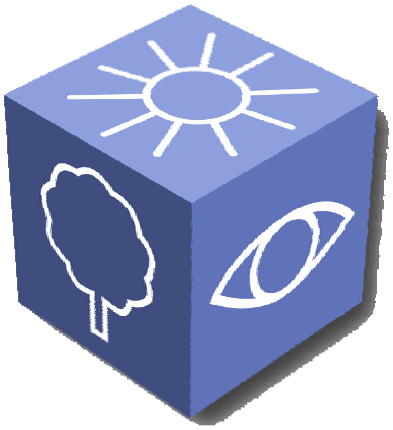


GAMETOOLS

ADVANCED TOOLS FOR DEVELOPING HIGHLY REALISTIC COMPUTER GAMES

REPORT ON VISIBILITY ALGORITHMS

Document identifier:	**GameTools-3-D3.2-02-1-1-Report on Visibility Algorithms**
Date: (use "update field" Word function, right mouse button)	**22/09/2005**
Work package:	**WP03: Visibility**
Partner(s):	**VUT**
Leading Partner:	**VUT**
Document status:	**APPROVED**
Deliverable identifier:	**D3.2**

Abstract: This technical report describes the different algorithms used on the implementation of the Visibility module.

Delivery Slip

	Name	Partner	Date	Signature
From	Jiri Bittner	VUT	15-09-2005	
Reviewed by	Moderator and reviewers	ALL		
Approved by	Moderator and reviewers	ALL		

Document Log

Issue	Date	Comment	Author
1-0	15-09-2005	First draft	Jiri Bittner
1-1	22-09-2005	Final Version	Michael Wimmer

Document Change Record

Issue	Item	Reason for Change

Files

Software Products	User files / URL
Word	gametools-ist-2-004363-3-d3.2-02-1-1-report on visibility algorithms.doc (use “update field” Word function)

Contents

Chapter 1

Introduction

1.1 Structure of the report

The report consists of two introductory chapters, which provide a theoretical background for description of the algorithms, and three chapters dealing with the actual visibility algorithms.

This chapter provides an introduction to visibility by using a taxonomy of visibility problems and algorithms. The taxonomy is used to classify the later described visibility algorithms. Chapter 2 provides an analysis of visibility in 2D and 3D polygonal scenes. This analysis also includes formal description of visibility using Plücker coordinates of lines. Plücker coordinates are exploited later in algorithms for mutual visibility verification (Chapter 5).

Chapter 3 describes a visibility culling algorithm used to implement the online visibility culling module. This algorithm can be used accelerate rendering of fully dynamic scenes using recent graphics hardware. Chapter 4 describes global visibility sampling algorithm which forms a core of the PVS computation module. This chapter also describes view space partitioning algorithms used in close relation with the PVS computation. Finally, Chapter 5 describes mutual visibility verification algorithms, which are used by the PVS computation module to generate the final solution for precomputed visibility.

1.2 Domain of visibility problems

Computer graphics deals with visibility problems in the context of 2D, $2\frac{1}{2}$D, or 3D scenes. The actual problem domain is given by restricting the set of rays for which visibility should be determined.

Below we list common problem domains used and the corresponding domain restrictions:

1. visibility along a line
   (a) line
   (b) ray (origin + direction)

2. visibility from a point (from-point visibility)
   (a) point
   (b) point + restricted set of rays
      i. point + raster image (discrete form)
      ii. point + beam (continuous form)
3. visibility from a line segment (from-segment visibility)
   (a) line segment
   (b) line segment + restricted set of rays
4. visibility from a polygon (from-polygon visibility)
   (a) polygon
   (b) polygon + restricted set of rays
5. visibility from a region (from-region visibility)
   (a) region
   (b) region + restricted set of rays
6. global visibility
   (a) no further input (all rays in the scene)
   (b) restricted set of rays

The domain restrictions can be given independently of the dimension of the scene, but the impact of the restrictions differs depending on the scene dimension. For example, visibility from a polygon is equivalent to visibility from a (polygonal) region in 2D, but not in 3D.

1.3 Dimension of the problem-relevant line set

The six domains of visibility problems stated in Section 1.2 can be characterized by the problem-relevant line set denoted $\mathcal{L}_R$. We give a classification of visibility problems according to the dimension of the problem-relevant line set. We discuss why this classification is important for understanding the nature of the given visibility problem and for identifying its relation to other problems.

For the following discussion we assume that a line in primal space can be mapped to a point in line space. For purposes of the classification we define the line space as a vector space where a point corresponds to a line in the primal space[1].

1.3.1 Parametrization of lines in 2D

There are two independent parameters that specify a 2D line and thus the corresponding set of lines is two-dimensional. There is a natural duality between lines and points in 2D. For example a line expressed as: $l : y = ax + c$ is dual to a point $p = (-c, a)$. This particular duality cannot handle vertical lines. See Figure 1.1 for an example of other dual mappings in the plane. To avoid the singularity in the mapping, a line $l : ax + by + c = 0$ can be represented as a point $p_l = (a, b, c)$ in 2D projective space $\mathcal{P}^2$ [Sto91]. Multiplying p_l by a non-zero scalar we obtain a vector that represents the same line l. More details about this singularity-free mapping will be discussed in Chapter 2.

[1] A classical mathematical definition says: Line space is a direct product of two Hilbert spaces [Wei99]. However, this definition differs from the common understanding of line space in computer graphics [Dur99]

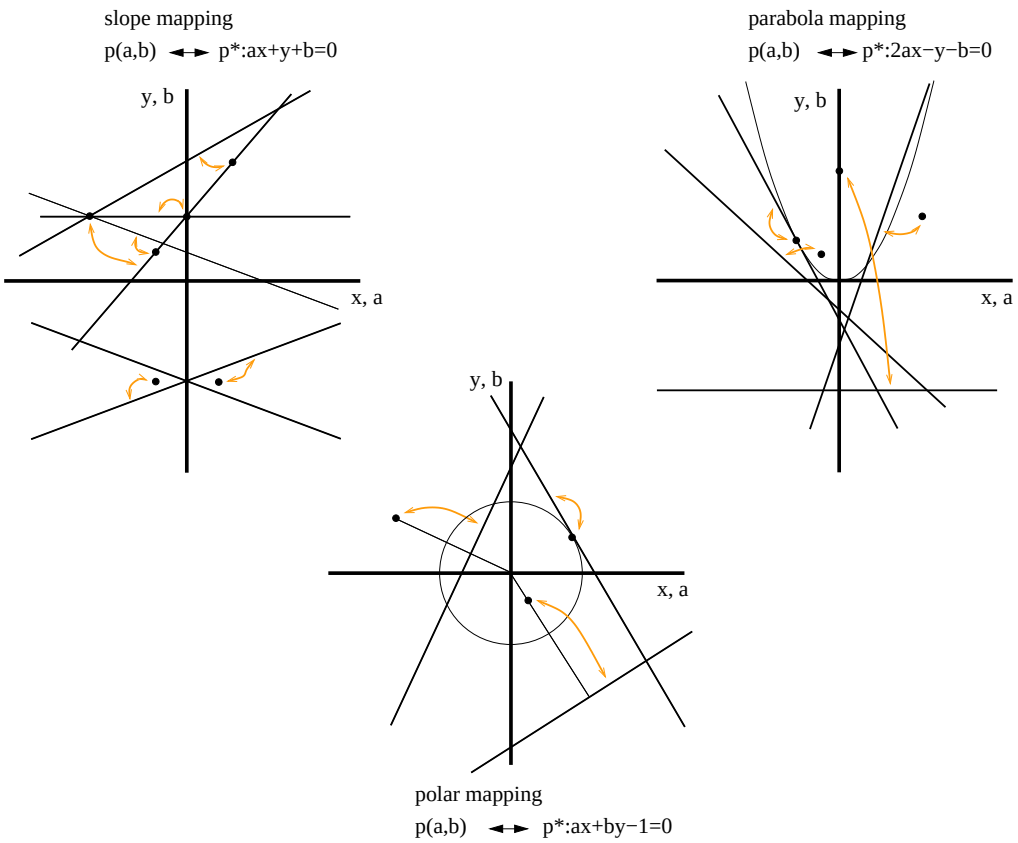


Figure 1.1: Duality between points and lines in 2D.

To sum up: In 2D there are two degrees of freedom in description of a line and the corresponding line space is two-dimensional. The problem-relevant line set $\mathcal{L}_R$ then forms a k-dimensional subset of $\mathcal{P}^2$, where $0 \leq k \leq 2$. An illustration of the concept of the problem-relevant line set is depicted in Figure 1.2.

1.3.2 Parametrization of lines in 3D

Lines in 3D form a four-parametric space [Pel97]. A line intersecting a given scene can be described by two points on a sphere enclosing the scene. Since the surface of the sphere is a two parametric space, we need four parameters to describe the line.

The two plane parametrization of 3D lines describes a line by points of intersection with the given two planes [GGC97]. This parametrization exhibits a singularity since it cannot describe lines parallel to these planes. See Figure 1.3 for illustrations of the spherical and the two plane parameterizations.

Another common parametrization of 3D lines are the Plücker coordinates. Plücker coordinates of an oriented 3D line are a six tuple that can be understood as a point in 5D oriented projective space [Sto91]. There are six coordinates in Plücker representation of a line although we know that the $\mathcal{L}_R$ is four-dimensional. This can be explained as follows:

- Firstly, Plücker coordinates are homogeneous coordinates of a 5D point. By multiplication of the coordinates by any positive scalar we get a mapping of the same line.
- Secondly, only 4D subset of the 5D oriented projective space corresponds to real lines. This subset is a 4D ruled quadric called the Plücker quadric or the Grassman manifold [Sto91, Pu98].

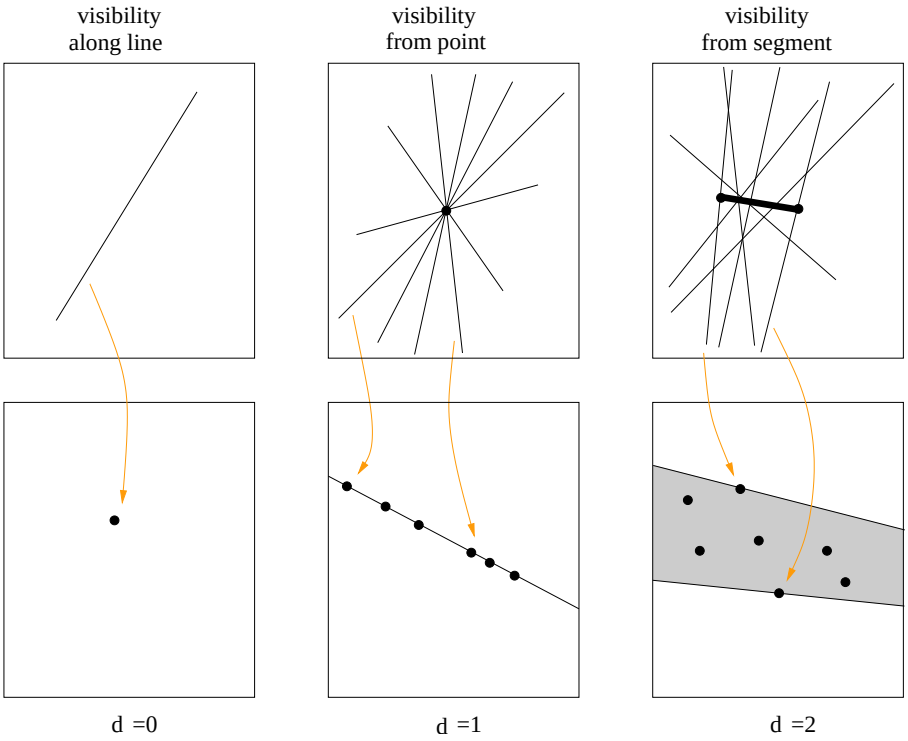


Figure 1.2: The problem-relevant set of lines in 2D. The $\mathcal{L}_R$ for visibility along a line is formed by a single point that is a mapping of the given line. The $\mathcal{L}_R$ for visibility from a point p is formed by points lying on a line. This line is a dual mapping of the point p. $\mathcal{L}_R$ for visibility from a line segment is formed by a 2D region bounded by dual mappings of endpoints of the given segment.

Although the Plücker coordinates need more coefficients they have no singularity and preserve some linearities: lines intersecting a set of lines in 3D correspond to an intersection of 5D hyperplanes. More details on Plücker coordinates will be discussed in Chapter 2 and Chapter 5 where they are used to solve the from-region visibility problem.

To sum up: In 3D there are four degrees of freedom in the description of a line and thus the corresponding line space is four-dimensional. Fixing certain line parameters (e.g. direction) the problem-relevant line set, denoted $\mathcal{L}_R$, forms a k-dimensional subset of $\mathcal{P}^4$, where $0 \leq k \leq 4$.

1.3.3 Visibility along a line

The simplest visibility problems deal with visibility along a single line. The problem-relevant line set is zero-dimensional, i.e. it is fully specified by the given line. A typical example of a visibility along a line problem is ray shooting.

A similar problem to ray shooting is the point-to-point visibility. The point-to-point visibility determines whether the line segment between two points is occluded, i.e. it has an intersection with an opaque object in the scene. Point-to-point visibility provides a visibility classification (answer 1a), whereas ray shooting determines a visible object (answer 2a) and/or a point of intersection (answer 3a). Note that the point-to-point visibility can be solved easily by means of ray shooting. Another constructive visibility along a line problem is determining the maximal free line segments on a given line. See Figure 1.4 for an illustration of typical visibility along a line problems.

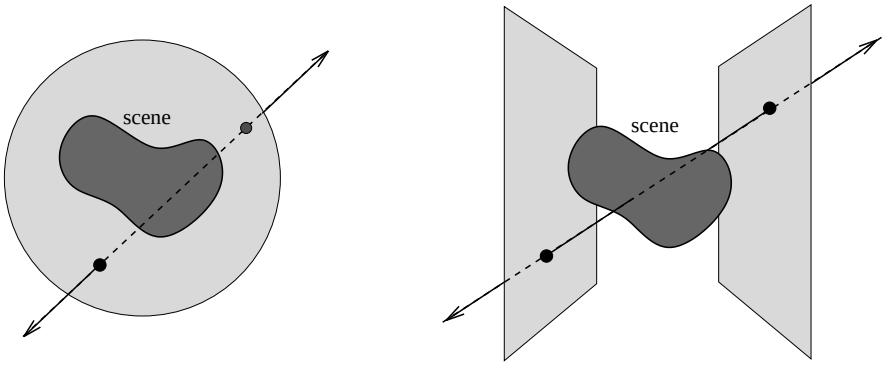


Figure 1.3: Parametrization of lines in 3D. (left) A line can be described by two points on a sphere enclosing the scene. (right) The two plane parametrization describes a line by point of intersection with two given planes.

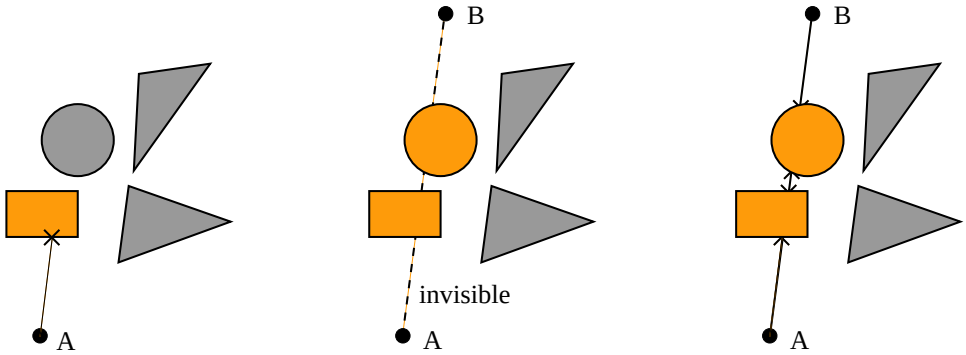


Figure 1.4: Visibility along a line. (left) Ray shooting. (center) Point-to-point visibility. (right) Maximal free line segments between two points.

1.3.4 Visibility from a point

Lines intersecting a point in 3D can be described by two parameters. For example the lines can be expressed by an intersection with a unit sphere centered at the given point. The most common parametrization describes a line by a point of intersection with a given viewport. Note that this parametrization accounts only for a subset of lines that intersect the viewport (see Figure 1.5).

In 3D the problem-relevant line set $\mathcal{L}_R$ is a 2D subset of the 4D line space. In 2D the $\mathcal{L}_R$ is a 1D subset of the 2D line space. The typical visibility from a point problem is the visible surface determination. Due to its importance the visible surface determination is covered by the majority of existing visibility algorithms. Other visibility from a point problem is the construction of the visibility map or the point-to-region visibility that classifies a region as visible, invisible, or partially visible with respect to the given point.

1.3.5 Visibility from a line segment

Lines intersecting a line segment in 3D can be described by three parameters. One parameter fixes the intersection of the line with the segment the other two express the direction of the line. The problem-relevant line set $\mathcal{L}_R$ is three-dimensional and it can be understood as a 2D cross section of $\mathcal{L}_R$ swept according to the translation on the given line segment (see Figure 1.6).

In 2D lines intersecting a line segment form a two-dimensional problem-relevant line set. Thus for the 2D case the $\mathcal{L}_R$ is a two-dimensional subset of 2D line space.

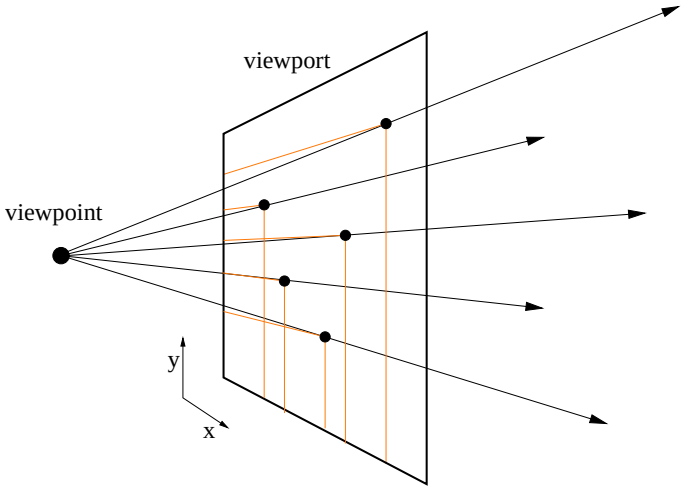


Figure 1.5: Visibility from a point. Lines intersecting a point can be described by a point of intersection with the given viewport.

1.3.6 Visibility from a region

Visibility from a region (or from-region visibility) involves the most general visibility problems. In 3D the $\mathcal{L}_R$ is a 4D subset of the 4D line space. In 2D the $\mathcal{L}_R$ is a 2D subset of the 2D line space. Consequently, in the presented classification visibility from a region in 2D is equivalent to visibility from a line segment in 2D.

A typical visibility from a region problem is the problem of region-to-region visibility that aims to determine if the two given regions in the scene are visible, invisible, or partially visible (see Figure 1.7). Another visibility from region problem is the computation of a potentially visible set (PVS) with respect to a given view cell. The PVS consists of a set of objects that are potentially visible from any point inside the view cell. Further visibility from a region problems include computing form factors between two polygons, soft shadow algorithms or discontinuity meshing.

1.3.7 Global visibility

According to the classification the global visibility problems can be seen as an extension of the from-region visibility problems. The dimension of the problem-relevant line set is the same ($k = 2$ for 2D and $k = 4$ for 3D scenes). Nevertheless, the global visibility problems typically deal with much larger set of rays, i.e. all rays that penetrate the scene. Additionally, there is no given set of reference points from which visibility is studied and hence there is no given priority ordering of objects along each particular line from $\mathcal{L}_R$. Therefore an additional parameter must be used to describe visibility (visible object) along each ray.

1.3.8 Summary

The classification of visibility problems according to the dimension of the problem-relevant line set is summarized in Table 1.1. This classification provides means for understanding how difficult it is to compute, describe, and maintain visibility for a particular class of problems. For example a data structure representing the visible or occluded parts of the scene for the visibility from a point problem needs to partition a 2D $\mathcal{L}_R$ into visible and occluded sets of lines. This observation conforms with the traditional visible surface algorithms – they partition a 2D viewport into empty/nonempty regions and associate each nonempty regions (pixels) with a visible object. In this case the viewport represents the $\mathcal{L}_R$ as each point of the viewport corresponds to a line through that point. To analytically describe visibility from a region a subdivision of 4D $\mathcal{L}_R$ should be performed. This is much more difficult than the 2D subdivision. Moreover the description of

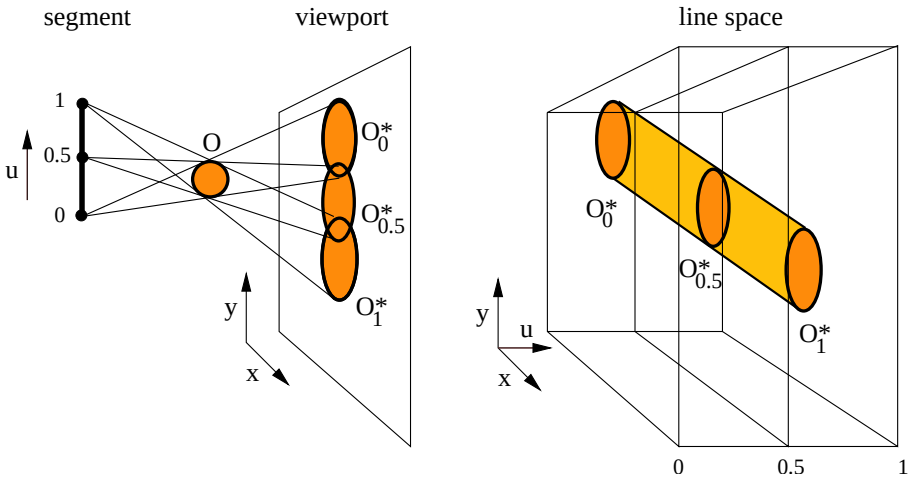


Figure 1.6: Visibility from a line segment. (left) Line segment, a spherical object O, and its projections O_0^*, $O_{0.5}^*$, O_1^* with respect to the three points on the line segment. (right) A possible parametrization of lines that stacks up 2D planes. Each plane corresponds to mappings of lines intersecting a given point on the line segment.

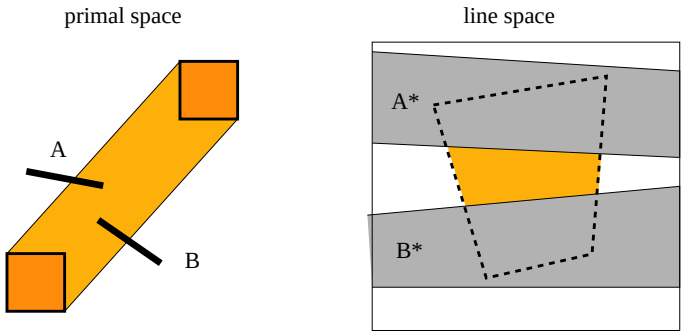


Figure 1.7: Visibility from a region — an example of the region-to-region visibility. Two regions and two occluders A, B in a 2D scene. In line space the region-to-region visibility can be solved by subtracting the sets of lines A^* and B^* intersecting objects A and B from the lines intersecting both regions.

visibility from a region involves non-linear subdivisions of both primal space and line space even for polygonal scenes [Tel92a, Dur99].

1.4 Classification of visibility algorithms

The taxonomy of visibility problems groups similar visibility problems in the same class. A visibility problem can be solved by means of various visibility algorithms. A visibility algorithm poses further restrictions on the input and output data. These restrictions can be seen as a more precise definition of the visibility problem that is solved by the algorithm.

Above we classified visibility problems according to the problem domain and the desired answers. In this section we provide a classification of visibility algorithms according to other important criteria characterizing a particular visibility algorithm.

2D		
domain	$d(\mathcal{L}_R)$	problems
visibility along a line	0	ray shooting, point-to-point visibility
visibility from a point	1	view around a point, point-to-region visibility
visibility from a line segment visibility from region global visibility	2	region-to-region visibility, PVS
3D		
domain	$d(\mathcal{L}_R)$	problems
visibility along a line	0	ray shooting, point-to-point visibility
from point in a surface	1	see visibility from point in 2D
visibility from a point	2	visible (hidden) surfaces, point-to-region visibility, visibility map, hard shadows
visibility from a line segment	3	segment-to-region visibility (rare)
visibility from a region global visibility	4	region-region visibility, PVS, aspect graph, soft shadows, discontinuity meshing

Table 1.1: Classification of visibility problems in 2D and 3D according to the dimension of the problem-relevant line set.

1.4.1 Scene restrictions

Visibility algorithms can be classified according to the restrictions they pose on the scene description. The type of the scene description influences the difficulty of solving the given problem: it is simpler to implement an algorithm computing a visibility map for scenes consisting of triangles than for scenes with NURBS surfaces. We list common restrictions on the scene primitives suitable for visibility computations:

- triangles, convex polygons, concave polygons,
- volumetric data,
- points,
- general parametric, implicit, or procedural surfaces.

Some attributes of scenes objects further increase the complexity of the visibility computation:

- transparent objects,
- dynamic objects.

The majority of analytic visibility algorithms deals with static polygonal scenes without transparency. The polygons are often subdivided into triangles for easier manipulation and representation.

1.4.2 Accuracy

Visibility algorithms can be classified according to the accuracy of the result as:

- exact,
- conservative,
- aggressive,
- approximate.

An exact algorithm provides an exact analytic result for the given problem (in practice however this result is typically influenced by the finite precision of the floating point arithmetics). A conservative algorithm overestimates visibility, i.e. it never misses any visible object, surface or point. An aggressive algorithm always underestimates visibility, i.e. it never reports an invisible object, surface or point as visible. An approximate algorithm provides only an approximation of the result, i.e. it can overestimate visibility for one input and underestimate visibility for another input.

The classification according to the accuracy is best illustrated on computing PVS: an exact algorithm computes an exact PVS. A conservative algorithm computes a superset of the exact PVS. An aggressive algorithm determines a subset of the exact PVS. An approximate algorithm computes an approximation to the exact PVS that is neither its subset or its superset for all possible inputs.

A more precise quality measure of algorithms computing PVSs can be expressed by the relative overestimation and the relative underestimation of the PVS with respect to the exact PVS. We can define a quality measure of an algorithm A on input I as a tuple $\boldsymbol{Q}^A(I)$:

$$\boldsymbol{Q}^A(I) = (Q_o^A(I), Q_u^A(I)), \qquad I \in \mathcal{D} \tag{1.1}$$

$$Q_o^A(I) = \frac{|S^A(I) \setminus S^{\mathcal{E}}(I)|}{|S^{\mathcal{E}}(I)|} \tag{1.2}$$

$$Q_u^A(I) = \frac{|S^{\mathcal{E}}(I) \setminus S^A(I)|}{|S^{\mathcal{E}}(I)|} \tag{1.3}$$

where I is an input from the input domain $\mathcal{D}$, $S^A(I)$ is the PVS determined by the algorithm A for input I and $S^{\mathcal{E}}(I)$ is the exact PVS for the given input. $Q_o^A(I)$ expresses the relative overestimation of the PVS, $Q_u^A(I)$ is the relative underestimation.

The expected quality of the algorithm over all possible inputs can be given as:

$$Q^A = E[\|\boldsymbol{Q}^A(I)\|] \tag{1.4}$$

$$= \sum_{\forall I \in \mathcal{D}} f(I).\sqrt{Q_o^A(I)^2 + Q_o^A(I)^2} \tag{1.5}$$

where f(I) is the probability density function expressing the probability of occurrence of input I. The quality measure $\boldsymbol{Q}^A(I)$ can be used to classify a PVS algorithm into one of the four accuracy classes according to Section 1.4.2:

1. exact
   $\forall I \in \mathcal{D} : Q_o^A(I) = 0 \wedge Q_u^A(I) = 0$
2. conservative
   $\forall I \in \mathcal{D} : Q_o^A(I) \geq 0 \wedge Q_u^A(I) = 0$
3. aggressive
   $\forall I \in \mathcal{D} : Q_o^A(I) = 0 \wedge Q_u^A(I) \geq 0$
4. approximate
   $\exists I_j, I_k \in \mathcal{D} : Q_o^A(I_j) > 0 \wedge Q_u^A(I_k) > 0$

1.4.3 Solution space

The solution space is the domain in which the algorithm determines the desired result. Note that the solution space does not need to match the domain of the result.

The algorithms can be classified as:

- discrete,
- continuous,
- hybrid.

A discrete algorithm solves the problem using a discrete solution space; the solution is typically an approximation of the result. A continuous algorithm works in a continuous domain and often computes an analytic solution to the given problem. A hybrid algorithm uses both the discrete and the continuous solution space.

The classification according to the solution space is easily demonstrated on visible surface algorithms: The z-buffer [Cat75] is a common example of a discrete algorithm. The Weiler-Atherton algorithm [WA77] is an example of a continuous one. A hybrid solution space is used by scan-line algorithms that solve the problem in discrete steps (scan-lines) and for each step they provide a continuous solution (spans).

Further classification reflects the semantics of the solution space. According to this criteria we can classify the algorithms as:

- primal space (object space),
- line space,
  - image space,
  - general,
- hybrid.

A primal space algorithm solves the problem by studying the visibility between objects without a transformation to a different solution space. A line space algorithm studies visibility using a transformation of the problem to line space. Image space algorithms can be seen as an important subclass of line space algorithms for solving visibility from a point problems in 3D. These algorithms cover all visible surface algorithms and many visibility culling algorithms. They solve visibility in a given image plane that represents the problem-relevant line set $\mathcal{L}_R$ — each ray originating at the viewpoint corresponds to a point in the image plane.

The described classification differs from the sometimes mentioned understanding of image space and object space algorithms that incorrectly considers all image space algorithms discrete and all object space algorithms continuous.

1.5 Summary

The presented taxonomy classifies visibility problems independently of their target application. The classification should help to understand the nature of the given problem and it should assist in finding relationships between visibility problems and algorithms in different application areas. The algorithms address the following classes of visibility problems:

- Visibility from a point in 3D $d(\mathcal{L}_R) = 2$.
- Global visibility in 3D $d(\mathcal{L}_R) = 4$.
- Visibility from a region in 3D, $d(\mathcal{L}_R) = 4$.

This chapter discussed several important criteria for the classification of visibility algorithms. This classification can be seen as a finer structuring of the taxonomy of visibility problems. We discussed important steps in the design of a visibility algorithm that should also assist in understanding the quality of a visibility algorithm. According to the classification the visibility algorithms described later in the report address algorithms with the following properties:

- Domain:
  - viewpoint (online visibility culling),
  - global visibility (global visibility sampling)
  - polygon or polyhedron (mutual visibility verification)
- Scene restrictions (occluders):
  - meshes consisting of convex polygons
- Scene restrictions (group objects):
  - bounding boxes
- Output:
  - PVS
- Accuracy:
  - conservative
  - exact
  - aggresive
- Solution space:
  - discrete (online visibility culling, global visibility sampling, conservative and approximate algorithm from the mutual visibility verification)
  - continuous (exact algorithm from mutual visibility verification)
- Solution space data structures: viewport (online visibility culling), ray stack (global visibility sampling, conservative and approximate algorithm from the mutual visibility verification), BSP tree (exact algorithm from the mutual visibility verification)
- Use of coherence of visibility:
  - spatial coherence (all algorithms)
  - temporal coherence (online visibility culling)
- Output sensitivity: expected in practice (all algorithms)
- Acceleration data structure: kD-tree (all algorithms)
- Use of graphics hardware: online visibility culling

Chapter 2

Analysis of Visibility in Polygonal Scenes

This chapter provides analysis of the visibility in polygonal scenes, which are the input for all developed algorithms. The visibility analysis uncoveres the complexity of the from-region and global visibility problems and thus it especially important for a good design of the global visibility preprocessor. To better undestand the visibility relationships in primal space we use mapping to line space, where the visibility interactions can be observed easily by interactions of sets of points. Additionally for the sake of clarity, we first analyze visibility in 2D and then extend the analysis to 3D polygonal scenes. Visibility in 3D is described using Plücker

2.1 Analysis of visibility in 2D

The proposed visibility algorithm uses a mapping of oriented 2D lines to points in 2D oriented projective space — line space. Such a mapping allows to handle sets of lines much easier than in the primal space [PV93].

We use a 2D projection of Plücker coordinates [Sto91] to parametrize lines in the plane. This mapping corresponds to an "oriented form" of the duality between points and lines in 2D. Let l be an oriented line in $\mathcal{R}^2$ and let $u = (u_x, u_y)$ and $v = (v_x, v_y)$ be two distinct points lying on l. Line l oriented from u to v can be described by the following matrix:

$$M_l = \begin{pmatrix} u_x & u_y & 1 \\ v_x & v_y & 1 \end{pmatrix}$$

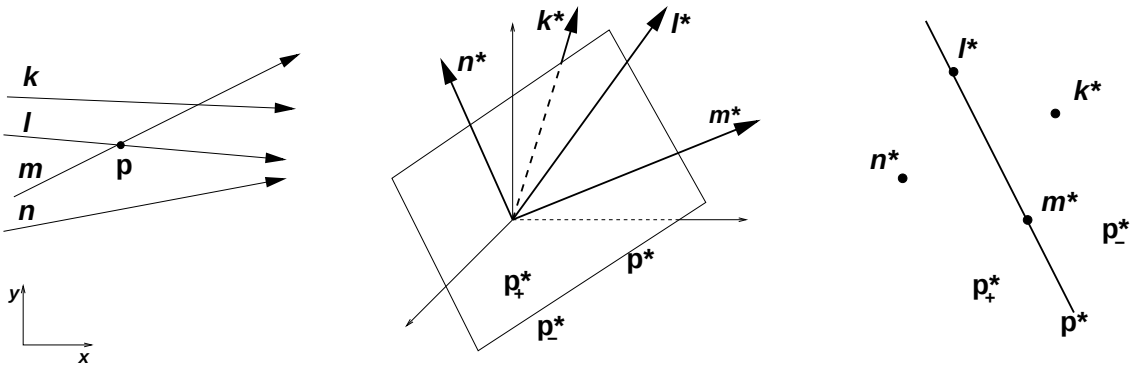
Plücker coordinates l^* of l are minors of M_l:

$$l^* = (l^*_x, l^*_y, l^*_z) = (u_y - v_y, v_x - u_x, u_x v_y - u_y v_x).$$

l^* can be interpreted as homogeneous coordinates of a point in 2D oriented projective space $\mathcal{P}^2$. Two oriented lines are equal if and only if their Plücker coordinates differ only by a positive scale factor. l^* also corresponds to coefficients of the implicit equation of a line: l' expressed as $l' : ax + by + c = 0$ induces two oriented lines l^*_1, l^*_2, with Plücker coordinates $l^*_1 = (a, b, c)$ and $l^*_2 = -(a, b, c)$. The Plücker coordinates of 2D lines defined in this chapter are a simplified form of the Plücker coordinates for 3D lines, which will be discussed below in this chapter: Plücker coordinates of a 2D line correspond to the Plücker coordinates of a 3D line embedded in the $z = 0$ plane after removal of redundant coordinates (equal to 0) and permutation of the remaining ones (including some sign changes).

Homogeneous coordinates are often normalized, e.g. $l^*_N = (a/b, 1, c/b)$. The normalization introduces a singularity — in our example vertical lines map to points at infinity. To avoid singularities we treat $\mathcal{P}^2$ as 3D linear space and call it line space denoted $\mathcal{L}$. Consequently, l^* represents a halfline in $\mathcal{L}$. All points on halfline l^* represent the same oriented line l.

To sum up: an oriented line in 2D is mapped to a halfline beginning at the origin in 3D. An example of the concept is depicted in Figures 2.1-(a) and 2.1-(b). Further in this chapter we will



(a) (b) (c)

Figure 2.1: (a) Four oriented lines in primal space. (b) Mappings of the four lines and point p. Lines intersecting p map to plane p^*. Lines passing clockwise (counterclockwise) around p, map to p^*_- (p^*_+). (c) The situation after projection to a plane perpendicular to p^*.

mostly use "projected" 2D illustrations of line space (such as in Figure 2.1-(c)). We will still talk about planes and halflines, but they will be depicted as lines and points, respectively, for the sake of clarity of the presentation.

2.1.1 Lines passing through a point

A pencil of oriented lines passing through a point $p = (p_x, p_y) \in \mathcal{R}^2$ maps to an oriented plane p^* in line space that is expressed as:

$$p^* = \{(x, y, z) | (x, y, z) \in \mathcal{L}, p_x x + p_y y + z = 0\}.$$

This plane subdivides $\mathcal{L}$ in two open halfspaces p^*_+ and p^*_-. Points in p^*_- correspond to oriented lines passing clockwise around p (see Figure 2.1). Points in p^*_+ correspond to oriented lines passing counterclockwise around p (these relations depend on the orientation of the primal space). We denote $-p^*$ an oriented plane opposite to p^* that can be expressed as:

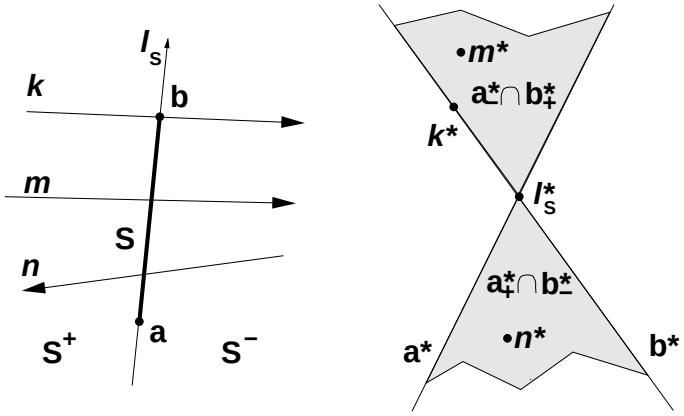
$$-p^* = \{(x, y, z) | (x, y, z) \in \mathcal{L}, -p_x x - p_y y - z = 0\}.$$

2.1.2 Lines passing through a line segment

Oriented lines passing through a line segment can be decomposed into two sets depending on their orientation. Consider a supporting line l_S of a line segment S, that partitions the plane in open halfspaces S^+ and S^-. Denote a and b the two endpoints of S and a^* and b^* their mappings to $\mathcal{L}$. Lines that intersect S and "come from" S^- can be expressed in line space as an intersection of halfspaces a^*_+ and b^*_-. The opposite oriented lines intersecting S are expressed as $a^*_- \cap b^*_+$ (see Figure 2.2-(a,b)).

2.1.3 Lines passing through two line segments

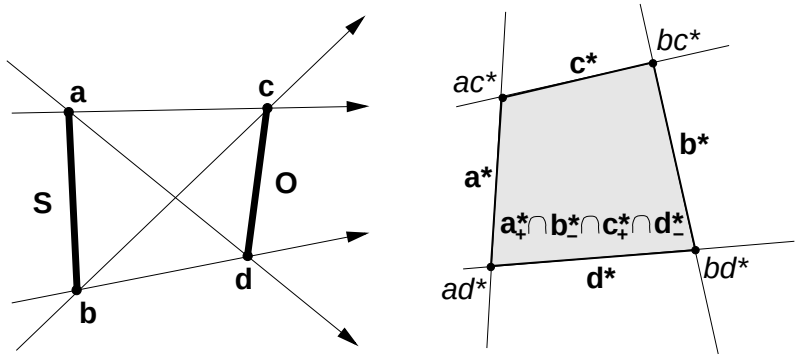
Consider two disjoint line segments such as those depicted in Figure 2.3-(a). The set of lines passing through the two line segments can be described as an intersection of four halfspaces in line space. The four halfspaces correspond to mappings of endpoints of the two line segments. Since the halfspaces pass through the origin of $\mathcal{L}$, their intersection is a pyramid with the apex at the origin. The boundary halflines of the pyramid correspond to mappings of the four extremal lines induced by the two segments. Denote $\mathcal{P}(S, O)$ a line space pyramid corresponding to oriented lines intersecting line segments S and O in this order. We represent the pyramid by a blocker



(a) (b)

Figure 2.2: (a) A line segment S and three oriented lines that intersect S. (b) The situation in line space: the projection of two wedges corresponding to lines intersecting S. Mappings of supporting line l_S of S are two halflines that project to point l^*_S. Line k intersects point b and therefore maps to plane b^*. Lines m and n map to the wedge corresponding to their orientation.

polygon $B(S, O)$ (see Figure 2.3-(b)). Since $B(S, O)$ only represents the pyramid $\mathcal{P}(S, O)$, it need not be a planar polygon, i.e. its vertices may lay anywhere on the boundary halflines of $\mathcal{P}(S, O)$. We normalize the vertex coordinates: they correspond to an intersection of the boundary halfline of $\mathcal{P}(S, O)$ and the unit sphere centered at the origin of $\mathcal{L}$.



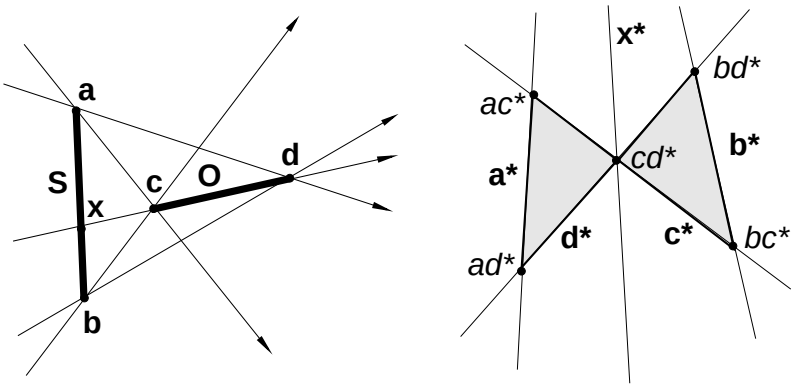
(a) (b)

Figure 2.3: (a) Two line segments and corresponding four extremal lines oriented from S to O. Separating lines ad and bc bound region of partial visibility of S behind O (penumbra). Supporting lines ac and bd bound region where S is invisible (umbra). (b) Blocker polygon $B(S, O)$ representing pyramid $\mathcal{P}(S, O)$.

In Figure 2.4-(a), the supporting line of cd intersects ab at point x. The set of rays passing through ab and cd can be decomposed to rays passing through ax and cd, and through xb and cd. Rays through ax and cd map to a pyramid that is described by intersection of only three halfspaces induced by mappings of a, x and d. Rays through xb and cd can be described similarly. The configuration in line space is depicted in Figure 2.4-(b).

2.1.4 Lines passing through a set of line segments

Consider a set of $n+1$ line segments. We call one line segment the source (denoted by S) and the other n segments we call occluders (denoted by O_k, $1 \leq k \leq n$). Further in the chapter we will



(a) (b)

Figure 2.4: (a) Degenerate configuration of line segments: the supporting line of cd intersects ab at point x. There are five extremal lines. Note, that there is no umbra region. (b) In line space the configuration yields two pyramids sharing a boundary that is a mapping of the oriented line cd.

use the term ray as a representative of an oriented line that is oriented from the source "towards" the occluders.

Assume that we can process all occluders in a strict front-to-back order with respect to the given source. We have already processed k occluders and we continue by processing O_{k+1}. O_{k+1} can be visible through rays that correspond to the pyramid $\mathcal{P}(S, O_{k+1})$. Nevertheless some of these rays can be blocked by combination of already processed occluders O_x $(1 \leq x \leq k)$. To determine if O_{k+1} is visible we subtract all $\mathcal{P}(S, O_x)$ from $\mathcal{P}(S, O_{k+1})$:

$$\mathcal{V}(S, O_{k+1}) = \mathcal{P}(S, O_{k+1}) \; - \bigcup_{1 \leq x \leq k} \mathcal{P}(S, O_x)$$

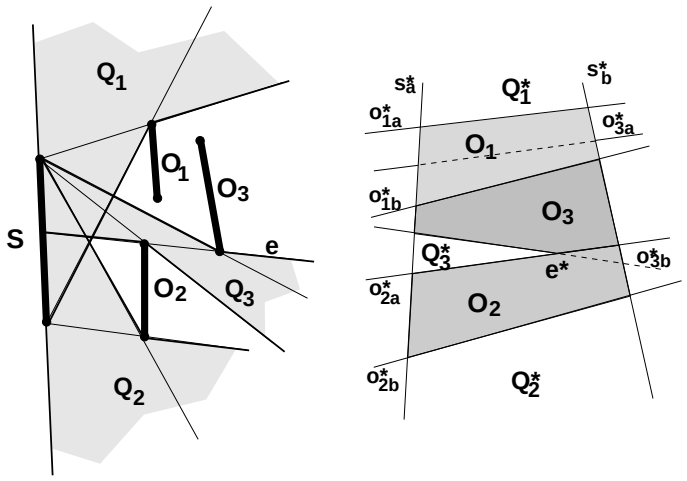
$\mathcal{V}(S, O_{k+1})$ is a set pyramids representing rays through which O_{k+1} is visible from S. In turn, all rays corresponding to $\mathcal{V}(S, O_{k+1})$ are blocked behind O_{k+1}. If $\mathcal{V}(S, O_{k+1})$ is an empty set, occluder O_{k+1} is invisible. This suggest incremental construction of an arrangement of pyramids $\mathcal{A}_k$ that corresponds to rays blocked by the k processed occluders. We determine $\mathcal{V}(S, O_{k+1})$ and $\mathcal{A}_{k+1}$ ($\mathcal{A}_0$ is empty):

$$\mathcal{V}(S, O_{k+1}) = \mathcal{P}(S, O_{k+1}) \; - \; \mathcal{A}_k,$$
$$\mathcal{A}_{k+1} = \mathcal{A}_k \cup \mathcal{P}(S, O_{k+1}) = \mathcal{A}_k \; \cup \; \mathcal{V}(S, O_{k+1}).$$

Figures 2.5-(a,b) depict a projection of an arrangement $\mathcal{A}_3$ of a source and three occluders. Note that the shorter the source line segment the narrower (s_a^* and s_b^* get closer) are the pyramids $\mathcal{P}(S, O_k)$.

Recall that the pyramid $\mathcal{P}(S, O_k)$ is represented by blocker polygon $B(S, O_k)$. The construction of the arrangement $\mathcal{A}_k$ resembles the from-point visibility problem, more specifically the hidden surface removal applied on the blocker polygons with respect to the origin of $\mathcal{L}$. The difference is that the depth information is irrelevant in line space. The priority of blocker polygons is either completely determined by the processing order of occluders or their depth must be compared in primal space. This observation is supported by the classification of visibility problems presented in Chapter 1. Visibility from point in 3D and visibility from region in 2D induce a two-dimensional problem-relevant line set. This suggests the possibility of mapping one problem to another.

In the next section we show how the arrangement $\mathcal{A}_k$ can be maintained consistently and efficiently using the occlusion tree.



(a) (b)

Figure 2.5: (a) The source line segment S and three occluders. Q_{1-3} denote unoccluded funnels. (b) The line space subdivision. For each cell, the corresponding occluder-sequence is depicted. Note the cells Q_1^*, Q_2^* and Q_3^* corresponding to unoccluded funnels.

2.2 Plücker coordinates of lines in 3D

We will use a mapping that describes an oriented 3D line as a point in a projective 5D space [BY98] by means of Plücker coordinates [Tel92b, Pel97, YN97, Pu98]. Plücker coordinates allow to represent sets of lines using 5D polyhedra and to compute visibility by means of polyhedra set operations in 5D.

A line in 3D can be described by homogeneous coordinates of two distinct points on that line. Let l be a line in $\mathcal{R}^3$ and let $\boldsymbol{u} = (u_x, u_y, u_z, u_w)$ and $\boldsymbol{v} = (v_x, v_y, v_z, v_w)$ be two distinct points in homogeneous coordinates lying on l. A line l oriented from $\boldsymbol{u}$ to $\boldsymbol{v}$ can be described by the following matrix:

$$l = \begin{pmatrix} u_x & u_y & u_z & u_w \\ v_x & v_y & v_z & v_w \end{pmatrix} \tag{2.1}$$

Minors of the matrix correspond to components of the Plücker coordinates $\boldsymbol{\pi}_l$ of line l:

$$\begin{aligned} \boldsymbol{\pi}_l &= (\pi_{l0}, \pi_{l1}, \pi_{l2}, \pi_{l3}, \pi_{l4}, \pi_{l5}) = \\ &= (\xi_{wx}, \xi_{wy}, \xi_{wz}, \xi_{yz}, \xi_{zx}, \xi_{xy}), \end{aligned} \tag{2.2}$$

where

$$\xi_{rs} = det \begin{pmatrix} u_r & u_s \\ v_r & v_s \end{pmatrix}. \tag{2.3}$$

Substituting $u_w = 1$ and $v_w = 1$ into Eq. 2.2 enumerates to:

$$\begin{aligned} \pi_{l0} &= v_x - u_x \\ \pi_{l1} &= v_y - u_y \\ \pi_{l2} &= v_z - u_z \\ \pi_{l3} &= u_y v_z - u_z v_y \\ \pi_{l4} &= u_z v_x - u_x v_z \\ \pi_{l5} &= u_x v_y - u_y v_x \end{aligned} \tag{2.4}$$

The Plücker coordinates $\boldsymbol{\pi}_l$ can be seen as homogeneous coordinates of a point in a projective five-dimensional space $\mathcal{P}^5$. We call this point a Plücker point $\hat{\boldsymbol{\pi}}_l$ of l. For a given oriented line l the Plücker coordinates $\boldsymbol{\pi}_l$ are unique and they do not depend on the choice of points p and

q. We will use the notation of a Plücker point $\hat{\boldsymbol{\pi}}_l$ in the case when we want to stress that the corresponding Plücker coordinates $\boldsymbol{\pi}_l$ are interpreted as a point in $\mathcal{P}^5$.

Using the projective duality the Plücker coordinates can be interpreted as coefficients of a hyperplane. The Plücker coefficients $\boldsymbol{\omega}_l$ of line l are given as:

$$\begin{aligned} \boldsymbol{\omega}_l &= (\omega_{l0}, \omega_{l1}, \omega_{l2}, \omega_{l3}, \omega_{l4}, \omega_{l5}) = \\ &= (\xi_{yz}, \xi_{zx}, \xi_{xy}, \xi_{wx}, \xi_{wy}, \xi_{wz}) \end{aligned} \tag{2.5}$$

Substituting Eq. 2.4 into Eq. 2.5 we get:

$$\begin{aligned} \omega_{l0} &= \pi_{l3} \\ \omega_{l1} &= \pi_{l4} \\ \omega_{l2} &= \pi_{l5} \\ \omega_{l3} &= \pi_{l0} \\ \omega_{l4} &= \pi_{l1} \\ \omega_{l5} &= \pi_{l2} \end{aligned} \tag{2.6}$$

The Plücker coefficients $\boldsymbol{\omega}_l$ define a Plücker hyperplane $\hat{\boldsymbol{\omega}}_l$. We will use the notation of a Plücker hyperplane $\hat{\boldsymbol{\omega}}_l$ when we want to stress that the corresponding Plücker coefficients $\boldsymbol{\omega}_l$ are interpreted as a hyperplane in $\mathcal{P}^5$. In terms of Plücker points the Plücker hyperplane can be expressed as:

$$\hat{\boldsymbol{\omega}}_l = \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\omega}_l \odot \boldsymbol{\pi} = 0\} \tag{2.7}$$

The Plücker hyperplane induces closed positive and negative halfspaces given as:

$$\begin{aligned} \hat{\boldsymbol{\omega}}_l^+ &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\omega}_l \odot \boldsymbol{\pi} \geq 0\} \\ \hat{\boldsymbol{\omega}}_l^- &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\omega}_l \odot \boldsymbol{\pi} \leq 0\} \end{aligned} \tag{2.8}$$

These definitions of Plücker coordinates and coefficients follow the "traditional" convention [Pu98]. They differ from the definitions used by Teller [Tel92b] who used a permuted order of the coordinates. The traditional convention provides an elegant interpretation of Plücker coordinates that will be discussed in Section 2.2.1.

If a and b are two directed lines, the relation $side(a, b)$ is defined as an inner product $\boldsymbol{\omega}_a \odot \boldsymbol{\pi}_b$ or permuted inner product $\boldsymbol{\pi}_a \times \boldsymbol{\pi}_b$:

$$\begin{aligned} side(a,b) &= \boldsymbol{\omega}_a \odot \boldsymbol{\pi}_b = \\ &= \omega_{a0}\pi_{b0} + \omega_{a1}\pi_{b1} + \omega_{a2}\pi_{b2} + \omega_{a3}\pi_{b3} + \omega_{a4}\pi_{b4} + \omega_{a5}\pi_{b5} = \\ &= \boldsymbol{\pi}_a \times \boldsymbol{\pi}_b = \\ &= \pi_{a0}\pi_{b3} + \pi_{a1}\pi_{b4} + \pi_{a2}\pi_{b5} + \pi_{a3}\pi_{b0} + \pi_{a4}\pi_{b1} + \pi_{a5}\pi_{b2} \end{aligned} \tag{2.9}$$

This relation can be interpreted with the right-hand rule (Figure 2.6). If the thumb of the right hand is directed along line a, then:

- $side(a, b) > 0$, if line b is oriented in the direction of the fingers,
- $side(a, b) = 0$, if lines a and b intersect or are parallel,
- $side(a, b) < 0$, if line b points against the direction of the fingers.

Plücker coordinates have an important property: Although every oriented line in $\mathcal{R}^3$ maps to a point in Plücker coordinates, not every tuple of six real numbers corresponds to a real line. Only the points $\hat{\boldsymbol{\pi}} \in \mathcal{P}^5$ Plücker coordinates of which satisfy the condition

$$\boldsymbol{\pi} \odot \boldsymbol{\pi} = 0 \qquad \equiv \qquad \pi_0\pi_3 + \pi_1\pi_4 + \pi_2\pi_5 = 0, \tag{2.10}$$

represent real lines in $\mathcal{R}^3$. All other points correspond to lines which are said to be imaginary. The set of points in $\mathcal{P}^5$ satisfying Eq. 2.10 forms a 4D hyperboloid of one sheet called the Plücker quadric, also known as the Klein quadric or the Grassman manifold (see Figure 2.7).

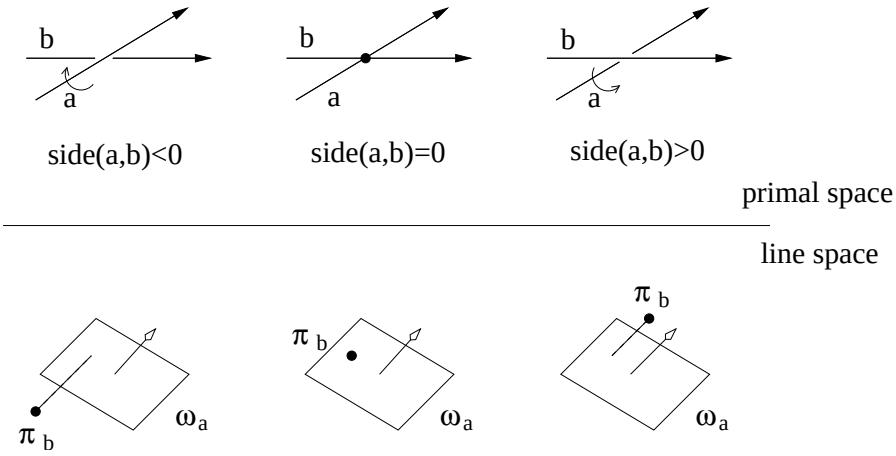


Figure 2.6: The $side(a, b)$, interpreted as the right-hand rule.

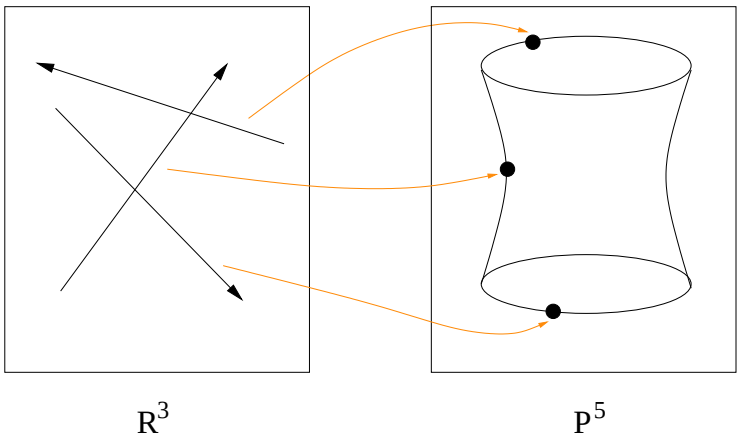


Figure 2.7: Real lines map on points on the Plücker quadric.

The six Plücker coordinates of a real line are not independent. Firstly, they describe an oriented projective space, secondly, they must satisfy the equation 2.10. Thus there are four degrees of freedom in the description of a 3D line, which conforms with the classification from Chapter 1.

Plücker coordinates allow to detect an incidence of two lines by computing an inner product of a homogeneous point (mapping of one line) with a hyperplane (mapping of the other). Lines l and l' intersect or are parallel (i.e. meet at infinity) if and only if $\hat{\boldsymbol{\pi}}_l \in \hat{\boldsymbol{\omega}}_{l'}$, i.e. $side(l, l') = 0$. Note that according to 2.10 any line always meets itself.

2.2.1 Geometric interpretation of Plücker coordinates

For a better understanding of Plücker coordinates it is natural to ask how each individual Plücker coordinate is related to the geometry of the corresponding line. The Plücker coordinates of a given line can be divided to the directional and the locational parts. The directional part encodes the direction of the line, the locational part encodes the position of the line. Given Plücker coordinates $\boldsymbol{\pi}_l$ of a line l we can write:

$$\begin{aligned} \boldsymbol{d}_l &= (\pi_{l0}, \pi_{l1}, \pi_{l2}), \\ \boldsymbol{l}_l &= (\pi_{l3}, \pi_{l4}, \pi_{l5}), \end{aligned} \tag{2.11}$$

where $\boldsymbol{d}_l$ is the directional vector of l and $\boldsymbol{l}_l$ is the locational vector of l. The Plücker coordinates

$\boldsymbol{\pi}_l$ and the Plücker coefficients $\boldsymbol{\omega}_l$ can be expressed as:

$$\begin{aligned} \boldsymbol{\pi}_l &= [\boldsymbol{d}_l; \boldsymbol{l}_l], \\ \boldsymbol{\omega}_l &= [\boldsymbol{l}_l; \boldsymbol{d}_l]. \end{aligned} \tag{2.12}$$

Extracting a point

Often we need to describe a line using a parametric representation by means of an anchor point and a directional vector. Given a line l the directional vector $\boldsymbol{d}_l$ is embedded in the Plücker coordinates of l (see Eq. 2.12). The anchor point $\boldsymbol{a}_l$ can be computed as:

$$\boldsymbol{a}_l = (a_x, a_y, a_z) = \frac{\boldsymbol{d}_l \times \boldsymbol{l}_l}{\parallel \boldsymbol{d}_l \parallel^2}. \tag{2.13}$$

Computing the distance

The distance between two lines l and l' can be expressed using their anchor points and the directional vectors:

$$dist(l, l') = \frac{|(\boldsymbol{a}_l - \boldsymbol{a}_{l'}) \cdot (\boldsymbol{d}_l \times \boldsymbol{d}_{l'})|}{\parallel \boldsymbol{d}_l \times \boldsymbol{d}_{l'} \parallel}. \tag{2.14}$$

The distance is the length of the projection of the line segment $\boldsymbol{a}_l, \boldsymbol{a}_{l'}$ onto the direction $\boldsymbol{d}_l \times \boldsymbol{d}_{l'}$.

2.3 Visual events

This section discusses visual events occurring in polygonal scenes [GM90]. We will focus on the boundaries of visual events and their relation to Plücker coordinates. The understanding of the visual events helps to comprehend the complexity of the from-region visibility in 3D.

Any scene can be decomposed into regions from which the scene has a topologically equivalent view [GM90]. Boundaries of such regions correspond to event surfaces. Crossing an event surface causes a visual event, i.e. a change in the topology of the view (visibility map). In polygonal scenes there are three types of event surfaces [GM90]:

- vertex-edge (VE) events involving an edge and a vertex of two distinct polygons.
- edge-edge-edge (EEE) events involving three edges of three distinct polygons.
- supporting events corresponding to supporting planes of scene polygons. The supporting event can be seen as a degenerated case of VE or EEE events.

The VE events correspond to planes, the EEE events in general form quadratic surfaces. The definitions assume that the scene polygons are in general non-degenerate position. In real world scenes the polygons or their edges polygons can be variously aligned. In such a case these definitions of visibility events form minimal sets of edges and vertices defining an event. For example a VE event can involve a vertex and several edges of scene polygons (see Figure 2.8).

The intersections of event surfaces correspond to extremal lines [Tel92b]. An extremal line intersects four edges of some scene polygons. There are four types of extremal lines: vertex-vertex (VV) lines, vertex-edge-edge (VEE) lines, edge-vertex-edge (EVE) and quadruple edge (4E) lines. Imagine "sliding" an extremal line (of any type) away from its initial position by relaxing exactly one of the four edge constraints determining the line. The section of the event surface swept out by the sliding line is called the swath. A swath is either planar if it corresponds to a VE event surface or a regulus if it is embedded in an EEE event surface.

Figure 2.9-(a) shows an extremal VV line tight on four edges A,B,C, and D. Relaxing constraint C yields a VE (planar) swath defined by A,B, and D. When the sliding line encounters an obstacle (edge E) it terminates at a VV extremal line defined by A,B,D, and E. Figure 2.9-(b) depicts an

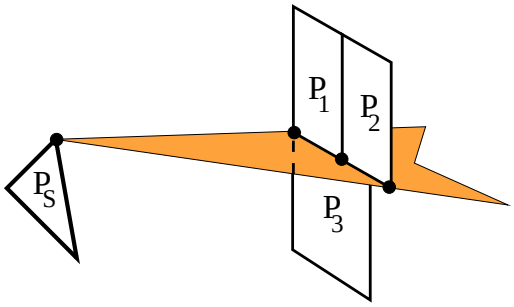


Figure 2.8: Degenerated VE event. The VE event is induced by a vertex and three edges of scene polygons.

extremal 4E line tight on the mutually skew edges A,B,C, and D. Relaxing constraint A produces an EEE event surface that is a regulus intersecting B,C, and D. When the sliding line encounters edge E the swath terminates at an VEE extremal line.

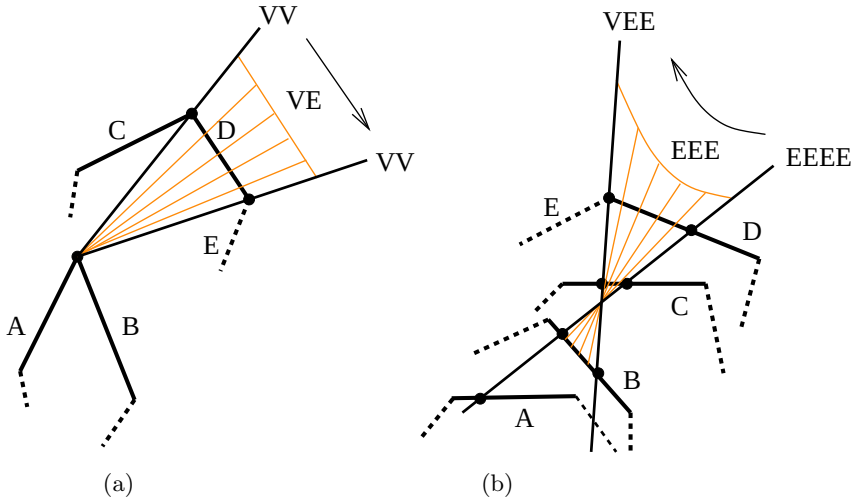


Figure 2.9: Swaths of event surfaces. (a) VE swath. (b) EEE swath.

2.3.1 Visual events and Plücker coordinates

Plücker coordinates allow an elegant description of event surfaces. An event surface can be expressed as an intersection of three Plücker hyperplanes, and thus avoiding explicit treatment of quadratic surfaces. The non-linear EEE surfaces correspond to curves embedded in the intersection of the Plücker hyperplanes.

Let $\mathcal{H}$ be an arrangement [GO97] of hyperplanes in $\mathcal{P}^5$ that correspond to Plücker coefficients of edges of the scene polygons. The intersection of the arrangement $\mathcal{H}$ and the Plücker quadric yields all visual events [Tel92b, Pel97, Pu98].

An extremal line l intersects four generator edges. Consequently, the corresponding Plücker point $\hat{\boldsymbol{\pi}}_l$ lies on four Plücker hyperplanes. In 5D the four hyperplanes define an edge of the arrangement $\mathcal{H}$. Thus, we can find all extremal lines of a given set of polygons by examining the edges of $\mathcal{H}$ for intersections with the Plücker quadric [Pu98].

Consider the situation depicted in Figure 2.9. In line space the event surfaces correspond to curves embedded in the Plücker quadric. In general these curves are conics defined by an intersection of the 2D-faces of $\mathcal{H}$ with the Plücker quadric (see Figure 2.10).

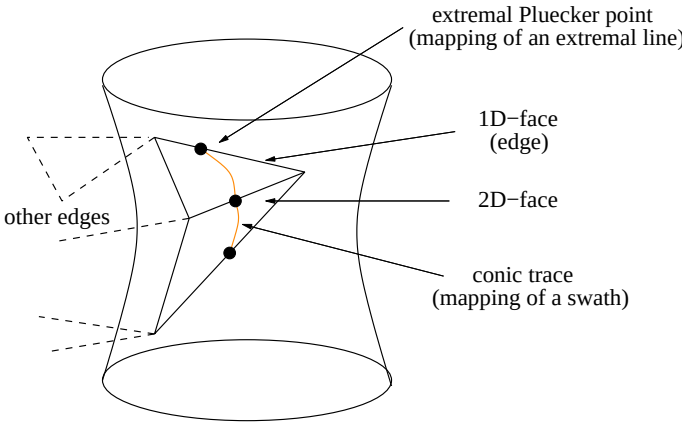


Figure 2.10: 3D swaths correspond to conics on the Plücker quadric.

2.4 Lines intersecting a polygon

Plücker coordinates provide a tool to map lines from primal space to points in line space. This mapping allows to perform operations of sets of lines using set theoretical operations on the corresponding sets of points. In polygonal scenes the elementary set of lines is formed by lines intersecting a given polygon.

Assume that a convex polygon P is defined by edges $e_i, 0 \leq i < n$ that are oriented counter-clockwise. The set of lines $\mathcal{L}_P$ intersecting the polygon that are oriented in the direction of the polygon's normal satisfies:

$$\mathcal{L}_P = \{l | l \in (R^3, R^3), side(\boldsymbol{\pi}_l, \boldsymbol{\pi}_{e_i}) \leq 0, \forall i \in \langle 0, n)\}, \tag{2.15}$$

where $\boldsymbol{\pi}_l$ are Plücker coordinates of line l and $\boldsymbol{\pi}_{e_i}$ are Plücker coordinates of i-th edge of the polygon. Substituting the Eq. 2.9 and rewriting the equation in terms of a set of Plücker points we get:

$$\begin{aligned} \mathcal{F}_P &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\pi} \times \boldsymbol{\pi}_{e_i} \leq 0, \forall i \in \langle 0, n)\} = \\ &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\pi} \odot \boldsymbol{\omega}_{e_i} \leq 0, \forall i \in \langle 0, n)\}, \end{aligned} \tag{2.16}$$

where $\mathcal{F}_P$ is a set of feasible Plücker points for polygon P. Substituting Eq. 2.8 into 2.16 we obtain:

$$\mathcal{F}_P = \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\pi} \in \hat{\boldsymbol{\omega}}_{e_i}^-, \forall i \in \langle 0, n)\} \tag{2.17}$$

Thus the set of feasible Plücker points is defined by an intersection of halfspaces defined by the Plücker hyperplanes corresponding to edges of the polygon. The set of stabbers $\mathcal{S}_P$ is then defined as an intersection of $\mathcal{F}_P$ with the Plücker quadric:

$$\mathcal{S}_P = \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{F}_P, \boldsymbol{\pi} \odot \boldsymbol{\pi} = 0\}. \tag{2.18}$$

The stabbers are Plücker points corresponding to the real lines intersecting the polygon that are oriented in the direction of the normal. Similarly we can define the sets of reverse feasible Plücker points $\mathcal{F}_P^-$ and reverse stabbers $\mathcal{S}_P^-$ that correspond to opposite oriented lines intersecting the polygon:

$$\begin{aligned} \mathcal{F}_P^- &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \hat{\boldsymbol{\pi}} \in \hat{\boldsymbol{\omega}}_{e_i}^+, \forall i \in \langle 0, n)\} \\ \mathcal{S}_P^- &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{F}_P^-, \boldsymbol{\pi} \odot \boldsymbol{\pi} = 0\}. \end{aligned} \tag{2.19}$$

2.5 Lines between two polygons

The above presented definitions of elementary line sets allow to handle visibility computations by means of set operations on the sets of feasible Plücker points. Visibility between two polygons P_j and P_k can be determined by constructing an intersection of feasible sets of the two polygons $\mathcal{F}_{P_j}$ and $\mathcal{F}_{P_k}$ and subtracting all feasible sets of polygons lying between P_j and P_k. To obtain the set of unoccluded stabbers we intersect the resulting feasible set with the Plücker quadric.

Further in this chapter we restrict our discussion to visibility from a given source polygon P_S. Given any occluder polygon P_j we first describe lines intersecting both P_S and P_j. Lines between P_S and P_j can be described by an intersection of their feasible line sets:

$$\mathcal{F}_{P_S P_j} = \mathcal{F}_{P_S} \cap \mathcal{F}_{P_j} \tag{2.20}$$

and thus

$$\mathcal{S}_{P_S P_j} = \mathcal{S}_{P_S} \cap \mathcal{S}_{P_j}. \tag{2.21}$$

The feasible Plücker points are defined by an intersection of halfspaces corresponding to edges of P_S and P_j. These halfspaces define a blocker polyhedron $B_{P_S P_j}$ that is described in the next section.

Blocker polyhedron

The blocker polyhedron describes lines intersecting the source polygon and the given occluder. The blocker polyhedron can be seen as an extension of the blocker polygon discussed above for the from-region visibility in 3D scenes. The blocker polyhedron is a 5D polyhedron in a 5D projective space. To avoid singularities in the projection from $\mathcal{P}^5$ to $\mathcal{R}^5$ the polyhedron can be embedded in $\mathcal{R}^6$ similarly to the embedding of blocker polygon in $\mathcal{R}^3$ (see Section 2.1.3). Then the polyhedron actually represents a 6D pyramid with an apex at the origin of $\mathcal{R}^6$.

Cap planes

The blocker polyhedron is defined by an intersection of halfspaces defined by Plücker planes that are mappings of edges of the source polygon and the occluder. As stated above the blocker polyhedron represents the set of feasible Plücker points $\mathcal{F}_{P_S P_j}$ including points not intersecting the Plücker quadric that correspond to imaginary lines. We bound the polyhedron by cap planes aligned with the Plücker quadric so that the resulting polyhedron is a tighter representation of the stabbers $\mathcal{S}_{P_S P_j}$. We need to ensure that the resulting polyhedron fully contains the stabbers $\mathcal{S}_{P_S P_j}$, i.e. contains the cross-section of the Plücker quadric and $\mathcal{F}_{P_S P_j}$.

The cap planes provide the following benefits:

- The computation is localized to the proximity of the Plücker quadric. This reduces the combinatorial complexity of data structure representing an arrangement of the blocker polyhedra.
- The blocker polyhedron is always bounded. Although the set of lines between two convex polygons is bounded, the set of feasible Plücker points can be unbounded at the "direction" of imaginary lines. Adding the cap planes we make sure that the polyhedron is bounded, which allows its easier treatment. By using the cap planes we avoid the handling of very oblong, almost unbounded polyhedra, which improves numerical stability of a floating point implementation of the algorithm.

We used two cap planes to bound the polyhedron, one for each side of the Plücker quadric (a side is given by the sign of $\boldsymbol{\pi} \odot \boldsymbol{\pi}$). The cap planes are computed as tangents to the Plücker quadric at the center of the set of stabbers $\mathcal{S}_{P_S P_j}$. The planes are translated each at the opposite direction making sure that they include the whole set $\mathcal{S}_{P_S P_j}$.

2.5.1 Intersection with the Plücker quadric

Given a blocker polyhedron representing the set of feasible lines $\mathcal{F}_{P_S P_j}$ we can compute an intersection of the edges of the polyhedron with the Plücker quadric to determine the set of extremal lines bounding the set of stabbers $\mathcal{S}_{P_S P_j}$. An intersection of an edge of the blocker polyhedron with the Plücker quadric results in at most two extremal Plücker points that correspond extremal lines[1]. Given an edge of the blocker polyhedron the intersection with the Plücker quadric is computed by solving the quadratic equation (Eq. 2.10). A robust algorithm for computing this intersection was developed by Teller [TH93a].

Intersecting all edges of the blocker polyhedron with the Plücker quadric yields all extremal lines of $\mathcal{S}_{P_S P_j}$ [Tel92a, Pu98]. See Figure 2.11 for an example of extremal lines computed for the given source polygon and a set of three occluders.

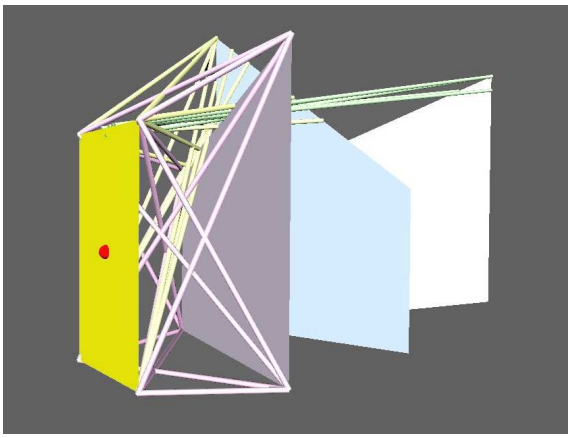


Figure 2.11: Extremal lines for the given source polygon (yellow) and three occluders.

The intersection of the 2D faces of the blocker polyhedron with the Plücker quadric yields swaths of event surfaces of the set of stabbers $\mathcal{S}_{P_S P_j}$ [Tel92b]. In general the intersection results in 1D conics.

We can avoid the explicit treatment of conics in 5D by computing the local topology of the edges of the blocker polyhedron and constructing the swaths in primal space between the topologically connected extremal lines [Tel92b]. The local topology of an extremal Plücker point is given by connections with extremal Plücker points embedded in the same 2D face of the blocker polyhedron. A 2D face of the blocker polyhedron is given by three Plücker hyperplanes. Thus the pairs of extremal Plücker points defined by the subset of the same three Plücker hyperplanes define a swath.

2.5.2 Size of the set of lines

Computing a size measure of a given set of lines is useful for most visibility algorithms. The computed size measure can be used to drive the subdivision of the given set of lines or to bound the maximal error of the algorithm. An analytic algorithm can use the computed size measure for thresholding by a given ϵ-size to discard very small line sets. A discrete algorithm can use the size measure to determine the required density of sampling.

The size of a set of lines for the from-point visibility can be formulated easily: the size is given by the area of the intersection of the line set with a plane. This corresponds to quantifying visibility of an object according to its projected area. Such a size is determined in the solution space (viewport). Alternatively we could use a "viewport independent measure" given by a solid

[1] Neglecting the case that the whole edge is embedded in the Plücker quadric, which results in infinite number of extremal lines.

angle formed by the visible part of an object. The size measure for the from-region visibility problems is more complicated for the following reasons:

- The domain of the solution space is four-dimensional.
- The solution space of the from-region visibility algorithm generally does not correspond to the solution space of the application. For example, a visible surface algorithm using a precomputed PVS works in a 2D domain induced by the given viewport.

General size measure

A size of a set of lines for the from-region visibility can be computed by evaluating a 4D integral. Using Plücker coordinates we can compute a volume of the 4D hyper-surface corresponding to the given set of lines. The volume however depends on a way of projecting the blocker polyhedron from $\mathcal{P}^5$ to $\mathcal{R}^5$. This projection has a similar role as the selection of the viewport for the from-point visibility problem. We can project the blocker polyhedron from $\mathcal{P}^5$ to $\mathcal{R}^5$ by projecting it to a 5D hyperplane defined by certain reference direction, e.g. the "center-line" of the given set of lines. Pu proposed a different size measure based on measuring the angular spread and the distance between lines [Pu98]. Both these quantities can be evaluated in terms of Plücker coordinates of the set of extremal lines of the given line set.

Size measure for the PVS computation

It can be difficult to relate the size measures described above to the domain of the result of a subsequently applied visibility algorithm. We need a simple scheme that fits to the context of the target application. In this section we suggest a size measure designed for the PVS computation. When computing a PVS we are interested in measuring the size of the set of unoccluded lines (stabbers) between the source polygon P_S and a given scene polygon. If this size is below an ϵ-threshold, we can possibly exclude the polygon from the PVS. We suggest to use an estimate of the minimal angle between the stabbers at a point inside P_S. The idea is to estimate the minimal projected diameter of a polygon visible through the given set of lines from any point inside P_S. This estimate can be used to bound a maximal error of an image synthesized with respect to any viewpoint inside P_S for the case that the corresponding set of lines is neglected.

Given a blocker polyhedron $\mathcal{F}_{P_S P_j}$ the proposed size measure can be evaluated as follows:

1. Compute the extremal lines of the corresponding set of stabbers $\mathcal{S}_{P_S P_j}$ as described in Section 2.5.1.
2. For each polygon edge e_i bounding the stabbers determine an extremal line l_{mi} with a maximal distance from the edge.
3. For each edge e_i compute a shortest line segment z_i connecting e_i and l_{mi}. The length of this line segment is then scaled according to its distance from the source polygon, i.e. we compute an angle α_i between the lines connecting the center of the source polygon and the endpoints of z_i.
4. Select a minimal angle α_m of all α_i as the estimate of the size of the given line set.

The evaluation of the size measure is depicted in Figure 2.12.

The angle α_m can be related to the angular resolution of the synthesized image. Given the resolution of the image we can threshold "small" line sets with α_m below the corresponding angular threshold to achieve a sub-pixel precision of the rendering algorithm. This measure can also be applied to deal with the finite precision of the floating point arithmetics by using a small ϵ-threshold to handle numerical inaccuracies.

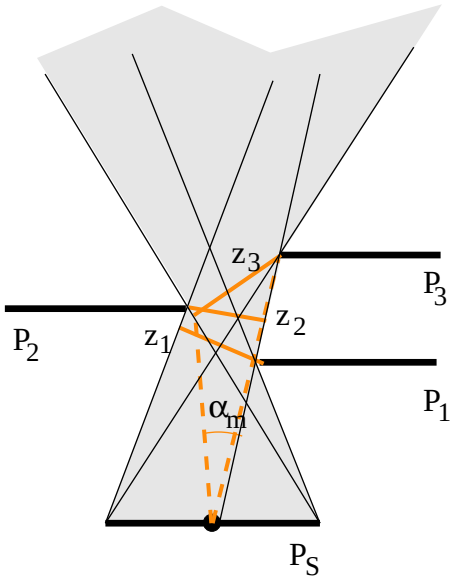


Figure 2.12: A 2D example of evaluation of the size of a set of lines. The three line segments z_1, z_2 and z_3 maximize the distance of the corresponding occluder edges from the extremal lines. The line segment z_3 spans a minimal angle α_m with respect to the center of the source polygon P_S.

2.6 Summary

In this chapter we discussed the relation of sets of lines in 3D and the polyhedra in Plücker coordinates. We proposed a general size measure for a set of lines described by a blocker polyhedron and a size measure designed for the computation of PVS.

Chapter 3

Online Visibility Culling

3.1 Introduction

Visibility culling is one of the major acceleration techniques for the real-time rendering of complex scenes. The ultimate goal of visibility culling techniques is to prevent invisible objects from being sent to the rendering pipeline. A standard visibility-culling technique is view-frustum culling, which eliminates objects outside of the current view frustum. View-frustum culling is a fast and simple technique, but it does not eliminate objects in the view frustum that are occluded by other objects. This can lead to significant overdraw, i.e., the same image area gets covered more than once. The overdraw causes a waste of computational effort both in the pixel and the vertex processing stages of modern graphic hardware. The elimination of occluded objects is addressed by occlusion culling. In an optimized rendering pipeline, occlusion culling complements other rendering acceleration techniques such as levels of detail or impostors.

Occlusion culling can either be applied offline or online. When applied offline as a preprocess, we compute a potentially visible set (PVS) for cells of a fixed subdivision of the scene. At runtime, we can quickly identify a PVS for the given viewpoint. However, this approach suffers from four major problems: (1) the PVS is valid only for the original static scene configuration, (2) for a given viewpoint, the corresponding cell-based PVS can be overly conservative, (3) computing all PVSs is computationally expensive, and (4) an accurate PVS computation is difficult to implement for general scenes. Online occlusion culling can solve these problems at the cost of applying extra computations at each frame. To make these additional computations efficient, most online occlusion culling methods rely on a number of assumptions about the scene structure and its occlusion characteristics (e.g. presence of large occluders, occluder connectivity, occlusion by few closest depth layers).

Recent graphics hardware natively supports an occlusion query to detect the visibility of an object against the current contents of the z-buffer. Although the query itself is processed quickly using the raw power of the graphics processing unit (GPU), its result is not available immediately due to the delay between issuing the query and its actual processing in the graphics pipeline. As a result, a naive application of occlusion queries can even decrease the overall application performance due the associated CPU stalls and GPU starvation. In this chapter, we present an algorithm that aims to overcome these problems by reducing the number of issued queries and eliminating the CPU stalls and GPU starvation. To schedule the queries, the algorithm makes use of both the spatial and the temporal coherence of visibility. A major strength of our technique is its simplicity and versatility: the method can be easily integrated in existing real-time rendering packages on architectures supporting the underlying occlusion query [WB05]. In figure 3.1, the same scene (top row) is rendered using view frustum culling (visualization in the bottom left image) versus online culling using occlusion queries (visualization in the bottom right image). It can be seen that with view frustum culling only many objects are still rendered.

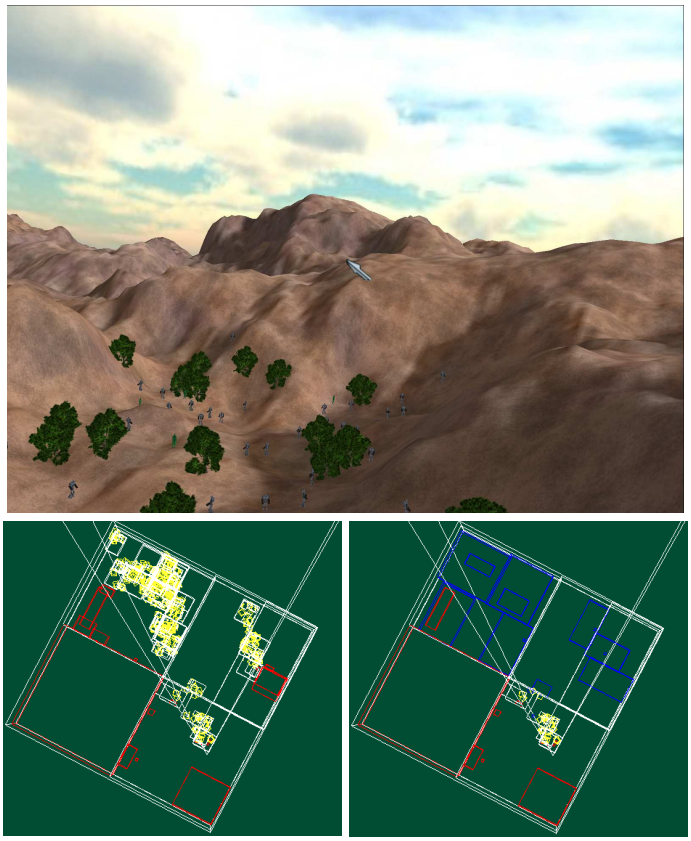


Figure 3.1: (top) The rendered terrain scene. (bottom) Visualizion of the rendered / culled objects. Using view frustum culling (left image) vs. occlusion queries (right image). The yellow boxes show the actually rendered scene objects. The red boxes depict the view frustum culled hierarchy nodes, the blue boxes depict the occlusion query culled hierarchy nodes.

3.2 Related Work

With the demand for rendering scenes of ever increasing size, there have been a number of visibility culling methods developed in the last decade. A comprehensive survey of visibility culling methods was presented by Cohen-Or et al. [COCSD03]. Another recent survey of Bittner and Wonka [BW03] discusses visibility culling in a broader context of other visibility problems.

According to the domain of visibility computation, we distinguish between from-point and from-region visibility algorithms. From-region algorithms compute a PVS and are applied offline in a preprocessing phase [ARB90, TS91a, LSCO03]. From-point algorithms are applied online for each particular viewpoint [GKM93, HMC+97, ZMHH97, BHS98, WS99, KS01]. In our further discussion we focus on online occlusion culling methods that exploit graphics hardware.

A conceptually important online occlusion culling method is the hierarchical z-buffer introduced by Greene et al. [GKM93]. It organizes the z-buffer as a pyramid, where the standard z-buffer is the finest level. At all other levels, each z-value is the farthest in the window corresponding to the adjacent finer level. The hierarchical z-buffer allows to quickly determine if the geometry in question is occluded. To a certain extent this idea is used in the current generation of graphics hardware by applying early z-tests of fragments in the graphics pipeline (e.g., Hyper-Z technology of ATI or Z-cull of NVIDIA). However, the geometry still needs to be sent to the GPU, transformed, and coarsely rasterized even if it is later determined invisible.

Zhang [ZMHH97] proposed hierarchical occlusion maps, which do not rely on the hardware support for the z-pyramid, but instead make use of hardware texturing. The hierarchical occlusion map is computed on the GPU by rasterizing and down sampling a given set of occluders. The occlusion map is used for overlap tests whereas the depths are compared using a coarse depth estimation buffer. Wonka and Schmalstieg [WS99] use occluder shadows to compute from-point visibility in $2\frac{1}{2}$D scenes with the help of the GPU. This method has been further extended to online computation of from-region visibility executed on a server [WWS01].

Bartz et al. [BMH98] proposed an OpenGL extension for occlusion queries along with a discussion concerning a potential realization in hardware. A first hardware implementation of occlusion queries came with the VISUALIZE fx graphics hardware [SOG98]. The corresponding OpenGL extension is called HP_occlusion_test. A more recent OpenGL extension, NV_occlusion_query, was introduced by NVIDIA with the GeForce 3 graphics card and it is now also available as an official ARB extension.

Hillesland et al. [HSLM02] have proposed an algorithm which employs the NV_occlusion_query. They subdivide the scene using a uniform grid. Then the cubes are traversed in slabs roughly perpendicular to the viewport. The queries are issued for all cubes of a slab at once, after the visible geometry of this slab has been rendered. The method can also use nested grids: a cell of the grid contains another grid that is traversed if the cell is proven visible. This method however does not exploit temporal and spatial coherence of visibility and it is restricted to regular subdivision data structures. Our new method addresses both these problems and provides natural extensions to balance the accuracy of visibility classification and the associated computational costs.

Recently, Staneker et al. [SBS04b] developed a method integrating occlusion culling into the OpenSG scene graph framework. Their technique uses occupancy maps maintained in software to avoid queries on visible scene graph nodes, and temporal coherence to reduce the number of occlusion queries. The drawback of the method is that it performs the queries in a serial fashion and thus it suffers from the CPU stalls and GPU starvation.

On a theoretical level, our method is related to methods aiming to exploit the temporal coherence of visibility. Greene et al. [GKM93] used the set of visible objects from one frame to initialize the z-pyramid in the next frame in order to reduce the overdraw of the hierarchical z-buffer. The algorithm of Coorg and Teller [CT96] restricts the hierarchical traversal to nodes associated with visual events that were crossed between successive viewpoint positions. Another method of Coorg and Teller [CT97] exploits temporal coherence by caching occlusion relationships. Chrysanthou and Slater have proposed a probabilistic scheme for view-frustum culling [SC97].

The above mentioned methods for exploiting temporal coherence are tightly interwoven with the particular culling algorithm. On the contrary, Bittner et al. [BH01] presented a general acceler-

ation technique for exploiting spatial and temporal coherence in hierarchical visibility algorithms. The central idea, which is also vital for the online occlusion culling, is to avoid repeated visibility tests of interior nodes of the hierarchy. The problem of direct adoption of this method is that it is designed for the use with instantaneous CPU based occlusion queries, whereas hardware occlusion queries exhibit significant latency. The method presented herein efficiently overcomes the problem of latency while keeping the benefits of a generality and simplicity of the original hierarchical technique. As a result we obtain a simple and efficient occlusion culling algorithm utilizing hardware occlusion queries.

3.3 Hardware Occlusion Queries

Hardware occlusion queries follow a simple pattern: To test visibility of an occludee, we send its bounding volume to the GPU. The volume is rasterized and its fragments are compared to the current contents of the z-buffer. The GPU then returns the number of visible fragments. If there is no visible fragment, the occludee is invisible and it need not be rendered.

3.3.1 Advantages of hardware occlusion queries

There are several advantages of hardware occlusion queries:

- Generality of occluders. We can use the original scene geometry as occluders, since the queries use the current contents of the z-buffer.
- Occluder fusion. The occluders are merged in the z-buffer, so the queries automatically account for occluder fusion. Additionally this fusion comes for free since we use the intermediate result of the rendering itself.
- Generality of occludees. We can use complex occludees. Anything that can be rasterized quickly is suitable.
- Exploiting the GPU power. The queries take full advantage of the high fill rates and internal parallelism provided by modern GPUs.
- Simple use. Hardware occlusion queries can be easily integrated into a rendering algorithm. They provide a powerful tool to minimize the implementation effort, especially when compared to CPU-based occlusion culling.

3.3.2 Problems of hardware occlusion queries

Currently there are two main hardware supported variants of occlusion queries: the HP test (HP_occlusion_test) and the more recent NV query (NV_occlusion_query, now also available as ARB_occlusion_query). The most important difference between the HP test and the NV query is that multiple NV queries can be issued before asking for their results, while only one HP test is allowed at a time, which severely limits its possible algorithmic usage. Additionally the NV query returns the number of visible pixels whereas the HP test returns only a binary visibility classification.

The main problem of both the HP test and the NV query is the latency between issuing the query and the availability of the result. The latency occurs due to the delayed processing of the query in a long graphics pipeline, the cost of processing the query itself, and the cost of transferring the result back to the CPU. The latency causes two major problems: CPU stalls and GPU starvation. After issuing the query, the CPU waits for its result and does not feed the GPU with new data. When the result finally becomes available, the GPU pipeline can already be empty. Thus the GPU needs to wait for the CPU to process the result of the query and to feed the GPU with new data.

A major challenge when using hardware occlusion queries is to avoid the CPU stalls by filling the latency time with other tasks, such as rendering visible scene objects or issuing other, independent occlusion queries (see Figure 3.2)

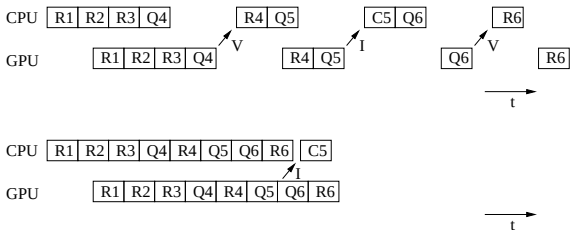


Figure 3.2: (top) Illustration of CPU stalls and GPU starvation. Qn, Rn, and Cn denote querying, rendering, and culling of object n, respectively. Note that object 5 is found invisible by Q5 and thus not rendered. (bottom) More efficient query scheduling. The scheduling assumes that objects 4 and 6 will be visible in the current frame and renders them without waiting for the result of the corresponding queries.

3.3.3 Hierarchical stop-and-wait method

Many rendering algorithms rely on hierarchical structures in order to deal with complex scenes. In the context of occlusion culling, such a data structure allows to efficiently cull large scene blocks, and thus to exploit spatial coherence of visibility and provide a key to achieving output sensitivity.

This section outlines a naive application of occlusion queries in the scope of a hierarchical algorithm. We refer to this approach as the hierarchical stop-and-wait method. Our discussion is based on kD-trees, which proved to be efficient for point location, ray tracing, and visibility culling [MB90, HMC+97, CT97, BH01]. The concept applies to general hierarchical data structures as well, though.

The hierarchical stop-and-wait method proceeds as follows: Once a kD-tree node passes view-frustum culling, it is tested for occlusion by issuing the occlusion query and waiting for its result. If the node is found visible, we continue by recursively testing its children in a front-to-back order. If the node is a leaf, we render its associated objects.

The problem with this approach is that we can continue the tree traversal only when the result of the last occlusion query becomes available. If the result is not available, we have to stall the CPU, which causes significant performance penalties. As we document in Section 3.6, these penalties together with the overhead of the queries themselves can even decrease the overall application performance compared to pure view-frustum culling. Our new method aims to eliminate this problem by issuing multiple occlusion queries for independent scene parts and exploiting temporal coherence of visibility classifications.

3.4 Coherent Hierarchical Culling

In this section we first present an overview of our new algorithm. Then we discuss its steps in more detail.

3.4.1 Algorithm Overview

Our method is based on exploiting temporal coherence of visibility classification. In particular, it is centered on the following three ideas:

- We initiate occlusion queries on nodes of the hierarchy where the traversal terminated in the last frame. Thus we avoid queries on all previously visible interior nodes [BH01].

- We assume that a previously visible leaf node remains visible and render the associated geometry without waiting for the result of the corresponding occlusion query.
- Issued occlusion queries are stored in a query queue until they are known to be carried out by the GPU. This allows interleaving the queries with the rendering of visible geometry.

The algorithm performs a traversal of the hierarchy that is terminated either at leaf nodes or nodes that are classified as invisible. Let us call such nodes the termination nodes, and interior nodes that have been classified visible the opened nodes. We denote sets of termination and opened nodes in the i-th frame $\mathcal{T}_i$ and $\mathcal{O}_i$, respectively. In the i-th frame, we traverse the kD-tree in a front-to-back order, skip all nodes of $\mathcal{O}_{i-1}$ and apply occlusion queries first on the termination nodes $\mathcal{T}_{i-1}$. When reaching a termination node, the algorithm proceeds as follows:

- For a previously visible node (this must be a leaf), we issue the occlusion query and store it in the query queue. Then we immediately render the associated geometry without waiting for the result of the query.
- For a previously invisible node, we issue the query and store it in the query queue.

When the query queue is not empty, we check if the result of the oldest query in the queue is already available. If the query result is not available, we continue by recursively processing other nodes of the kD-tree as described above. If the query result is available, we fetch the result and remove the node from the query queue. If the node is visible, we process its children recursively. Otherwise, the whole subtree of the node is invisible and thus it is culled.

In order to propagate changes in visibility upwards in the hierarchy, the visibility classification is pulled up according to the following rule: An interior node is invisible only if all its children have been classified invisible. Otherwise, it remains visible and thus opened. The pseudo-code of the complete algorithm is given in Figure 3.3. An example of the behavior of the method on a small kD-tree for two subsequent frames is depicted Figure 3.4.

The sets of opened nodes and termination nodes need not be maintained explicitly. Instead, these sets can be easily identified by associating with each node an information about its visibility and an id of the last frame when it was visited. The node is an opened node if it is an interior visible node that was visited in the last frame (line 23 in the pseudocode). Note that in the actual implementation of the pull up we can set all visited nodes to invisible by default and then pull up any changes from invisible to visible (lines 25 and line 12 in Figure 3.3). This modification eliminates checking children for invisibility during the pull up.

3.4.2 Reduction of the number of queries

Our method reduces the number of visibility queries in two ways: Firstly, as other hierarchical culling methods we consider only a subtree of the whole hierarchy (opened nodes + termination nodes). Secondly, by avoiding queries on opened nodes we eliminate part of the overhead of identification of this subtree. These reductions reflect the following coherence properties of scene visibility:

- Spatial coherence. The invisible termination nodes approximate the occluded part of the scene with the smallest number of nodes with respect to the given hierarchy, i.e., each invisible termination node has a visible parent. This induces an adaptive spatial subdivision that reflects spatial coherence of visibility, more precisely the coherence of occluded regions. The adaptive nature of the subdivision allows to minimize the number of subsequent occlusion queries by applying the queries on the largest spatial regions that are expected to remain occluded.
- Temporal coherence. If visibility remains constant the set of termination nodes needs no adaptation. If an occluded node becomes visible we recursively process its children (pull-down). If a visible node becomes occluded we propagate the change higher in the hierarchy

(pull-up). A pull-down reflects a spatial growing of visible regions. Similarly, a pull-up reflects a spatial growing of occluded regions.

By avoiding queries on the opened nodes, we can save $1/k$ of the queries for a hierarchy with branching factor k (assuming visibility remains constant). Thus for the kD-tree, up to half of the queries can be saved. The actual savings in the total query time are even larger: the higher we are at the hierarchy, the larger boxes we would have to check for occlusion. Consequently, the higher is the fill rate that would have been required to rasterize the boxes. In particular, assuming that the sum of the screen space projected area for nodes at each level of the kD-tree is equal and the opened nodes form a complete binary subtree of depth d, the fill rate is reduced $(d + 2)$ times.

3.4.3 Reduction of CPU stalls and GPU starvation

The reduction of CPU stalls and GPU starvation is achieved by interleaving occlusion queries with the rendering of visible geometry. The immediate rendering of previously visible termination nodes and the subsequent issuing of occlusion queries eliminates the requirement of waiting for the query result during the processing of the initial depth layers containing previously visible nodes. In an optimal case, new query results become available in between and thus we completely eliminate CPU stalls. In a static scenario, we achieve exactly the same visibility classification as the hierarchical stop-and-wait method.

If the visibility is changing, the situation can be different: if the results of the queries arrive too late, it is possible that we initiated an occlusion query on a previously occluded node A that is in fact occluded by another previously occluded node B that became visible. If B is still in the query queue, we do not capture a possible occlusion of A by B since the geometry associated with B has not yet been rendered. In Section 3.6 we show that the increase of the number of rendered objects compared to the stop-and-wait method is usually very small.

3.4.4 Front-to-back scene traversal

For kD-trees the front-to-back scene traversal can be easily implemented using a depth first traversal [BH01]. However, at a modest increase in computational cost we can also use a more general breadth-first traversal based on a priority queue. The priority of the node then corresponds to an inverse of the minimal distance of the viewpoint and the bounding box associated with the given node of the kD-tree [KS01, SBS04b].

In the context of our culling algorithm, there are two main advantages of the breadth-first front-to-back traversal :

- Better query scheduling. By spreading the traversal of the scene in a breadth-first manner, we process the scene in depth layers. Within each layer, the node processing order is practically independent, which minimizes the problem of occlusion query dependence. The breadth-first traversal interleaves occlusion-independent nodes, which can provide a more accurate visibility classification if visibility changes quickly. In particular, it reduces the problem of false classifications due to missed occlusion by nodes waiting in the query queue (discussed in Section 3.4.3).

- Using other spatial data structures. By using a breadth-first traversal, we are no longer restricted to the kD-tree. Instead we can use an arbitrary spatial data structure such as a bounding volume hierarchy, octree, grid, hierarchical grid, etc. Once we compute a distance from a node to the viewpoint, the node processing order is established by the priority queue.

When using the priority queue, our culling algorithm can also be applied directly to the scene graph hierarchy, thus avoiding the construction of any auxiliary data structure for spatial partitioning. This is especially important for dynamic scenes, in which maintenance of a spatial classification of moving objects can be costly.

3.4.5 Checking the query result

The presented algorithm repeatedly checks if the result of the occlusion query is available before fetching any node from the traversal stack (line 6 in Figure 3.3). Our practical experiments have proven that the cost of this check is negligible and thus it can used frequently without any performance penalty. If the cost of this check were significantly higher, we could delay asking for the query result by a time established by empirical measurements for the particular hardware. This delay should also reflect the size of the queried node to match the expected availability of the query result as precise as possible.

3.5 Further Optimizations

This section discusses a couple of optimizations of our method that can further improve the overall rendering performance. In contrast to the basic algorithm from the previous section, these optimizations rely on some user specified parameters that should be tuned for a particular scene and hardware configuration.

3.5.1 Conservative visibility testing

The first optimization addresses the reduction of the number of visibility tests at the cost of a possible increase in the number of rendered objects. This optimization is based on the idea of skipping some occlusion tests of visible nodes. We assume that whenever a node becomes visible, it remains visible for a number of frames. Within the given number of frames we avoid issuing occlusion queries and simply assume the node remains visible [BH01].

This technique can significantly reduce the number of visibility tests applied on visible nodes of the hierarchy. Especially in the case of sparsely occluded scenes, there is a large number of visible nodes being tested, which does not provide any benefit since most of them remain visible. On the other hand, we do not immediately capture all changes from visibility to invisibility, and thus we may render objects that have already become invisible from the moment when the last occlusion test was issued.

In the simplest case, the number of frames a node is assumed visible can be a predefined constant. In a more complicated scenario this number should be influenced by the history of the success of occlusion queries and/or the current speed of camera movement.

3.5.2 Approximate visibility

The algorithm as presented computes a conservative visibility classification with respect to the resolution of the z-buffer. We can easily modify the algorithm to cull nodes more aggressively in cases when a small part of the node is visible. We compare the number of visible pixels returned by the occlusion query with a user specified constant and cull the node if this number drops below this constant.

3.5.3 Complete elimination of CPU stalls

The basic algorithm eliminates CPU stalls unless the traversal stack is empty. If there is no node to traverse in the traversal stack and the result of the oldest query in the query queue is still not available, it stalls the CPU by waiting for the query result. To completely eliminate the CPU stalls, we can speculatively render some nodes with undecided visibility. In particular, we select a node from the query queue and render the geometry associated with the node (or the whole subtree if it is an interior node). The node is marked as rendered but the associated occlusion query is kept in the queue to fetch its result later. If we are unlucky and the node remains invisible, the effort of rendering the node's geometry is wasted. On the other hand, if the node has become visible, we have used the time slot before the next query arrives in an optimal manner.

To avoid the problem of spending more time on rendering invisible nodes than would be spent by waiting for the result of the query, we select a node with the lowest estimated rendering cost and compare this cost with a user specified constant. If the cost is larger than the constant we conclude that it is too risky to render the node and wait till the result of the query becomes available.

3.5.4 Initial depth pass

To achieve maximal performance on modern GPU's, one has to take care of a number of issues. First, it is very important to reduce material switching. Thus modern rendering engines sort the objects (or patches) by materials in order to eliminate the material switching as good as possible.

Next, materials can be very costly, sometimes complicated shaders have to be evaluated several times per batch. Hence it should be avoided to render the full material for fragments which eventually turn out to be occluded. This can be achieved by rendering an initial depth pass (i.e., enabling only depth write to fill the depth buffer). Afterwards the geometry is rendered again, this time with full shading. Because the depth buffer is already established, invisible fragments will be discarded before any shading is done calculated.

This approach can be naturally adapted for use with the CHC algorithm. Only an initial depth pass is rendered in front-to-back order using the CHC algorithm. The initial pass is sufficient to fill the depth buffer and determine the visible geometry. Then only the visible geometry is rendered again, exploiting the full optimization and material sorting capability of the rendering engine.

If the materials requires several rendering passes, we can use a variant of the depth pass method. We render only the first passes using the algorithm (e.g., the solid passes), determining the visibility of the patches, and render all the other passes afterwards. This approach can be used when there are passes which require a special kind of sorting to be rendered correctly (e.g., transparent passes, shadow passes). In figure 3.6, we can see that artifacts occur in the left image if the transparent passes are not rendered in the correct order after applying the hierarchical algorithm (right image). In a similar fashion, we are able to handle shadows 3.7.

3.5.5 Batching multiple queries

When occlusion queries are rendered interleaved with geometry, there is always a state change involved. To reduce state changes, it is beneficial not to execute one query at a time, but multiple queries at once [SBS04a]. Instead of immediately executing a query for a node when we fetch it from the traversal stack, we add it to the pending queue. If n of these queries are accumulated in the queue, we can execute them at once. To optain an optimal value for n, several some heuristics can be applied, e.g., a fraction of the number of queries issued in the last frame. The pseudo-code of the algorithm including the batching is given in Figure 3.5.

3.6 Results

We have incorporated our method into an OpenGL-based scene graph library and tested it on three scenes of different types. All tests were conducted on a PC with a 3.2GHz P4, 1GB of memory, and a GeForce FX5950 graphics card.

3.6.1 Test scenes

The three test scenes comprise a synthetic arrangement of 5000 randomly positioned teapots (11.6M polygons); an urban environment (1M polygons); and the UNC power plant model (13M polygons). The test scenes are depicted in Figure 3.12. All scenes were partitioned using a kD-tree constructed according to the surface-area heuristics [MB90].

Although the teapot scene would intuitively offer good occlusion, it is a complicated case to handle for occlusion culling. Firstly, the teapots consist of small triangles and so only the effect

of fused occlusion due to a large number of visible triangles can bring a culling benefit. Secondly, there are many thin holes through which it is possible to see quite far into the arrangement of teapots. Thirdly, the arrangement is long and thin and so we can see almost half of the teapots along the longer side of the arrangement.

The complete power plant model is quite challenging even to load into memory, but on the other hand it offers good occlusion. This scene is an interesting candidate for testing not only due to its size, but also due to significant changes in visibility and depth complexity in its different parts.

The city scene is a classical target for occlusion culling algorithms. Due to the urban structure consisting of buildings and streets, most of the model is occluded when viewed from the streets. Note that the scene does not contain any detailed geometry inside the buildings. See Figure 3.8 for a visualization of the visibility classification of the kD-tree nodes for the city scene.

3.6.2 Basic tests

We have measured the frame times for rendering with only view-frustum culling (VFC), the hierarchical stop-and-wait method (S&W), and our new coherent hierarchical culling method (CHC). Additionally, we have evaluated the time for an "ideal" algorithm. The ideal algorithm renders the visible objects found by the S&W algorithm without performing any visibility tests. This is an optimal solution with respect to the given hierarchy, i.e., no occlusion culling algorithm operating on the same hierarchy can be faster. For the basic tests we did not apply any of the optimizations discussed in Section 3.5, which require user specified parameters.

For each test scene, we have constructed a walkthrough which is shown in full in the accompanying video. Figures 3.9, 3.10, and 3.11 depict the frame times measured for the walkthroughs. Note that Figure 3.11 uses a logarithmic scale to capture the high variations in frame times during the power plant walkthrough. To better demonstrate the behavior of our algorithm, all walkthroughs contain sections with both restricted and unrestricted visibility. For the teapots, we viewed the arrangement of teapots along the longer side of the arrangement (frames 25–90). In the city we elevated the viewpoint above the roofs and gained sight over most of the city (frames 1200–1800). The power plant walkthrough contains several viewpoints from which a large part of the model is visible (spikes in Figure 3.11 where all algorithms are slow), viewpoints along the border of the model directed outwards with low depth complexity (holes in Figure 3.11 where all algorithms are fast), and viewpoints inside the power plant with high depth complexity where occlusion culling produces a significant speedup over VFC (e.g. frame 3800).

As we can see for a number frames in the walkthroughs, the CHC method can produce a speedup of more than one order of magnitude compared to VFC. The maximum speedup for the teapots, the city, and the power plant walkthroughs is 8, 20, and 70, respectively. We can also observe that CHC maintains a significant gain over S&W and in many cases it almost matches the performance of the ideal algorithm. In complicated scenarios the S&W method caused a significant slowdown compared to VFC (e.g. frames 1200–1800 of Figure 3.10). Even in these cases, the CHC method maintained a good speedup over VFC except for a small number of frames.

Next, we summarized the scene statistics and the average values per frame in Table 3.1. The table shows the number of issued occlusion queries, the wait time representing the CPU stalls, the number of rendered triangles, the total frame time, and the speedup over VFC.

We can see that the CHC method practically eliminates the CPU stalls (wait time) compared to the S&W method. This is paid for by a slight increase in the number of rendered triangles. For the three walkthroughs, the CHC method produces average speedups of 4.6, 4.0, and 4.7 over view frustum culling and average speedups of 2.0, 2.6, and 1.6 over the S&W method. CHC is only 1.1, 1.7, and 1.2 times slower than the ideal occlusion culling algorithm. Concerning the accuracy, the increase of the average number of rendered triangles for CHC method compared to S&W was 9%, 1.4%, and 1.3%. This increase was always recovered by the reduction of CPU stalls for the tested walkthroughs.

scene	method	#queries	wait time [ms]	rendered triangles	frame time [ms]	speedup
Teapots	VFC	—	—	11,139,928	310.42	1.0
11,520,000 triangles	S&W	4704	83.19	2,617,801	154.95	2.3
21,639 kD-Tree nodes	CHC	2827	1.31	2,852,514	81,18	4.6
	Ideal	—	—	2,617,801	72.19	5.2
City	VFC	—	—	156,521	19.79	1.0
1,036,146 triangles	S&W	663	9.49	30,594	19.9	1.5
33,195 kD-Tree nodes	CHC	345	0.18	31,034	8.47	4.0
	Ideal	—	—	30,594	4.55	6.6
Power Plant	VFC	—	—	1,556,300	138.76	1.0
12,748,510 triangles	S&W	485	16.16	392,962	52.29	3.2
18,719 kD-Tree nodes	CHC	263	0.70	397,920	38.73	4.7
	Ideal	—	—	392,962	36.34	5.8

Table 3.1: Statistics for the three test scenes. VFC is rendering with only view-frustum culling, S&W is the hierarchical stop and wait method, CHC is our new method, and Ideal is a perfect method with respect to the given hierarchy. All values are averages over all frames (including the speedup).

scene	t_{av}	n_{vp}	#queries	frame time [ms]
Teapots	0	0	2827	81.18
	2	0	1769	86.31
	2	25	1468	55.90
City	0	0	345	8.47
	2	0	192	6.70
	2	25	181	6.11
Power Plant	0	0	263	38.73
	2	0	126	31.17
	2	25	120	36.62

Table 3.2: Influence of optimizations on the CHC method. t_{av} is the number of assumed visibility frames for conservative visibility testing, n_{vp} is the pixel threshold for approximate visibility.

3.6.3 Optimizations

First of all we have observed that the technique of complete elimination of CPU stalls discussed in Section 3.5.3 has a very limited scope. In fact for all our tests the stalls were almost completely eliminated by the basic algorithm already (see wait time in Table 3.1). We did not find constants that could produce additional speedup using this technique.

The measurements for the other optimizations discussed in Section 3.5 are summarized in Table 3.2. We have measured the average number of issued queries and the average frame time in dependence on the number of frames a node is assumed visible and the pixel threshold of approximate visibility. We have observed that the effectiveness of the optimizations depends strongly on the scene. If the hierarchy is deep and the geometry associated with a leaf node is not too complex, the conservative visibility testing produces a significant speedup (city and power plant). For the teapot scene the penalty for false rendering of actually occluded objects became larger than savings achieved by the reduction of the number of queries. On the other hand since the teapot scene contains complex visible geometry the approximate visibility optimization produced a significant speedup. This is however paid for by introducing errors in the image proportional to the pixel threshold used.

3.6.4 Comparison to PVS-based rendering

We also compared the CHC method against precalculated visibility. In particular, we used the PVS computed by an offline visibility algorithm [WWS00]. While the walkthrough using the PVS was 1.26ms faster per frame on average, our method does not require costly precomputation and

can be used at any general 3D position in the model, not only in a predefined view space.

3.7 Summary

We have presented a method for the optimized scheduling of hardware accelerated occlusion queries. The method schedules occlusion queries in order to minimize the number of the queries and their latency. This is achieved by exploiting spatial and temporal coherence of visibility. Our results show that the CPU stalls and GPU starvation are almost completely eliminated at the cost of a slight increase in the number of rendered objects.

Our technique can be used with practically arbitrary scene partitioning data structures such as kD-trees, bounding volume hierarchies, or hierarchical grids. The implementation of the method is straightforward as it uses a simple OpenGL interface to the hardware occlusion queries. In particular, the method requires no complicated geometrical operations or data structures. The algorithm is suitable for application on scenes of arbitrary structure and it requires no preprocessing or scene dependent tuning.

We have experimentally verified that the method is well suited to the NV_occlusion_query supported on current consumer grade graphics hardware. We have obtained an average speedup of 4.0–4.7 compared to pure view-frustum culling and 1.6–2.6 compared to the hierarchical stop-and-wait application of occlusion queries.

```
Algorithm: Traversal of the kD-tree
 1:  TraversalStack.Push(kDTree.Root);
 2:  while ( not TraversalStack.Empty() or
 3:          not QueryQueue.Empty() ) {
 4:    //—- PART 1: processing finished occlusion queries
 5:    while ( not QueryQueue.Empty() and
 6:           (ResultAvailable(QueryQueue.Front()) or
 7:            TraversalStack.Empty()) ) {
 8:      N = QueryQueue.Dequeue();
 9:      // wait if result not available
10:      visiblePixels = GetOcclussionQueryResult(N);
11:      if ( visiblePixels ¿ VisibilityThreshold ) {
12:        PullUpVisibility(N);
13:        TraverseNode(N);
14:      }
15:    }
16:    //—- PART 2: kd-tree traversal
17:    if ( not TraversalStack.Empty() ) {
18:      N = TraversalStack.Pop();
19:      if ( InsideViewFrustum(N) ) {
20:        // identify previously visible nodes
21:        wasVisible = N.visible && (N.lastVisited == frameID -1);
22:        // identify previously opened nodes
23:        opened = wasVisible && !IsLeaf(N);
24:        // reset node's visibility classification
25:        N.visible = false;
26:        // update node's visited flag
27:        N.lastVisited = frameID;
28:        // skip testing all previously opened nodes
29:        if ( !opened ) {
30:         IssueOcclusionQuery(N); QueryQueue.Enqueue(N);
31:        }
32:        // traverse a node unless it was invisible
33:        if ( wasVisible )
34:         TraverseNode(N);
35:      }
36:    }
37:  }
38:  TraverseNode(N) {
39:    if ( IsLeaf(N) )
40:      Render(N);
41:    else
42:      TraversalStack.PushChildren(N);
43:  }
44:  PullUpVisibility(N) {
45:    while (!N.visible) { N.visible = true; N = N.parent; }
46:  }
```

Figure 3.3: Pseudo-code of coherent hierarchical culling.

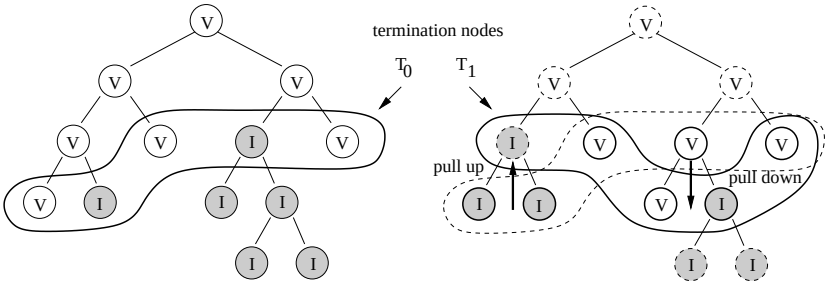


Figure 3.4: (left) Visibility classification of a node of the kD-tree and the termination nodes. (right) Visibility classification after the application of the occlusion test and the new set of termination nodes. Nodes on which occlusion queries were applied are depicted with a solid outline. Note the pull-up and pull-down due to visibility changes.

```
Algorithm: Traversal of the kD-tree
 1:  TraversalStack.Push(kDTree.Root);
 2:  while ( not TraversalStack.Empty() or
 3:          not QueryQueue.Empty() ) {
 4:    //—- PART 1: processing finished occlusion queries
 5:    while ( not QueryQueue.Empty() and
 6:            (ResultAvailable(QueryQueue.Front()) or
 7:             TraversalStack.Empty()) ) {
 8:      N = QueryQueue.Dequeue();
 9:      // wait if result not available
10:      visiblePixels = GetOcclussionQueryResult(N);
11:      if ( visiblePixels ¿ VisibilityThreshold ) {
12:        PullUpVisibility(N);
13:        TraverseNode(N);
14:      }
15:    }
16:    //—- PART 2: kd-tree traversal
17:    if ( not TraversalStack.Empty() ) {
18:      N = TraversalStack.Pop();
19:      if ( InsideViewFrustum(N) ) {
20:        // identify previously visible nodes
21:        wasVisible = N.visible && (N.lastVisited == frameID -1);
22:        // identify previously opened nodes
23:        opened = wasVisible && !IsLeaf(N);
24:        // reset node's visibility classification
25:        N.visible = false;
26:        // update node's visited flag
27:        N.lastVisited = frameID;
28:        // skip testing all previously opened nodes
29:        if ( !opened ) {
30:         IssueOcclusionQuery(N); QueryQueue.Enqueue(N);
31:        }
32:        // traverse a node unless it was invisible
33:        if ( wasVisible )
34:          TraverseNode(N);
35:      }
36:    }
37:  }
38:  TraverseNode(N) {
39:    if ( IsLeaf(N) )
40:      Render(N);
41:    else
42:      TraversalStack.PushChildren(N);
43:  }
44:  PullUpVisibility(N) {
45:    while (!N.visible) { N.visible = true; N = N.parent; }
46:  }
```

Figure 3.5: Pseudo-code of coherent hierarchical culling using multiple queries.

Figure 3.6: (left) all passes are rendered with CHC. Note that the soldiers are visible through the tree. (right) Only the solid passes are rendered using CHC, afterwards the transparent passes.

Figure 3.7: We can correctly handle shadow volumes together with CHC.

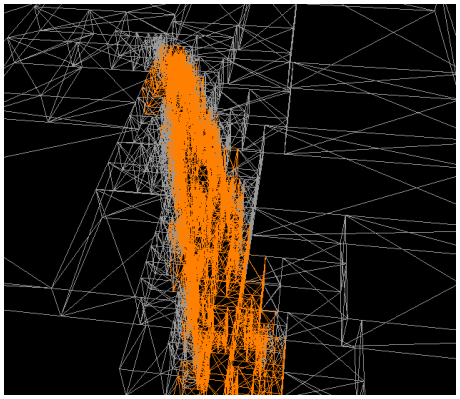


Figure 3.8: Visibility classification of the kD-tree nodes in the city scene. The orange nodes were found visible, all the other depicted nodes are invisible. Note the increasing size of the occluded nodes with increasing distance from the visible set.

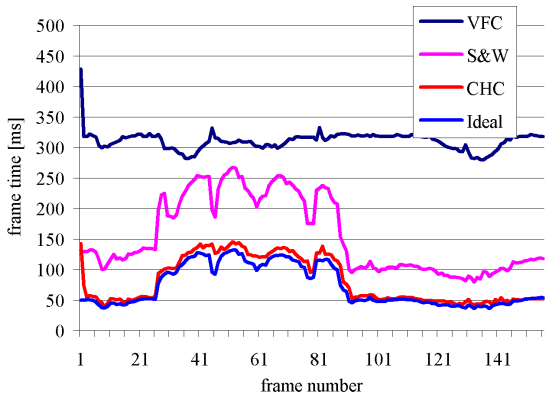


Figure 3.9: Frame times for the teapot scene.

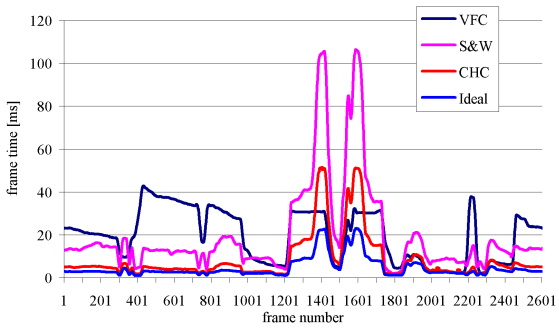


Figure 3.10: Frame times for the city walkthrough. Note the spike around frame 1600, where the viewpoint was elevated above the roofs, practically eliminating any occlusion.

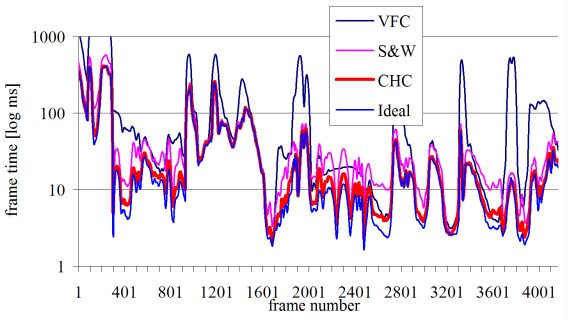


Figure 3.11: Frame times for the power plant walkthrough. The plot shows the weakness of the S&W method: when there is not much occlusion it becomes slower than VFC (near frame 2200). The CHC can keep up even in these situations and in the same time it can exploit occlusion when it appears (e.g. near frame 3700).

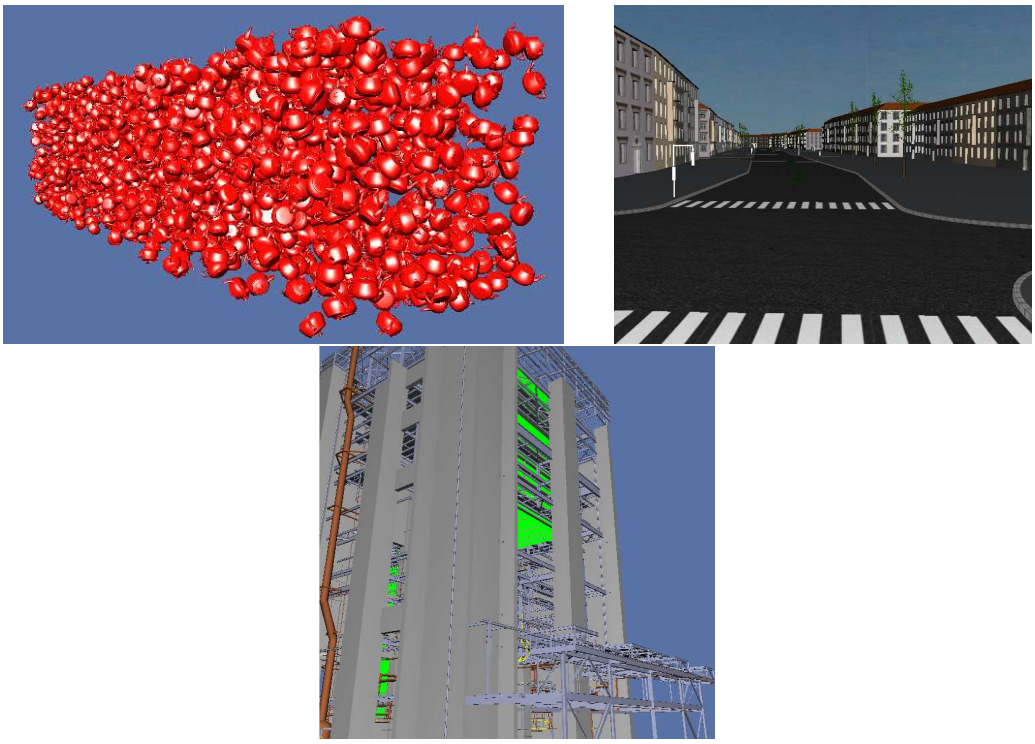


Figure 3.12: The test scenes: the teapots, the city, and the power plant.

Chapter 4

Global Visibility Sampling

The proposed visibility preprocessing framework consists of two major steps.

- The first step is an aggressive visibility sampling which gives initial estimate about global visibility in the scene. The sampling itself involves several strategies which will be described bellow. The important property of the aggressive sampling step is that it provides a fast progressive solution to global visibility and thus it can be easily integrated into the game development cycle. The aggressive sampling will terminate when the average contribution of new ray samples falls below a predefined threshold.
- The second step is mutual visibility verification. This step turns the previous aggressive visibility solution into either exact, conservative or error bound aggressive solution. The choice of the particular verifier is left on the user in order to select the best one for a particular scene, application context and time constrains. For example, in scenes like a forest an error bound aggressive visibility can be the best compromise between the resulting size of the PVS (and frame rate) and the visual quality. The exact or conservative algorithm can however be chosen for urban scenes where omission of even small objects can be more distracting for the user. The mutual visibility verification will be described in the next chapter.

In traditional visibility preprocessing the view space is subdivided into view cells and for each view cell the set of visible objects — potentially visible set (PVS) is computed. This framework has been used for conservative, aggressive and exact algorithms.

We propose a different strategy which has several advantages for sampling based aggressive visibility preprocessing. The strategy is based on the following fundamental ideas:

- Compute progressive global visibility instead of sequential from-region visibility
- Replace the roles of view cells and objects for some parts of the computation

Both these points will be addressed in this chapter in more detail.

4.1 Related work

Below we briefly discuss the related work on visibility preprocessing in several application areas. In particular we focus on computing from-region which has been a core of most previous visibility preprocessing techniques.

4.1.1 Aspect graph

The first algorithms dealing with from-region visibility belong to the area of computer vision. The aspect graph [GM90, PDS90, Soj95] partitions the view space into cells that group viewpoints

from which the projection of the scene is qualitatively equivalent. The aspect graph is a graph describing the view of the scene (aspect) for each cell of the partitioning. The major drawback of this approach is that for polygonal scenes with n polygons there can be $\Theta(n^9)$ cells in the partitioning for unrestricted view space. A scale space aspect graph [EBD+93, SP93] improves robustness of the method by merging similar features according to the given scale.

4.1.2 Potentially visible sets

In the computer graphics community Airey [ARB90] introduced the concept of potentially visible sets (PVS). Airey assumes the existence of a natural subdivision of the environment into cells. For models of building interiors these cells roughly correspond to rooms and corridors. For each cell the PVS is formed by cells visible from any point of that cell. Airey uses ray shooting to approximate visibility between cells of the subdivision and so the computed PVS is not conservative.

This concept was further elaborated by Teller et al. [Tel92b, TS91b] to establish a conservative PVS. The PVS is constructed by testing the existence of a stabbing line through a sequence of polygonal portals between cells. Teller proposed an exact solution to this problem using Plücker coordinates [Tel92a] and a simpler and more robust conservative solution [Tel92b]. The portal based methods are well suited to static densely occluded environments with a particular structure. For less structured models they can face a combinatorial explosion of complexity [Tel92b]. Yagel and Ray [YR95] present an algorithm, that uses a regular spatial subdivision. Their approach is not sensitive to the structure of the model in terms of complexity, but its efficiency is altered by the discrete representation of the scene.

Plantinga proposed a PVS algorithm based on a conservative viewspace partitioning by evaluating visual events [Pla93]. The construction of viewspace partitioning was further studied by Chrysanthou et al. [CCOZ98], Cohen-Or et al. [COFHZ98] and Sadagic [SS00]. Sudarsky and Gotsman [SG96] proposed an output-sensitive visibility algorithm for dynamic scenes. Cohen-Or et al. [COZ98] developed a conservative algorithm determining visibility of an ϵ-neighborhood of a given viewpoint that was used for network based walkthroughs.

Conservative algorithms for computing PVS developed by Durand et al. [DDTP00] and Schaufler et al. [SDDS00] make use of several simplifying assumptions to avoid the usage of 4D data structures. Wang et al. [WBP98] proposed an algorithm that precomputes visibility within beams originating from the restricted viewpoint region. The approach is very similar to the 5D subdivision for ray tracing [SD94] and so it exhibits similar problems, namely inadequate memory and preprocessing complexities. Specialized algorithms for computing PVS in $2\frac{1}{2}$D scenes were proposed by Wonka et al. [WWS00], Koltun et al. [KCCO01], and Bittner et al. [BWW01].

The exact mutual visibility method presented later in the report is based on method exploting Plücker coordinates of lines [Bit02, NBG02, HMN05]. This algorithm uses Plücker coordinates to compute visibility in shafts defined by each polygon in the scene.

4.1.3 Rendering of shadows

The from-region visibility problems include the computation of soft shadows due to an areal light source. Continuous algorithms for real-time soft shadow generation were studied by Chin and Feiner [CF92], Loscos and Drettakis [LD97], and Chrysanthou [Chr96] and Chrysanthou and Slater [CS97b]. Discrete solutions have been proposed by Nishita [NN85], Brotman and Badler [BB84], and Soler and Sillion [SS98]. An exact algorithm computing an antipenumbra of an areal light source was developed by Teller [Tel92a].

4.1.4 Discontinuity meshing

Discontinuity meshing is used in the context of the radiosity global illumination algorithm or computing soft shadows due to areal light sources. First approximate discontinuity meshing algorithms were studied by Campbell [CF90, Cam91], Lischinski [LTG92], and Heckbert [Hec92]. More elaborate methods were developed by Drettakis [Dre94, DF94], and Stewart and Ghali [SG93, SG94].

These methods are capable of creating a complete discontinuity mesh that encodes all visual events involving the light source.

The classical radiosity is based on an evaluation of form factors between two patches [SH93]. The visibility computation is a crucial step in the form factor evaluation [TH93b, HW94, TFFH94, NS96, TT]. Similar visibility computation takes place in the scope of hierarchical radiosity algorithms [SS96, DS97, DSSD97].

4.1.5 Global visibility

The aim of global visibility computations is to capture and describe visibility in the whole scene [DDP96]. The global visibility algorithms are typically based on some form of line space subdivision that partitions lines or rays into equivalence classes according to their visibility classification. Each class corresponds to a continuous set of rays with a common visibility classification. The techniques differ mainly in the way how the line space subdivision is computed and maintained. A practical application of most of the proposed global visibility structures for 3D scenes is still an open problem. Prospectively these techniques provide an elegant method for ray shooting acceleration — the ray shooting problem can be reduced to a point location in the line space subdivision.

Pocchiola and Vegter introduced the visibility complex [PV93] that describes global visibility in 2D scenes. The visibility complex has been applied to solve various 2D visibility problems [Riv95, Riv97b, Riv97a, ORDP96]. The approach was generalized to 3D by Durand et al. [DDP96]. Nevertheless, no implementation of the 3D visibility complex is currently known. Durand et al. [DDP97] introduced the visibility skeleton that is a graph describing a skeleton of the 3D visibility complex. The visibility skeleton was verified experimentally and the results indicate that its $O(n^4 \log n)$ worst case complexity is much better in practice. Pu [Pu98] developed a similar method to the one presented in this chapter. He uses a BSP tree in Plücker coordinates to represent a global visibility map for a given set of polygons. The computation is performed considering all rays piercing the scene and so the method exhibits unacceptable memory complexity even for scenes of moderate size. Recently, Duguet and Drettakis [DD02] developed a robust variant of the visibility skeleton algorithm that uses robust epsilon-visibility predicates.

Discrete methods aiming to describe visibility in a 4D data structure were presented by Chrysanthou et al. [CCOL98] and Blais and Poulin [BP98]. These data structures are closely related to the lumigraph [GGSC96, BBM+01] or light field [LH96]. An interesting discrete hierarchical visibility algorithm for two-dimensional scenes was developed by Hinkenjann and Müller [HM96]. One of the biggest problems of the discrete solution space data structures is their memory consumption required to achieve a reasonable accuracy. Prospectively, the scene complexity measures [CS97a] provide a useful estimate on the required sampling density and the size of the solution space data structure.

4.1.6 Other applications

Certain from-point visibility problems determining visibility over a period of time can be transformed to a static from-region visibility problem. Such a transformation is particularly useful for antialiasing purposes [Gra85]. The from-region visibility can also be used in the context of simulation of the sound propagation [FCE+98]. The sound propagation algorithms typically require lower resolution than the algorithms simulating the propagation of light, but they need to account for simulation of attenuation, reflection and time delays.

4.2 Algorithm Description

This section first describes the setup of the global visibility sampling algorithm. In particular we describe the view cell representation and the novel concept of from-object based visibility. The we outline the different visibility sampling strategies.

4.2.1 View Space Partitioning

Before the visibility computation itself, we subdivide the space of all possible viewpoints and viewing directions into view cells. A good partition of the scene into view cells is an essential part of every visibility system. If they are chosen too large, the PVS (Potentially Visible Set) of a view cells is too large for efficient culling. If they are chosen too small or without consideration, then neighbouring view cells contain redundant PVS information, hence boosting the PVS computation and storage costs for the scene. In the left image of figure 4.1 we see view cells of the Vienna model, generated by triangulation of the streets. In the closeup (right image) we can see that each triangle is extruded to a given height to form a view cell prism.

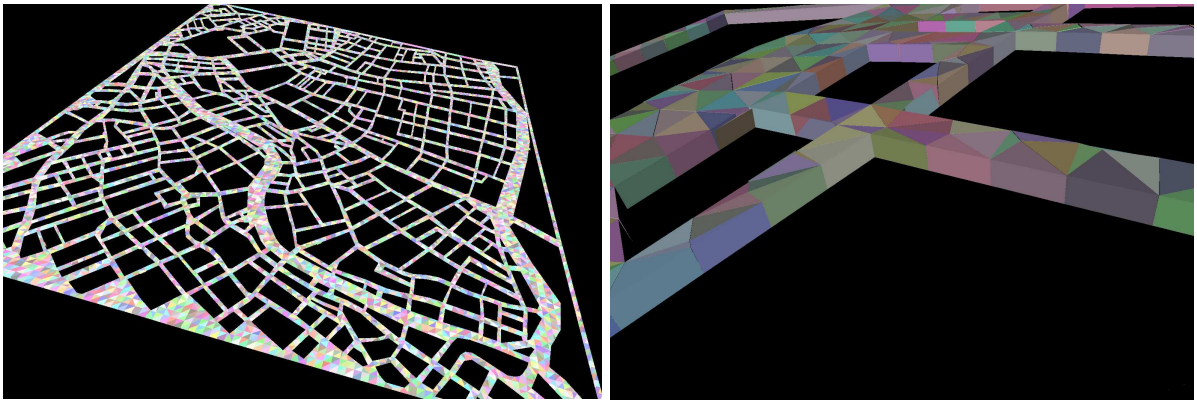


Figure 4.1: (left) Vienna view cells. (right) The view cells are prisms with a triangular base.

In order to efficiently use view cells with our sampling method, we require a view cell representation which is

- optimized for view cell - ray intersection.
- flexible, i.e., it can represent arbitrary geometry.
- naturally suited for a hierarchical approach.

We meet these requirements by employing spatial subdivisions (i.e., KD trees and BSP trees), to store the view cells. The initial view cells are associated with the leaves. The reason why we chose BSP trees as view cell representation is that they are very flexible. View cells forming arbitrary closed meshes can be closely matched. Therefore we are able to find a view cells with only a few view ray-plane intersections. Furthermore, the hierarchical structures can be exploited as hierarchy of view cells. Interior nodes form larger view cells containing the children. If necessary, a leaf can be easily subdivided into smaller view cells.

Currently we consider three different approaches to generate the initial view cell BSP tree. The third method is not restricted to BSP trees, but BSP trees are preferred because of their greater flexibility.

- A number of input view cells is given in advance, and we insert them into a BSP tree (i.e., we are changing their representation for a fast BSP tree lookup). As input view cell any closed mesh can be applied. The only requirement is that the any two view cells do not overlap. The view cell polygons are extracted, storing a pointer to the parent view cell with the polygon. The BSP is build from these polygons using some global optimizations like tree balancing or least splits. The polygons guide the split process as they are filtered down the tree. The subdivision terminates when there is only one polygon left, which is coincident to the last split plane. Then two leaves are created and the view cell pointer (stored with the polygon) is inserted into the leaf representing the inside of the view cell. One input view cell

Input	Vienna view cells selection	Vienna view cells full	Vienna simple scene
method	insert input viewcells	insert input view cells	generate from scene polygons
#view cells	105	16447	4867
#input polygons	525	82235	16151
BSP tree generation time	0.016s	10.328s	0.61s
#nodes	1137	597933	9733
#interior nodes	568	298966	4866
#leaf nodes	569	298967	4867
#splits	25	188936	2010
max tree depth	13	27	17
avg tree depth	9.747	21.11	12.48

Table 4.1: Statistics for the view cell BSP tree. In the first column we insert a selection of given view cells from the Vienna scene into a BSP tree. In the second column we do the same for the full Vienna view cell set. In the third column we generate new view cells using a BSP tree subdivision of the Vienna simple scene. The termination criterion was to stop subdivision if there are 3 or less polygons per node.

can be associated with many leaves in case a view cell was split during the traversal. On the other hand, each leafs corresponds to exactly one or no view cell.

However, sometimes a good set of view cells is not available. Or the scene is changed frequently, and the designer does not want to create new view cells on each change. In such a case one of the following two methods should rather be chosen, which generate view cells automatically.

- We apply a BSP tree subdivision to the scene geometry. Whenever the subdivision terminates in a leaf, a view cell is associated with the leaf node. This simple approach is justified because it places the view cell borders along some discontinuities in the visibility function.

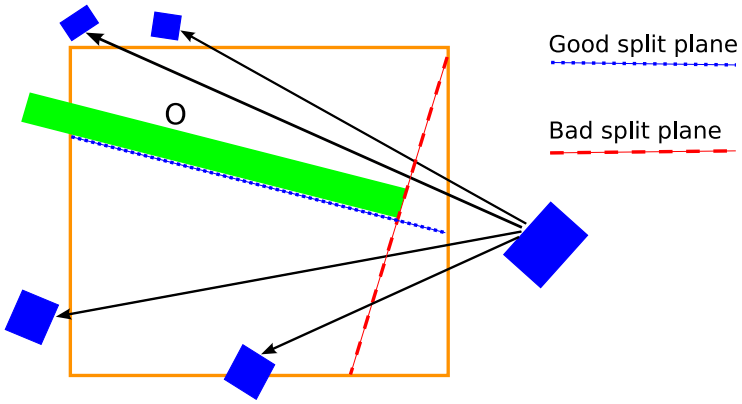


Figure 4.2: A good view cell partition with respect to the sample rays piercing the scene objects and the view cell minimizes the number of rays piercing more than one view cell. During subdivision, this can be achieved by aligning the split plane with one of the long sides of occluder O.

- The view cell generation can be guided by the sampling process. We start with with a single initial view cell representing the whole space. If a given threshold is reached during the preprocessing (e.g., the view cell is pierced by too many rays resulting in a large PVS), the view cell is subdivided into smaller cells using some criteria.

  In order to evaluate the best split plane, we first have to define the characteristics of a good view cell partition: The view cells should be quite large, while their PVS stays rather small.

The PVS of each two view cells should be as distinct as possible, otherwise they could be merged into a larger view cell if the PVSs are too similar. E.g., for a building, the perfect view cells are usually the single rooms connected by portals.

Hence we can define some useful criteria for the split: 1) the number of rays should be roughly equal among the new view cells. 2) The split plane should be chosen in a way that the ray sets are disjoint, i.e., the number of rays contributing to more than one cell should be minimized. 3) For BSP trees, the split plane should be aligned with some scene geometry which is large enough to contribute a lot of occlusion power. This criterion can be naturally combined with the second one. As termination criterion we can choose the minimum PVS / piercing ray size or the maximal tree depth. An illustration of a good and a bad choice of a split plane is given in figure 4.2.

Some statistics about the first two methods (i.e., the insertion of the view cells into the BSP tree, and the automatic generation from the scene polygons using a BSP tree subdivision) is given in table 4.1. We used a selection from given view cells for the Vienna scene for the first column, the full set for the second column, and the Vienna simple scene geometry for the automatic view cell generation. The measurements were conducted on a PC with 3.4GHz P4 CPU.

4.2.2 From-Object Based Visibility

Our framework is based on the idea of sampling visibility by casting casting rays through the scene and collecting their contributions. A visibility sample is computed by casting a ray from an object towards the view cells and computing the nearest intersection with the scene objects. All view cells pierced by the ray segment can the object and thus the object can be added to their PVS. If the ray is terminated at another scene object the PVS of the pierced view cells can also be extended by this terminating object. Thus a single ray can make a number of contributions to the progressively computed PVSs. A ray sample piercing n view cells which is bound by two distinct objects contributes by at most $2 * n$ entries to the current PVSs. Apart from this performance benefit there is also a benefit in terms of the sampling density: Assuming that the view cells are usually much larger than the objects (which is typically the case) starting the sampling deterministically from the objects increases the probability of small objects being captured in the PVS.

At this phase of the computation we not only start the samples from the objects, but we also store the PVS information centered at the objects. Instead of storing a PVS consisting of objects visible from view cells, every object maintains a PVS consisting of potentially visible view cells. While these representations contain exactly the same information as we shall see later the object centered PVS is better suited for the importance sampling phase as well as the visibility verification phase.

4.2.3 Naive Randomized Sampling

The naive global visibility sampling works as follows: At every pass of the algorithm visits scene objects sequentially. For every scene object we randomly choose a point on its surface. Then a ray is cast from the selected point according to the randomly chosen direction (see Figure 4.3). We use a uniform distribution of the ray directions with respect to the half space given by the surface normal. Using this strategy the samples at deterministically placed at every object, with a randomization of the location on the object surface. The uniformly distributed direction is a simple and fast strategy to gain initial visibility information.

The described algorithm accounts for the irregular distribution of the objects: more samples are placed at locations containing more objects. Additionally every object is sampled many times depending on the number of passes in which this sampling strategy is applied. This increases the chance of even a small object being captured in the PVS of the view cells from which it is visible.

Each ray sample can contribute by a associating a number of view cells with the object from which the sample was cast. If the ray does not leave the scene it also contributes by associating the pierced view cells to the terminating object. Thus as the ray samples are cast we can measure

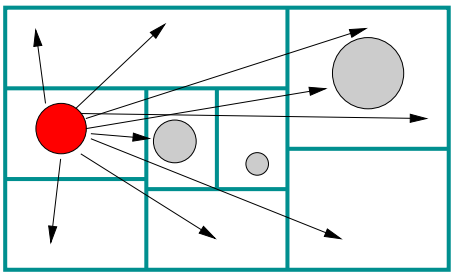


Figure 4.3: Three objects and a set of view cells corresponding to leaves of an axis aligned BSP tree. The figure depicts several random samples cast from a selected object (shown in red). Note that most samples contribute to more view cells.

the average contribution of a certain number of most recent samples. If this contribution falls below a predefined constant we move on to the next sampling strategy, which aim to discover more complicated visibility relations.

4.2.4 Accounting for View Cell Distribution

The first modification to the basic algorithm accounts for irregular distribution of the view cells. Such a case is common for example in urban scenes where the view cells are mostly distributed in a horizontal direction and more view cells are placed at denser parts of the city. The modification involves replacing the uniformly distributed ray direction by directions distributed according to the local view cell directional density. This means placing more samples at directions where more view cells are located: We select a random view cell which lies at the half space given by the surface normal at the chosen point. We pick a random point inside the view cell and cast a ray towards this point.

4.2.5 Accounting for Visibility Events

Visibility events correspond to appearance and disappearance of objects with respect to a moving view point. In polygonal scenes the events defined by event surfaces defined by three distinct scene edges. Depending on the edge configuration we distinguish between vertex-edge events (VE) and triple edge (EEE) events. The VE surfaces are planar planes whereas the EEE are in general quadratic surfaces.

To account for these events we explicitly place samples passing by the object edges which are directed to edges and/or vertices of other objects. In this way we perform stochastic sampling at boundaries of the visibility complex [DDP96].

The first strategy starts similarly to the above described sampling methods: we randomly select an object and a point on its surface. Then we randomly pickup an object from its PVS. If we have mesh connectivity information we select a random silhouette edge from this object and cast a sample which is tangent to that object at the selected edge.

The second strategy works as follows: we randomly pickup two objects which are likely to see each other. Then we determine a ray which is tangent to both objects. For simple meshes the determination of such rays can be computed geometrically, for more complicated ones it is based again on random sampling. The selection of the two objects works as follows: first we randomly select the first object and a random non-empty view cell for which we know that it can see the object. Then we randomly select an object associated with that view cell as the second object.

4.3 Summary

This chapter described the global visibility sampling algorithm which forms a core of the visibility preprocessing framework. The global visibility sampling computes aggressive visibility, i.e. it computes a subset of the exact PVS for each view cell. The aggressive sampling provides a fast progressive solution and thus it can be easily integrated into the game development cycle. The sampling itself involves several strategies which aim to progressively discover more visibility relationships in the scene.

The methods presented in this chapter give a good initial estimate about visibility in the scene, which can be verified by the mutual visibility algorithms described in the next chapter.

Chapter 5

Mutual Visibility Verification

The aggressive visibility sampling discussed in the previous chapter is a good candidate if a fast visibility solution is required (e.g. during the development cycle). However for final solution (production) we should either be sure that there can be no visible error due to the preprocessed visibility or the error is sufficiently small and thus not noticeable.

The mutual visibility verification starts with identifying a minimal set of object/view cell pairs which are classified as mutually invisible. We process the objects sequentially and for every given object we determine the view cells which form the boundary of the invisible view cell set.

This boundary it is based on the current view cell visibility classifications. We assume that there is a defined connectivity between the view cells which is the case for both BSP-tree or kD-tree based view cells. Then given a PVS consisting of visible view cells (with respect to an object) we can find all the neighbors of the visible view cells which are invisible. In order words a view cell belongs to this boundary if it is classified invisible and has at least one visible neighbor. If all view cells from this boundary are proven invisible, no other view cell (behind this boundary) can be visible. If the verification determines some view cells as visible, the current invisible boundary is extended and the verification is applied on the new view cells forming the boundary.

The basic operation of the verification is mutual visibility test between an object and a view cell. This test works with a bounding volume of the object and the boundary of the view cell. In the most general case both are defined bound by a convex polyhedron, in a simpler case both are defined by an axis aligned bounding box.

Below, we describe three different mutual visibility verification algorithms. The first algorithm which is described in most detail computes exact visibility. The second one is a conservative algorithm and the third one is an approximate algorithm with a guaranteed error bound.

5.1 Exact Verifier

The exact mutual visibility verifier computes exact visibility between two polyhedrons in the scene. This is computed by testing visibility between all pairs of potentially visible polygons of these polyhedrons. For each pair of tested polygons the computation is localized into the shaft defined by their convex hull. This shaft is used to determine the set of relevant occluders [HW94].

5.1.1 Occlusion tree

The occlusion tree for the visibility from region problem is a 5D BSP tree maintaining a collection of the 5D blocker polyhedra [Bit02]. The tree is constructed for each source polygon P_S and represents all rays emerging from P_S. Each node N of the tree represents a subset of line space $\mathcal{Q}_N^*$. The root of the tree represents the whole problem-relevant line set $\mathcal{L}_R$. If N is an interior node, it is associated with a Plücker plane $\hat{\boldsymbol{\omega}}_N$. Left child of N represents $\mathcal{Q}_N^* \cap \hat{\boldsymbol{\omega}}_N^-$, right child $\mathcal{Q}_N^* \cap \hat{\boldsymbol{\omega}}_N^+$, where $\hat{\boldsymbol{\omega}}_N^-$ and $\hat{\boldsymbol{\omega}}_N^+$ are halfspaces induced by $\hat{\boldsymbol{\omega}}_N$.

Leaves of the occlusion tree are classified *in* or *out*. If N is an *out*-leaf, $\mathcal{Q}_N^*$ represents unoccluded rays emerging from the source polygon P_S. If N is an *in*-leaf, it is associated with an occluder O_N that blocks the corresponding set of rays $\mathcal{Q}_N^*$. Additionally N stores a fragment of the blocker polyhedron B_N representing $\mathcal{Q}_N^*$. The intersection of B_N and the Plücker quadric corresponds to a set of stabbers $\mathcal{S}_N$ through which O_N is visible from P_S. A 2D example of an occlusion tree is shown at Figure 5.1.

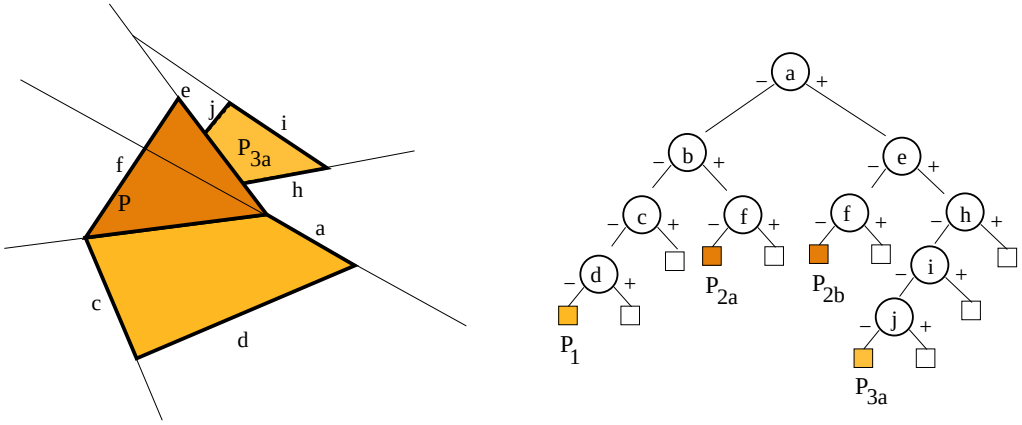


Figure 5.1: A 2D example of an occlusion tree. Rays intersecting scene polygons are represented by polyhedra P1, P2 and P3. The occlusion tree represents a union of the polyhedra. Note that P2 has been split during the insertion process.

5.1.2 Occlusion tree construction

The occlusion tree is constructed incrementally by inserting blocker polyhedra in the order given by the size of the polygons. When processing a polygon P_j the algorithm inserts a polyhedron $B_{P_S P_j}$ representing the feasible line set between the source polygon P_S and the polygon P_j. The polyhedron is split into fragments that represent either occluded or unoccluded rays.

We describe two methods that can be used to insert a blocker polyhedron into the occlusion tree. The first method inserts the polyhedron by splitting it using hyperplanes encountered during the traversal of the tree. The second method identifies hyperplanes that split the polyhedron and uses them later for the construction of polyhedron fragments in leaf nodes.

Insertion with splitting

The polyhedron insertion algorithm maintains two variables — the current node N_c and the current polyhedron fragment B_c. Initially N_c is set to the root of the tree and B_c equals to $B_{P_S P_j}$. The insertion of a polyhedron in the tree proceeds as follows: If N_c is an interior node, we determine the position of B_c and the hyperplane $\hat{\boldsymbol{\omega}}_{N_c}$ associated with N_c. If B_c lies in the positive halfspace induced by $\hat{\boldsymbol{\omega}}_{N_c}$ the algorithm continues in the right subtree. Similarly, if B_c lies in the negative halfspace induced by $\hat{\boldsymbol{\omega}}_{N_c}$, the algorithm continues in the left subtree. If B_c intersects both halfspaces, it is split by $\hat{\boldsymbol{\omega}}_{N_c}$ into two parts B_c^+ and B_c^- and the algorithm proceeds in both subtrees of N_c with relevant fragments of B_c.

If N_c is a leaf node then we make a decision depending on its classification. If N_c is an *out*-leaf then B_c is visible and N_c is replaced by the elementary occlusion tree of B_c. If N_c is an *in*-leaf it corresponds to already occluded rays and no modification to the tree necessary. Otherwise we need to merge B_c into the tree. The merging replaces N_c by the elementary occlusion tree of B_c and inserts the old fragment B_{N_c} in the new subtree.

Insertion without splitting

The above described polyhedron insertion algorithm requires that the polyhedron is split by the hyperplanes encountered during the traversal of the occlusion tree. Another possibility is an algorithm that only tests the position of the polyhedron with respect to the hyperplane and remembers the hyperplanes that split the polyhedron on the path from the root to the leaves. Reaching a leaf node these hyperplanes are used to construct the corresponding polyhedron fragment using a polyhedron enumeration algorithm.

The splitting-free polyhedron insertion algorithm proceeds as follows: we determine the position of the blocker polyhedron and the hyperplane $\hat{\boldsymbol{\omega}}_{N_c}$ associated with the current node N_c. If B_c lies in the positive halfspace induced by $\hat{\boldsymbol{\omega}}_{N_c}$ the algorithm continues in the right subtree. Similarly if B_c lies in the negative halfspace induced by $\hat{\boldsymbol{\omega}}_{N_c}$ the algorithm continues in the left subtree. If B_c intersects both halfspaces the algorithm proceeds in both subtrees of N_c and $\hat{\boldsymbol{\omega}}_{N_c}$ is added to the list of splitting hyperplanes with a correct sign for each subtree. Reaching an *out*-leaf the list of splitting hyperplanes and the associated signs correspond to halfspaces bounding the corresponding polyhedron fragment. The polyhedron enumeration algorithm is applied using these halfspaces and the original halfspaces defining the blocker polyhedron. Note that it is possible that no feasible polyhedron exists since the intersection of halfspaces is empty. Such a case occurs due to the conservative traversal of the tree that only tests the position of the inserted polyhedron with respect to the splitting hyperplanes. If the fragment is not empty, the tree is extended as described in the previous section.

Reaching an *in*-leaf the polygon positional test is applied. If the inserted polygon is closer than the polygon associated with the leaf, the polyhedron fragment is constructed and it is merged in the tree as described in the previous section. The splitting-free polyhedron insertion algorithm has the following properties:

- If the polyhedron reaches only *in*-leaves the 5D set operations on the polyhedron are avoided completely.
- If the polyhedron reaches only a few leaves the application of the polyhedron enumeration algorithm is potentially more efficient than the sequential splitting. On the contrary, when reaching many *out*-leaves the splitting-free method makes less use of coherence, i.e. the polyhedron enumeration algorithm is applied independently in each leaf even if the corresponding polyhedra are bound by coherent sets of hyperplanes.
- An existing implementation of the polyhedron enumeration algorithm can be used [Fuk, Avi02].

The polyhedron enumeration algorithm constructs the polyhedron as an intersection of a set of halfspaces. The polyhedron is described as a set of vertices and rays and their adjacency to the hyperplanes bounding the polyhedron [FP96, AF96]. The adjacency information is used to construct a 1D skeleton of the polyhedron that is required for computation of the intersection with the Plücker quadric.

5.1.3 Visibility test

The visibility test classifies visibility of a given polygon with respect to the source polygon. The test can be used to classify visibility of a polyhedral region by applying it on the boundary faces of the region and combining the resulting visibility states.

The exact visibility test for a given polyhedral region proceeds as follows: for each face of the region facing the given source polygon we construct a blocker polyhedron. The blocker polyhedron is then tested for visibility by the traversal of the occlusion tree. The visibility test proceeds as the algorithms described in Section 5.1.2, but no modifications to the tree are performed. If the polyhedron is classified as visible in all reached leaves, the current face is fully visible. If the polyhedron is invisible in all reached leaves, the face is invisible. Otherwise it is partially visible

since some rays connecting the face and the source polygon are occluded and some are unoccluded. The visibility of the whole region can computed using a combination of visibility states of its boundary faces according to Table 5.1. However, since we are interested only in verification of mutual invisibility as soon as we find a first visible fragment the test can be terminated.

Fragment A	Fragment B	A ∪ B
F	F	F
I	I	I
I	F	P
F	I	P
P	∗	P
∗	P	P

I – invisible
P – partially visible
F – fully visible
∗ – any of the I,P,F states

Table 5.1: Combining visibility states of two fragments.

5.1.4 Optimizations

Below we discuss several optimization techniques that can be used to improve the performance of the algorithm. The optimizations do not alter the accuracy of the visibility algorithm.

Visibility estimation

The visibility estimation aims to eliminate the polyhedron enumeration in the leaves of the occlusion tree. If we find out that the currently processed polygon is potentially visible in the given leaf-node (it is an *out*-leaf or it is an *in*-leaf and the positional test reports the polygon as the closest), we estimate its visibility by shooting random rays. We can use the current occlusion tree to perform ray shooting in line space. We select a random ray connecting the source polygon and the currently processed polygon. This ray is mapped to a Plücker point and this point is tested for inclusion in halfspaces defined by the Plücker planes splitting the polyhedron on the path from to root to the given leaf. If the point is contained in all tested halfspaces the corresponding ray is unoccluded and the algorithm inserts the blocker polyhedron into the tree. Otherwise it continues by selecting another random ray until a predefined number of rays was tested.

The insertion of the blocker polyhedron devotes further discussion. Since the polyhedron was not enumerated we do not know which of its bounding hyperplanes really bound the polyhedron fragment and which are redundant for the given leaf. Considering all hyperplanes defining the blocker polyhedron could lead to inclusion of many redundant nodes in the tree. We used a simple conservative algorithm that tests if the given hyperplane is bounding the (unconstructed) polyhedron fragment. For each hyperplane H_i bounding the blocker polyhedron the algorithm tests the position of extremal lines embedded in this hyperplane with respect to each hyperplane splitting the polyhedron. If mappings of all extremal lines lay in the same open halfspace defined by a splitting hyperplane, hyperplane H_i does not bound the current polyhedron fragment and thus it can be culled.

Visibility merging

Visibility merging aims to propagate visibility classifications from the leaves of the occlusion tree up into the interior nodes of the hierarchy. Visibility merging is connected with the approximate occlusion sweep, which simplifies the treatment of the depth of the scene polygons.

The algorithm classifies an interior node of the occlusion tree as occluded (*in*) if the following conditions hold:

- Both its children are *in*-nodes.

- The occluders associated with both children are strictly closer than the closest unswept node of the spatial hierarchy.

The first condition ensures that both child nodes correspond to occluded nodes. The second condition ensures that any unprocessed occluder is behind the occluders associated with the children. Using this procedure the effective depth of the occlusion becomes progressively smaller if more and more rays become occluded.

5.2 Conservative Verifier

A conservative verifier is a faster alternative to the exact visibility verifier described above. The verifier is an extension of the strong occlusion algorithm of Cohen-Or et al. [COFHZ98]. In particular our verifier refines the search for a strong occluder by using a hierarchical subdivision of space of lines connecting the two regions tested for mutual visibility. Initially the shaft bounding the the tested regions is constructed. Rays bounding the shaft are traced through the scene and we compute all intersections with the scene objects between the tested regions. The the algorithm proceeds as follows:

- In the case that any ray does not intersect any object the tested regions are classified as visibility and the algorithm terminates.
- If the rays intersect the same convex object (at any depth) this object is a strong occluder with respect to the shaft and thus it also hides all rays in the corresponding shaft.
- If the rays do not intersect a single convex object four new shafts are constructed by splitting both regions in half and the process is repeated recursively.

If the subdivision does not terminate till reaching a predefined subdivision depth, we conservatively classify the tested regions as mutually visible. The conservative verifier is illustrated at Figure 5.2.

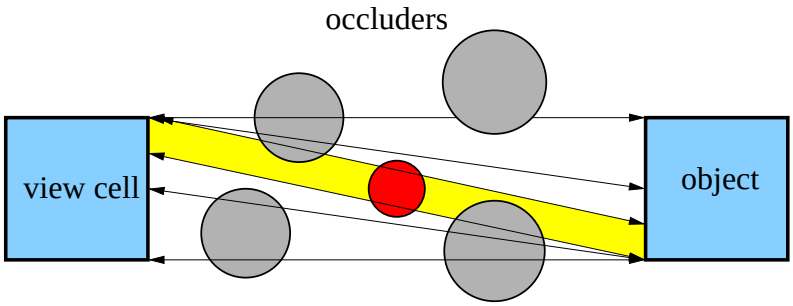


Figure 5.2: An example of the conservative visibility verification. The figure shows two tested regions (the view cell and the object) and several occluders. For sake of clarity we only show samples which are boundaries of shafts on one path of the line space subdivision tree. The subdivision terminates at the yellow shaft since a single strong occluder has been found.

5.3 Error Bound Approximate Verifier

The approximate verifier will be based on the similar idea as the conservative one. However it will behave differently in the finer subdivision of the ray shafts. The idea is to use the above algorithm as far as the shafts get small enough that we can neglect objects which can be seen through such a shaft. Even if not all rays inside the shaft are not blocked by scene objects, a pixel error induced due to omission of objects potential visible behind the shaft will be below a given threshold.

For the computation of the maximal error due to the current shaft we assume that one tested region is a view cell, whereas the other is an object bounding box or cell of the spatial hierarchy. The threshold is computed as follows: We first triangulate the farthest intersection points in the shaft as seen from the view cell side of the shaft. Then for each computed triangle we calculate a point in the view cell which maximizes the projected area of the triangle (see Figure 5.3). The conservative estimate of the maximal error is then given by a sum of the computed projected areas. If this error is below a specified threshold we terminate the subdivision of the current shaft.

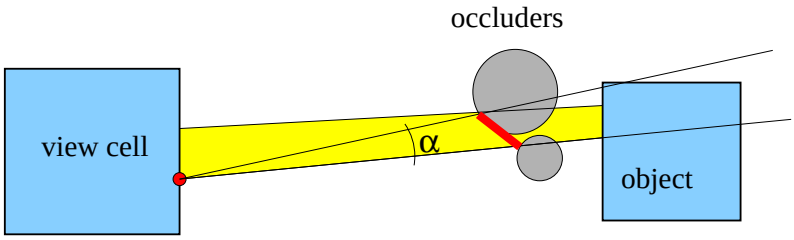


Figure 5.3: An example of the approximate error bound visibility verification. The figure shows two tested regions (the view cell and the object) and two occluders. Due to the 2D nature of the example the triangulation is depicted as the red line segment. The maximal error for the possible viewpoints in the shaft corresponds to the angle α.

5.4 Summary

This chapter presented a mutual visibility verification algorithms, which determine whether two regions in the scene are visible.

First, we have described an exact visibility algorithm which is a simple of modification of an algorithm for computing from-region visibility in polygonal scenes. The key idea is a hierarchical subdivision of the problem-relevant line set using Plücker coordinates and the occlusion tree. Plücker coordinates allow to perform operations on sets of lines by means of set theoretical operations on the 5D polyhedra. The occlusion tree is used to maintain a union of the polyhedra that represent lines occluded from the given region (polygon). We described two algorithms for construction of the occlusion tree by incremental insertion of blocker polyhedra. The occlusion tree was used to test visibility of a given polygon or region with respect to the source polygon/region. We proposed several optimization of the algorithm that make the approach applicable to large scenes. The principal advantage of the exact method over the conservative and the approximate ones is that it does not rely on various tuning parameters that are characterizing many conservative or approximate algorithms. On the other hand the exactness of the method requires higher computational demands and implementation effort.

Second, we have described a conservative verifier which is an extension of the algorithm based on strong occluders. The conservative verifier requires a specification of the shaft size at which the tested regions are conservatively classified as visible.

Third, we have described an approximate error bound verifier which extends the conservative verifier by using automatic termination criteria based on the estimation of maximal error for the given shaft.

Bibliography

[AF96] David Avis and Komei Fukuda. Reverse search for enumeration. Discrete Applied Mathematics, 6:21–46, 1996.

[ARB90] John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. Towards image realism with interactive update rates in complex virtual building environments. In 1990 Symposium on Interactive 3D Graphics, pages 41–50. ACM SIGGRAPH, 1990.

[Avi02] David Avis. Lrs polyhedra enumeration library, 2002. Available at http://cgm.cs.mcgill.ca/~avis/C/lrs.html.

[BB84] Lynne Shapiro Brotman and Norman I. Badler. Generating soft shadows with a depth buffer algorithm. IEEE Computer Graphics and Applications, 4(10):5–12, October 1984. CODEN ICGADZ. ISSN 0272-1716.

[BBM+01] Chris Buehler, Michael Bosse, Leonard McMillan, Steven J. Gortler, and Michael F. Cohen. Unstructured lumigraph rendering. In Computer Graphics (SIGGRAPH '01 Proceedings), pages 425–432, 2001.

[BH01] Jiří Bittner and Vlastimil Havran. Exploiting coherence in hierarchical visibility algorithms. Journal of Visualization and Computer Animation, John Wiley & Sons, 12:277–286, 2001.

[BHS98] Jiří Bittner, Vlastimil Havran, and Pavel Slavík. Hierarchical visibility culling with occlusion trees. In Proceedings of Computer Graphics International '98 (CGI'98), pages 207–219. IEEE, 1998.

[Bit02] Jiří Bittner. Hierarchical Techniques for Visibility Computations. PhD thesis, Czech Technical University in Prague, October 2002.

[BMH98] Dirk Bartz, Michael Meissner, and Tobias Hüttner. Extending graphics hardware for occlusion queries in opengl. In Proceedings of the 1998 Workshop on Graphics Hardware, pages 97–104, 1998.

[BP98] M. Blais and P. Poulin. Sampling visibility in three-space. In Proc. of the 1998 Western Computer Graphics Symposium, pages 45–52, April 1998.

[BW03] Jiří Bittner and Peter Wonka. Visibility in computer graphics. Environment and Planning B: Planning and Design, 30(5):729–756, September 2003.

[BWW01] Jiří Bittner, Peter Wonka, and Michael Wimmer. Visibility preprocessing for urban scenes using line space subdivision. In Proceedings of Pacific Graphics (PG'01), pages 276–284. IEEE Computer Society, Tokyo, Japan, 2001.

[BY98] J.-D. Boissonnat and M. Yvinec. Algorithmic Geometry. cambridge University Press, 1998.

[Cam91] A. T. Campbell, III. Modeling Global Diffuse Illumination for Image Synthesis. PhD thesis, CS Dept, University of Texas at Austin, December 1991. 155 pp. Tech. Report TR-91-39.

[Cat75] Edwin E. Catmull. Computer display of curved surfaces. In Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure, pages 11–17, May 1975.

[CCOL98] Y. Chrysanthou, D. Cohen-Or, and D. Lischinski. Fast approximate quantitative visibility for complex scenes. In Proceedings of Computer Graphics International '98 (CGI'98), pages 23–31. IEEE, NY, Hannover, Germany, June 1998.

[CCOZ98] Y. Chrysanthou, D. Cohen-Or, and E. Zadicario. Viewspace partitioning of densely occluded scenes. Abstract of a video presentation, at the 13th Annual ACM Symposium on Computational Geometry, Minnesota, pages 413–414, June 1998.

[CF90] A. T. Campbell, III and Donald S. Fussell. Adaptive mesh generation for global diffuse illumination. In Computer Graphics (SIGGRAPH '90 Proceedings), volume 24, pages 155–164, August 1990.

[CF92] Norman Chin and Steven Feiner. Fast object-precision shadow generation for areal light sources using BSP trees. In David Zeltzer, editor, Computer Graphics (1992 Symposium on Interactive 3D Graphics), volume 25, pages 21–30, March 1992.

[Chr96] Yiorgos Chrysanthou. Shadow Computation for 3D Interaction and Animation. PhD thesis, QMW, Dept of Computer Science, January 1996.

[COCSD03] D. Cohen-Or, Y. Chrysanthou, C. Silva, and F. Durand. A survey of visibility for walkthrough applications. IEEE Transactions on Visualization and Computer Graphics, 9(3):412–431, 2003.

[COFHZ98] D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario. Conservative visibility and strong occlusion for viewspace partitioning of densely occluded scenes. In EUROGRAPHICS'98, 1998.

[COZ98] Daniel Cohen-Or and Eyal Zadicario. Visibility streaming for network-based walkthroughs. In Graphics Interface, pages 1–7, June 1998.

[CS97a] F. Cazals and M. Sbert. Some integral geometry tools to estimate the complexity of 3d scenes. Technical Report RR-3204, The French National Institue for Research in Computer Science and Control (INRIA), July 1997.

[CS97b] Yiorgos Chrysanthou and Mel Slater. Incremental updates to scenes illuminated by area light sources. In Proceedings of Eurographics Workshop on Rendering, pages 103–114. Springer Verlag, June 1997.

[CT96] Satyan Coorg and Seth Teller. Temporally coherent conservative visibility. In Proceedings of the Twelfth Annual ACM Symposium on Computational Geometry, pages 78–87, May 1996.

[CT97] Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In Proceedings of the Symposium on Interactive 3D Graphics, pages 83–90. ACM Press, 1997. ISBN 0-89791-884-3.

[DD02] Florent Duguet and George Drettakis. Robust epsilon visibility. To appear in Computer Graphics (SIGGRAPH'02 Proceedings), 2002.

[DDP96] Frédo Durand, George Drettakis, and Claude Puech. The 3D visibility complex: A new approach to the problems of accurate visibility. In Proceedings of Eurographics Rendering Workshop '96, pages 245–256. Springer, June 1996.

[DDP97] Frédo Durand, George Drettakis, and Claude Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In Computer Graphics (Proceedings of SIGGRAPH '97), pages 89–100, 1997.

[DDTP00] Frédo Durand, George Drettakis, Joëlle Thollot, and Claude Puech. Conservative visibility preprocessing using extended projections. In Computer Graphics (Proceedings of SIGGRAPH 2000), pages 239–248, 2000. URL http://visinfo.zib.de/EVlib/Show?EVL-2000-60.

[DF94] George Drettakis and Eugene Fiume. A Fast Shadow Algorithm for Area Light Sources Using Backprojection. In Computer Graphics (Proceedings of SIGGRAPH '94), pages 223–230, 1994.

[Dre94] George Drettakis. Structured Sampling and Reconstruction of Illumination for Image Synthesis. CSRI Technical Report, Department of Computer Science, University of Toronto, Toronto, Ontario, January 1994.

[DS97] George Drettakis and François Sillion. Interactive update of global illumination using A line-space hierarchy. In Turner Whitted, editor, SIGGRAPH 97 Conference Proceedings, Annual Conference Series, pages 57–64. ACM SIGGRAPH, Addison Wesley, August 1997. ISBN 0-89791-896-7.

[DSSD97] Katja Daubert, Hartmut Schirmacher, François X. Sillion, and George Drettakis. Hierarchical lighting simulation for outdoor scenes. In Julie Dorsey and Philipp Slusallek, editors, Eurographics Rendering Workshop 1997, pages 229–238. Eurographics, Springer Wein, New York City, NY, June 1997. ISBN 3-211-83001-4.

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

[EBD+93] D.W. Eggert, K.W. Bowyer, C.R. Dyer, H.I. Christensen, and D.B. Goldgof. The scale space aspect graph. PAMI, 15(11):1114–1130, November 1993.

[FCE+98] Thomas Funkhouser, Ingrid Carlbom, Gary Elko, Gopal Pingali, Mohan Sondhi, and Jim West. A beam tracing approach to acoustic modeling for interactive virtual environments. In Computer Graphics (Proceedings of SIGGRAPH '98), pages 21–32, July 1998.

[FP96] K. Fukuda and A. Prodon. Double description method revisited. Lecture Notes in Computer Science, 1120:91–111, 1996. CODEN LNCSD9. ISSN 0302-9743.

[Fuk] Komei Fukuda. Cdd home page. http://www.ifor.math.ethz.ch.

[GGC97] Xianfeng Gu, Steven J. Gortier, and Michael F. Cohen. Polyhedral geometry and the two-plane parameterization. In Julie Dorsey and Philipp Slusallek, editors, Eurographics Rendering Workshop 1997, pages 1–12. Eurographics, Springer Wein, New York City, NY, June 1997. ISBN 3-211-83001-4.

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

[GKM93] N. Greene, M. Kass, and G. Miller. Hierarchical Z-buffer visibility. In Computer Graphics (Proceedings of SIGGRAPH '93), pages 231–238, 1993.

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

[GO97] Jacob E. Goodman and Joseph O'Rourke, editors. Handbook of Discrete and Computational Geometry. CRC Press, 1997.

[Gra85] C. W. Grant. Integrated analytic spatial and temporal anti-aliasing for polyhedra in 4-space. In B. A. Barsky, editor, Computer Graphics (SIGGRAPH '85 Proceedings), volume 19, pages 79–84, July 1985.

[Hec92] Paul S. Heckbert. Discontinuity meshing for radiosity. In Third Eurographics Workshop on Rendering, pages 203–216. Bristol, UK, May 1992.

[HM96] André Hinkenjann and Heinrich Müller. Hierarchical blocker trees for global visibility calculation. Research Report 621/1996, University of Dortmund, August 1996.

[HMC+97] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated occlusion culling using shadow frusta. In Proceedings of the Thirteenth ACM Symposium on Computational Geometry, pages 1–10. ACM Press, 1997.

[HMN05] Denis Haumont, Otso Mäkinen, and Shaun Nirenstein. A low dimensional framework for exact polygon-to-polygon occlusion queries. In Proceedings of Eurographics Symposium on Rendering, pages 211–222, 2005.

[HSLM02] Karl Hillesland, Brian Salomon, Anselmo Lastra, and Dinesh Manocha. Fast and simple occlusion culling using hardware-based depth queries. Technical Report TR02-039, Department of Computer Science, University of North Carolina - Chapel Hill, September 12 2002. URL ftp://ftp.cs.unc.edu/pub/publications/techreports/02-039.pdf.

[HW94] Eric A. Haines and John R. Wallace. Shaft culling for efficient ray-traced radiosity. In P. Brunet and F. W. Jansen, editors, Photorealistic Rendering in Computer Graphics (Proceedings of the Second Eurographics Workshop on Rendering). Springer-Verlag, New York, NY, 1994. also available via FTP from princeton.edu:/pub/Graphics/Papers.

[KCCO01] Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. Hardware-accelerated from-region visibility using a dual ray space. In Proceedings of the 12th EUROGRAPHICS Workshop on Rendering, 2001.

[KS01] James T. Klosowski and Cláudio T. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. IEEE Transactions on Visualization and Computer Graphics, 7(4):365–379, October 2001. CODEN ITVGEA. ISSN 1077-2626. URL http://www.computer.org/tvcg/tg2001/v0365abs.htm;http://dlib.computer.org/tg/books/tg2001/pdf/v0365.pdf.

[LD97] Celine Loscos and George Drettakis. Interactive high-quality soft shadows in scenes with moving objects. Computer Graphics Forum, 16(3):C219–C230, September 4–8 1997. CODEN CGFODY. ISSN 0167-7055.

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. In Holly Rushmeier, editor, SIGGRAPH 96 Conference Proceedings, Annual Conference Series, pages 31–42. ACM SIGGRAPH, Addison Wesley, August 1996. held in New Orleans, Louisiana, 04-09 August 1996.

[LSCO03] Tommer Leyvand, Olga Sorkine, and Daniel Cohen-Or. Ray space factorization for from-region visibility. ACM Transactions on Graphics (Proceedings of SIGGRAPH '03), 22(3):595–604, July 2003.

[LTG92] Dani Lischinski, Filippo Tampieri, and Donald P. Greenberg. Discontinuity meshing for accurate radiosity. IEEE Computer Graphics and Applications, 12(6):25–39, November 1992.

[MB90] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. Visual Computer, 6(6):153–65, 1990.

[NBG02] Shaun Nirenstein, Edwin Blake, and James Gain. Exact From-Region visibility culling. In Proceedings of EUROGRAPHICS Workshop on Rendering, pages 199–210, 2002.

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

[NS96] K. Nechvile and J. Sochor. Form-factor evaluation with regional BSP trees. In Winter School of Computer Graphics 1996, February 1996. held at University of West Bohemia, Plzen, Czech Republic, 12-16 February 1996.

[ORDP96] Rachel Orti, Stephane Riviere, Fredo Durand, and Claude Puech. Using the Visibility Complex for Radiosity Computation. In Lecture Notes in Computer Science (Applied Computational Geometry: Towards Geometric Engineering), volume 1148, pages 177–190. Springer-Verlag, Berlin, Germany, May 1996.

[PDS90] Harry Plantinga, Charles R. Dyer, and W. Brent Seales. Real-time hidden-line elimination for a rotating polyhedral scene using the aspect representation. In Proceedings of Graphics Interface '90, pages 9–16, May 1990.

[Pel97] M. Pellegrini. Ray shooting and lines in space. In Jacob E. Goodman and Joseph O'Rourke, editors, Handbook of Discrete and Computational Geometry, chapter 32, pages 599–614. CRC Press LLC, Boca Raton, FL, 1997.

[Pla93] Harry Plantinga. Conservative visibility preprocessing for efficient walkthroughs of 3D scenes. In Proceedings of Graphics Interface '93, pages 166–173. Canadian Information Processing Society, Toronto, Ontario, Canada, May 1993.

[Pu98] Fan-Tao Pu. Data Structures for Global Illumination and Visibility Queries in 3-Space. PhD thesis, University of Maryland, College Park, MD, 1998.

[PV93] M. Pocchiola and G. Vegter. The visibility complex. In Proc. 9th Annu. ACM Sympos. Comput. Geom., pages 328–337, 1993.

[Riv95] Stéphane Rivière. Topologically sweeping the visibility complex of polygonal scenes. In Proc. 11th Annu. ACM Sympos. Comput. Geom., pages C36–C37, 1995.

[Riv97a] S. Rivière. Dynamic visibility in polygonal scenes with the visibility complex. In Proc. 13th Annu. ACM Sympos. Comput. Geom., pages 421–423, 1997.

[Riv97b] Stéphane Rivière. Walking in the visibility complex with applications to visibility polygons and dynamic visibility. In Proc. 9th Canad. Conf. Comput. Geom., pages 147–152, 1997.

[SBS04a] Dirk Staneker, Dirk Bartz, and Wolfgang Strasser. Efficient multiple occlusion queries for scene graph systems. WSI Report (WSI-2004-6), 2004.

[SBS04b] Dirk Staneker, Dirk Bartz, and Wolfgang Straßer. Occlusion culling in OpenSG PLUS. Computers and Graphics, 28(1):87–92, February 2004. CODEN COGRD2. ISSN 0097-8493.

[SC97] M. Slater and Y. Chrysanthou. View volume culling using a probabilistic caching scheme. In Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST'97), pages 71–78. ACM Press, 1997.

[SD94] G. Simiakakis and A. M. Day. Five-dimensional adaptive subdivision for ray tracing. Computer Graphics Forum, 13(2):133–140, June 1994. CODEN CGFODY. ISSN 0167-7055.

[SDDS00] Gernot Schaufler, Julie Dorsey, Xavier Decoret, and François X. Sillion. Conservative volumetric visibility with occluder fusion. In Computer Graphics (Proceedings of SIGGRAPH 2000), pages 229–238, 2000. URL http://visinfo.zib.de/EVlib/Show?EVL-2000-59.

[SG93] A. James Stewart and Sherif Ghali. An Output Sensitive Algorithm for the Computation of Shadow Boundaries. In Canadian Conference on Computational Geometry, pages 291–296, August 1993.

[SG94] A. James Stewart and Sherif Ghali. Fast computation of shadow boundaries using spatial coherence and backprojections. In Computer Graphics (Proceedings of SIGGRAPH '94), pages 231–238, 1994.

[SG96] Oded Sudarsky and Craig Gotsman. Output-sensitive visibility algorithms for dynamic scenes with applications to virtual reality. Computer Graphics Forum, 15(3): C249–C258, September 1996. CODEN CGFODY. ISSN 0167-7055.

[SH93] Peter Schröder and Pat Hanrahan. On the form factor between two polygons. In Computer Graphics (Proceedings of SIGGRAPH '93), pages 163–164, 1993.

[SOG98] N. D. Scott, D. M. Olsen, and E. W. Gannett. An overview of the VISUALIZE fx graphics accelerator hardware. Hewlett-Packard Journal: technical information from the laboratories of Hewlett-Packard Company, 49(2):28–34, May 1998. CODEN HPJOAX. ISSN 0018-1153. URL http://www.hp.com/hpj/98may/tc-05-98.htm.

[Soj95] E. Sojka. Aspect graphs of three dimensional scenes. In Winter School of Computer Graphics 1995, pages 289–299, February 1995. held at University of West Bohemia, Plzen, Czech Republic, 14-18 February 1995.

[SP93] I. Shimshoni and J. Ponce. Finite resolution aspect graphs of polyhedral objects. In WQV93, pages 140–150, 1993.

[SS96] Cyril Soler and François Sillion. Accurate error bounds for multi-resolution visibility. In Xavier Pueyo and Peter Schröder, editors, Eurographics Rendering Workshop 1996, pages 133–142. Eurographics, Springer Wein, New York City, NY, June 1996. ISBN 3-211-82883-4.

[SS98] Cyril Soler and François Sillion. Fast calculation of soft shadow textures using convolution. In Computer Graphics (Proceedings of SIGGRAPH '98). ACM SIGGRAPH, July 1998.

[SS00] Amela Sadagic and Mel Slater. Dynamic polygon visibility ordering for head-slaved viewing in virtual environments. In Computer Graphics Forum, volume 19(2), pages 111–122. Eurographics Association, 2000. URL http://visinfo.zib.de/EVlib/Show?EVL-2000-336.

[Sto91] J. Stolfi. Oriented Projective Geometry: A Framework for Geometric Computations. Academic Press, 1991.

[Tel92a] Seth J. Teller. Computing the antipenumbra of an area light source. In Computer Graphics (Proceedings of SIGGRAPH '92), pages 139–148, July 1992.

[Tel92b] Seth J. Teller. Visibility Computations in Densely Occluded Polyhedral Environments. PhD thesis, CS Division, UC Berkeley, October 1992. Tech. Report UCB/CSD-92-708.

[TFFH94] Seth Teller, Celeste Fowler, Thomas Funkhouser, and Pat Hanrahan. Partitioning and ordering large radiosity computations. In Andrew Glassner, editor, Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994), Computer Graphics Proceedings, Annual Conference Series, pages 443–450. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[TH93a] S. Teller and M. Hohmeyer. Computing the lines piercing four lines. Technical Report UCB/CSD 93/161, UC Berkeley, April 1993.

[TH93b] Seth Teller and Pat Hanrahan. Global visibility algorithms for illumination computations. In Computer Graphics Proceedings, Annual Conference Series, 1993, pages 239–246, 1993.

[TS91a] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In Proceedings of SIGGRAPH '91, pages 61–69, July 1991.

[TS91b] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. In Computer Graphics (Proceedings of SIGGRAPH '91), pages 61–69, 1991.

[TT] Marek Teichmann and Seth Teller. A weak visibility algorithm with an application to an interactive walkthrough. Downloaded from the WWW.

[WA77] Kevin Weiler and Peter Atherton. Hidden surface removal using polygon area sorting. In Computer Graphics (SIGGRAPH '77 Proceedings), pages 214–222, July 1977.

[WB05] Michael Wimmer and Jiří Bittner. Hardware occlusion queries made useful. GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation, pages 91–108, 2005.

[WBP98] Yigang Wang, Hujun Bao, and Qunsheng Peng. Accelerated walkthroughs of virtual environments based on visibility preprocessing and simplification. In Eurographics '98, volume 17, pages 187–194, 1998.

[Wei99] Eric W. Weisstein. The CRC Concise Encyclopedia of Mathematics. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1999. ISBN 0-8493-9640-9. 1969 pp. LCCN QA5.W45 1999. US$79.95.

[WS99] Peter Wonka and Dieter Schmalstieg. Occluder shadows for fast walkthroughs of urban environments. In Computer Graphics Forum (Proceedings of EUROGRAPHICS '99), pages 51–60, September 1999.

[WWS00] Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In Proceedings of EUROGRAPHICS Workshop on Rendering, pages 71–82, 2000.

[WWS01] Peter Wonka, Michael Wimmer, and François X. Sillion. Instant visibility. In Computer Graphics Forum (Proceedings of EUROGRAPHICS '01), pages 411–421, 2001. URL http://visinfo.zib.de/EVlib/Show?EVL-2001-201.

[YN97] F. Yamaguchi and M. Niizeki. Some basic geometric test conditions in terms of pluecker coordinates and pluecker coefficients. Visual Comput., 13(1):29–41, 1997.

[YR95] R. Yagel and W. Ray. Visibility computation for efficient walkthrough of complex environments. Presence: Teleoperators and Virtual Environments, 5(1), 1995.

[ZMHH97] Hansong Zhang, Dinesh Manocha, Thomas Hudson, and Kenneth E. Hoff III. Visibility culling using hierarchical occlusion maps. In Computer Graphics (Proceedings of SIGGRAPH '97), pages 77–88, 1997.