TEL-AVIV UNIVERSITY

The Raymond and Beverly Sackler Faculty of Exact Sciences

School of Computer Science

RAY SPACE FACTORIZATION FOR FROM-REGION VISIBILITY

Submitted in partial fulfillment of the requirements

for the degree of “master of science”

By

TOMMER LEYVAND

This work was supervised by

Professor DANIEL COHEN-OR

Submitted to the senate of Tel-Aviv University

June 2005

Abstract

In this thesis we present a conservative, from-region, occlusion culling method based on factorizing the 4D visibility problem into horizontal and vertical components. The visibility of the two components is solved asymmetrically. The horizontal component is based on parameterization of the ray space using two concentric squares. The visibility of the vertical component is solved by incrementally merging umbrae within vertical slices. Based on the horizontal parameterization we decompose the difficult from-region visibility problem in 3D into many simpler from-region problems in 2D vertical slices. The technique is designed so that the horizontal and vertical operations can be efficiently realized together and accelerated by modern graphics hardware. Similar to image-based from-point methods, we use an occlusion map to encode visibility; however, the image-space occlusion map is in the ray space rather than in the primal space. This map maintains the conservative aggregated umbra created by the occluders processed so far. Our results show that the culling time and the size of the computed potentially visible set depend on the size of the viewcell. For moderate viewcells for walk-through applications, conservative occlusion culling of large urban scenes takes less than a second, and the size of the potentially visible set is only about two times larger than the size of the exact visible set.

This work was published in *ACM Transactions on Graphics*[LSCO03].

Contents

List of Figures

Chapter 1

Introduction

Modem graphics workstations are capable of rendering virtual scenes with a few million polygons at interactive frame-rates. Yet, for scenes describing large virtual environments (such as urban scenes) or containing extremely detailed structures, rendering in real time remains a challenge.

In many such complex virtual scenes the visible part of the model from various viewpoints is quite small with respect to the full model (see Figure 1.1). Graphics hardware is capable of removing the hidden parts by applying z-buffer or similar techniques. However, such techniques require ”feeding” the hardware with huge amounts of geometry that is rendered, colored and lighted, where ultimately most of it turns out to be occluded and is discarded. As expected this is inefficient and better solutions are available.

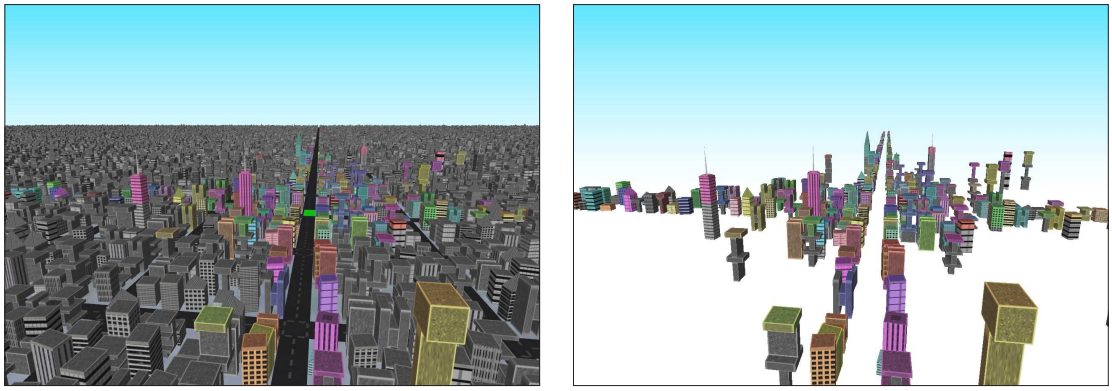


Figure 1.1: A view of a large urban model consisting of 26.8M triangles. In the left image, the parts visible from a region located at the junction of two streets (in green) are colored. In the right image, only the buildings with some visible parts are displayed.

Visibility culling is the process of efficiently identifying the *Visibility Set* (VS), the set of objects that are visible and should be rendered, thereby significantly reducing the size of the model required for rendering. Conservative solutions gain speedup by overestimating the VS and identifying the *Potential Visibility Set* (PVS), the set of objects that might be visible ($VS \in PVS$). Clearly, the hidden parts depend on the position of the viewer, the view frustum and the actual model.

Methods that compute the visibility from a point are necessarily applied in each frame during rendering [GKM93, CT96, ZMHH97, HMC+97, BHS98]. Recently, based on earlier methods [ARB90, TS91, Tel92a, FST92], more attention is devoted to *from-region* methods where the computed visibility is valid for a region rather than a single point [COFHZ98, SVNB99, GSF99, SDDS00, DDTP00, WWS00]. These methods take advantage of time and spatial coherence, and the computational cost of the visibility calculations is amortized over consecutive frames. Still, it is desirable to be able to compute from-region visibility on-the-fly, not having to resort to off-line methods that require excessive storage space.

The from-region visibility problem is considered significantly harder than the from-point visibility one. To decide whether an object S is (partially) visible or occluded from a region C requires detecting whether there exists at least a single ray that leaves C and intersects S before it intersects an occluder. This is inherently a 4-dimensional problem [Tel92b, Dur99]. Although exact solutions are possible [BP01, NBG02], they are overly expensive. Some methods have been restricted to dealing with 2.5D scenes [KCOC01, BWW01, WWS01]. The assumption of 2.5D occluders is quite reasonable in practice, especially in walkthrough applications where architectural models can be well approximated conservatively by 2.5D shapes. However, for more general scenes, it is necessary to have a culling method that can correctly deal with the occlusion of a larger domain of shapes.

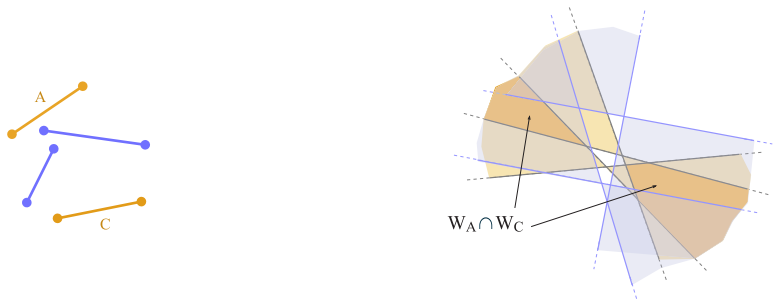
1.1 Ray-space techniques

Visibility problems and in particular occlusion problems are often expressed as problems in line spaces. Consider for example, the following basic occlusion problem in 2D. Given a line segment C, an object A is occluded from any point on C by a set of objects B_i, if all the lines intersecting C and A also intersect the union of B_i. Using the line space, lines in the primal space are mapped to

points. To tell whether the union of B_i is blocking all the rays that leave C towards A can be solved by simple Boolean operation in line space.

The mapping between 2D lines and points is commonly defined by the coefficients of the lines in the primal space. The line $y = ax + b$ is mapped to the point $(a, -b)$ in the line parameter space. All the lines that intersect a segment A in the primal space are mapped to a double wedge in the parameter space, and we call this the *footprint* of A. All the lines intersecting the union of segments B_i are mapped to the union of their footprints. All the lines passing through two given segments A and C are the intersection of their footprints.

Now, the above occlusion problem can be expressed as a simple Boolean set operation of the footprints (see Figure 1.2). One nice property of this simple idea is that it yields an exact solution, and considers all types of possible occlusions. However, since an exact solution is not an important property, one can use a discrete space, and get a conservative solution. Still, as we shall see, this conservative solution captures all types of occlusions, but slightly overestimates the visibility due to the conservative discretization.



(a) (b)

Figure 1.2: Boolean operations in dual space can be used to determine visibility between two line segments. In (a), the orange segments are mutually occluded by the blue segments. In (b), Boolean set operations are applied to the double-wedge footprints to test visibility. $W_A \cap W_C$ (in dark orange) represents all lines passing through both A and C; thus, A and C are mutually hidden if and only if $W_A \cap W_C$ is a subset of the union of occluder footprints (in blue).

In 2D the above (double wedge) footprints can be discretized and drawn as polygons, and their union and intersections can be applied in image-space, taking advantage of fast per-pixel Boolean operations preformed by common graphics hardware. Although it seems simple at first, the above solution has a number of problems that prevent a straightforward implementation.

First, the above-mentioned choice of mapping is not optimal as the line space is unbounded. This causes serious problems for all lines with large coefficients, because for practical reasons the line space must be bounded.

Second, a direct extension to the 3D case is not as easy as in the 2D case since the parameter space of 3D lines has high dimensionality. In general, 3D lines have four degrees of freedom. However, since we are usually interested in an asymmetric visibility problem, another dimension is required to specify the origin of the lines, which are thus regarded as *rays*. But the visibility in 2D alone is not of much interest. It might be the case that most of the objects are detected as occluded in flatland, while actually being visible in 2.5D due to their various heights.

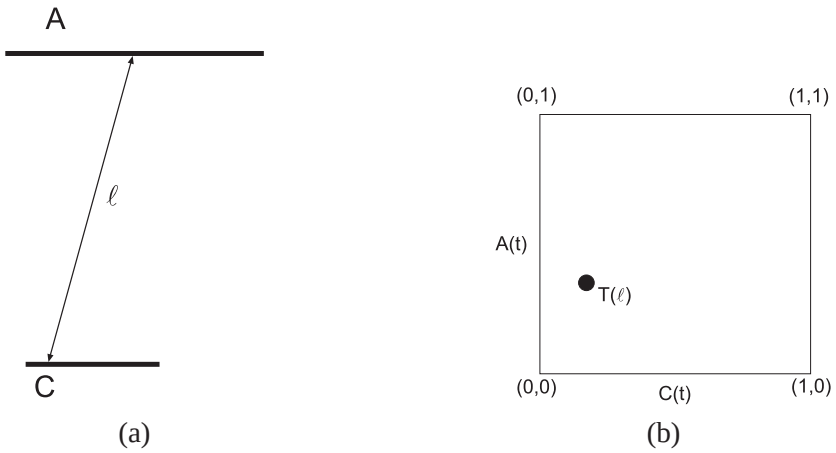
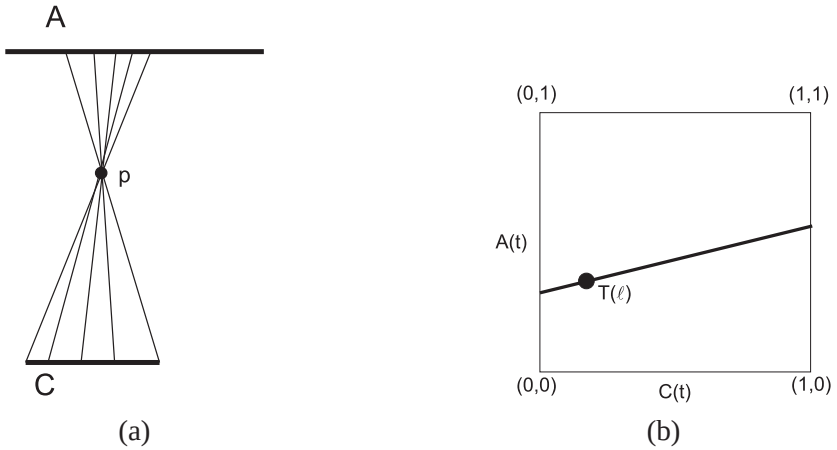


Figure 1.3: A ray l in the primal space is mapped to a point $T(l)$ is the ray space.

1.1.1 Visibility in Line space

To better understand how visibility can be solved in line space, let us consider a simple case where one tests the visibility between two given parallel segments C and A. The segment C represents the viewing region, and A the object in question for which we test whether A is visible or not by a collection of occluding segments O. For that simple case we can use the parameterization introduced by [KCOC01].

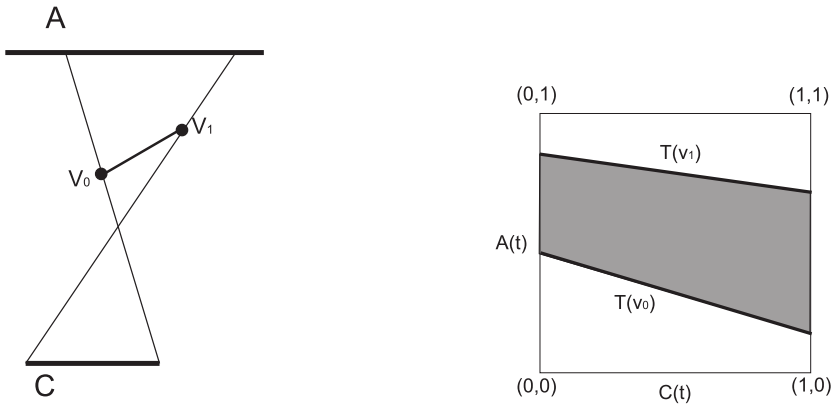
We define a bounded two-dimensional ray space, such that every ray originating on C and intersecting A corresponds to a point in this space. The visibility is determined by "marking"all points in the ray space that represent rays that pass through occluding segments. Visibility is then



(a) (b)

Figure 1.4: All the rays passing through a point p are mapped to a line in the ray space, which in particular includes $T(l)$.

detected by checking whether there is at least one point that has not been "marked".



(a) (b)

Figure 1.5: Rays passing through a segment are mapped to a trapezoid.

More precisely, parameterize C and A as $\{C(t)|\,0 \le t \le 1\}$ and $\{A(t)|\,0 \le t \le 1\}$, respectively. Let $\mathbb{U}$ be the unit square $\{(x,y)|\,0 \le x,y \le 1\}$, such that a point (x,y) in $\mathcal{U}$ corresponds to the ray originating at $C(x)$ and passing through $A(y)$ (see Figure 1.3).

Define a mapping $\mathcal{T}:\mathbb{R}^2 \to \mathcal{U}$ that maps each point $p \in \mathbb{R}^2$ to the collection of points in $\mathcal{U}$ that correspond to rays passing through p. For any $p \in \mathbb{R}^2$, $\mathcal{T}(p)$ is a line segment in $\mathcal{U}$ (see Figure 1.4) . It should be noted that this property is valid only provided that A and C are parallel lines (see [LSCO03]).

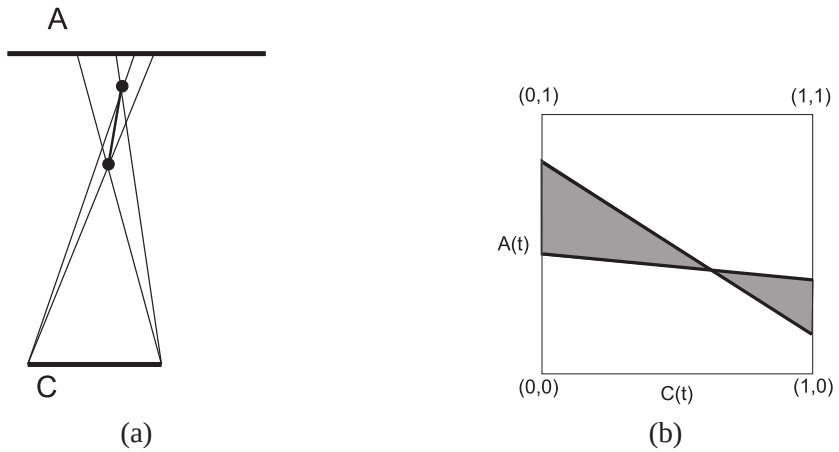


Figure 1.6: Rays passing through a segment are mapped to a double-wedge.

For a segment $v \in O$, parameterized as $\{v(t) | 0 \leq t \leq 1\}$, $\mathcal{T}(v)$ is defined to be the continuous collection of segments $\{\mathcal{T}(v(t)) | 0 \leq t \leq 1\}$. This collection is bounded by the segments $\mathcal{T}(v(0))$ and $\mathcal{T}(v(1))$ that correspond to the end-points of v. In general, it forms either a trapeze (Figure 1.5) or a double-triangle (Figure 1.6), depending on whether the line containing v intersects the interior of both parallel segments A and C. As mentioned earlier the mapping of a segment in primal space is called the *footprint* of the segment.

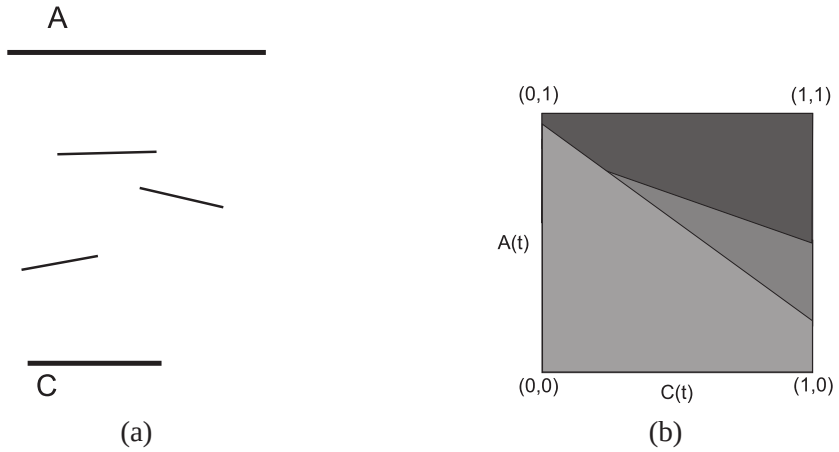


Figure 1.7: The footprints of the three segments cover the entire ray space, implying that A is hidden from C.

This implies a simple exact algorithm for determining whether A is visible from C: for each segment $v \in O$ create its footprint in $\mathcal{U}$ and compute the union of these footprints (i.e. $\bigcup_{v \in O} \mathcal{T}(v)$).

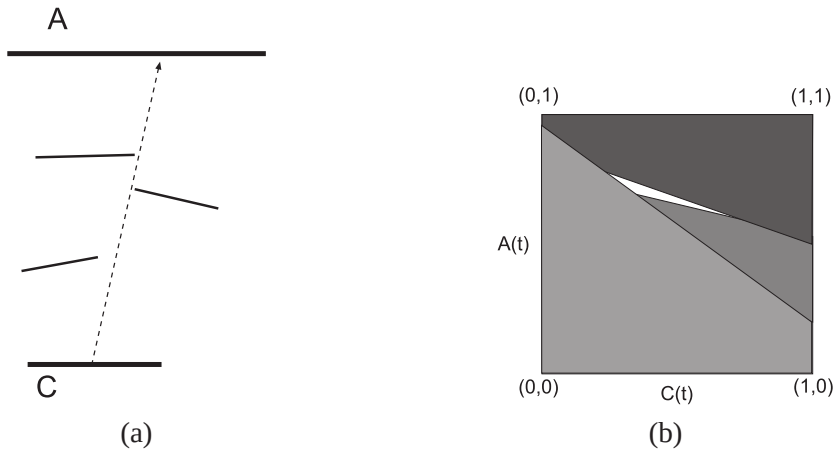


Figure 1.8: The footprints of the three segments do not cover the entire ray space, implying that A is visible from some point in C.

This computation can be performed in worst-case optimal $O(n^2)$ time without employing complex data structures [dBvKOS97]. If C and A are mutually visible, there is a point $p_o \in \mathcal{U}$ that is not contained in this union. The point p_o corresponds to a visibility ray that intersect both C and A (see Figures 1.7and 1.8).

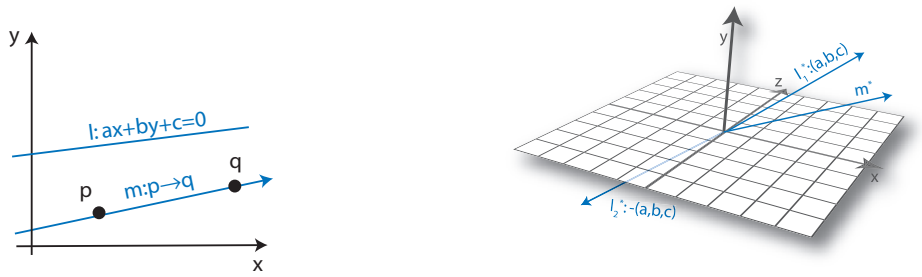
This mapping defines a bounded ray space as opposed to the infinite line space of the above-mentioned transforms. This is a crucial advantage that permits the discretization of the ray space, and the rasterization of the footprints by by graphics cards.

However, this ray space is not a global one, but a view-dependent ray space. It requires the construction of a dedicated ray space to determine the visibility of each individual object A. Also this ray space is valid only for the portion of the segments that intersect the shaft defined by C and A. This requires clipping, that is, to first compute the portion of the segments that intersect the shaft among all the candidate occluding segments in the scene. In contrast, a global ray space allows the construction of a single occlusion maps by mapping all the occluders and then using this map in ray space to test the visibility of many objects. In the following section we will study such a global ray space.

1.2 Plücker coordinates

The simple mapping of lines $y = ax + b$ in primal space to points $(a, -b)$ in line space defines an unbounded line space. Instead of the explicit line representation, we can use an implicit equation for a line l. Let $l : ax+by+c = 0$. Now, the line l is mapped to a homogeneous line space, where l is mapped to the points $l_1^* = (a,b,c)$ and $l_2^* = -(a,b,c)$, where l_1^* and l_2^* represent two opposite oriented lines. If we are given the oriented line by two ordered points $l : p \rightarrow q$ then $l^* = (p_y - q_y, p_x - q_x, p_x q_y - p_y q_q)$ or the minors of $\begin{pmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \end{pmatrix}$.

Normalizing the coefficients will introduce singularities. Instead, the homogenous coordinates are kept as points in a 3D linear space [BP01]. This means that all the points along a half-line intersecting the origin, represent the same line in the primal space. In other words, an oriented line in 2D is mapped to a half-line emerging from the origin in the 3D line space (see Figure 1.9).



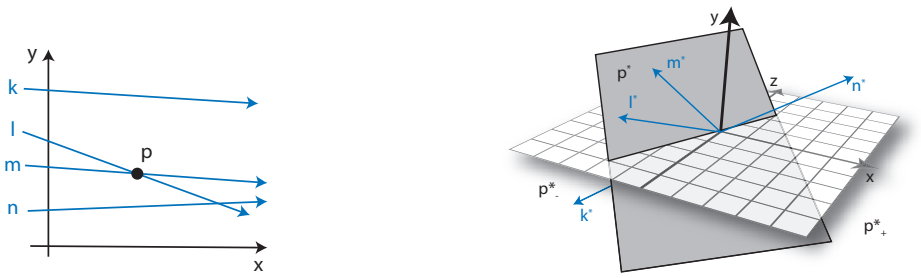
(a) (b)

Figure 1.9: The implicit equation for line l is mapped to two opposite oriented lines, l_1^* and l_2^*. The oriented line m is mapped to the single oriented line m^* in the ray space.

All the oriented lines intersecting a point p map to a plane $p^* = \{(x,y,z)|p_x x + p_y y + z = 0\}$, subdividing the line space into two half spaces, p_+^* and p_-^*. Oriented lines that see p from the left (counterclockwise around p) reside in p_+^*, whereas oriented lines that see p from the right reside in p_-^* (see Figure 1.10).

1.2.1 Footprint of a line segment

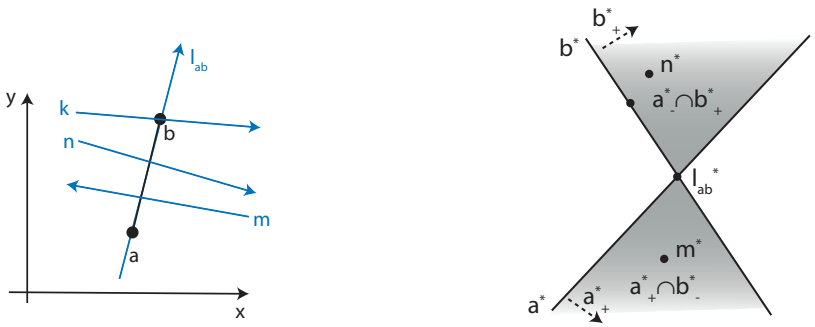
All the oriented lines that intersect a lines segment $\overline{ab}$ map to its footprint $\overline{ab}^*$, a 3D volumetric double-wedge (hourglass) in the line space. We denote this footprint by $F(\overline{ab})$. To understand how



(a) (b)

Figure 1.10: The mapping of the point p and the four oriented lines k,l,m and p to the Plücker ray-space. The mapping of p is the plane p^* which coincides with l^* and m^*. The oriented line k sees p at its right, thus, it maps to the p^*_- half-space. Similarly, n maps to p^*_+ half-space.

the footprint is built we need to examine the mapping of both end-points to line space (see Figure 1.11). The points a and b map to the planes a^* and b^* and the supporting line l^*_{ab} is mapped to their intersection. Oriented lines that intersect $\overline{ab}$ can either see point a at their right and point b at their left (such as the oriented line n) and thus $n \in \{a^*_- \cap b^*_+\}$. Similarly, $m \in \{a^*_+ \cap b^*_-\}$.



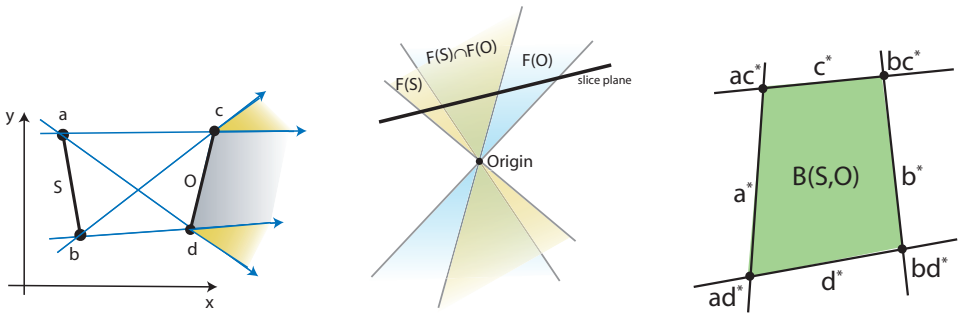
(a) (b)

Figure 1.11: (a) A line segment $\overline{ab}$ and some oriented lines that intersect it. (b) A slice view of the resulting 3D Plücker ray space.

1.2.2 Oriented lines intersecting two line segments

The footprint of the set of lines that intersect two disjoint line segments S and O is the intersection of their 3D footprints $F(S)$ and $F(O)$. The result of this intersection are two pyramids with a joint apex at the origin (see Figure 1.12a,b). The footprint of the ordered set of lines that first intersect S and then intersect O is a single pyramid. This pyramid, $P(S,O)$ is represented by its *blocker polygon*

$B(S,O)$, whose vertices lie on the boundary half-lines of the pyramid (see Figure 1.12c). Notice that the blocker polygon need not be planar since its vertices only represent the induced pyramid. The vertices can be normalized to the unit sphere centered at the origin of line-space.



(a) (b) (c)

Figure 1.12: (a) Two line segments S and O in primal space. The supporting lines $\overline{ac}$ and $\overline{bd}$ construct the umbra cast by O from S (in cyan). The separating lines $\overline{ad}$ and $\overline{bc}$ construct the partial visible area from S behind O (in yellow). (b) The intersection of the two footprints $F(S)$ and $F(O)$ in ray space represents all the oriented lines that pass through both S and O. (c) A different slice-view from the slicing plane in (b) showing the blocker polygon $B(S,O)$.

1.3 Visibility Preprocessing for Urban Scenes using Line Space Subdivision

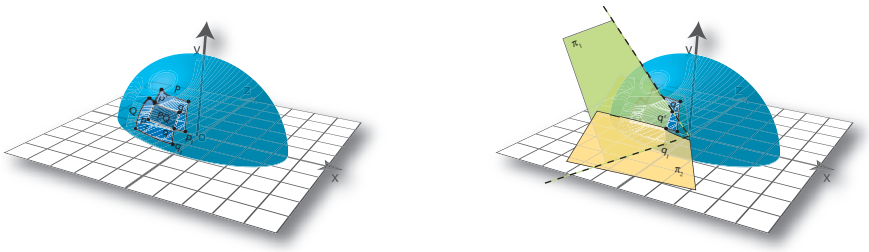
In [BWW01] they present a method for conservatively preprocessing the visibility in a 2.5D scene, where occluders are limited to vertical trapezoids such as building facades and the viewcell is a polyhedron made up of several vertical faces. This allows the algorithm to mainly operate on the top view 2D orthographic projection of the scene. The actual heights are considered when necessary.

The algorithm hierarchically traverses the scene using a kd-tree like partitioning, maintaining the accumulated occlusion in line-space. Each kd-cell, in a top-down approximate front-to-back order, is tested against the current occlusion in line-space. A non-leaf cell that is visible is recursively tested, a visible leaf cell contributes its contained vertical trapezoids as occluders, augmenting the occlusion. The above mentioned hierarchical framework is based on two operations: viewcell to kd-cell visibility test and aggregating occlusion in line-space. Since the viewcell and the kd-cell are both composed of vertical trapezoids, these operations reduce to the following operations between

vertical trapezoids: maintaining the visibility information in line space cast by a set of vertical occluders, testing the visibility between a face of the viewcell and a face of a kd-tree cell and aggregating occlusion of newly found to be visible vertical trapezoids.

1.3.1 Maintaining the occlusion information in ray space

To capture the occlusion information from a viewcell face S and a set of previously identified occluders $\{O_i\}$, we maintain the set of *blocker polygons* $\bigcup B_i(S,O_i)$. For reasons that will be clear later, we maintain this union as a series of disjoint polygonal cells, forming a subdivision of the ray space. Notice, that since these polygons do not reside on the same plane, their intersection can be performed either on the unit sphere or simply by pair-wise intersecting edges in accordance to their induced plane with the origin (see Figure 1.13). Each cell C in the projective ray space is associated with a list of blocker polygons B_{C_i} (see Figure 1.14), arranged in the order they are seen. Meaning, for every point $l^* \in C$, the associated ray l in primal space intersects the exact set of occluders O_{C_i} in the specified order. The subdivision of the ray space is built incrementally, separately for all the faces of the viewcell. For each occluder O (and a viewcell face S) we compute the blocker polygon $P(S,O)$ and identify the cells already in the ray space subdivision that intersect P. For each such cell C we test the visibility of O in primary space with respect to the set of occluders within C (this will be explained in the following section). If O turns out to be visible then it is either also associated with C or C is subdivided, otherwise O is hidden by all rays in C and thus can be ignored.



(a) (b)

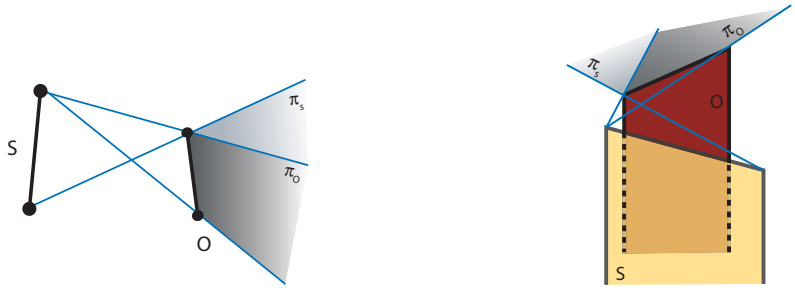
Figure 1.13: (a) Two blocker polygons P and Q and their intersection PQ on the unit sphere. (b) Calculating the intersection point q' of $\overline{q_1q_2}$ and $\overline{p_1p_2}$ by intersecting their associated planes π_1 and π_2, also passing through the origin, and normalizing back to the unit sphere.

(a) (b)

Figure 1.14: (a) The projection of a viewcell face S and three occluder O_1,O_2 and O_3 onto the ground plane. The non-occluded funnels from S are denoted by E_A, E_B and E_C. (b) A slice-view of the plücker ray space subdivision induced by the occluders. Note the ordering of the occluders in the relevant cells.

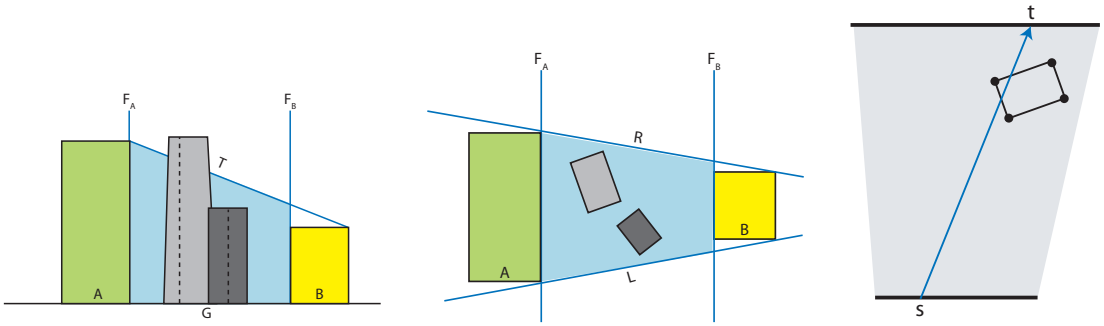
1.3.2 Testing visibility using the ray space

An important aspect of this algorithm is determining visibility from a viewcell face S to an object O (either a potential occluder or one of the faces of some kd-cell). Initially, the blocker polygon $P(S,O)$ is constructed and segmented into disjoint subdivision cells Q_i^*, according to the existing line space subdivision. Separately, with the ray space reduced to such a cell Q^*, we solve the visibility of O from S. For each $O_i \in Q^*$, build two shadow planes, π_1 and π_2, against which clip the top edge of O (see Figure 1.15). Due to concavity, this results in at most one visible fragment. If there is indeed such a fragment (O is not completely occluded) then it's end-points are mapped to an edge in line space which is used to sub-divide the cell Q^*, adding O to the occluder lists in the appropriate cells.



(a) (b)

Figure 1.15: The two shadow planes π_O and π_S. The first plane, π_O is constructed by connecting the top edge of O with a vertex of the top edge of S. The second plane, π_S is constructed by connecting the top edge of S with a vertex of the top edge of O. In (a) a top view and in (b) a front view.



(a) side view (b) top view (c) slice view

Figure 1.16: The shaft (light cyan) between two parallel boxes A and B is defined between the six supporting planes: T (top) and G (ground), L and R (left and right) and F_A and F_B (induced by the relevant faces of the kd-cell and the viewcell). The slice view (c) shows the 2D visibility within the plane, parameterizing the rays source and target with two parameters s and t.

1.4 Hardware-accelerated from-region visibility using a dual ray space

In [KCOC01] they introduce a new approach for solving 2.5D visibility using the 2D ray-space between parallel line segments, as described in 1.1.1. Similarly to other hierarchical frameworks, a kd-hierarchy of the entire scene is used to accelerate the culling algorithm. Occluders must be 2.5D primitives, but not limited to vertical trapezoids as in [BWW01]. The viewcell however, must be a box aligned with the kd-tree axes.

The key-observation behind this work is that the visibility between two aligned boxes (the viewcell and some kd-tree cell) in a 2.5D world reduces to a 2D visibility problem on the plane through the roofs (see Figure 1.15a).

Their framework works as follows. First a kd-tree hierarchy is built for the scene. The culling algorithm traverses the tree top-down, testing the box to box visibility (viewcell to kd-cell). A visible non-leaf kd-cell is further traversed, a visible leaf cell is added to the PVS whereas an occluded kd-cell is culled along with its sub-hierarchy.

1.4.1 Visibility between two aligned boxes

In a 2.5D world, two axis aligned boxes A and B are mutually visible if and only if there upper rims (roof-tops) are. The set of occluders reduces to the *shaft*, the volume between the six supporting planes (see Figure 1.16a,b).

The visibility test between A and B is performed as follows. First, the occluders in the shaft are intersected with the top visibility plane. Occluders that are shorter than this plane do not occlude and are omitted. The intersections produces polygons on the top plane which are considered as individual line segment (see Figure 1.16c). Using the parameterization described in 1.1.1, each line segment induces a simple polygonal footprint. Each footprint is rastered conservatively to the frame-buffer using the graphics hardware (see Figure 1.17). If the entire ray space is not fully covered by these footprints then A and B are mutually visible (see Figures 1.7 and 1.7). Testing for full coverage can also be performed in hardware by using the occlusion query extension.

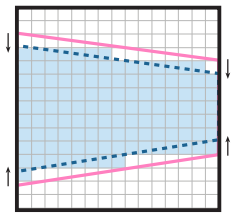


Figure 1.17: The footprint of an occluder is conservatively rasterized by coloring only fully covered pixels.

1.5 Occluder fusion

There are two main approaches for aggregating umbrae. The first approach uses ray-parameterization to capture the visibility information of an object into some geometric region (usually a polygonal footprint) and apply Boolean operations to determine visibility [KCOC01, BP01]. The second approach fuses individual intersecting umbrae into large umbrae [DDTP00, SDDS00].

Ray space techniques, as described in 1.1, are well suited for aggregating umbrae, yet are difficult to extend to 3D due to the high dimensionality of the parameter space (5 degrees of freedom).

A different approach for occluder fusion is to incrementally construct an aggregate umbra in primal space. An occluder fusion occurs whenever individual umbrae intersect [SDDS00, DDTP00]. It should be noted that umbra intersection is a sufficient condition but not a necessary one; see Figure 1.18, where three non-intersecting umbrae yield a large aggregate umbra.

Occluder fusion algorithms that are based on umbra intersection are realized in discrete space.

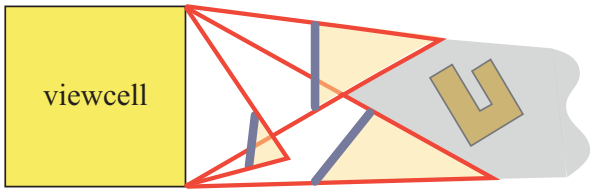
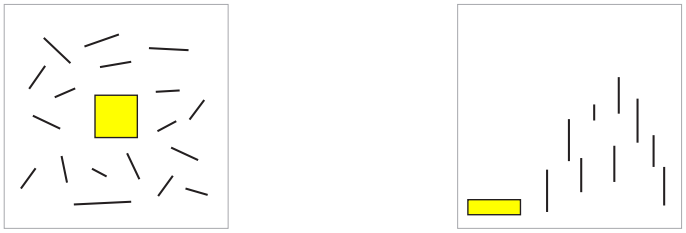


Figure 1.18: Example of occlusion due to non-intersecting umbrae. The brown object is fully occluded only by the aggregation of the umbrae of all occluders. However, none of the individual umbrae intersects.

Schaufler et al. [SDDS00] maintain discretized versions of the umbrae and extend them by generating large boxes that intersect the discrete umbrae. Durand et al. [DDTP00] use discrete projection planes placed near each occluder to capture their umbrae. Umbra fusion is accomplished by projecting the projected planes from one to the next based on a discrete convolution operation.

As shall be described next, our solution combines both of the above techniques. We use a ray space technique (horizontally) combined with an umbra merging occluder fusion technique (vertically).



(a) (b)

Figure 1.19: The two components of our factorization. (a) The axonometric (top-view) projection of a scene consisting of the viewcell (in yellow) and projected scene objects (vertical triangles). (b) A vertical plane where the line segments are the intersections of this plane with some of the scene triangles. If the triangles are non-vertical, the top-view consists of triangles and the vertical plane consists of arbitrary oriented segments, rather than vertical ones.

1.6 Overview

Our technique is based on a factorization of 4D visibility into horizontal and vertical components. We define a bounded non-singular parameterization for the 2D horizontal component by a vertical (axonometric) projection of the objects onto the ground (z = 0 plane). In Section 2.1 we show that

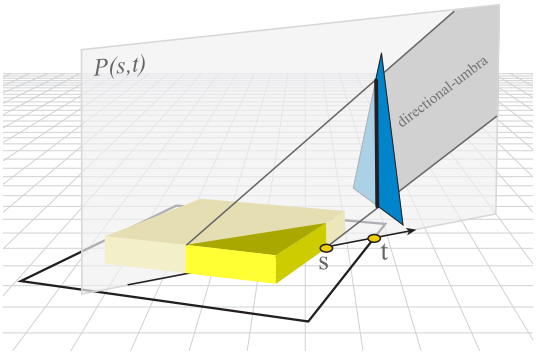


Figure 1.20: The *directional-plane* $P(s,t)$ and the *directional-umbra*. $P(s,t)$ is the vertical plane defined by the horizontal ray direction (s,t). The intersection of a polygon with $P(s,t)$ is a line segment which casts a directional-umbra with respect to the viewcell.

the footprint of the projected object is composed of a few polygons in the parameter space.

Each point (s,t) in the parameter space represents a horizontal direction. All the rays in the 3D primal space that agree with a given (s,t) direction define a vertical slice, which we call the *directional plane* (see Figure 1.20). Note that the intersection of a triangle with the directional plane is a line segment, and it casts a *directional umbra*. For each directional plane we maintain an aggregated umbra created by the occluders. The aggregated umbra for all horizontal directions is maintained in the *occlusion map*. A hierarchical front to back traversal of the objects is used to perform visibility queries and to update the occlusion map. The footprints of the objects are conservatively discretized and drawn by the graphics hardware, and the occlusion map is represented by a discrete ray space.

The rest of the thesis is organized as follows. In Chapter 2 we describe the 2D (horizontal) parameterization and the treatment of the vertical component, concluding with how to combine the horizontal and vertical components to solve the visibility in 3D. The hardware implementation details are described in Chapter 2.4 and the results are presented in Chapter 3.

Chapter 2

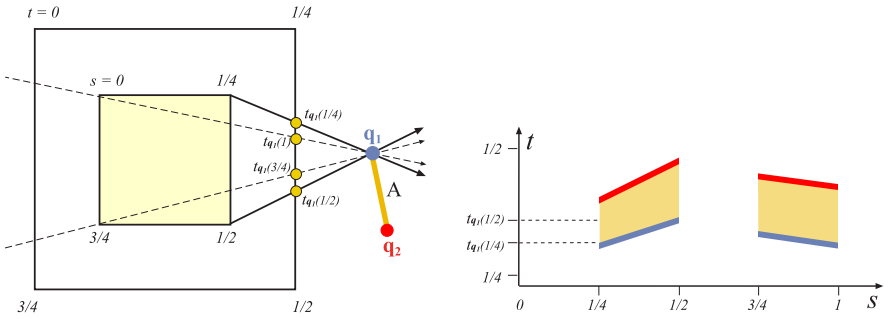
Ray Space Factorization

The common duality of $\mathbb{R}^2$ is a correspondence between 2D lines $y = ax + b$ in the primal space and points $(a, -b)$ in the dual space. This parameterization is unbounded, which prevents its simple discretization. We present a different parameterization of 2D lines. We choose to parameterize only the oriented lines (rays) that emerge from a given 2D square viewcell. This parameterization does not have singularities and the parameter space is bounded. In addition, as shown below, all the rays that leave the viewcell and intersect a triangle, form a footprint in the parameter space that can be represented by a few polygons.

2.1 The parameter space

Given a square viewcell, a representation of rays that originate from this viewcell is defined as follows. Each ray is represented by its two intersections with two concentric squares: an inner square (which is the viewcell) and an outer square (see Figure 2.1(a)). This representation can be regarded as the 2D case of the two-plane parameterization using multiple slabs [GGSC96, LH96].

Parameters s and t are associated with the inner and outer squares, respectively (see Figure 2.1(a)). They are assigned an arbitrary range, for example, the unit square $0 \leq s, t < 1$. We choose the size of the outer square edge to be of about twice the size of the inner square edge. Any ray that starts inside the viewcell must intersect both the inner and outer squares. Thus, each such ray r is represented by a pair of parameters s_r and t_r that correspond to its intersections with the inner and



(a) (b)

Figure 2.1: The footprint of a point is a line segment. The rays passing through a point in the primal space (in (a)) are mapped to line segments in the parameter space (b). The rays passing through a segment A in the primal space are mapped to the area bounded by the two footprints of the endpoints of A.

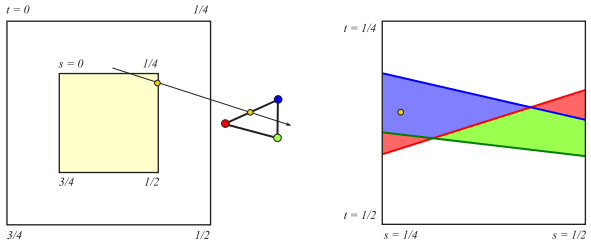
outer squares, respectively. The parameter space is bounded and each ray has a mapping.

The intersection point t of a ray with the outer square is either on a vertical edge or a horizontal edge. We choose to map the ray only to points (s,t) such that the edges of the inner and outer squares associated with s and t, respectively, are parallel. This parameterization still captures all emanating rays, since each ray intersects at least one parallel inner and outer edge pair. It is also possible that the same ray intersects the inner square twice on parallel edges, at s_1 and s_2. We choose to map such rays to two points (s_1,t_1) and (s_2,t_1) to avoid the need to distinguish between them. It should be noted that although some rays are mapped to two points, the representation is still unique.

2.1.1 The footprint of a 2D triangle

Let us define the *footprint* of a geometric primitive as the set of all points in the parameter space that refer to rays that intersect the primitive. We will now describe the shape of the footprint of a point, a segment and finally a triangle.

All the rays that intersect some point q are mapped to a set of segments in the parameter space. To compute the footprint of q we need to consider the eight pairs of parallel edges of the squares. Each pair defines a line $t_q(s) = \alpha s + \beta$ in the parameter space. Since the range of both s and t is bounded, the footprint of q is a segment on the line $t_q(s)$ bounded by the domain of s and t (see Figure 2.1(b)).



(a) (b)

Figure 2.2: The subdivision of the footprint of a triangle. (a) The orthographic projection of the triangle. Each vertex has a distinct color. (b) Part of the parameter space footprint. The line segments are shown in the same color as their corresponding vertices. The footprint is divided into different regions, each region representing rays that have the same pair of entry and exit edges. The color of each region corresponds to the vertex that is shared by both edges.

The footprint of a segment $A = \overline{q_1 q_2}$ is a set of polygons in the parameter space. Let us look at each pair of parallel edges of the parameter squares separately, and capture the rays that hit the parallel edges and the segment. For a fixed value of s on the inner edge, the range of corresponding t's defines a vertical segment $\{(s,t) : t \in [t_{q_1}(s), t_{q_2}(s)]\}$ in the parameter space (see Figure 2.1). Since $t_{q_1}(s)$ and $t_{q_2}(s)$ are linear functions of s, the set of the vertical segments of all s defines a polygon (maybe non-simple) in the parameter space. Given an arbitrary segment, its footprint consists of up to six polygons out of the eight possible pairs of parallel square edges. However, in most cases no more than four are required.

The footprint of a 2D triangle is the union of the footprints of its edges. In general, the triangle is subdivided into three regions according to the pairs of entry and exit edges, and the footprint of each region is generated as above (see Figure 2.2).

2.2 Visibility within a vertical plane

So far we have shown the first part of our factorization, that is, the parameterization of horizontal rays leaving a viewcell and passing through a 2D triangle. Now we continue to describe the second part of the factorization – the visibility within a vertical directional plane. We traverse the cells of a kd-tree in a front-to-back order and interleave occlusion tests against the occlusion map and umbra merging to maintain it. It consists of: (i) How to perform visibility queries, and (ii) How to perform occluder fusion.

2.2.1 Vertical visibility query

Let (s,t) be a point in the parameter space representing some fixed horizontal ray. We denote by $P(s,t)$ the *vertical plane* that corresponds to that direction, and by K the intersection of the axis-aligned viewcell with $P(s,t)$ (see Figure 2.3). Let R be an arbitrary 3D triangle; the line segment $B = \overline{p_1 p_2}$ is its intersection with $P(s,t)$. The segment B casts a *directional umbra* with respect to K within $P(s,t)$, which is defined by the supporting lines ℓ_t and ℓ_b ("t" stands for *top* and "b" for *bottom*, see Figure 2.3). The two values α_t and α_b denote the *supporting-angles* corresponding to ℓ_t and ℓ_b, respectively. These two values encode the directional umbra of B within the vertical plane $P(s,t)$. The angle values are represented by their tangents as functions of (s,t) (see the appendix). Hereafter we refer to these values as *angles* while we mean their tangents.

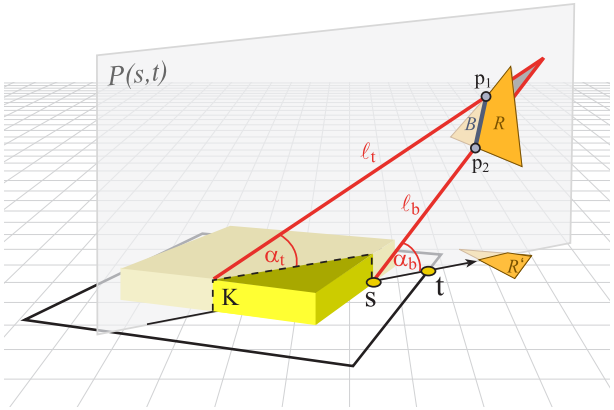


Figure 2.3: The directional plane. $P(s,t)$ is the vertical plane that corresponds to the horizontal direction (s,t). Its intersection with the viewcell is the rectangle K, and the intersection with the triangle R is the segment $B = \overline{p_1 p_2}$. The directional umbra of B with respect to K is defined by the supporting lines ℓ_t and ℓ_b or, alternatively, by the supporting angles α_t and α_b. The horizontal visibility component of R is computed by parameterizing the horizontal rays that hit R' (the orthogonal projection of R onto the ground).

Let Q be some other line segment within $P(s,t)$ that is behind B according to the front-to-back order with respect to the viewcell. Determining whether Q is occluded by B translates into testing whether the umbra of B contains Q. This test is fairly simple using the pair of supporting-angles as it only requires testing whether both endpoints of Q are inside the umbra of B. This is done by comparing the supporting-angles of B with those of Q, as illustrated in Figure 2.4. The front-to-back order guarantees that the tested segment is always behind the occluders; therefore the supporting

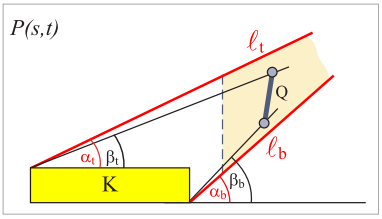


Figure 2.4: Visibility test within a vertical plane. The accumulated umbra is represented by the supporting angles α_t and α_b corresponding to the supporting lines ℓ_t and ℓ_b, respectively. The line segment Q is occluded if both of its endpoints are within the accumulated umbra, i.e. $\beta_t \leq \alpha_t$ and $\beta_b \geq \alpha_b$.

lines are sufficient for the visibility test.

2.2.2 Occluder fusion

In the vertical directional plane, umbra aggregation is performed by testing umbra intersection and fusing occluders. In general, the supporting lines alone do not uniquely describe the umbra. As depicted in Figure 2.5(a), adding the separating lines to the representation defines the umbra uniquely by pinpointing the endpoints, and allows to test whether the umbrae intersect. Whether the umbra of Q intersects the umbra of B, can be determined by a series of simple angle comparisons of Q and B (see Figure 2.5). If umbra intersection occurs, Q and B can be fused into a new "virtual occluder" segment that represents the aggregated occlusion of Q and B. Figure 2.5(g–h) describes how to create this segment. It is important to note that the fused umbra is valid only for testing occlusion of ocludees placed behind all the occluders that created the umbra.

Note that the test of merging the umbra of Q with the umbra of B is order-independent. Since often Q does not merge with B, we maintain B as the union of a number of umbrae B_i. Either Q merges with one of the B_i's or it creates another umbra component. We believe that typically, a small number of umbrae is enough to converge into a large augmented umbra, but the number of B_i's required is often order-dependent. Processing adjacent or nearby triangles is likely to rapidly merge small umbrae into one larger umbra. Thus, an approximate ordering of the occluders is more efficient. This implies that maintaining only a small number of umbra components, possibly even a single one, might not be too conservative. The visibility test based on a small number of umbrae is conservative, since if an object is visible, its footprint is not fully contained in the

full aggregated umbra, and therefore cannot be contained in any of its subsets. In particular, for the scenes with low vertical complexity that we tested, maintaining a single umbra in the occlusion map is efficient enough in the sense that it captures most of the occlusion and produces a tight PVS. In our current implementation, we ignore the last umbra component that didn't merge with the existing umbra. Typically man-made scenes have a strong vertical coherence and after some umbra-merging steps, the umbra grows to capture most of the occlusion. However, in true 3D models, with no preferable orientation, such as a flying asteroid, maintaining only one umbra is overly conservative and ineffective.

2.3 Putting it all together

In the previous section we described the visibility within a directional plane. We combine the two parts of our factorization: the vertical (directional) visibility and the horizontal footprints, together with front to back scene traversal, exploiting the fact that all the directional (vertical) computations can be performed in parallel.

The footprints are conservatively discretized before rendering, as in [WS99, DDTP00, KCOC01]. Each pixel in the discrete footprint represents a directional plane (s,t). We augment the "flat" discrete footprint of a given triangle by adding four values $\{v_0,...,v_3\}$ to each of its pixels. These values represent the angles of the four supporting and separating lines associated with the triangle and the viewcell. More precisely, let R be a 3D triangle and let R' denote the vertical projection of R (see Figure 2.6). Let $F(R')$ denote the footprint of R' in the parameter space. Each pair $(s,t)\in F(R')$ defines a directional plane $P(s,t)$ that emanates from the viewcell and intersects R'. Along each direction (s,t), the occlusion of R is expressed by the two supporting angles $\alpha_t(s,t)$ and $\alpha_b(s,t)$, and the two separating angles $\overline{\alpha}_t(s,t)$ and
overline$\alpha_b(s,t)$. In the following, we denote these four values by $v_i = \{v_0,...,v_3\}$.

For each v_i, the footprint $F(R')$ is augmented into a 3D (s,t,v) parameter space, yielding four 3D footprints. These footprints are surfaces, which can be computed using shading operations available on advanced graphics hardware. Alternatively, they can be conservatively approximated using simple polygons (see the appendix for details). This enables the use of conventional graphics hardware to generate them rapidly. Note that the discrete footprint is a conservative discretization

of the domain and the values of a continuous function, rather than their sampling.

By conservatively discretizing the bounded parameter space, all the per-(s,t) visibility operations, described in Section 2.2, are performed using per-pixel operations supported by modern graphics hardware. This takes advantage of the fact that the directional operations are independent of each other. The visibility tests and umbra merging operations are performed in parallel across the parameter space. In this setting the discrete occlusion map is an array, where each of its entries contains a series of four values v_i, that is, four values for each umbra component. The details of the hardware implementation are described in Section 2.4.

2.3.1 Hierarchical visibility culling

The original objects of the scene are inserted into a kd-tree. During the algorithm execution, the kd-tree has two functions. First, it serves as a means to traverse the scene in a front-to-back order. Second, it allows culling of large portions of the model. The kd-tree is traversed top-down, so that early on, large kd-tree cells of the hierarchy can be detected as hidden and culled with all their subtrees. If a leaf of the tree is still visible, then the visibility of each bounding box associated with it is tested. If a bounding box is visible, the triangles bounded in it are defined as potentially visible. In scenes with significant occlusion, the objects close to the viewcell rapidly fill the occlusion map, and most of the back larger kd-cells are detected as hidden. We emphasize that the from-region front-to-back order of the kd-cells is a strict order rather than approximate [BP01]. The strict order is guaranteed by using large kd-cells whose splitting planes never intersect any viewcells.

The umbra merging is applied while traversing the kd-tree front-to-back. Whenever a leaf node is detected as visible, the polygons in that leaf are considered as occluders and their footprints are inserted into the occlusion map, while possibly merging with the existing umbrae created so far during the traversal. Merging umbrae simplifies the occlusion map and increases the occlusion. Optionally, before updating the occlusion map, the visibility of individual polygons in the potentially visible leaf can be tested to tighten the PVS. The pseudocode in Figure 2.7 summarizes the entire process.

2.4 Hardware implementation

We first describe a simpler scheme that uses only 2.5D occluders. To handle 2.5D occluders we use a "half-umbra", where only the top supporting angle (α_t) is stored. The bottom supporting angle is always zero, and no separating angles are needed since the occluder umbrae always intersect. We encode α_t using the z-coordinate of the 2D footprint by conservatively linearly approximating it from below/above for occluder/occludee, respectively, as described in the appendix. Testing visibility of an occludee translates into testing the visibility of its footprint while disabling z-buffer updates to protect the occlusion map. This is accelerated using the hardware occlusion flag. Umbra fusion is implemented by enabling z-buffer updates and drawing the occluder footprint.

To handle 3D occluders we need to use a series of top and bottom supporting and separating angles. However, due the limitation of currently available hardware, the occlusion map stores only a single directional umbra per direction, compactly placed in different regions of a single z-buffer. Testing visibility of an occludee translates into testing whether the top or bottom supporting footprints are visible when rendered using the top/bottom occlusion map z-buffers. Again, rendering is performed while z-buffer updates are disabled. Umbra fusion is implemented by testing whether the new umbra intersects the current umbra in the occlusion map. For a given occluder, its four supporting and separating angles are required to be tested with the occlusion map. This is implemented in two passes, where in the first one the stencil buffer is used to mask locations where the umbra intersects, and in the second pass the z-buffer is actually updated where the stencil test passes.

Note that this two-pass scheme is expensive since each occluder needs to be drawn twice by the hardware. Our current implementation is reported for NVIDIA GeForce4 Ti graphics card. The NVIDIA GeForce FX card allows more flexibility. Since we implemented it only with an emulator, we cannot report the acceleration expected. However, the GeForce FX card provides stronger fragment shader functionality to support the calculation of the directional umbra within each slice. In particular, $\tan\alpha = H/L$ (see appendix) can be computed without approximation.

Our implementation is using NVIDIA's Cg shader language. We use the available 32bit floating-point PBuffers (denoted occlusionPB) to store the global occlusion map, thus allowing four 32bit floating-point precision values in each (s,t) pixel to store the exact v_i values. We use an additional 32bit floating-point buffer (denoted tempPB) for temporary storage. Augmenting the occlusion

map with the umbra of an occluder triangle R is performed as follows. We use occlusionPB as input texture and tempPB for output. We render the 2D footprint of R, thus triggering fragment-shader code in (s,t) pixels that represent directional planes that intersect R. At each such pixel, the fragment shader calculates the intersection segment of R and the corresponding directional plane, and extracts the exact separating and supporting angles (v_i values). The current umbra at each pixel is read from occlusionPB. The read occlusion angles are compared to the newly calculated angles of R as explained in Figure 2.5. Where fusion is possible, the fused umbra, represented by four new v_i values, is output to tempPB; otherwise the pixel is killed. The above yields the set of pixels within tempPB containing the fused umbra. Whether any umbra fusion has occurred is identified by the occlusion query extension that tests if not all pixels were killed. In that case, the 2D footprint is rendered again while setting tempPB as input texture and occlusionPB as output. The fragment shader simply copies the updated values to occlusionPB.

Testing the visibility of a potential occludee R is performed by setting occlusionPB as input texture and tempPB as output. The fragment-shader calculates only the v_i values that represent supporting angles, reads the occlusion supporting angles from occlusionPB and performs the comparison between them as explained in Figure 2.4. The shader kills the pixels that represent directional planes wherein R is occluded and outputs some arbitrary value for pixels where R is visible. The visibility test is performed by using the occlusion query extension to test if not all pixels were killed.

As a future extension to support multiple umbrae per slice, we suggest to pack 8 half-float (16 bit floating point) values in the 32bit floating point PBuffers. This allows storing two umbrae per pixel in the occlusion map. Another, more general approach would be to use multiple 32bit floating point PBuffers to store the occlusion map. However, since it is currently not possible to output to more than a single buffer, augmenting such an occlusion map probably requires multiple rendering passes which is thus overly expensive.

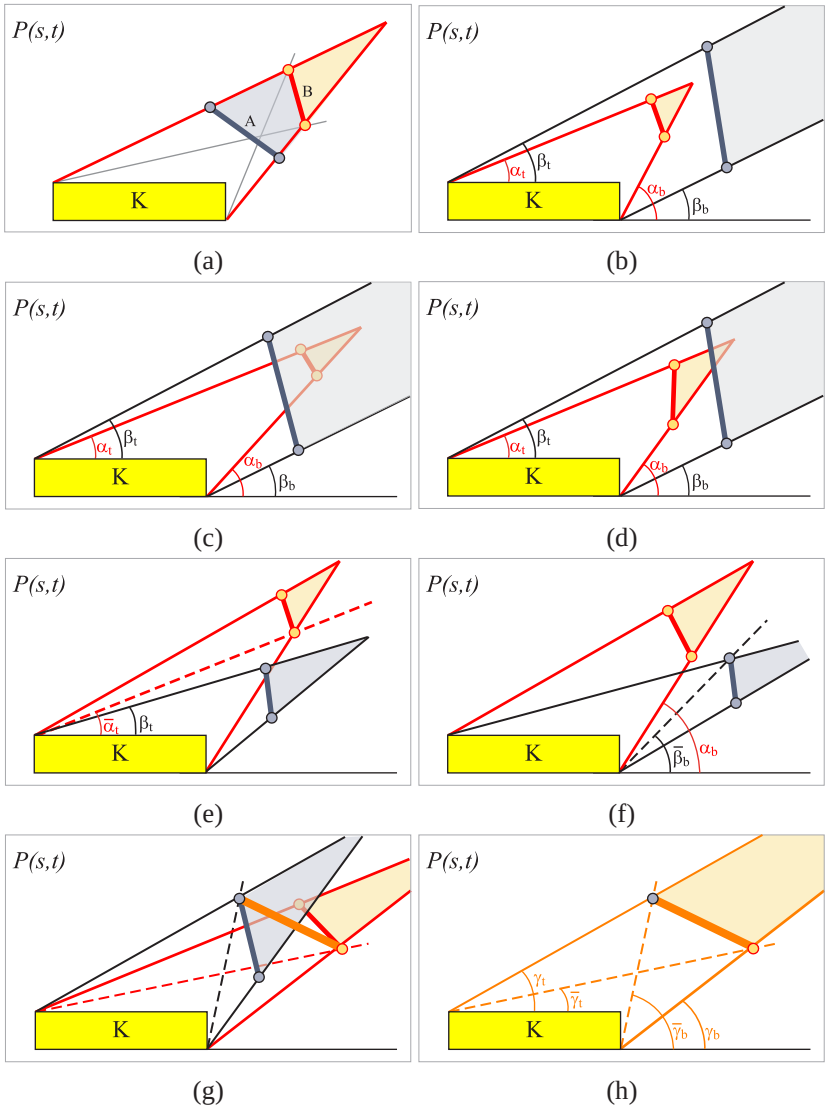
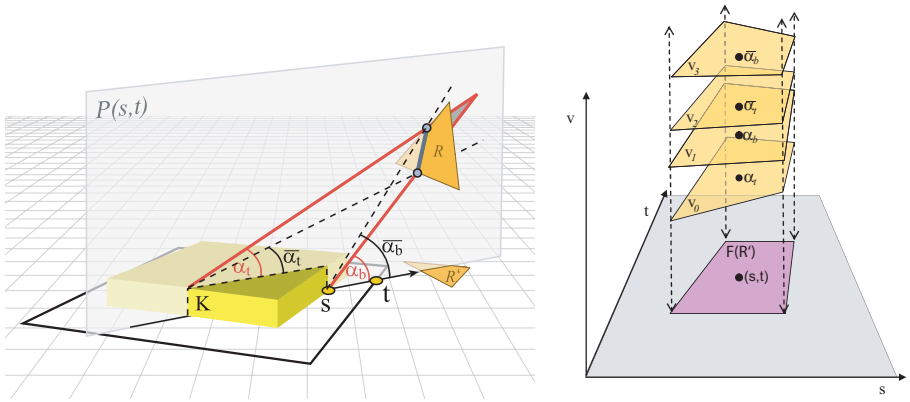


Figure 2.5: (a) Representing umbrae solely by the supporting lines does not provide a unique definition. The segments A and B have the same supporting lines, though they cast different umbrae. Adding separating lines pinpoints the location of the endpoints of the segment, thus defining the umbra uniquely. (b)–(h) Different cases of occluder fusion within a vertical slice. The separating lines are dashed and supporting lines are solid. The supporting angles of the segments are α_t, α_b (red) and β_t, β_b (blue). Separating angles are denoted by $\bar{\alpha}_t$, $\bar{\alpha}_b$, $\bar{\beta}_t$, $\bar{\beta}_b$. In cases (b)–(d), the umbra of one segment is *contained* in the area between the supporting lines of the other segment, therefore the first segment's occlusion makes no contribution even if the two umbrae intersect. This case happens when $\alpha_t < \beta_t$ and $\alpha_b > \beta_b$ (or vice versa, i.e. $\alpha \leftrightarrow \beta$). In such a case we "throw out" the contained segment and keep just the other one. If the above comparison condition doesn't hold, we test whether we are in situations (e) or (f), where no umbra intersection occurs. The tests are: $\bar{\alpha}_t > \beta_t$ (e) or $\alpha_b > \bar{\beta}_b$ (f). If these tests fail as well, then there must be umbra intersection (g). We replace the two segments by a new "virtual" occluder segment (see the orange segment in (g)), i.e. we insert into the occlusion map the following angles: $\gamma_t = \max\{\alpha_t, \beta_t\}$, $\gamma_b = \min\{\alpha_b, \beta_b\}$, $\bar{\gamma}_t = \min\{\bar{\alpha}_t, \bar{\beta}_t\}$, $\bar{\gamma}_b = \max\{\bar{\alpha}_b, \bar{\beta}_b\}$ (see (h)). It is easy to prove that any ray that leaves the viewcell and intersects the new segment must be also blocked by at least one of the two old segments. Thus, any occludee placed *behind* both old segments, is occluded by the virtual occluder iff it is occluded by the old segments.



(a) (b)

Figure 2.6: Footprint of the triangle R in the (s,t,v) parameter space. (a) The supporting and separating angles of R (α_t, α_b, $\overline{\alpha}_t$, $\overline{\alpha}_b$) in the primal space. (b) The augmented footprint $F(R')$ into the 3D (s,t,v) parameter space. Each $(s,t)\in F(R')$ is associated with the 4 v_i values (denoting the supporting and separating angles within the vertical plane $P(s,t)$. Note that the v_i surfaces above $F(R')$ are rational surfaces that can be conservatively approximated using polygons (see the appendix).

```
PVS ← ∅
CalculateVisibility( kd_tree.root )

CalculateVisibilty( curr_kd_cell )
    if TestVisibility(curr_kd_cell) // current kd-cell is visible
        if cur_kd_cell.IsLeaf()
            PVS ← PVS ∪ curr_kd_cell.getTriangles()
            AugmentOcclusionMap(curr_kd_cell.getTriangles())
        else
            foreach kd_child ∈ curr_kd_cell.children in f2b order
                CalculateVisibility( kd_child )
            end foreach
        endif
    else
        return // current kd-cell occluded ⇒ terminate
    endif
end
```

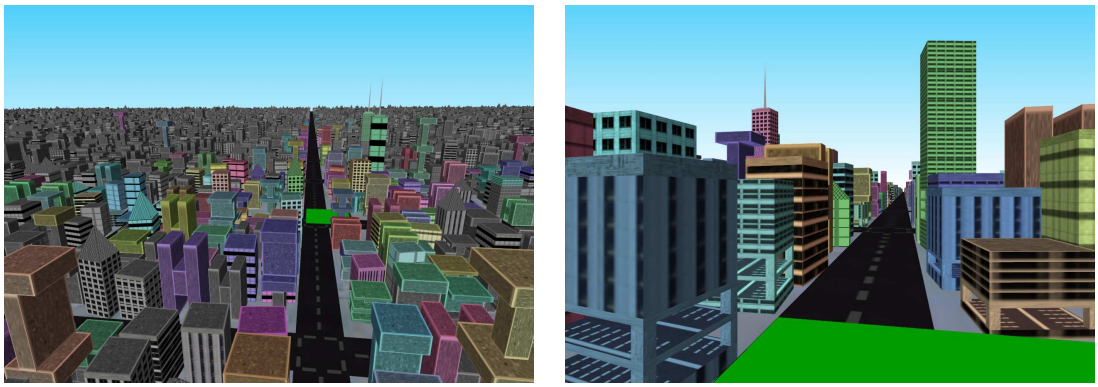
Figure 2.7: Pseudocode of the overall occlusion culling scheme.

Chapter 3

Results

We have implemented the technique and integrated it into a hierarchical occlusion culling mechanism based on a kd-tree data-structure. The results we report here are of our current implementation on a 2GHz P4 machine with NVIDIA GeForce4 Ti graphics card. We used a randomly generated urban model controlled by a large set of parameters which define model size, density, distribution of heights, regularity, etc. Some of the buildings consist of parts of the shape of the letters **H** and **T**, and some buildings have a number of parking decks in their first floors as a means of increasing the vertical complexity of the urban model. Unlike common reconstructed urban models, here the buildings consist of several floors, each being a 3D box. The buildings are between 9–12 units in width and length and are rotated by at most 30°, thus they are not axis-aligned. See Figure 3.1 where only the visible buildings are colored, while the occluded ones are in gray.

We also generated a non-realistic model, which we call the "box-field", that consists of randomly generated boxes. The boxes have arbitrary size up to 10×10×10 units and arbitrary orientation (see Figure 3.2) that form a highly complex 3D model. With this model we tested and analyzed the behavior of our technique in the vertical direction. The complexity of the box-field model is apparent in Figure 3.2(b). The model is not too dense, so that some geometry deep inside the box-field is visible. Since a single visible node necessarily causes an entire branch of the kd-tree to be visible, it avoids the early culling of a large portion of the tree, reducing the effectiveness of the use of a hierarchy.



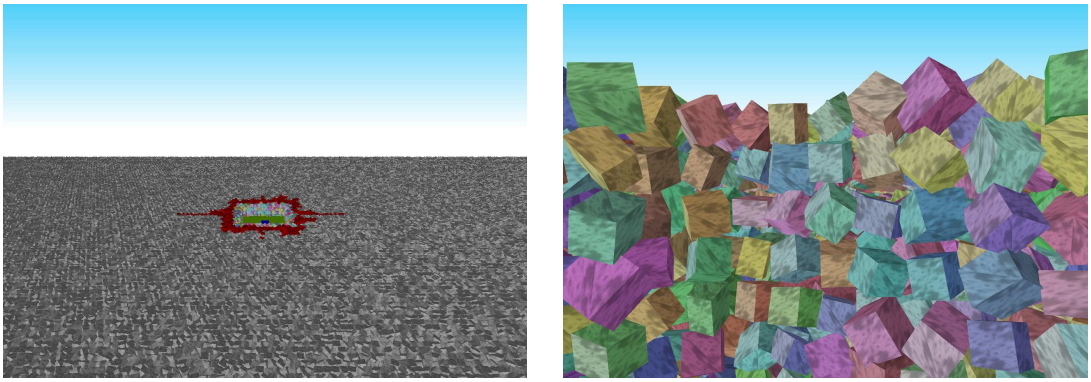
(a) (b)

Figure 3.1: The city model consisting of 26.8M triangles. (a) The view from above. The viewcell is located in a junction (in green) where distant geometry is visible. (b) A typical view from inside the viewcell during a walkthrough. The viewcell size is 25×25×2.5 units.

Cell size	Off-junction viewcells		In-junction viewcells	
	Time	PVS / VS	Time	PVS / VS
3	0.31	2412 / 1760	0.40	8832 / 6824
9	0.41	2840 / 2632	0.51	12184 / 9072
14	0.58	3592 / 2894	0.96	13576 / 9184
20	0.71	4568 / 2928	1.56	18304 / 9888
25	0.78	5224 / 2960	2.01	21608 / 10072

Table 3.1: Results for the urban scene consisting of 26.8M triangles. *Cell size* denotes the length and width of the viewcell; the height of all viewcells is 2.5 units. *PVS* and *VS* size is given in triangles, *Time* refers to culling-time in seconds. The results are taken from two types of viewcells: *Off-junction* viewcells are within some city block, whereas *in-junction* viewcells are positioned on the junction of two long avenues. The buildings are between 9–12 units in width and length and are rotated by at most 30° about the axes.

We approximated the exact visibility set (VS) by sampling many random viewpoints and view-directions with a from-point algorithm. By running enough samples (around 1000), at some point the sampled VS converges and gets very close to the exact VS. Tables 3.1 and 3.2 summarize the results for the urban model and the box-field model. The tables compare the performance of the technique in terms of speed and culling conservativeness (the size of the PVS vs. the VS). When the viewcell is placed in the junction of two long avenues the VS and consequently the PVS are much larger than the VS of a viewcell placed elsewhere. Similarly, as shown in Table 3.2, by moving the viewcell away from the box-field we increased the size of the PVS. When the viewcell is closer to



(a) (b)

Figure 3.2: The box field model, 20.7M polygons. In (a), the entire model is displayed. Gray polygons are those detected as hidden, colored ones are visible and the red polygons belong to the PVS although they are occluded. The area around the viewcell was cleared to increase the portion of the visible geometry. (b) A view from inside the viewcell shows the complexity of the scene. The viewcell size is 20×20×10.

Cell size	Near viewcell		Far viewcell	
	Time	PVS / VS	Time	PVS / VS
5	0.93	4864 / 1312	3.22	35456 / 14224
10	1.10	6032 / 1424	3.45	37072 / 14256
15	1.27	7184 / 1552	3.59	38912 / 14304
20	1.39	8176 / 1632	3.71	40080 / 14416
30	1.87	13136 / 2400	4.04	43248 / 14672

Table 3.2: Results for the box-field scene consisting of 20.7M triangles with respect to different viewcell sizes. *Near viewcell* is a type of a viewcell located from within the field (as seen in Figure 3.2(a)) whereas *Far viewcell* is a type located outside and far from the field. The height of all viewcells is 10 units.

the scene, the objects are larger and occlude much more.

The degree of conservativeness of the vertical umbrae merging technique depends on the number of umbrae maintained in the occlusion map. Table 3.3 reports the results of computing the PVS using a half umbra and a full umbra for the urban scene. Half an umbra captures only the occlusion of 2.5D occluders. Maintaining a single full umbra reduces the size of the PVS, at the expense of longer computation time. On current cards it would be too expensive to implement an occlusion map with multiple umbrae, since it would require a stack of textures and a multi-pass which significantly slows down per-pixel operations (see also Section 2.4).

Viewcell size	VS size	Half-umbra		Full umbra	
		PVS size	Time	PVS size	Time
3	7536	28296	0.966	18968	1.354
9	7720	31936	1.149	22368	1.704
14	7808	33504	1.184	22576	1.745
25	8304	36480	1.253	24520	1.845

Table 3.3: Comparison between culling effectiveness using only 2.5D occluders and by using 3D occluders on a urban city model consisting of 20M triangles. Non-2.5D occluders compose 30% of the scene occluders. VS and PVS sizes are in triangles.

All our tests show that the culling time is directly dependent on two interdependent factors: (i) the number of visibility queries, and (ii) the number of visible triangles (occluders). Due to the hierarchy, the technique can deal with a huge model, but this is only as long as the size of the VS is small. A small VS means that only a small number of triangles is visible, which implies that only a small number of kd-cells in the hierarchy is tested.

Regarding the conservativeness of the technique, as shown in the tables, our current implementation yields a PVS which is quite conservative. However, the absolute size of the PVS is relatively small (in our tests it is less than 0.1% of the full model) and bounded for dense scenes, so common graphics hardware can render it in real-time. Moreover, the effectiveness of the technique is not measured for a single view, but over time, for a large number of frames. Assuming the scene is rendered at 30 frames per second, at least hundreds of frames are generated within one viewcell, and the actual cost of generating the PVS can be amortized over time.

Another important factor is the resolution of the ray space. Clearly it takes more time to render in high resolution than in lower resolution. Moreover, the cost of a hardware-based visibility test is also proportional to the resolution. Our tests show that the cost is a linear function of the resolution. In general, a high resolution ray space yields a less conservative PVS. Table 3.4 reports these numbers. We found that 512×512 is an effective resolution, and all the results reported here are with this ray space resolution.

Our results analyze the method as function of the viewcell size. We can see a clear dependency between the viewcell size and the size of the PVS, and consequently the culling time. The challenge is to treat viewcells that are large relative to the average occluder. Our results show that the culling time grows more slowly than the viewcell size, suggesting that for these scenes our technique is not

too sensitive to the viewcell size, but rather to the size of the VS. This implies that at the negligible cost of testing the visibility of another large kd-cell, we could cull a scene which is twice as large.

Finally, we tested the algorithm on the Vienna2000 model which represents about 3×3km of the city [Vienna2000]. We checked two types of viewcells: small and large, and generated two different kd-trees, respectively. Polygons crossing the kd-cell boundaries are split. Figure 3.3 is a view of a culling result from a large viewcell. The results are reported in Table 3.5. It can be seen that for the smaller viewcells, the PVS is reasonably tight, ranging at about 2–2.6 times of the VS size, while for large viewcells the PVS/VS ratio grows to about 6–7. This is because the conservative approximations (see appendix) are sensitive to the viewcell size and thus play a significant role for larger viewcells.

Resolution	PVS size	Percentage	Time (sec.)
128×128	36936	0.14%	0.626
256×256	27920	0.10%	1.217
512×512	21608	0.08%	2.011
1024×1024	14168	0.05%	5.335

Table 3.4: The effect of the frame-buffer resolution on the PVS size and the computation time. PVS size is given in triangles and *percentage* is the size of the PVS relative to the full model consisting of 26.8M triangles. The PVS was computed from a viewcell of size 25×25×2.5 units. For comparison, the VS size is 10072 triangles.

3.1 Discussion

Our factorization method decomposes the 3D from-region visibility problem into many simpler from-region problems in 2D vertical slices. The solution is asymmetric as it treats the vertical dimension differently than the horizontal one. While the latter is almost exact (it is conservative only due to discretization), the former assumes vertical coherence in the scene that guarantees that a significant portion of individual umbrae intersect and merge. Due to gravity, realistic models tend to have a general vertical orientation and typically their vertical complexity is low. One can argue that if the simplest vertical complexity is 2.5D, then more vertically complex models are somewhere between 2.5D and 3D. In that sense we claim that conceptually, our technique can treat "3D-ϵ" scenes, while in practice due to current hardware limitations we treat "2.5D+ϵ" scenes.

The methods presented by Durand et al. [DDTP00] and Schaufler et al. [SDDS00] handle 3D from-region visibility by fusing occluders based on 3D umbra intersection. As we mentioned above, umbra intersection is a conservative approach, which we employ only within the vertical slices, where the visibility coherence is typically large. In contrast, in the horizontal direction we use a ray-parameterization which is exact up to discretization of the polygonal footprint. This allows to increase the horizontal dimensions of the viewcell with respect to the average occluder size.

The from-region techniques [KCCO00, KCOC01, BWW01] are limited to 2.5D occlusion. Bittner et al. [BWW01] also use a 2D ray space parameterization. In contrast to us, their 2D visibility is used to detect the set of potential occluders for a given 2.5D object, while we use the parameterization for the decomposition of the from-region problem. A key point in the design of our algorithm is that it uses an *occludee-independent* ray space. This is in contrast to the *occludee-dependent* ray parameterization used in [KCOC01], where the parameterization is valid only for objects in the shaft defined by the viewcell and the given kd-cell. The view-dependent parameterization necessarily requires a repeated clipping of the entire scene with each of the view-dependent shafts. The view-independent ray space allows the scene to be traversed in front-to-back order, where each visible object is accessed only once to update the occlusion map.

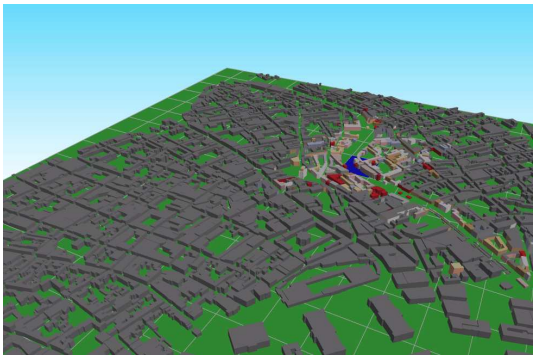


Figure 3.3: A view of the culling result for the Vienna 2000 model. The viewcell is colored in blue. Viewcell size is 150×150×2m.

Nirenstein et al. [NBG02] also solve the 3D from-region problem in ray space. However, they provide an exact solution using high-dimensional spaces, which cannot compete in speed with conservative solutions.

Small viewcells			Large viewcells		
Size	**Time**	**PVS / VS**	**Size**	**Time**	**PVS / VS**
3	0.18	964 / 424	50	0.36	3396 / 502
14	0.19	1216 / 514	100	0.45	4424 / 744
25	0.22	1546 / 592	150	0.58	5720 / 930

Table 3.5: Results for the Vienna2000 model. The height of the buildings is 2.2–50.8m (18.6m on average). *Size* denotes the length and width of the viewcell in meters; the height of all viewcells is 2m. The total number of triangles is 87k (for small viewcells) and 72k (for larger cells). The reason for the difference is that the 67k triangles of the original model are split across kd-cells boundaries.

3.2 Conclusions

We have presented a from-region visibility technique that takes advantage of the capabilities of the advanced graphics hardware to compute from-region visibility. The technique is based on the factorization of the 4D problem into horizontal and vertical components. A notable property of our solution is that it is asymmetric, since it favors the horizontal component over the vertical one. The footprints drawn on the horizontal plane are exact, up to discretization, while in the vertical direction we employ a conservative occluder fusion technique. This is based on the observation that in common scenes there is a preferable orientation since the vertical direction is less complex.

As discussed above, the current implementation of the occluder fusion in the vertical direction is conservative. One can consider other hardware-assisted implementations and adapt them to the available hardware capabilities. We believe that in the future, graphics cards will include new features that will facilitate the implementation of from-region visibility techniques. This is especially vital for remote walkthrough applications where from-point calculations on the server are not applicable due to network latency.

3.3 Acknowledgments

We would like to thank Frédo Durand, Peter Wonka, Yiorgos Chrysanthou and Shuly Lev-Yehudi for fruitful discussions and for reviewing an early version of this work. We also thank the anonymous reviewers and referees for their comments. This work was supported in part by the Israel Science Foundation founded by the Israel Academy of Sciences and Humanities and by the Israeli Ministry of Science.

Chapter 4

Appendix

Here we describe the calculation of the supporting and separating angles and their conservative rasterization over the footprint. We choose to consider the tangents of the angles because they can be easily computed and maintain the order relation between angles.

Given a 3D triangle R and a directional slice $P(s,t)$, denote the intersection of R with $P(s,t)$ by Q. The bottom supporting angle α is given by $v = \tan\alpha = H/L$ (see Figure 4.1(b)). We will only discuss the bottom supporting angles; the other supporting and separating angles can be handled in a similar fashion. H and L are functions of (s,t), and we would like to interpolate $v(s,t) = H(s,t)/L(s,t)$ across the horizontal (s,t) footprint of R to obtain the 3D footprint in the (s,t,v) parameter space.

L is the horizontal distance between the considered endpoint of Q and the front of the viewcell (i.e. the part of the viewcell closest to Q). In Figure 4.1(a), $L = \lambda_1 \|\mathbf{v_1} - \mathbf{v_0}\|$. Define $\mathbf{n_1} = \perp \overline{\mathbf{u_0}\mathbf{u_1}}$. It can be easily verified that:

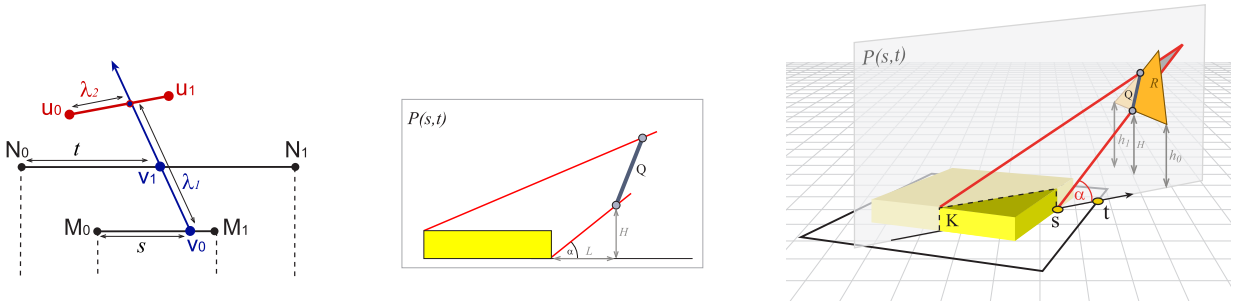
$$\lambda_1 = \frac{< \mathbf{u_0} - \mathbf{v_0},\ \mathbf{n_1} >}{< \mathbf{v_1} - \mathbf{v_0},\ \mathbf{n_1} >}.$$

Since $\mathbf{v_0} = M_0 + s(M_1 - M_0)$ and $\mathbf{v_1} = N_0 + t(N_1 - N_0)$, we obtain by substitution that

$$\lambda_1 = \frac{a_1 s + a_3}{b_1 s + b_2 t + b_3},$$

where

$$a_1 = - < M_1 - M_0,\ \mathbf{n_1} >,\ a_3 = < \mathbf{u_0} - M_0,\ \mathbf{n_1} >,$$



(a) (b) (c)

Figure 4.1: (a) The red line denotes the 3D segment, as seen from above. The height of endpoint $\mathbf{u_0}$ is h_0 and the height of $\mathbf{u_1}$ is h_1. A ray is shot from point $\mathbf{v_0}$ on the viewcell boundary to the point $\mathbf{v_1}$ on the outer square boundary. (b) Measuring angles by their tangents. Here, $\tan\alpha = H/L$, where α is the bottom supporting angle of the segment Q, H is the height of the lower endpoint of Q, and L is its the horizontal distance from the viewcell front. (c) Representing H as an interpolation of the heights of the endpoints, h_0 and h_1.

$$b_1 = a_1,\ b_2 =< N_1 - N_0,\ \mathbf{n_1} >,\ b_3 =< N_0 - M_0,\ \mathbf{n_1} > .$$

Now, let us compute the function H, which is the height of the considered endpoint of Q (note that when we compute the top supporting and separating angle of Q, height is measured from the top of the viewcell). Let h_0 and h_1 be the heights of the endpoints of the considered edge of R (see Figures 4.1(a),(c)). Then $H = h_0 + \lambda_2(h_1 - h_0)$. It is thus sufficient to calculate λ_2 as a function of s and t. Denote $\mathbf{n_2} =\perp \overline{\mathbf{v_0 v_1}}$. Then similarly to λ_1, λ_2 can be written as

$$\lambda_2 = \frac{< \mathbf{v_0} - \mathbf{u_0},\ \mathbf{n_2} >}{< \mathbf{u_1} - \mathbf{u_0},\ \mathbf{n_2} >}.$$

Assume $M_i = (M_{ix}, M_{iy})$, $N_i = (N_{ix}, N_{iy})$, *i=0,1*. We can write $\mathbf{n_2}$ explicitly as

$$\mathbf{n_2} = (\Delta N_y\, t - \Delta M_y\, s + N_{0y} - M_{0y},\ \Delta M_x\, s - \Delta N_x\, t + M_{0x} - N_{0x}).$$

After substituting this explicit form in the equation of λ_2, we get

$$\lambda_2 = \frac{a_1 s + a_2 t + a_3}{b_1 s + b_2 t + b_3},$$

where the numbers a_j and b_j, *j=1,2,3,* are constants (they depend on M_i, N_i and the segment endpoints only). Thus, H is a first-order rational function of s and t.

Linear approximation. The value of $\|\mathbf{v_1} - \mathbf{v_0}\|$ is between some d_{min} and d_{max} over the entire (s,t) footprint. Thus, L can be conservatively approximated by a first order rational function of s, t:

$$d_{min}\lambda_1(s,t) \le L(s,t) \le d_{max}\lambda_1(s,t).$$

How to linearly approximate H/L? First, either H or L are conservatively approximated by a constant. It is preferable to take the "less changing" function for that purpose. If the height does not change much ($h_0 \approx h_1$), then H should be set to a constant. Otherwise, L is set to a constant which is the minimal/maximal distance of a point in the viewcell from the segment, for occluder/occludee accordingly. The function that is not set to a constant can be further approximated by a plane. Suppose we have

$$F(s,t) = \frac{a_1 s + a_2 t + a_3}{b_1 s + b_2 t + b_3} = \frac{F_1(s,t)}{F_2(s,t)},$$

where $F = H$ or $F = L$. The point (s,t) is in a compact polygonal domain P. $F_1(s,t)$ has the same sign as $F_2(s,t)$, because $F(s,t)$ is positive. Thus, we only need to compute $m_1 = \max F_2(s,t)$ and $m_2 = \min F_2(s,t)$ over the domain P, by testing its vertices. Then, supposing w.l.o.g. that F_2 is positive:

$$\frac{a_1 s + a_2 t + a_3}{m_1} \le F(s,t) \le \frac{a_1 s + a_2 t + a_3}{m_2}.$$

For even tighter approximation, we triangulate P using "cutting ears" algorithm [Mei75] and compute linear approximations of F over each triangle.

Bibliography

[ARB90] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics (1990 Symposium on Interactive 3D Graphics)*, 24(2):41–50, March 1990.

[BHS98] Jiri Bittner, Vlastimil Havran, and Pavel Slavik. Hierarchical visibility culling with occlusion trees. In *Proceedings of Computer Graphics International '98*, pages 207–219, June 1998.

[BP01] Jiri Bittner and Jan Prikryl. Exact regional visibility using line space partitioning. Technical Report TR-186-2-01-06, Institute of Computer Graphics, Vienna University of Technology, Karlsplatz 13/186, A-1040 Vienna, Austria, 2001.

[BWW01] Jiri Bittner, Peter Wonka, and Michael Wimmer. Visibility preprocessing for urban scenes using line space subdivision. *9th Pacific Conference on Computer Graphics and Applications*, pages 276–284, October 2001.

[COFHZ98] Daniel Cohen-Or, Gadi Fibich, Dan Halperin, and Eyal Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. *Computer Graphics Forum*, 17(3):243–254, 1998.

[CT96] Satyan Coorg and Seth Teller. Temporally coherent conservative visibility. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 78–87, 1996.

[dBvKOS97] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

[Vienna2000] VIENNA. Available at *http://www.cg.tuwien.ac.at/research/vr/urbanmodels/*

[DDTP00] Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. Conservative visibility preprocessing using extended projections. *Proceedings of ACM SIGGRAPH 2000*, pages 239–248, July 2000.

[Dur99] Frédo Durand. *3D Visibility: Analytical study and Applications*. PhD thesis, Universite Joseph Fourier, Grenoble, France, July 1999.

[FST92] Thomas A. Funkhouser, Carlo H. Séquin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. *1992 Symposium on Interactive 3D Graphics*, 25(2):11–20, March 1992.

[GGSC96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proceedings of ACM SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 43–54, August 1996.

[GKM93] Ned Greene, M. Kass, and Gary Miller. Hierarchical z-buffer visibility. *Proceedings of ACM SIGGRAPH 93*, pages 231–240, 1993.

[GSF99] Craig Gotsman, Oded Sudarsky, and Jeffry Fayman. Optimized occlusion culling. *Computer & Graphics*, 23(5):645–654, 1999.

[HMC+97] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frustra. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 1–10, 1997.

[KCCO00] Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. Virtual occluders: An efficient intermediate pvs representation. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 59–70, June 2000.

[KCOC01] Vladlen Koltun, Daniel Cohen-Or, and Yiorgos Chrysanthou. Hardware-accelerated from-region visibility using a dual ray space. *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, June 2001.

[LH96] Marc Levoy and Patrick M. Hanrahan. Light field rendering. In *Proceedings of ACM SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 31–42, August 1996.

[LSCO03] Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. Ray space factorization for from-region visibility. *ACM Transactions on Graphics (TOG)*, 22(3):595–604, 2003.

[Mei75] Gary H. Meisters. Polygons have ears. *American Mathematical Monthly*, 82:648–751, 1975.

[NBG02] Shaun Nirenstein, Edwin Blake, and James Gain. Exact from-region visibility culling. In *Rendering Techniques 2002 (Proceedings of the 13th EUROGRAPHICS Workshop on Rendering*, pages 199–210. Springer, 2002.

[SDDS00] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. Conservative volumetric visibility with occluder fusion. *Proceedings of ACM SIGGRAPH 2000*, pages 229–238, July 2000.

[SVNB99] Carlos Saona-Vazquez, Isabel Navazo, and Pere Brunet. The visibility octree: A data structure for 3d navigation. *Computer & Graphics*, 23(5):635–644, 1999.

[Tel92a] Seth Teller. *Visibility Computations in Densely Occluded Environments*. PhD thesis, University of California, Berkeley, 1992.

[Tel92b] Seth J. Teller. Computing the antipenumbra of an area light source. *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, 26(2):139–148, July 1992.

[TS91] Seth J. Teller and Carlo H. Sequin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):61–69, 1991.

[WS99] Peter Wonka and Dieter Schmalstieg. Occluder shadows for fast walkthroughs of urban environments. In Hans-Peter Seidel and Sabine Coquillart, editors, *Computer Graphics Forum*, volume 18, pages C51–C60. Eurographics Association and Blackwell Publishers Ltd 1999, 1999.

[WWS00] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pages 71–82, June 2000.

[WWS01] Peter Wonka, Michael Wimmer, and François X. Sillion. Instant visibility. In A. Chalmers and T.-M. Rhyne, editors, *Computer Graphics Forum (Proc. of Eurographics '01)*, volume 20(3), Manchester, UK, September 2001.

[ZMHH97] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. In Turner Whitted, editor, *Proceedings of ACM SIGGRAPH 97*, Annual Conference Series, pages 77–88. ACM SIGGRAPH, Addison Wesley, August 1997.

תקציר

אנו מציגים מתודולוגיה חדשנית לחישוב קונסרבטיבי של נראות מתא המתבססת על פירוק מרחב הקרניים ה 4-מימדי לרכיבים אופקיים ואנכיים. הנראות מחושבת באופן שונה ואסימטרי בכל רכיב. הרכיב האופקי מבוסס על מרחב קרניים חדשני שבבסיסו פרמטריזציה באמצעות שני ריבועים קונסנטריים. ברכיב האנכי אנחנו ממזגים את אזורי הצל בתוך פרוסות אופקיות. באמצעות הרכיב האופקי אנחנו מפרידים את בעיית הנראות הקשה מתא בתלת-ממד להרבה בעיות נראות פשוטות יותר בפרוסות אופקיות בדו-ממד. השיטה מתוכננת כך שניתן להיעזר בחומרת הגרפיקה המודרניות להאצה משמעותית של המימוש.

בדומה לשיטות דיסקרטיות אחרות לחישוב נראות מנקודה, אנחנו מתחזקים מפת הסתרות המשמשת כעזר לחישוב; אולם, מפת ההסתרות הדיסקרטית נמצאת במרחב הקרניים ולא במרחב הרגיל. מפה זו שומרת באופן קונסרבטיבי את סך ההסתרות של האובייקטים בסצנה שטופלו עד כה.

התוצאות שלנו מראות כי זמני חישוב הראות וגודל ה PVS תלויים בגודל התא. לתאים סבירים לאפליקציות walk-through עבור סצנה אורבנית גדולה, זמן חישוב הראות הקונסרבטיבי באמצעות שיטה זו הינו קטן משנייה, וכן גודל ה PVS הוא עד פי שניים מגודל ה VS המדויק.

עבודה זאת פורסמה ב ACM Transactions on Graphics [LSCO03] .

אוניברסיטת תל-אביב
הפקולטה למדעים מדוייקים על שם ריימונד ובברלי סאקלר
בית הספר למדעי המחשב

חישוב נראות מתא באמצעות פירוק מרחב הקרניים

חיבור זה הוגש כחלק מהדרישות לקבלת התואר
"מוסמך אוניברסיטה" - M.Sc. באוניברסיטת תל-אביב

ע"י

תומר ליבנד

העבודה הוכנה בהדרכתו של פרופ' דניאל כהן-אור

יוני 2005