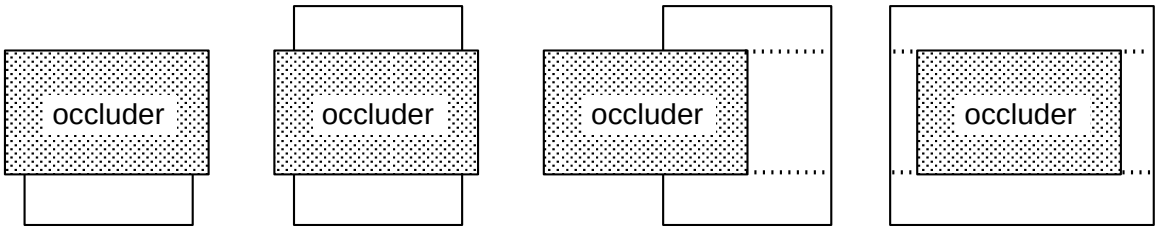


Figure 5.3: The primitive operation subtracts one rectangle from another. Assuming the two rectangles intersect, the result must be zero (not shown), one, two, three, or four new rectangles (left to right, above).

5.3 Static Visibility Operations

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

5.3.1 Cell-to-Cell Visibility

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

Before considering the problem in three dimensions, we consider the two dimensional analog. In the plane, we are looking for an oriented line that intersects each of n vertical or horizontal line segments. In addition, we are given the direction in which the stabbing line must traverse the line segments: either bottom to top, or top to bottom for the horizontal line segments, and either left to right, or right to left for the vertical line segments. This is equivalent to the following problem. Given two sets of points, $\{p_i\}$ and $\{q_i\}$, find a line that passes above the $\{p_i\}$ and below the $\{q_i\}$. Such a line is called a feasible line.

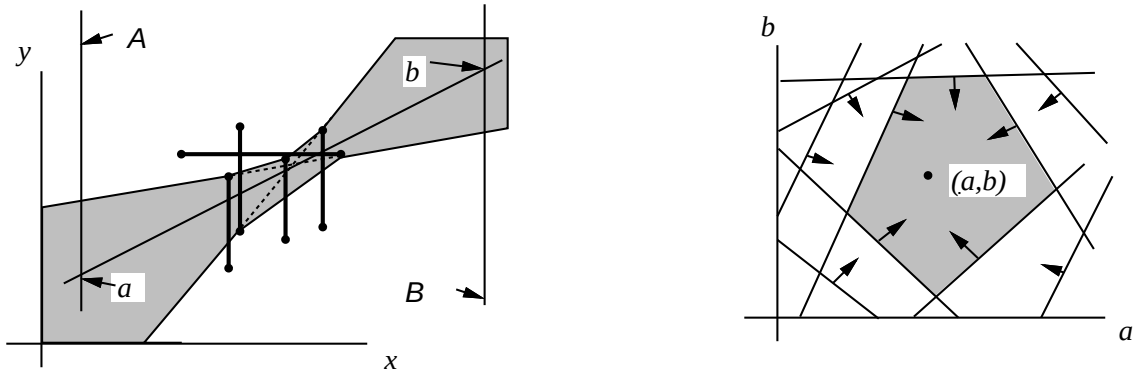


Figure 5.4: Lines in $\mathbf{R}^2$ and their dual representation.

While the set of feasible lines appears as an hourglass-shaped shaded region, as depicted in Figure 5.4, in a dual space that suitably parametrizes all lines, the set of feasible lines is a convex polygon.

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

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

Stabbing Isothetic Rectangles in $\mathbf{R}^3$

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

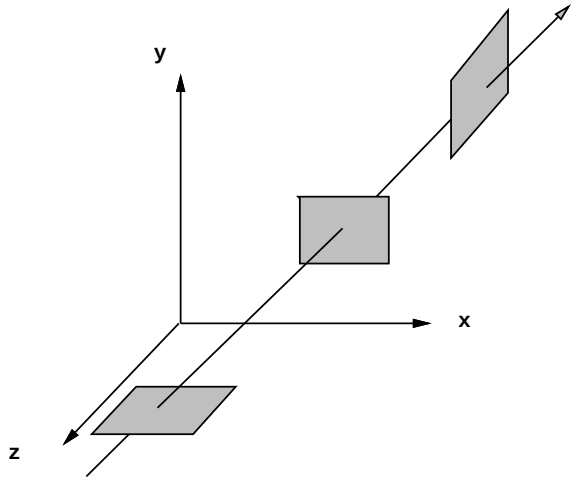


Figure 5.5: An axial portal sequence and stabbing.

A priori, we should consider all pairs of x-aligned edges, all pairs of y-aligned edges, and all pairs of

z-aligned edges. However, if there are m x-aligned edges there will be at most m pairs defining planes that contain *any* feasible lines (Figure 5.6). Consider a line L in the plane defined by the x-aligned edge pair A and B. A plane Q perpendicular to the x axis is shown, along with the intersections of

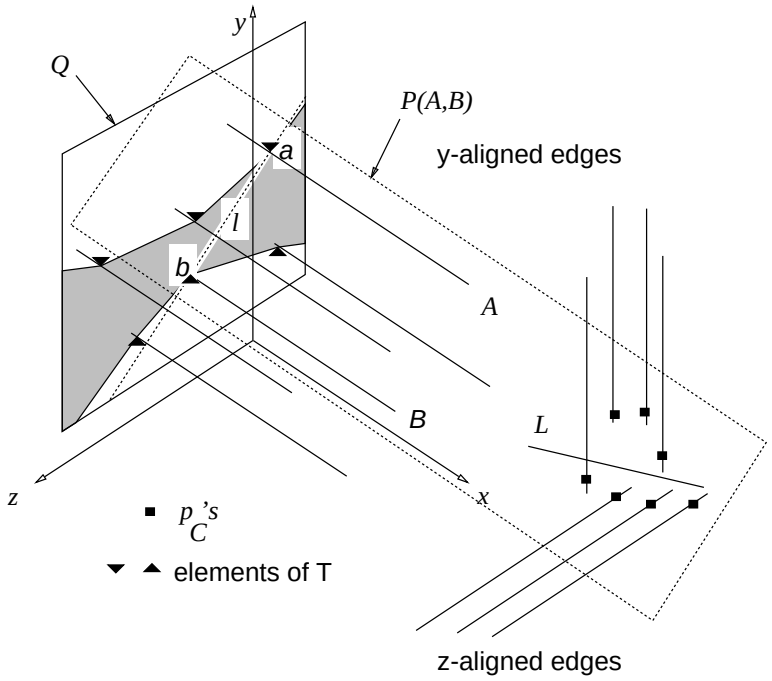


Figure 5.6: Reducing three-dimensional stabbing to a series of two-dimensional problems.

Q with the x-aligned edges. Also shown is l, the projection of L onto Q. Note that L will be feasible with respect to the x-aligned edges if and only if l is feasible with respect to the intersection points on Q. If we can determine the pairs of intersection points that define feasible lines in Q, then we have also determined the pairs of x-aligned edges that are feasible with respect to the other x-aligned edges. This is exactly the two-dimensional problem of the previous section. By explicitly constructing the polygon shown in Figure 5.4, only a small list of pairs of x-aligned edges must be consider. In fact, the number of pairs will be no greater than the number of x-aligned edges. Clearly, this holds in the y and z directions as well.

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

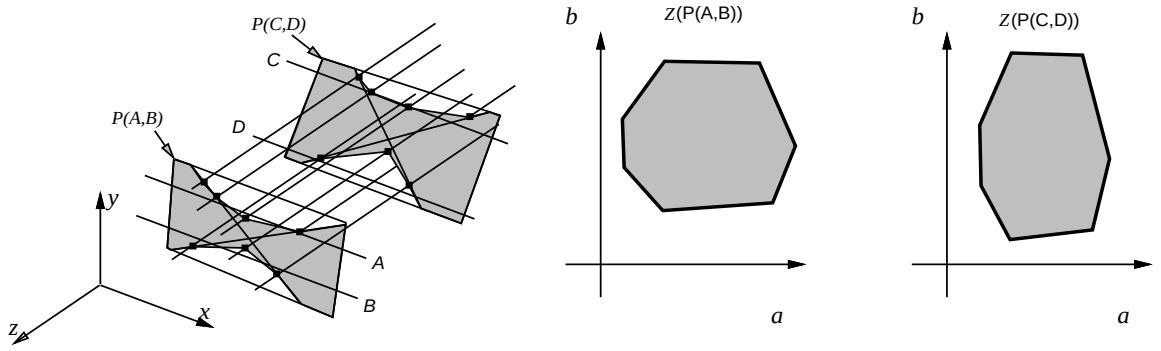


Figure 5.7: The structure of the region of feasible lines is independent of P.

points of the lines from the other two directions with the plane P. Solve the resulting two-dimensional stabbing problem as a linear programming problem. In pseudo code:

Stabbing_Line_1 (X, Y, Z)
(1) $S = X \cup Y \cup Z$
(2) **for** $G = X,\ Y,\ Z$
(3) $F = S \setminus G$
(4) *let Q be a plane perpendicular to the lines in G*
(5) *let T be the intersections of G with Q*
(6) *let H be the convex hull (in the dual space) of the lines in Q feasible with respect to T*
(7) **for** *each corner $l \in H$*
(8) *determine the plane P defined by l and the direction of the lines in G.*
(9) **for** *each $C \in F$*
(10) *let p_C be the point where C intersects P*
(11) **endfor**
(12) *use a linear programming algorithm to determine if there exists an L in P satisfying the p_C's.*
(13) **if** *there exists an L* **return** L
(14) **endfor**
(15) **endfor**
(16) **return** *infeasible*

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

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

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

Stabbing_Line_2 (X, Y, Z)
(1) *Let* $\mathcal{S}[0] = \mathcal{X}()$ *(the logarithmic query structure for x)*
(2) *Let* $\mathcal{S}[1] = \mathcal{Y}()$ *(the logarithmic query structure for y)*
(3) *Let* $\mathcal{S}[2] = \mathcal{Z}()$ *(the logarithmic query structure for z)*
(5) **for** $i = 0, 1, 2$
(4) **for** *each vertex L in* $\mathcal{S}[i]$ *(L is a line)*
(6) *determine the plane P defined by L and the direction of the lines in* $\mathcal{S}[i]$*;*
(7) **if** $\mathcal{S}[(i+1) \bmod 3](P)$ *intersects* $\mathcal{S}[(i+2) \bmod 3](P)$
return *the line corresponding to the point of intersection*
(8) **endfor**
(9) **endfor**
(10) **return** *infeasible*

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

5.3.2 Cell-to-Region Visibility

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

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

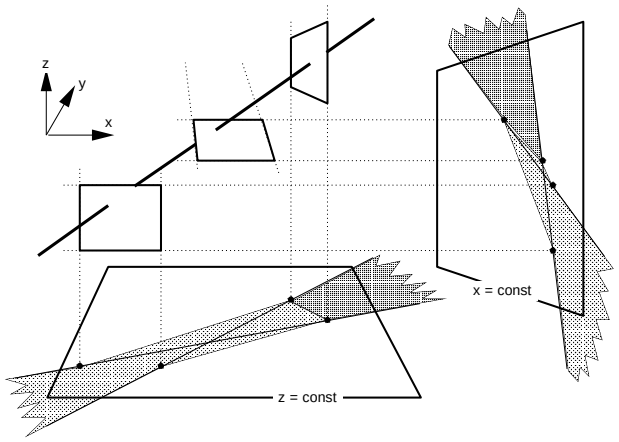


Figure 5.8: An axial portal sequence can be decomposed into three sets of 2D hourglass constraints.

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

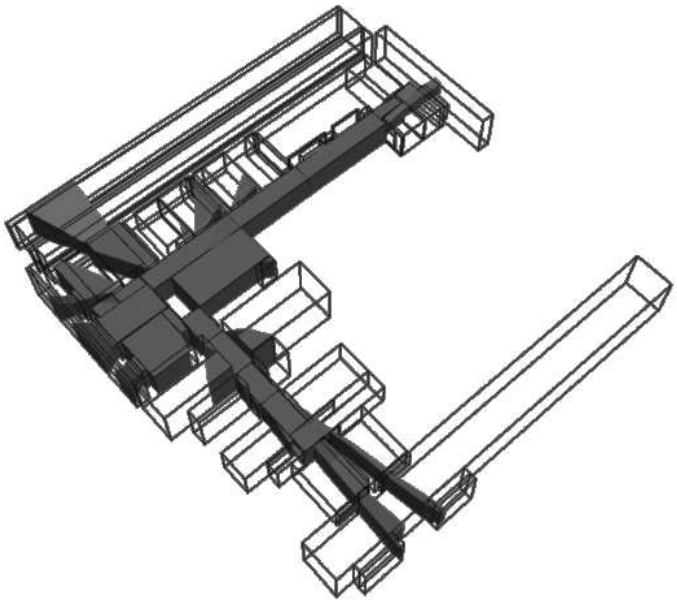


Figure 5.9: The linearized visibility volumes emanating from an axial source cell.

5.3.3 Cell-to-Object Visibility

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

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

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

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

5.4 Dynamic Visibility Queries

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

5.4.1 Observer View Variables

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

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

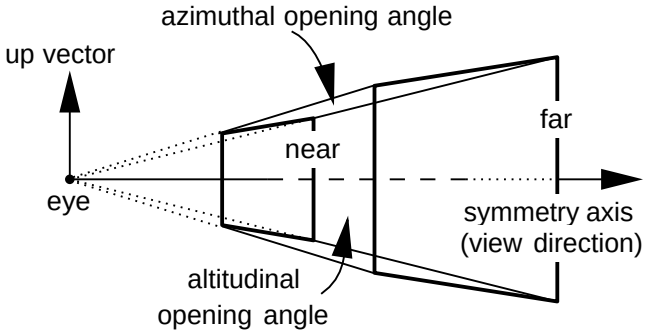


Figure 5.10: Three-dimensional observer view variables.

5.4.2 Eye-to-Cell Visibility

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

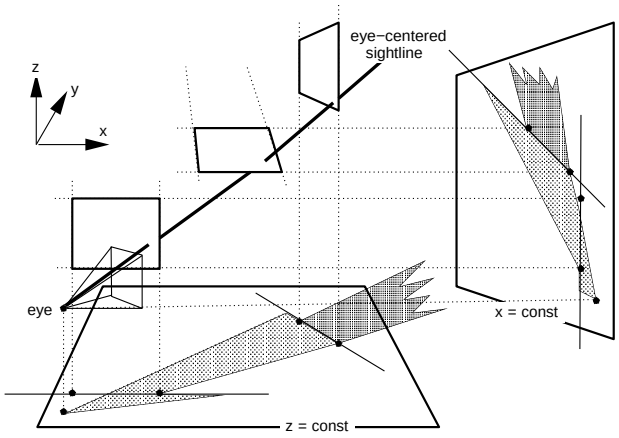


Figure 5.11: Decomposing axial portals into constituent axial eye-centered constraints.

Two extremal lines are maintained on each of the three principal planes. As each portal is encountered, it is decomposed, and the set of extremal lines is suitably narrowed in the appropriate axial plane, using the 2D method described in §4.4.2. If the extremal lines exclude the portal entirely, the DFS is immediately terminated. Otherwise, the three resulting pairs of lines together imply six constraint planes in three-space. These six planes are combined with the four planes defining the view frustum to produce at most ten linear constraints on the existence of a stabbing line through the eye. These constraints can be cast as a linear program of three-coefficient constraints, as in §8.4.2. We can therefore examine the ten linear constraints for a feasible solution (a stabbing line through the eye and the portal sequence) in constant time. If the linear program fails to find a stabbing line through the eye, the most recent portal is impassable, and the active branch of the DFS terminates.

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

5.4.3 Eye-to-Region Visibility

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

5.4.4 Eye-to-Object Visibility

Eye-to-object visibility can also be formulated as a DFS augmentation. Surprisingly, examining each object for an eye-centered stabbing line can be done with at most ten halfspaces, in constant time. This is due to the fact that the object bounding box is axial, and induces at most six silhouette planes containing the eye (Figure 5.12). The set of silhouette planes is assembled via a table-lookup on the $26 = 3^3 - 1$ possible positions of the eye with respect to the three infinite axial slabs whose intersection is the box. Each of the three pairs of silhouette planes can be compared to the three pairs of portal-induced constraint planes, just as if the silhouette planes arose as portal boundaries. The result is a two-dimensional linear program with at most ten normal constraints (analogous to that of Figure 4.12, §4.4.2), solvable in constant time. As in the cell-to-object computation, if the object tested is not entirely contained in the reached cell, six more constraints must be added to the linear program to ensure that the computed sightline intersects the object bounding box inside the cell.

Analogously to the cell-to-object case, there is an alternative method for computing eye-to-object visibility. Rather than performing an explicit sightline test per-object, we can construct a constant-size description of the eye-to-region visibility (a polyhedron) in the reached cell, and compare objects to the interior of this polyhedron. This construction is attractive in that it discards irrelevant halfspace constraints. However, we do not use this method for three reasons. First, the polyhedron construction

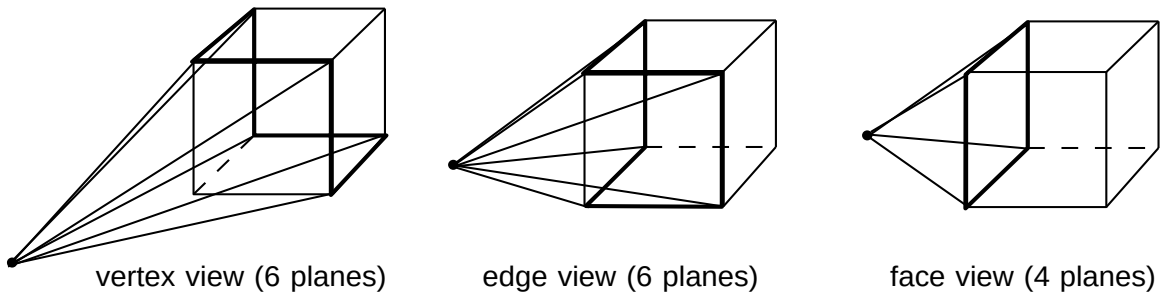


Figure 5.12: The four to six faces of an eye-centered bounding box pyramid.

is a 3D operation and is hard to implement both efficiently and robustly. Second, the resulting polyhedron/bounding box intersection check requires a 3D linear program, rather than 2D as in the sightline check, and the linear programming algorithm we use has $O(d!n)$ time complexity; the result is that the linear program slows down by a factor of three in the worst case (for which no superfluous constraints are eliminated). Finally, since the sightline check is constant-time in the first place, we can improve its efficiency by at most a constant factor.

Chapter 6

Line Coordinates

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

6.1 Plücker Coordinates

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

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

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

$$\begin{aligned} \pi_{\ell 0} &= p_x q_y - q_x p_y \\ \pi_{\ell 1} &= p_x q_z - q_x p_z \\ \pi_{\ell 2} &= p_x - q_x \\ \pi_{\ell 3} &= p_y q_z - q_y p_z \\ \pi_{\ell 4} &= p_z - q_z \\ \pi_{\ell 5} &= q_y - p_y \end{aligned}$$

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

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

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

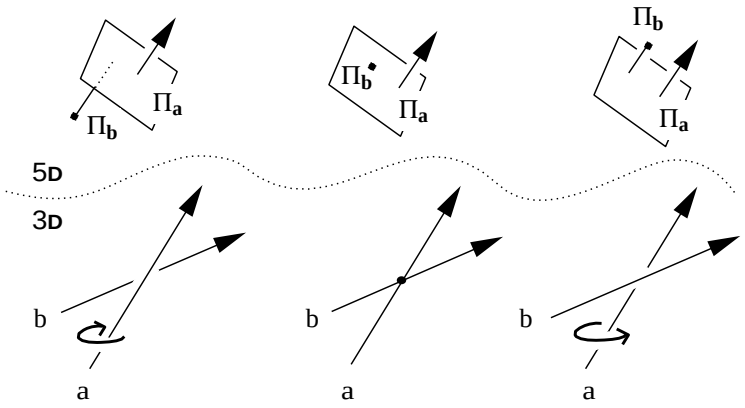


Figure 6.1: The right-hand rule applied to *side*(a, b).

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

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

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

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

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

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

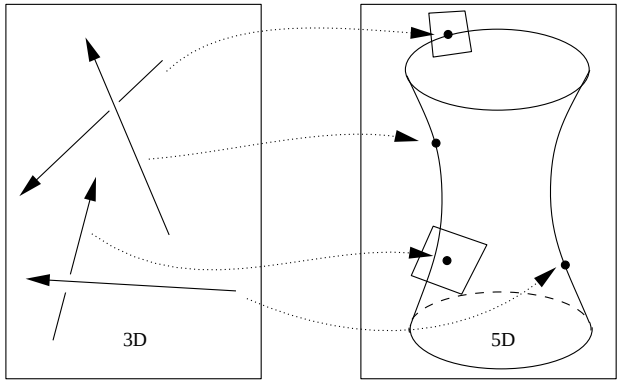


Figure 6.2: Directed lines map to points on, or hyperplanes tangent to, the Plücker surface.

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

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

6.2 Degrees of Freedom in Line Space

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

Suppose five lines are specified in 3D (Figure 6.3), and we wish to determine whether there exists a sixth line incident on the given five. This question can be answered generically with a DOF argument. Starting with five degrees (the dimensionality of the problem), each of the five specified lines induces

a hyperplane in Plücker space, and removes a DOF, leaving zero degrees of freedom, i.e., a point. The Plücker quadric removes one more degree, resulting in -1 degrees of freedom, an overspecified problem that generally has no solution. In other words, the five hyperplanes determine a unique point, which almost always lies off of the Plücker surface. In general, therefore, the 3D solution lines have complex coefficients.

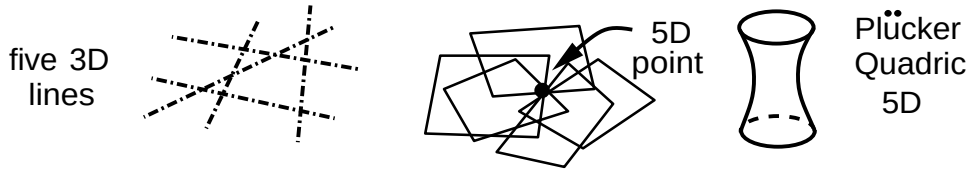


Figure 6.3: Generically, no real line is incident to five given lines.

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

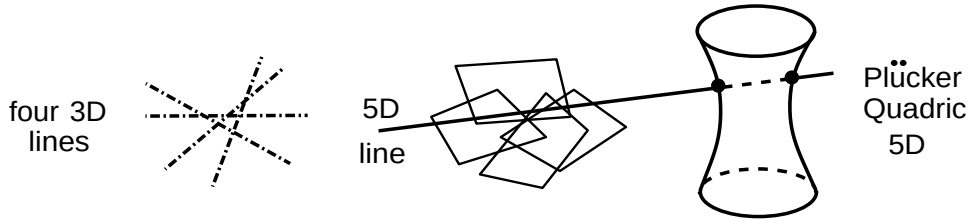


Figure 6.4: Generically, two real lines are incident to four given lines.

The surprising result is that, through four lines in three dimensions, there generally exist *two* other lines. This single fact captures the special behavior of lines in three dimensions, and the profound difference between their essentially quadratic nature and the essentially linear nature of points and planes.

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

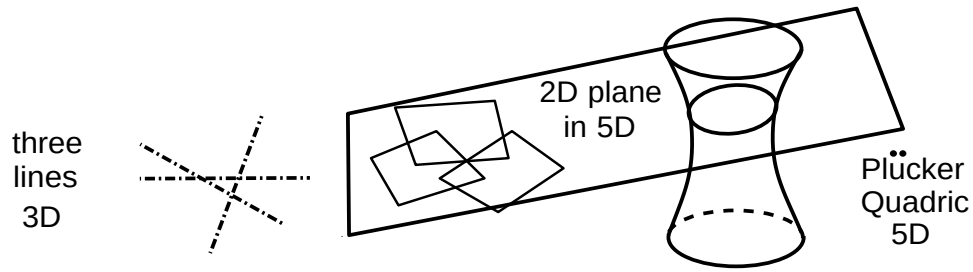


Figure 6.5: Generically, a one-parameter family of lines is incident to three given lines.

6.3 Computing the Incident Lines

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

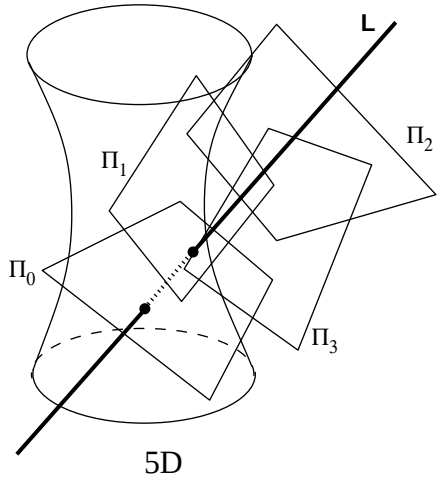


Figure 6.6: The four Π_k determine a line to be intersected with the Plücker quadric.

Thus the incidence computation has two parts. The first is an intersection of four hyperplanes Π_k, to form a line (the *null space*) of the Π_k. The second is an intersection of this line with a quadric

surface to produce a discrete result. We have implemented this computation in the C language using a FORTRAN singular value decomposition package from Netlib [DG87]. Figure 6.7 depicts the algorithm applied to four generic lines. Note that the input lines (thick) are mutually skew, and that each of the two solution lines (thin) pierces the input lines in a distinct order.

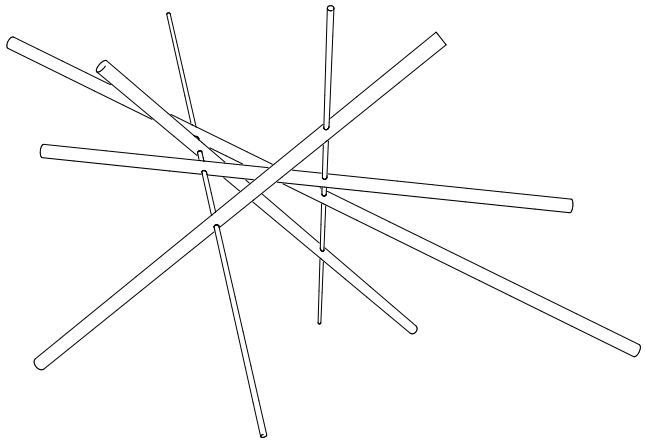


Figure 6.7: The two lines incident through four generic lines in 3D.

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

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

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

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

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

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

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

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

The null space property implies that

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

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

$$\mathbf{\Lambda}(t) \odot \mathbf{\Lambda}(t) = 0.$$

This is a quadratic equation in t:

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

or

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

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

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

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

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

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

If $\sigma_3 = 0$ and $\sigma_2 \neq 0$ then $\mathbf{M}$ has rank three. That is, only three rows of $\mathbf{M}$ are linearly independent. In this case, the set of lines incident on the l_k are those lines whose Plücker coefficients are orthogonal

to the first three rows of $\mathbf{V}$. Thus, by the SVD, the last three rows of $\mathbf{V}$ span the space of lines (real and imaginary) incident on the l_k. Consider the map $\mathbf{\Lambda}(u,v) : \mathbf{P^2} \rightarrow \mathbf{P^5}$ given by

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

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

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

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

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

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

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

6.4 Application

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

Chapter 7

Stabbing 3D Portal Sequences

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

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

7.1 Stabbing General Portal Sequences

In the following analysis, we call the light source and holes the *generator polygons*. In total, these polygons have n directed *generator edges* E_k, $k \in 1, \ldots, n$ (we assume at first that no two edges from different polygons are coplanar). Each edge E_k is a segment of a directed line e_k. Since the polygons are oriented, the e_k can be arranged so that if some directed line s *stabs* (intersects) each polygon, it must have the same sidedness relation with respect to each e_k (cf. Equation 6.2). That is, any stabbing line s must satisfy (Figure 7.1):

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

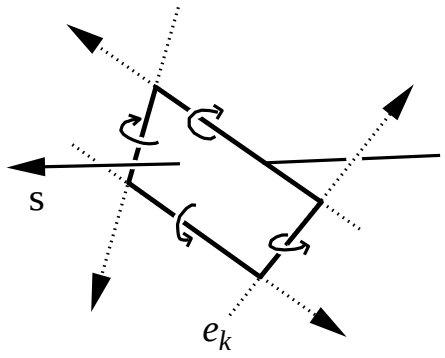


Figure 7.1: The stabbing line s must pass to the same side of each e_k.

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

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

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

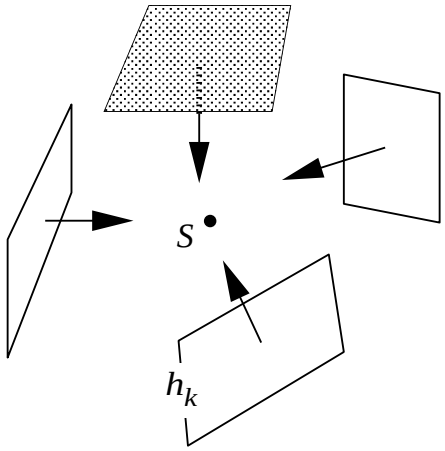


Figure 7.2: The 5D point $S = \Pi(s)$ must be on or above each hyperplane h_k.

The face structure of the polytope $\bigcap_k h_k^+$ has worst-case complexity quadratic in the number of halfspaces defining it [Grü67], and can be computed by a randomized algorithm in optimal $O(n^2)$ expected time [CS89]. We define the *extremal* stabbing lines as those lines incident on four generator

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

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

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

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

7.2 Penumbrae and Antipenumbrae

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

Imagine that the occluders are replaced by convex portals in otherwise opaque planes. In a sense complementary to that above, every point in space can again be naturally characterized. We define the *antiumbra* cast by the light source as that volume from which the entire light source can be seen, and the *antipenumbra* as that volume from which some, but not all, of the light source can be seen (Figure

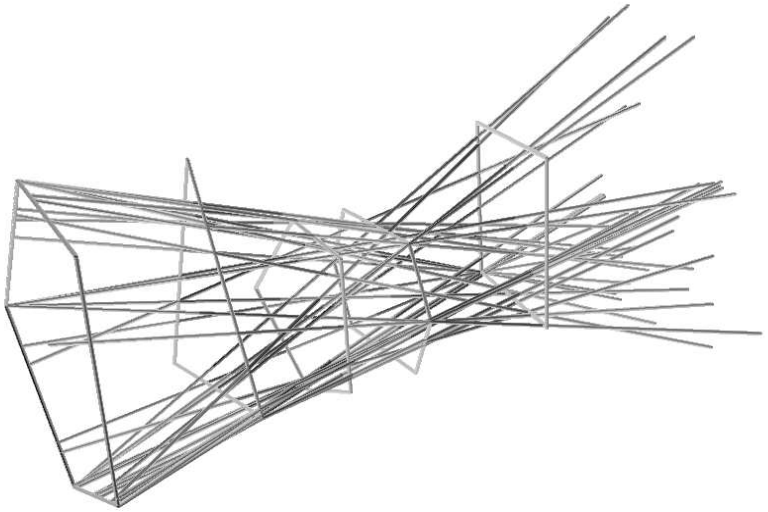


Figure 7.3: The forty extremal stabbing lines of five oriented polygons in 3D. The total edge complexity n is twenty-three. Note the hourglass-shape of the line bundle stabbing the sequence.

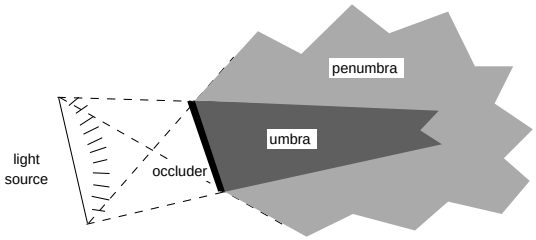


Figure 7.4: Umbra and penumbra of an occluder (bold) in 2D.

7.5). For a given light source and set of portals or occluders, the umbra is the spatial complement of the union of antiumbra and antipenumbra; similarly, the antiumbra is the spatial complement of the union of umbra and penumbra. Both the antiumbra and antipenumbra may be empty beyond the plane of the final portal, i.e., they may contain no points.

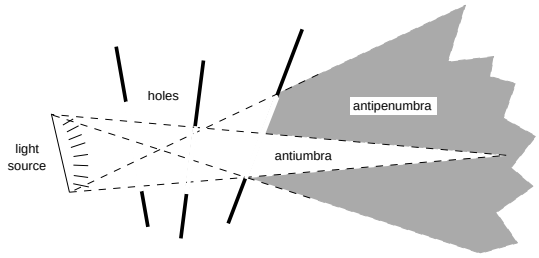


Figure 7.5: Antiumbra and antipenumbra admitted by a 3D portal sequence.

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

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

7.2.1 Event Surfaces and Extremal Swaths

There are three types of extremal stabbing lines: vertex-vertex, or VV lines; vertex-edge-edge, or VEE lines; and quadruple edge, or 4E lines. Imagine "sliding" an extremal stabbing line (of any type) away from its initial position, by relaxing exactly one of the four edge constraints determining the line (Figure 7.6). The surface, or *swath*, swept out by the sliding line must either be planar (if the line remains tight on a vertex) or a regulus, whose three generator lines embed edges of three different polygons. In the terminology of [GCS91], swaths are VE or EEE *event surfaces* important in the construction of aspect graphs, since they are loci at which qualitative changes in occlusion occur.

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

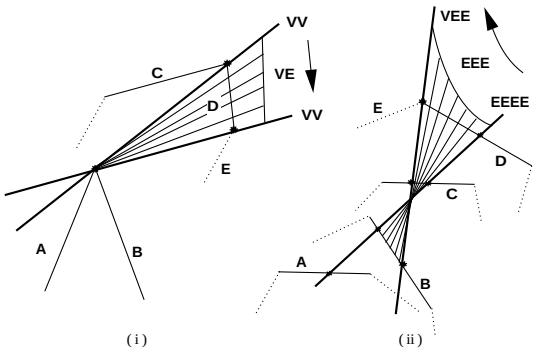


Figure 7.6: Sliding a stabbing line away from various extremal lines in 3D generates a VE planar swath (i) or a EEE quadratic surface (ii).

produces a EEE (regulus) swath tight on B, C, and D. The sliding line eventually encounters edge E and induces an extremal VEE line, terminating the swath.

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

7.2.2 Boundary and Internal Swaths

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

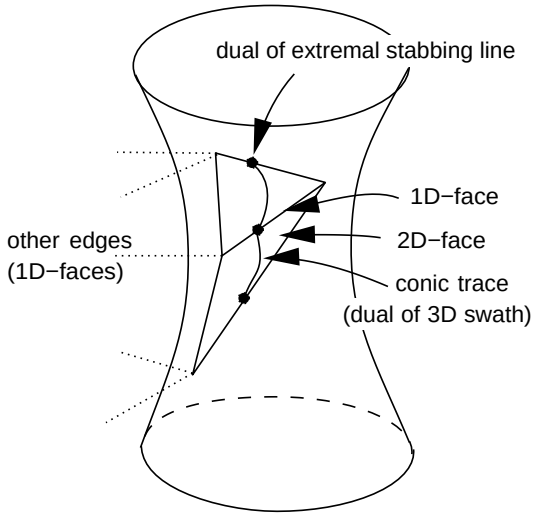


Figure 7.7: Traces (intersections) of extremal lines and swaths on the Plücker surface in 5D (higher-dimensional faces are not shown).

7.2.3 Two-Dimensional Example

The notions of boundary and internal swaths are most easily illustrated in two dimensions (Figure 7.8). As in §4.3, portals and the light source are line segments. The hourglass of lines that stabs the portal sequence contains an extremal swath (in 2D, a line segment) wherever a stabbing line can be incident on two linear constraints (i.e., portal endpoints). The illuminated volume beyond the final portal is a convex volume bounded by the two crossover hourglass edges and the final portal itself. The crossover edges are therefore boundary swaths in two dimensions, since they separate the region of partial illumination from that of zero illumination. The remaining extremal edges are valid stabbing lines, by construction, but are internal swaths.

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

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

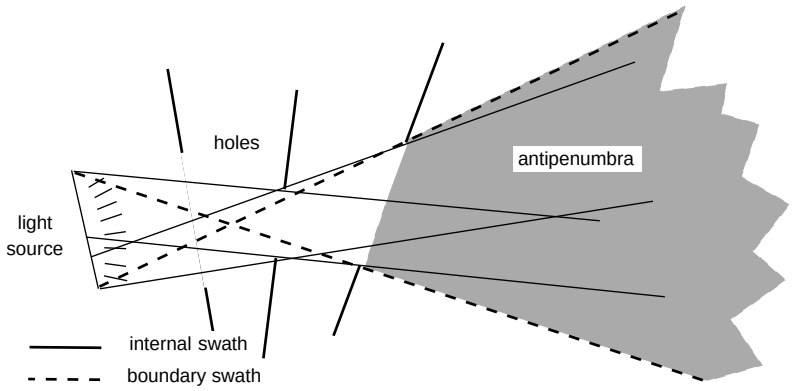


Figure 7.8: Extremal swaths arising from a two-dimensional portal sequence.

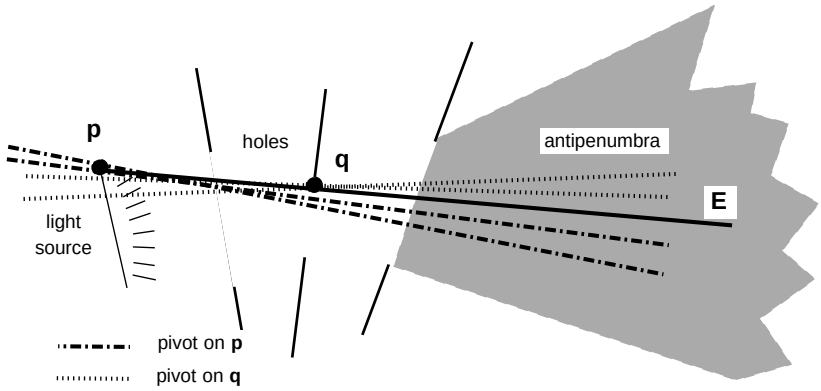


Figure 7.9: An internal swath in a two-dimensional portal sequence.

antipenumbra). Therefore, **E** is a boundary swath.

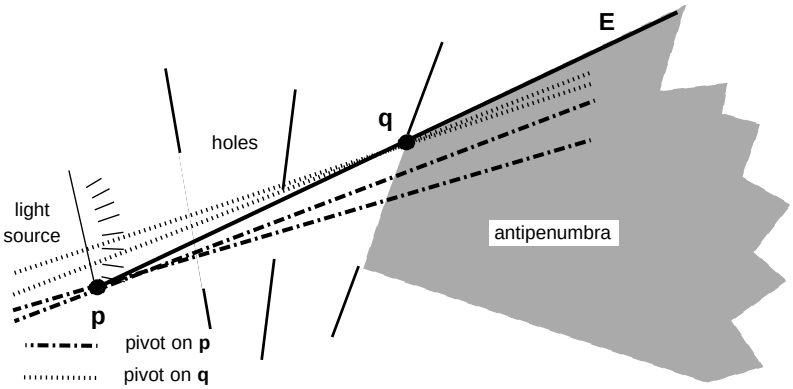


Figure 7.10: A boundary swath in a two-dimensional portal sequence.

7.2.4 Edge-Edge-Edge Swaths

In three dimensions, internal and boundary EEE swaths can be distinguished as follows. Suppose three mutually skew generator edges A, B, and C give rise to an extremal EEE swath. Choose some line **L** incident on the generators respectively at points **a**, **b**, and **c** (Figure 7.11-i). At these points, erect three vectors $\mathbf{N}_a$, $\mathbf{N}_b$, and $\mathbf{N}_c$, perpendicular both to **L** and to the relevant generator edge, with their signed directions chosen so as to have a positive dot product with a vector pointing into the interior of the edge's generator portal. (We refer to these vectors collectively as the $\mathbf{N}_i$.)

We say that three coplanar vectors can be *contained* if there exists a vector whose dot product with all three vectors is strictly positive. We claim that a swath is internal if and only if its corresponding $\mathbf{N}_i$ can be contained.

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

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

In contrast, the $\mathbf{N}_i$ of Figure 7.12-i cannot be contained. Thus, any vector (including one chosen from the gray region of Figure 7.12-ii) will have a negative dot product with at least one, and at most two, of the $\mathbf{N}_i$. Suppose it has a negative dot product with $\mathbf{N}_c$. Slice the configuration with a plane

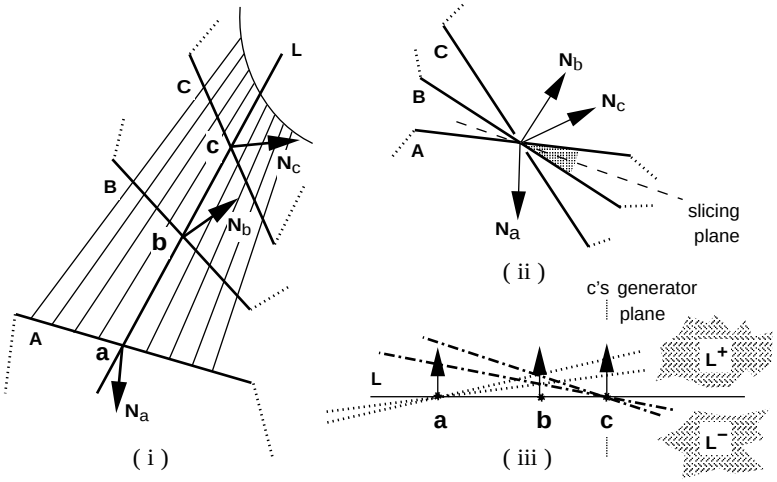


Figure 7.11: An internal EEE swath (i), viewed along $\mathbf{L}$ (ii) and transverse to $\mathbf{L}$ (iii). The $\mathbf{N}_i$ can be contained, and the swath is therefore internal.

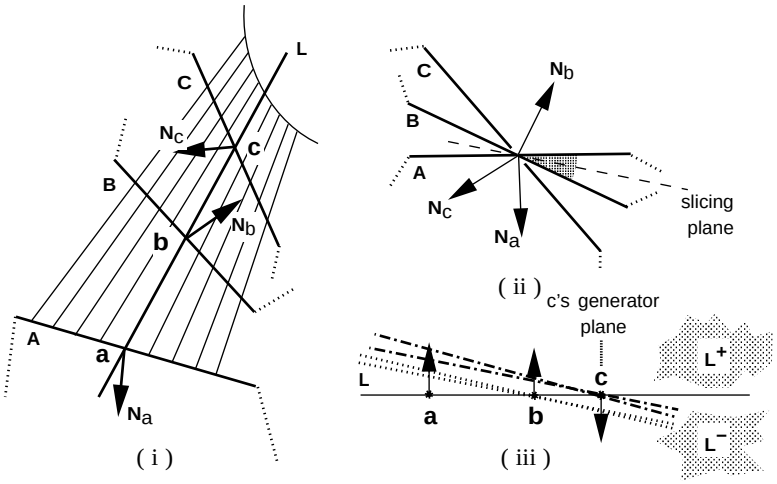


Figure 7.12: A boundary EEE swath (i), viewed along $\mathbf{L}$ (ii) and transverse to $\mathbf{L}$ (iii). The $\mathbf{N}_i$ cannot be contained, and the swath is therefore a boundary swath.

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

7.2.5 Vertex-Edge Swaths

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

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

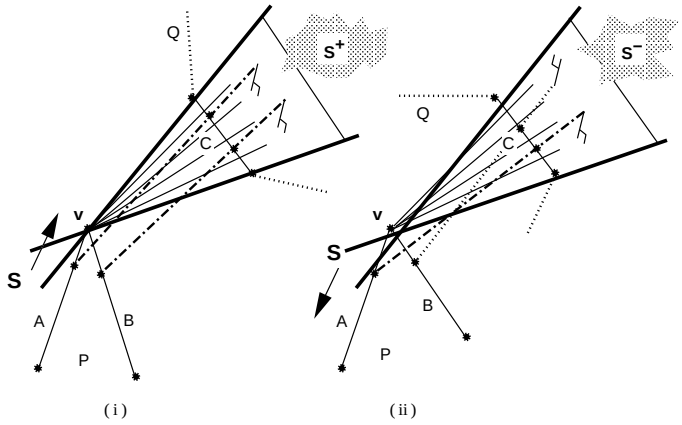


Figure 7.13: Boundary (i) and internal (ii) VE swaths.

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

on $\mathbf{v}$, it can only intersect $\mathbf{S}^+$. The swath admits only stabbing lines in $\mathbf{S}^+$, and is therefore a boundary swath.

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

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

7.2.6 The Containment Function

The containment function is a criterion for distinguishing between boundary and internal swaths. However, it applies only along a single stabbing line, not over an entire swath. Fortunately, evaluating the containment function anywhere along a swath produces the same result, since the basic configuration of the normals erected at the constraining portal edges remains the same. To see why this is so, consider any configuration of three coplanar vectors $\mathbf{N}_a$, $\mathbf{N}_b$, and $\mathbf{N}_c$. Suppose that the configuration changes continuously from containable to non-containable (e.g., by rotation of vector $\mathbf{N}_c$), as in Figure 7.14. At either moment of transition (marked with dotted lines in the figure), $\mathbf{N}_c$ and one of $\mathbf{N}_a$ or $\mathbf{N}_b$ must be antiparallel.

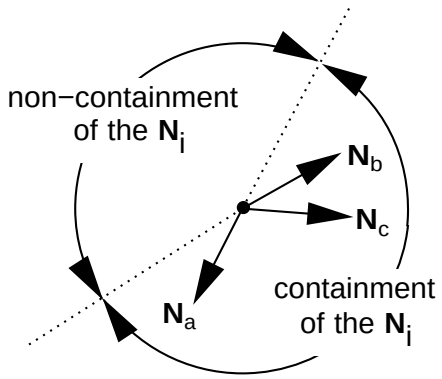


Figure 7.14: Directions of containment and non-containment, with moving $\mathbf{N}_c$, for fixed $\mathbf{N}_a$ and $\mathbf{N}_b$. Transition directions are marked.

For this to occur in the EEE swath construction (Figure 7.12) two of the three generator edges must be coplanar. Similarly, in the VE swath construction (Figure 7.13), edge C and one of the edges

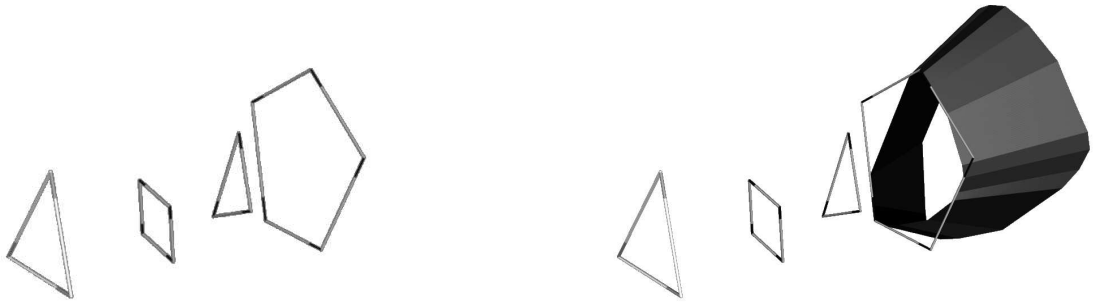


Figure 7.15: (i) The antipenumbra cast by a triangular light source through three convex portals ($n = 15$); (ii) VE boundary swaths (dark), EEE boundary swaths (light). (Figure 7.16 depicts the traces of the boundary swaths on a plane beyond that of the final hole.)

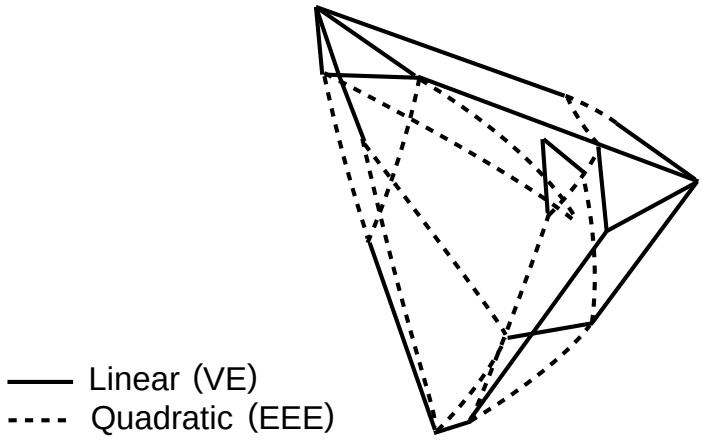


Figure 7.16: The internal swaths induced by the portal sequence of Figure 7.15, intersected with a plane beyond that of the final portal to yield linear and conic traces.

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

7.2.7 Computing the Antipenumbra

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

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

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

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

The algorithm can be described in pseudocode as:

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

convert directed edges to directed lines e_k

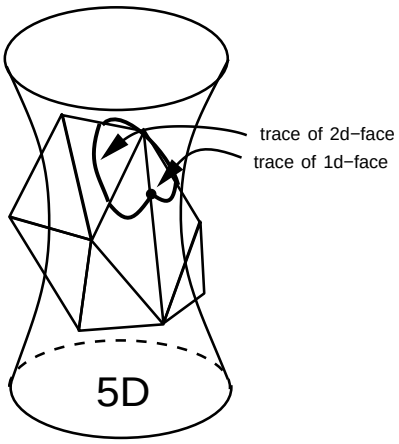


Figure 7.17: Loops in 5D line space. Each piecewise-conic path in 5D is isomorphic to the boundary of a connected component of the antipenumbra. One such loop is shown.

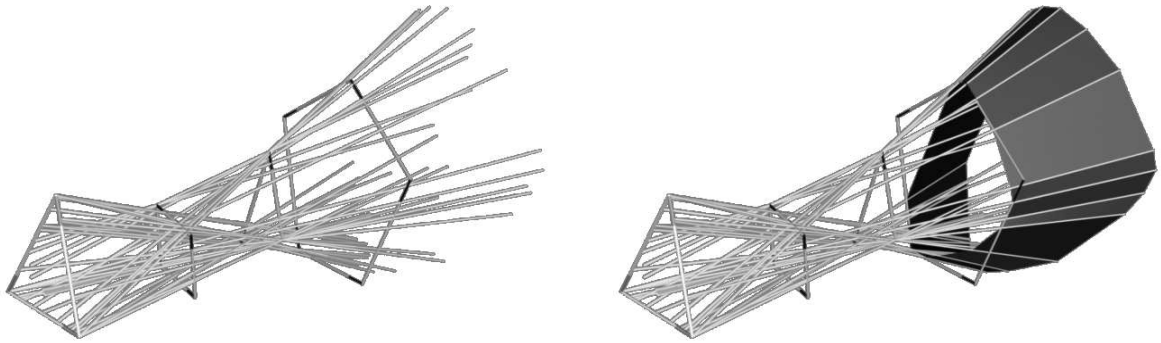
transform e_k to Plücker halfspaces $h_k = \Pi(e_k)$
compute 5D convex polytope $\bigcap_k h_k^+$ as winged-edge data structure
identify 2D-face intersections of $\bigcap_k h_k^+$ with Plücker surface
classify resulting extremal swaths as boundary or internal
    (i.e., mark 2D-faces in 5D with containment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

7.3 Implementation Issues

7.3.1 Current Implementation and Test Cases

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

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



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

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

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

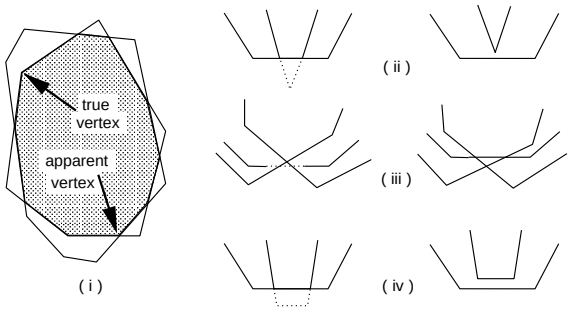
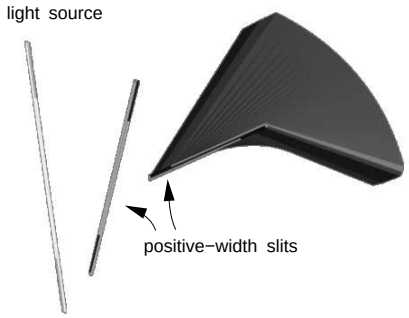
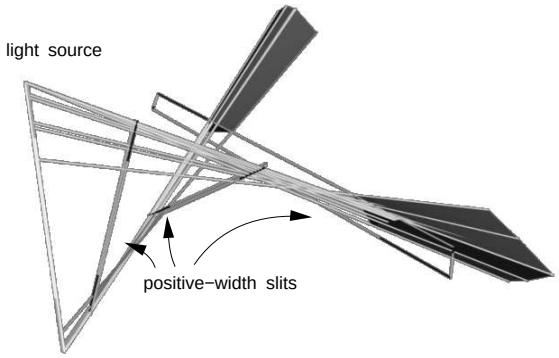


Figure 7.19: An observer's view of the light source (i). Crossing an extremal VE (ii), EEE (iii), or degenerate (iv) swath.



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



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

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

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

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

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

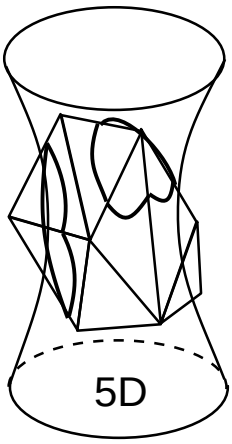


Figure 7.21: A disconnected antipenumbra boundary in 3D is isomorphic to a collection of loops in 5D line space.

7.3.2 Parametric Swath Representations

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

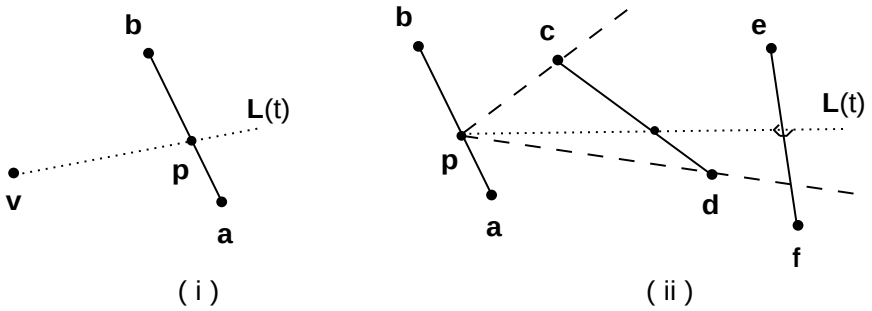


Figure 7.22: Parametrizing VE swaths (i) and EEE swaths (ii).

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

7.22-(ii)).

7.3.3 Implicit Swath Representations

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

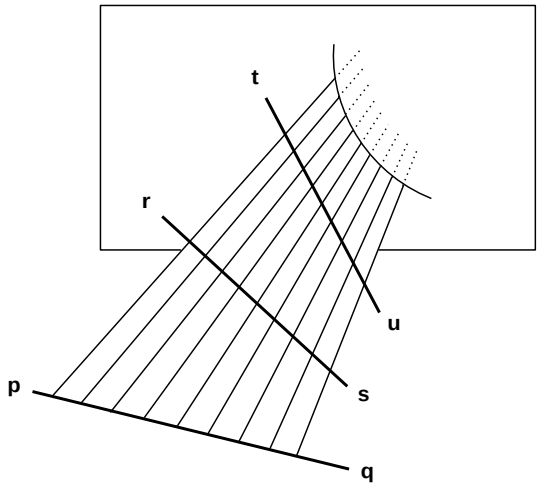


Figure 7.23: Three generator lines and part of the induced regulus of incident lines.

regulus through the lines is [Som59]:

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

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

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

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

or

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

where

$$A \quad = \quad \alpha_X \beta_X - \gamma_X \delta_X$$

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

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

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

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

7.4 Applications

7.4.1 Weak Visibility in Three Dimensions

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

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

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

7.4.2 Aspect Graphs

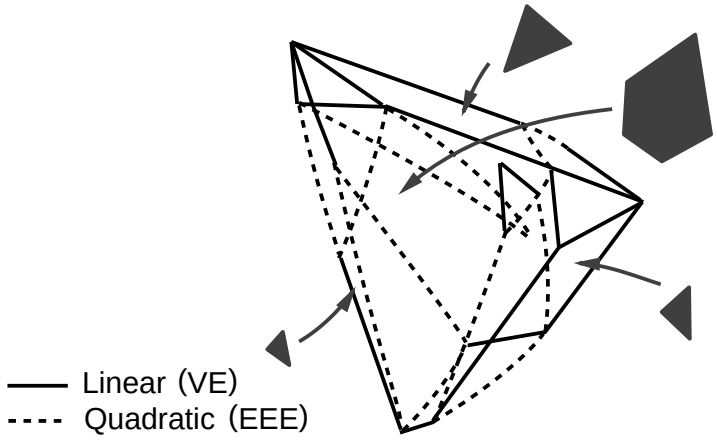


Figure 7.24: The aspect of the light source, as seen through the portal sequence.

Finally, we note that the computation of extremal swaths in the antipenumbra algorithm constitutes a complete *aspect graph* description of the light source (Figure 7.24. Outside the antipenumbra, the light source aspect is empty; inside the antipenumbra, event surfaces of the aspect are simply the internal VE and EEE swaths generated by the antipenumbra algorithm. Thus, for the specific case in which the input comprises a portal sequence, the aspect graph computation requires only $O(n^2)$ time, a substantial improvement over the $O(n^6 \lg n)$ general case.

Chapter 8

Three-Dimensional Polyhedral Environments

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

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

8.1 Major Occluders and Detail Objects

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

8.2 Spatial Subdivision

In 3D, the n occluders are finite, generally oriented convex polygons, and we desire a spatial subdivision whose cells are convex polyhedra, and whose portals are convex polygons. The BSP tree [FKN80] data

is a recursive subdivision of space based on splitting planes, and can be straightforwardly augmented to become a conforming spatial subdivision.

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

8.2.1 Splitting Criteria

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

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

8.2.2 Point Location

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

8.2.3 Object Population

Object population in BSP trees is more complex than for 2D or axial 3D spatial subdivisions. Since each cell is a convex polyhedron, detecting object incidence amounts to deciding whether the intersection of the cell with the object bounding box is non-empty. This is a 3D linear programming problem, and

can be solved for a particular cell in $O(s+b)$ time, where s is the number of planar faces bounding the polyhedral cell, and b is the (constant) number of faces comprising the object bounding box. These $s+b$ halfspaces are simply checked for a common intersection; if a more discriminating population method is desired, the convex hull of the detail object's vertices may be precomputed and stored with the detail object, to be intersected with BSP leaf cells as they are encountered and populated.

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

8.2.4 Neighbor Finding

Neighbor finding in BSP trees can be done robustly in an entirely two-dimensional fashion. Given a particular cell and boundary face, the BSP tree is ascended to find the cell that was split along the plane of the given boundary face. The appropriate subtree of this cell is then searched for all *leaf* cells with a boundary face affine to this plane (there may be no such cells). The matching boundaries of all cells so identified are then subjected to a two-dimensional intersection test on the shared boundary plane. Only cells with a 2D (i.e., superlineal, or positive area) intersection with the original boundary are marked as neighbors of the given cell.

8.2.5 Portal Enumeration

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

When spatial cells are general polyhedra, portals are planar non-convex regions formed by (convex) cell boundaries minus unions of opaque occluders on the boundary planes. Non-convex egresses can be handled by partitioning them into convex fragments. Alternatively, any collection of egresses can be coalesced into a single portal by performing a 2D convex hull computation on the egress vertices. This tactic, although increasing portal area above the minimum possible, can only increase the computed cell-to-cell visibility estimation, ensuring that it remains a superset of the actual visibility; on the other hand, it will reduce the number of portal sequences that must be examined.

Analogously to the two dimensional case of §4.2.4 (cf. Figure 4.4), a cell whose boundary has only a zero-D or 1D (i.e., point or lineal) intersection with an occluder need not take this occluder into account during portal enumeration, as the occluder can have no effect on the visibility of the generalized

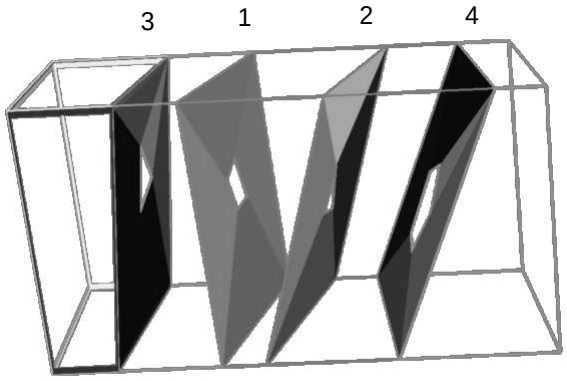


Figure 8.1: Five leaf cells of an augmented BSP tree, with convex portals. The four splitting planes are affine to the four coaffine occluder sets shown; they are numbered by the order in which they occurred.

observer in the cell. Both backfacing and frontfacing 2D incident occluders are relevant, and should not be ignored during portal enumeration.

8.3 Static Visibility Operations

8.3.1 Cell-to-Cell Visibility

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

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

8.3.2 Cell-to-Region Visibility

We wish to characterize the volume illuminated by a light source (the first portal in a sequence) within a cell that is reached by a line stabbing the sequence. Characterizing this volume for a sequence of a single portal is trivial: the antipenumbra (illuminated volume) in this case is the entire halfspace beyond the portal, intersected with the reached cell, which is, of course, an immediate neighbor of the source cell. By the convexity of cells, this intersection yields exactly the volume of the reached cell. Computing the antipenumbra of portal sequences with length two or more involves interactions among vertices and edges of different portals and, as we showed in Chapter 7, requires both linear and quadratic implicit primitives to correctly describe the illuminated volume.

8.3.3 Representing the Cell-to-Region Visibility

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

8.3.4 Cell-to-Object Visibility

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

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

Computing exact box incidence with the interior halfspace of a VE swath is straightforward. Computing box incidence with a quadratic halfspace is a more interesting problem. It is an instance of quadratic programming; we wish to determine if the (linear) box interior and the (quadratic) regulus interior have any common volume. Invoking a fully general quadratic programming algorithm is unnecessary, however, given the special nature of the problem. First, we have a constant number of constraints.

Second, the quadratic constraint is of a particular class, eliminating a class of intersection events *a priori* that a fully general algorithm would have to treat explicitly.

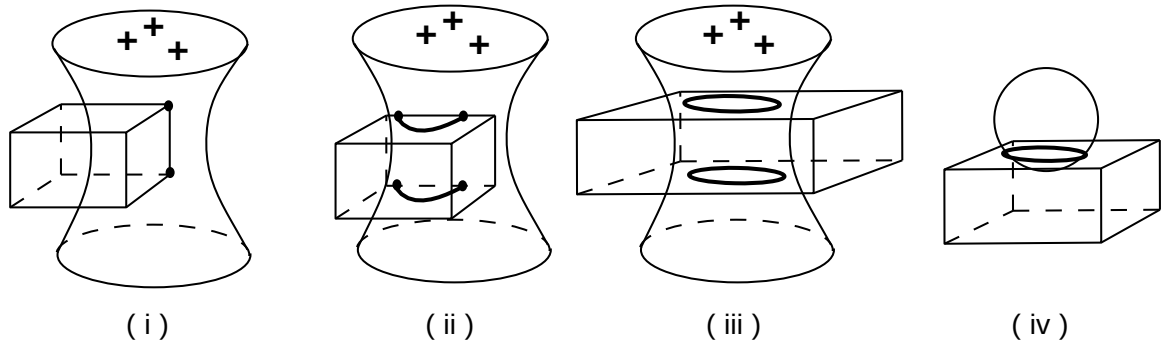


Figure 8.2: A convex polyhedron and a regulus can intersect in only three ways.

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

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

8.3.5 Conservatively Approximating the Antipenumbra

In practice, the antipenumbra and cell-to-object computations are difficult to implement robustly. They depend on high-dimensional convex-hull computations, singular value decompositions of matrices, and robust root finding, all of which have specific and sometimes algorithmically idiosyncratic dependencies on numerical tolerances and geometric degeneracies. Consequently, we have developed an algorithm that bounds the antipenumbra of a light source with a convex polyhedron, using only 3D linear operations such as point-plane and vector-vector inner products. This algorithm is a straightforward assembly of separating tangent planes, which can be thought of as an "internal giftwrap" over the edges of the portal

sequence. Asymptotically, this algorithm remains $O(n^2)$ in complexity for general portals with n total edges; however, in practice it runs one to two orders of magnitude faster than the exact antipenumbra algorithm above for portal sequences of a few tens of edges.

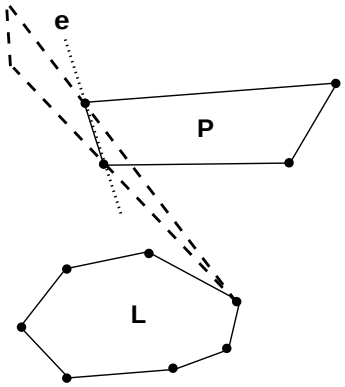


Figure 8.3: The plane through edge **e**, separating portals **P** and **L**.

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

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

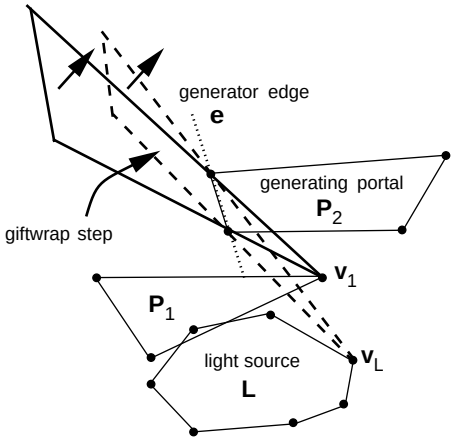


Figure 8.4: A giftwrap step on edge $\mathbf{e}$, from $\mathbf{v}_1$ to $\mathbf{v}_L$.

8.4 Dynamic Visibility Queries

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

8.4.1 Observer View Variables

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

8.4.2 Eye-to-Cell Visibility

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

Computing eye-to-cell visibility consequently reduces to the *existence* problem of determining eye-centered stabbing lines through 3D portal sequences. A list of normal vectors is initialized to the four normals of the frustum bounding planes, each oriented toward the interior of the frustum (Figure 8.6). As each stab tree portal is encountered, it is traversed in a specific direction. Each portal edge is oriented by the traversal; the portal edge and the eye span a plane, oriented so that its positive halfspace contains

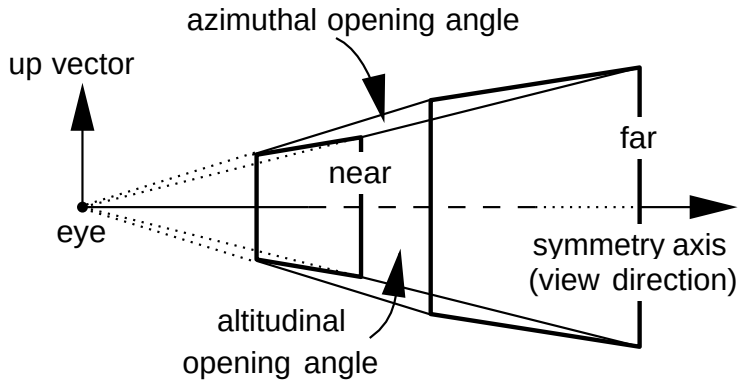


Figure 8.5: Three-dimensional observer view variables.

the interior of the portal. The plane normal is concatenated to a list, and linear programming is used to determine whether there exists a suitable stabbing line, i.e., a vector with a positive inner product with each of the constraint vectors.

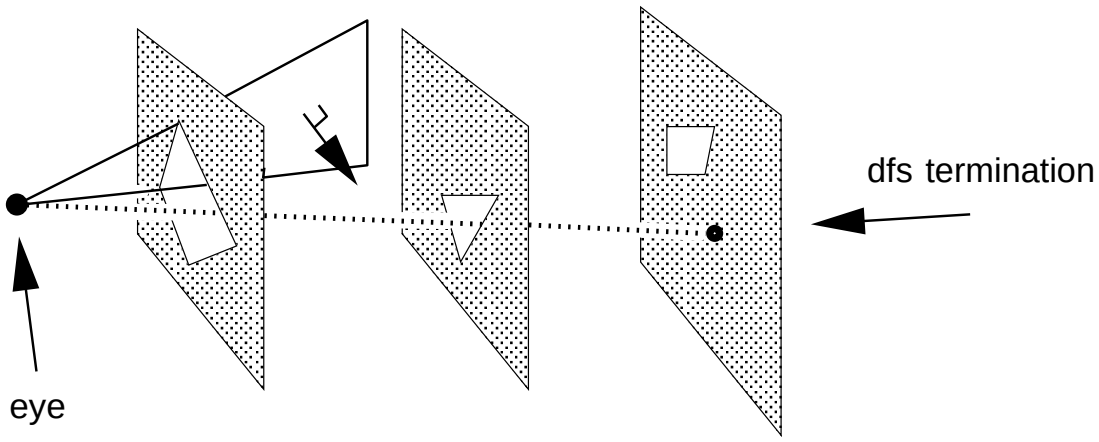


Figure 8.6: Encountering a portal in the eye-to-cell DFS.

These constraints are cast as a two-dimensional linear program as follows. If the k^{th} plane equation has normal $\mathbf{n}_k$, any stabbing line through the eye must have a direction vector $\mathbf{v}$ such that

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

Note that the linear program is two-dimensional since all of the planes contain the eyepoint; therefore, we need only compute a *vector* that has a non-negative dot product with each of the plane normals. We

examine this collection of three-coefficient linear constraints for a feasible solution in linear time using a linear programming algorithm. If the linear program fails to find a stabbing line through the eye, the most recent portal is impassable, the reached cell is not visible through the current portal sequence, and the active branch of the stab tree DFS terminates.

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

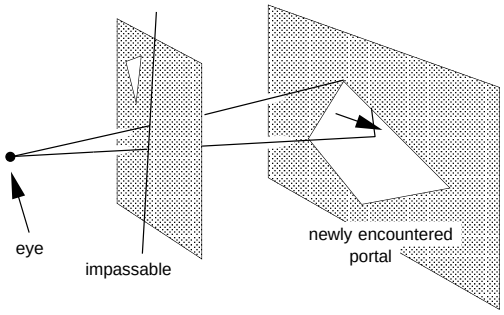


Figure 8.7: Detecting an impassable portal edge constraint.

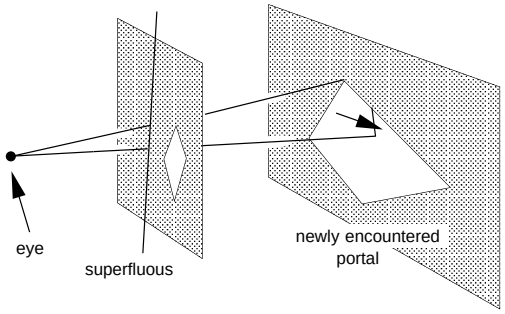


Figure 8.8: Detecting a superfluous portal edge constraint.

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

Recall Jones' 1971 visibility algorithm [Jon71], which traversed a spatial subdivision adjacency graph, projecting portals onto the view plane and solving the generalized clipping problem for each portal sequence (cf. Figure 2.1). Our algorithm is significantly different from that of Jones. First, we construct the spatial subdivision and portals automatically as a function of the occluders, rather than by hand. Second, we need not ensure that every occluder and detail object polygon end up on the boundary of a cell. Any particular entity can be ignored (treated as invisible); doing so will make our algorithms produce larger potentially visible sets, but not incorrect answers. Third, our off-line (*cell-to-cell*) stabbing line computation typically causes the eye-to-cell visibility computation to terminate sooner than Jones' query, since the eye-to-cell query will not examine portal sequences that are known *a priori* not to admit sightlines. This allows predictive visibility computations, i.e., knowledge that particular cells or objects cannot become visible within specified time intervals. Fourth, our algorithm produces an *unordered* set of potentially visible polygons, and assumes the existence of a depth-buffer

to perform hidden-surface elimination. Finally, we use linear programming to compute the *existence* of the intersection of a collection of convex polygons, not the actual intersection (i.e., a convex polygon or the empty set), and therefore compute a superset of the visible objects, not an unnecessarily exact representation of their visible fragments. Consequently, for portal sequences with n edges, our approach has $O(n)$ complexity compared to the $O(n \lg n)$ complexity of Jones' approach. Indeed, since Jones made no distinction between occluders and objects, his algorithm as originally stated would not be applicable to any but the simplest environments in practice. We conclude that the eye-to-cell algorithm presented here is more storage-efficient and substantially faster than that of Jones, in both theory and practice.

Degeneracy

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

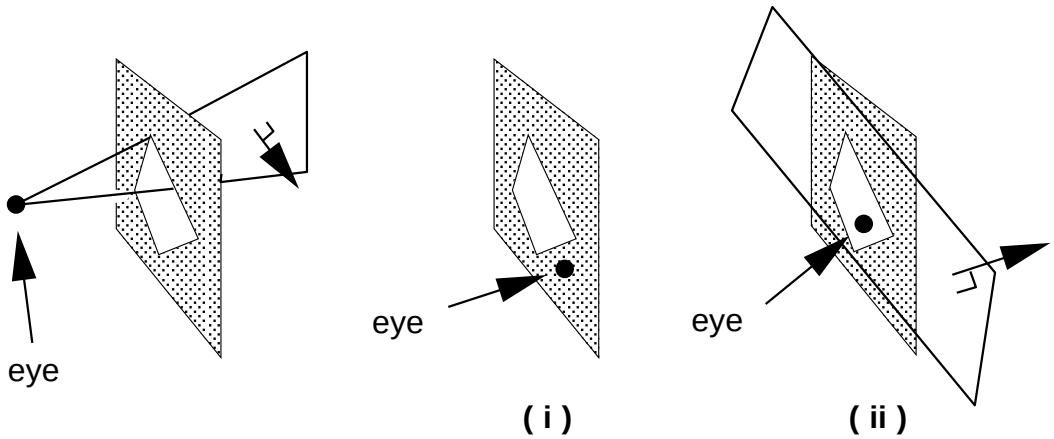


Figure 8.9: Handling degeneracies in the eye-to-cell DFS.

8.4.3 Eye-to-Region Visibility

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

8.4.4 Eye-to-Object Visibility

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

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

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