Hierarchical Techniques for Visibility Computations

by

Jiří Bittner

A dissertation submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfillment of the requirements for the degree of Doctor.

October 2002

Thesis Supervisor:
Assoc. Prof. Pavel Slavík, CSc.
Department of Computer Science and Engineering
Faculty of Electrical Engineering
Czech Technical University
Karlovo náměstí 13
121 35 Praha 2
Czech Republic

Abstract

Visibility computation is crucial for computer graphics from its very beginning. The first visibility algorithms aimed to determine visible lines or surfaces in a synthesized image of a 3D scene. Nowadays there is a plethora of visibility algorithms for various applications. The thesis proposes a taxonomy of visibility problems based on the dimension of the problem-relevant line set, i.e. set of lines forming the domain of the visibility problem. The taxonomy helps to identify relationships between visibility problems from various application areas by grouping together problems of similar complexity. The thesis presents a general concept of a visibility algorithm suitable for solution of several classes of visibility problems. The concept is based on three main ideas: an approximate occlusion sweep, an occlusion tree, and hierarchical visibility tests. The central idea of the concept is the occlusion tree representing an aggregated occlusion map by means of a hierarchical subdivision in line space. The thesis presents several applications of the concept. The major focus is placed on visibility culling algorithms for real-time rendering acceleration. First, we describe an algorithm for real-time visibility culling. The algorithm is used to accelerate rendering of large densely occluded scenes. It improves previous results by efficiently performing occluder fusion in real-time. We propose several techniques exploiting temporal and spatial coherence applicable to most existing hierarchical visibility culling methods. Second, we propose an algorithm for computing visibility maps in polygonal scenes. The algorithm provides a comprehensive description of the topology of the given view of the scene. We discuss an application of the algorithm to discontinuity meshing. Third, the proposed concept is applied to computation of from-region visibility in 2D scenes. We present and evaluate an exact analytic algorithm for computing a potentially visible set for a polygonal region in the plane. Fourth, we propose two algorithms for computing from-region visibility in $2\frac{1}{2}$D scenes. Both algorithms are targeted at visibility preprocessing for walkthroughs of outdoor urban scenes. The methods are compared with another recent technique. Finally, we describe an exact analytic algorithm for computing from-region visibility in 3D scenes suitable for various applications. The algorithm uses Plücker coordinates and maintains a hierarchical subdivision of 5D space. We discuss its application to visibility preprocessing, occluder synthesis and discontinuity meshing.

Acknowledgments

First of all, I would like to express my gratitude to my supervisor Pavel Slavík. He has been a constant source of encouragement and always assisted me with problems I have encountered during my Ph.D. studies.

I want to thank Vlastimil Havran, Peter Wonka and Michael Wimmer for many insightful discussions and for being great partners in our common work. I want to thank my colleagues from our lab, namely Roman Berka, Martin Brachtl, Jaroslav Křivánek, Lukáš Mikšíček, Jaroslav Sloup and Jiří Žára for their support and for creating a pleasant environment to work in. I am much in debt to people from the CG lab at the TU Vienna, namely Katja Bühler, Eduard Gröller, Markus Hadwiger, Robert Tobler, Thomas Theußl, Anna Vilanova i Bartrolí, and Alexander Wilkie, for being great hosts during my research visits in Vienna. I also want to express my gratitude to my former colleagues Jan Buriánek, Petr Chlumský, Aleš Holeček, Jan Přikryl and Jan Vorlíček for always willing to share their extensive knowledge.

I wish to thank Yiorgos Chrysanthou and Daniel Cohen-Or for many fruitful discussions that contributed greatly to the quality of the thesis. My gratitude belongs to professors Werner Purgathofer and Walter Pätzold for their amazing support during my research visits at their institutes. I want to thank the head of our department Josef Kolář for taking care of my financial support during the first years of my study. I am much indebted to Václav Hlaváč, the head of the Machine perception group of the Center for Applied Cybernetics, for providing my financial support during the last two years of my studies. My work was supported by the Czech Ministry of Education under Project LN00B096 and a grant No. 1252/1998, a grant from the Internal Grant Agency of the Czech Technical University No. 309810103, and the Aktion Kontakt OE/CZ grant No. 1999/17.

Last but certainly not least I want to thank all my friends and my family for their constant support and encouragement, and for their patience during the period of writing up the thesis.

Contents

Chapter 1

Introduction

Visibility is a phenomenon studied in various research areas. Computation of visibility is important for computer graphics, computational geometry, computer vision, robotics, telecommunications, and other research areas. The thesis focuses on visibility computations in computer graphics with the emphasis placed on real-time rendering.

The first visibility algorithms in computer graphics aimed to determine which lines or surfaces are visible in a synthesized image of a 3D scene. These algorithms solve the problems known as *visible line* or *visible surface* determination. A different formulation of these tasks states the goal as to eliminate hidden lines or surfaces in the image. Therefore these tasks are often called *hidden line* or *hidden surface* removal. The first visibility algorithms developed in late 60's and the beginning of 70's were designed for the use with vector displays. Later with increasing availability of raster devices the traditional techniques were replaced by the z-buffer algorithm [Catm75]. Nowadays, there are two widely spread visibility algorithms: the z-buffer for visible surface determination and ray shooting for computing visibility along a given ray. The z-buffer dominates the area of *real-time rendering* whereas ray shooting is used in the scope of *realistic rendering*. Besides these two well known algorithms there is a plethora of techniques designed for various specific applications. The next section provides a motivation for further research in the area of visibility computations.

1.1 Motivation

Visibility is inherently connected with *rendering* that aims to synthesize images of a 3D scene. Generally, we distinguish between real-time rendering and realistic rendering. In real-time rendering the time devoted to synthesis of an image is severely limited by the desirable image refresh rate. In realistic rendering the ultimate goal is to provide an accurate simulation of the light propagation at the expense of a higher computational cost. Prospectively, real-time rendering and realistic rendering merge as the computational resources enable to implement realistic rendering techniques in real time.

Below we discuss visibility problems occurring in the context of real-time rendering, realistic rendering, and other application areas. The discussion is followed by a summary of challenges in the area of visibility computations.

1.1.1 Real-time rendering

The desirable image refresh rate for real-time applications is at least 20 Hz [Moll02] and preferably it should match the refresh rate of the display device to provide a visually pleasing results. Although the power of computational resources permanently increases so does the user expectation of the quality of the visualization. The user wants to interact with complex scenes rendered in the highest possible quality. Consequently, many today's applications such as architectural walkthroughs, driving simulation, or visual impact analysis, have to cope with large amount of data representing the scene.

Many acceleration techniques were developed to speedup rendering of such large scenes [Moll02]. Among these the *visibility culling* methods aim to efficiently cull the invisible part of the scene so that it does not have to be processed by the rendering pipeline [Cohe02]. Visibility culling provides a great benefit in large densely occluded scenes where only a fraction of scene objects are visible to the observer. This is a typical situation for architectural walkthroughs in both indoor and outdoor environments.

We can distinguish between *offline* visibility culling methods that precompute visibility, and *online* methods that are applied in real-time. Offline techniques typically subdivide the scene into *view cells* and for each view cell they compute a *potentially visible set* (PVS) of scene objects [Aire90, Tell91, Cohe98a, Dura00, Scha00, Wonk00, Kolt01]. During an interactive session the precalculated information is used to render only objects that are potentially visible from the view cell the observer is located in. The final visibility is usually resolved using the z-buffer algorithm. See Figure 1.1 for an example of the offline visibility culling.

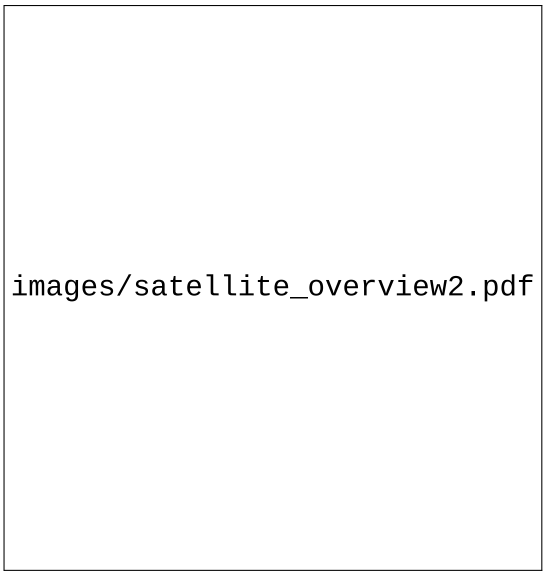


Figure 1.1: Offline visibility culling. The figure depicts 8km^2 of Vienna and a view cell of 0.1km^2 (shown in blue). The red objects form the PVS for the view cell.

Online visibility culling techniques do not preprocess visibility in advance, but they recompute the PVS for each change of the viewpoint [Lueb95, Huds97, Zhan97b, Coor97, Wonk99, Klos01]. Some online techniques compute the PVS for a small neighborhood of the viewpoint to amortize the cost of the PVS computation over several frames [Cohe98c, Wonk01b]. See Figure 1.2 for an illustration of an online visibility culling.

Most online visibility culling algorithms solve the *from-point* visibility problem, i.e. they compute visibility with respect to the given point. On the contrary, most offline visibility culling methods compute the *from-region* visibility accounting for visibility from any point in the given view cell.

1.1.2 Realistic rendering

One of the ultimate goals of computer graphics is to synthesize realistic images of three-dimensional virtual scenes using a physically based simulation of the light propagation [Glas95]. Visibility is a crucial factor in this simulation.

In the simplest case rendering uses only a local illumination model, i.e. a visible surface is shaded according to its material properties and light sources with a global impact. This model does not account

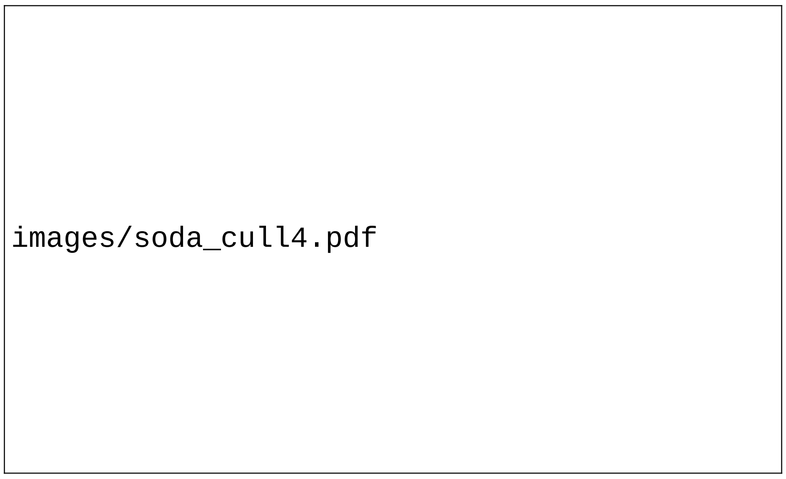


Figure 1.2: Online visibility culling. The figure depicts a top view of an indoor scene. The blue regions were found invisible using an online hierarchical visibility culling.

for shadows and secondary illumination. The only visibility algorithm involved in this model is the final visible surface determination with respect to the viewpoint.

A more advanced simulation accounts for shadows due to primary light sources. A visible surface algorithm can be used to determine shadows with respect to a point light source [Will78, Chin89, Woo90b] (see Figure 1.3).

Figure 1.3: (a) A view of the scene from the point light source. (b) The figure depicts a mesh resulting from subdividing the scene into polygons visible and invisible to the light source. Parts of the scene invisible from the light source are in shadow.

Much more complicated is the illumination due to an areal light source. A shadow due to an areal light source consists of umbra and penumbra [Cook84, Camp90, Chin92, Heck97, Chry97]. An analytic description of illumination in penumbra is complicated and thus it is advantageous to determine loci of discontinuities in the illumination function. This task is called discontinuity meshing [Heck92, Lisc92, Dret94b, Stew94]. Discontinuity meshing aims to subdivide the scene surfaces into patches so that each patch sees a topologically equivalent part of the light source (see Figure 1.4). Discontinuity meshing is used in the scope of the radiosity global illumination algorithm to capture illumination due to primary light sources. Less important light transfers are typically solved by an approximation of mutual visibility [Wall89, Cohe85].

Figure 1.4: A subset of a discontinuity mesh due to a rectangular light source.

View-depend global illumination algorithms such as ray tracing [Whit79], distributed ray tracing [Cook86] or path tracing [Kaji86] use huge amounts of rays to sample light paths. The task of solving visibility along the given ray is called *ray shooting*. Efficient ray shooting is a key factor influencing the total computational time [Arvo89, Havr00a]. Ray shooting can be accelerated by precomputing visibility from light sources [Hain86, Choi92] or other techniques restricting the set of objects potentially hit by the given ray [Simi94, Havr00a]. Global visibility algorithms describing visibility in the whole scene can be used to reduce the ray shooting problem to a point location in the global visibility data structure [Pocc93, Dura96, Dura97, Cho99].

1.1.3 Other applications

There are many other applications where visibility is important [Dura99]. In computer graphics visibility is significant for finding a representative view of an object for presentation purposes. In robotics visibility is important for planning the optimal path for a mobile robot or sweeping the scene using mobile robots. In telecommunications visibility analysis is used in design of cellular networks in mountainous or urban areas. The work developed in the thesis can possibly be applied to some of these problems, but such an application is not discussed.

1.1.4 Challenges in visibility computations

The most important requirements on an ideal visibility algorithm can be stated as follows:

1. It is suitable for scenes with a general structure (possibly dynamic).
2. It is accurate (e.g. accounts for occluder fusion, finds precise shadow boundaries, etc.).
3. It is efficient in terms of the expected running time and memory requirements, particularly:
   (a) it exploits visibility coherence,
   (b) it is output-sensitive.
4. It requires minimal preprocessing of the input data.
5. It is simple to implement.

The current visibility algorithms fail to provide at least one of the above stated goals. The priority of the goals differs depending on the context of the application, e.g. the type of the typically processed scene, the available hardware, the time available for the implementation of the algorithm, etc. Consequently, most visibility algorithms address rather narrow application area (e.g. computing form-factors, computing hard/soft shadows, etc.), often using various ad hoc design decisions. A challenging topic is thus providing a solution that is potentially useful for a large class of applications while satisfying most of the requirements stated above.

The thesis aims to address most of the goals in the context of several application areas. We present a general concept of a visibility algorithm that serves as a basis of all algorithms proposed in the thesis. The most complex application of the concept is an algorithm solving the from-region visibility in general 3D scenes. This method provides a comprehensive description of from-region visibility and it is applicable in the context of visibility culling as well as rendering shadows, discontinuity meshing, or visibility preprocessing for ray tracing acceleration.

1.2 Contributions of the thesis

The thesis aims to provide the following contributions:

1. A *new classification* of visibility problems based on the dimension of the *problem-relevant line set* and the *type of the answer* to the given problem (published in [Bitt02b]).

2. A *general concept* of a visibility algorithm suitable for several classes of visibility problems. The concept aims at exploiting coherence of visibility and achieving output-sensitivity of the algorithm.

3. Several visibility algorithms were designed as an application of the proposed concept. These algorithms address the following problems:

   (a) *Real-time visibility culling.*

   We present an algorithm for real-time visibility culling designed for walkthroughs of large scenes (published in [Bitt98]). The algorithm extends previous results achieved by continuous visibility culling methods [Huds97, Coor97] by efficiently accounting for occluder fusion in real-time. Further, we discuss a general technique making use of temporal coherence that can be applied to accelerate most existing visibility culling methods (published in [Bitt01b, Bitt01a]).

   (b) *Computing visibility maps.*

   We describe a new output-sensitive algorithm for computing visibility maps and discuss its application to discontinuity meshing (published in [Bitt02a]). The algorithm illustrates the applicability of the proposed concept for solving a from-point visibility problem requiring a comprehensive description of visibility from the given point.

   (c) *Computing from-region visibility in 2D and $2\frac{1}{2}$D scenes.*

   We present new algorithms for computing from-region visibility in 2D and $2\frac{1}{2}$D scenes (published in [Bitt01c, Bitt01e, Bitt01d]). The proposed algorithms include an exact analytic solution of the $2\frac{1}{2}$D from-region visibility problem that is suitable for large scenes. The algorithms extend previous results achieved in the context of the from-region visibility in $2\frac{1}{2}$D scenes [Wonk00, Kolt00, Kolt01].

   (d) *Computing from-region visibility in 3D scenes.*

   We propose a new analytic solution to the from-region visibility in 3D scenes suitable for various applications. We discuss the application of the method to computing potentially visible sets, occluder synthesis, and discontinuity meshing.

1.3 Structure of the thesis

Chapter 2 presents an overview of the visibility problems and algorithms. It contains a taxonomy of visibility problems based on the problem domain and the type of the desired answer. Further, it classifies visibility algorithms according to several important criteria and discusses the important concepts in the design of a visibility algorithm.

Chapter 3 presents the concept of a visibility algorithm suitable for solution of various classes of visibility problems. In the following chapters we present algorithms based on the concept introduced herein. These chapters also summarize and discuss related work in the particular application area.

Chapter 4 presents an application of the proposed concept for real-time visibility culling. The chapter contains the description of the proposed algorithm and evaluation of its implementation on several test scenes. The second part of this chapter presents a general techniques exploiting temporal and spatial coherence in the scope of hierarchical visibility algorithms. Chapter 5 presents a new algorithm for computing a visibility map of a given view of the scene. The algorithm is evaluated on several scenes of moderate complexity. Chapter 6 presents an algorithm for computing from-region visibility in 2D scenes. The description of the method is followed by its evaluation on several test scenes. Chapter 7 presents an extension of the algorithm from Chapter 6 that deals with $2\frac{1}{2}$D scenes. We describe and evaluate a conservative and an exact algorithms for from-region visibility in $2\frac{1}{2}$D. Chapter 8 presents an algorithm dealing with from-region visibility in 3D scenes. The algorithm provides an analytic solution to the problem suitable for complex 3D scenes of general structure. We discuss an application of the method to visibility preprocessing, occluder synthesis, and discontinuity meshing. Chapter 9 draws conclusions and presents suggestions for further research.

The appendix consists of three chapters presenting an overview of the important visibility algorithms. These chapters provide an extended context for the thesis with the aim to make the thesis self-contained. Appendix A discusses the traditional visible surface algorithms. Appendix B discusses visibility algorithms in real-time rendering. Appendix C discusses visibility algorithms exploited in the context of realistic rendering.

Chapter 2

Overview of visibility problems and algorithms

This chapter provides a taxonomy of visibility problems encountered in computer graphics based on the *problem domain* and the *type of the answer*. The taxonomy helps to understand the nature of a particular visibility problem and provides a tool for grouping problems of similar complexity independently of their target application. We discuss typical visibility problems encountered in computer graphics and identify their relation to the proposed taxonomy. A visibility problem can be solved by means of various visibility algorithms. We classify visibility algorithms according to several important criteria and discuss important concepts in the design of a visibility algorithm. Finally, we discuss specific issues of visibility in urban scenes that are exploited in Chapters 4, 6, and 7 of the thesis. The taxonomy and the discussion of the algorithm design sums up ideas and concepts that are independent of any specific algorithm. This can help algorithm designers to transfer the existing algorithms to solve visibility problems in other application areas.

For a more detailed discussion of the state of the art visibility algorithms we refer an interested reader to Appendices A, B, and C.

2.1 Taxonomy of visibility problems

We propose a taxonomy of visibility problems that is based on the *problem domain* and the *type of the answer*. The specification of the problem domain and the scene description provide an input to an algorithm solving the given visibility problem. The type of the output of the algorithm is given by the type of the answer for the given problem. In this taxonomy we do not classify visibility problems according to the type of the scene description. This criterion will be used later for classification of visibility algorithms (Section 2.3.1).

2.1.1 Problem domain

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.

2.1.2 Type of the answer

Visibility problems can further be distinguished according to the type of the desired answer to the given problem. The taxonomy treats the desired answer as an integral part of the problem formulation. This enables a more precise specification of the given class of problems and allows better understanding of its complexity. The type of the answer restricts the domain of the output of an algorithm solving the problem. We identify three classes of answers to visibility problems. Each class consists of several types of answers:

1. *visibility classification*
   (a) visible/invisible
   (b) visible/invisible/partially visible
2. *subset of the input*
   (a) a single scene object
   (b) set of scene objects (e.g. triangles)
   (c) set of group objects (e.g. nodes of a bounding volume hierarchy)
   (d) set of regions (e.g. nodes of a spatial subdivision)
   (e) set of rays

3. *a constructed data structure*
   (a) set of points
   (b) set of rays
   (c) set of curves or line segments (e.g. visibility discontinuities on surfaces)
   (d) set of surfaces (e.g. visible/invisible patches)
   (e) a volume (e.g. antipenumbra/umbra/penumbra)
   (f) set of volumes (e.g. cells of the aspect graph)

The first two classes of answers induce a finite domain of the possible result. The third class consists of answers for which the solution is constructed from generally infinite number of choices in solution space. Consequently, problems with answers from the first two classes (visibility classification, subset of the input) are often simpler to solve than those with an answer from the third class (a constructed data structure). Furthermore, algorithms solving such a decision-making problems are often more robust with respect to numerical stability.

2.2 Dimension of the problem-relevant line set

The six domains of visibility problems stated in Section 2.1.1 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].

2.2.1 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 2.1 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 $\mathcal{P}^2$ [Stol91]. 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 6.

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 2.2.

2.2.2 Parametrization of lines in 3D

Lines in 3D form a four-parametric space [Pell97]. 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 [Gu97]. This parametrization exhibits a singularity since it cannot describe lines parallel to these planes. See Figure 2.3 for illustrations of the spherical and the two plane parameterizations.

[1]A classical mathematical definition says: Line space is a direct product of two Hilbert spaces [Weis99]. However, this definition differs from the common understanding of line space in computer graphics [Dura99]

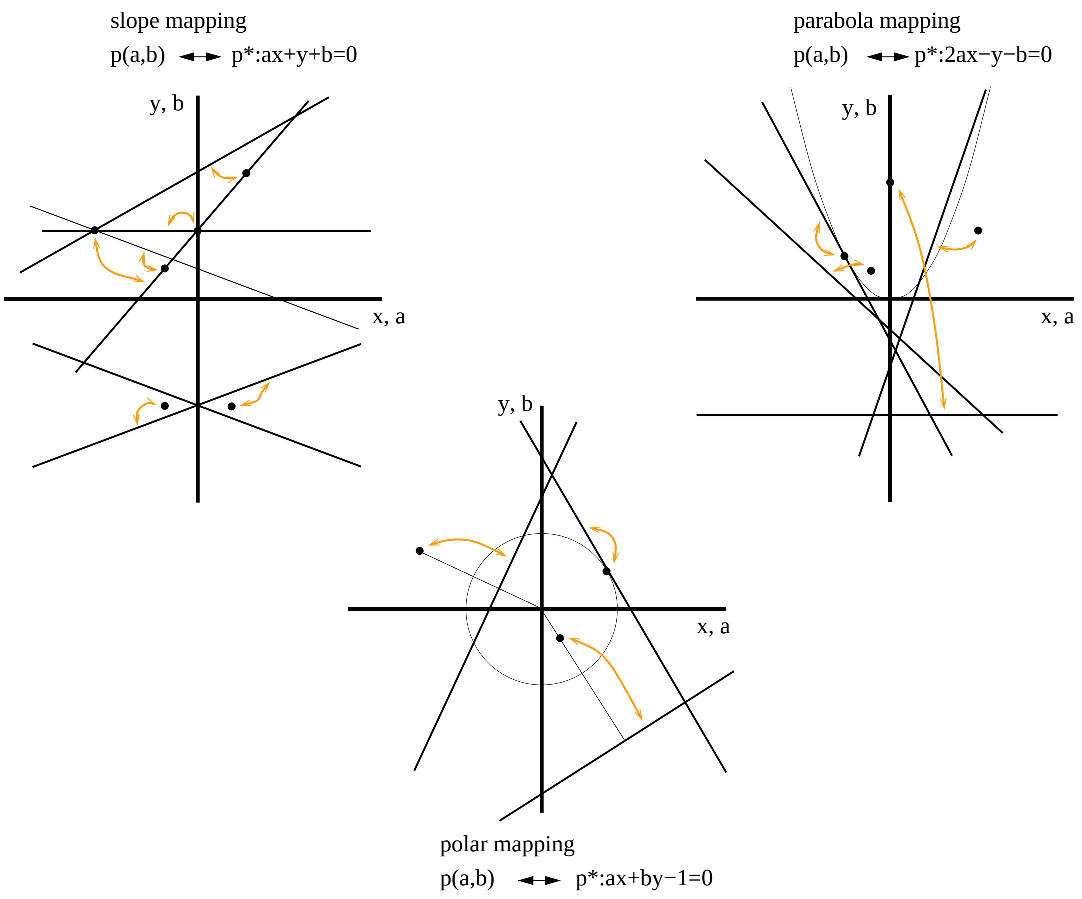


Figure 2.1: Duality between points and lines in 2D.

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 [Stol91]. 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* [Stol91, Pu98].

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 Chapters 7 and 8 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 $0 \leq k \leq 4$.

2.2.3 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*.

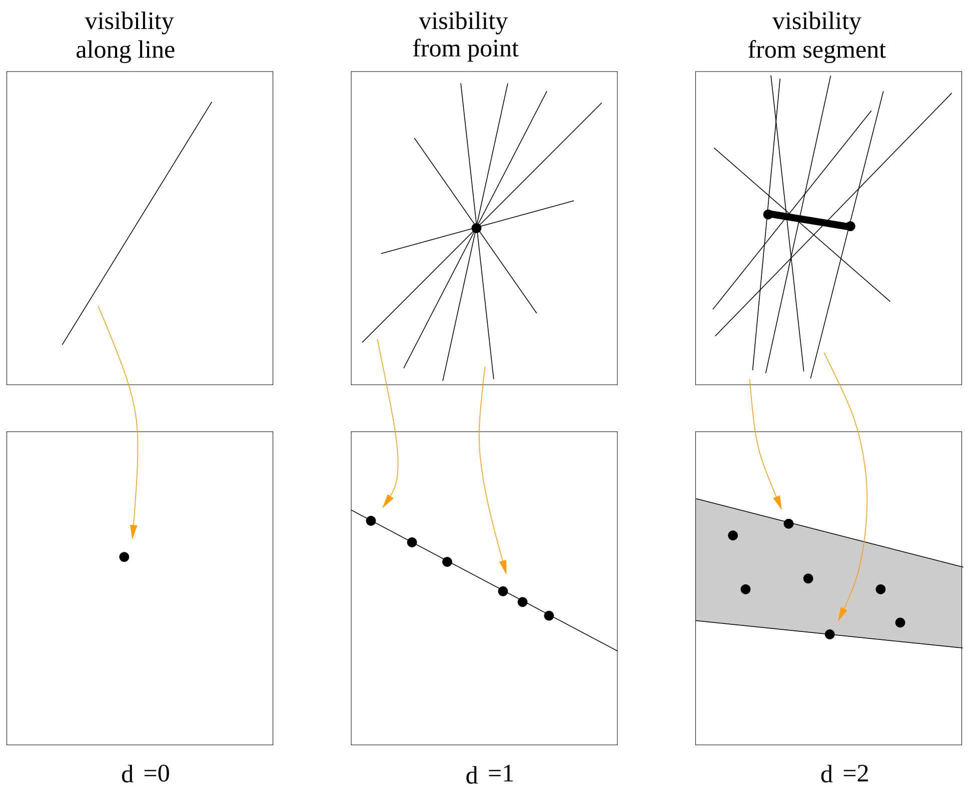


Figure 2.2: The problem-relevant set of lines in 2D. The $\mathcal{L}_R$ for visibility along a line is formed by a single point that is a mapping of the given line. The $\mathcal{L}_R$ for visibility from a point p is formed by points lying on a line. This line is a dual mapping of the point p. $\mathcal{L}_R$ for visibility from a line segment is formed by a 2D region bounded by dual mappings of endpoints of the given segment.

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 2.4 for an illustration of typical visibility along a line problems.

2.2.4 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 2.5).

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.

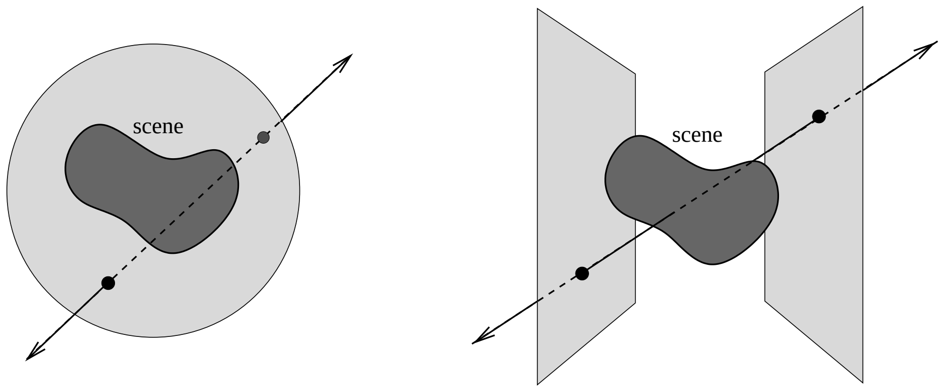


Figure 2.3: Parametrization of lines in 3D. (left) A line can be described by two points on a sphere enclosing the scene. (right) The two plane parametrization describes a line by point of intersection with two given planes.

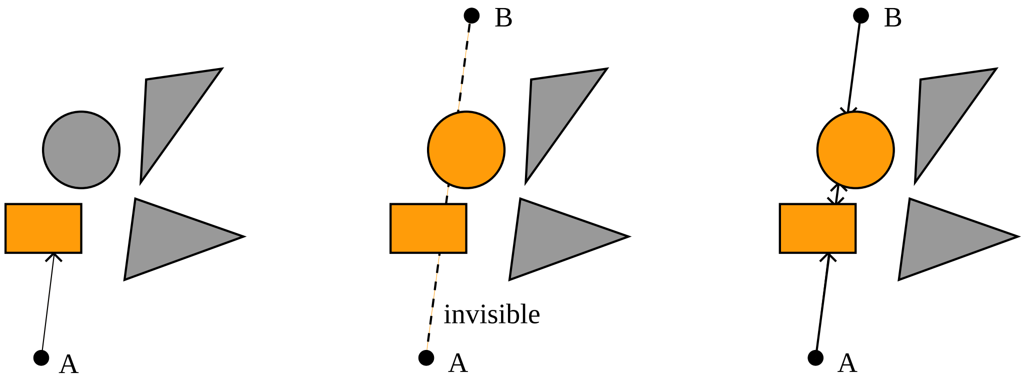


Figure 2.4: Visibility along a line. (left) Ray shooting. (center) Point-to-point visibility. (right) Maximal free line segments between two points.

2.2.5 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 2.6).

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.

2.2.6 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 proposed 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 2.7). 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.

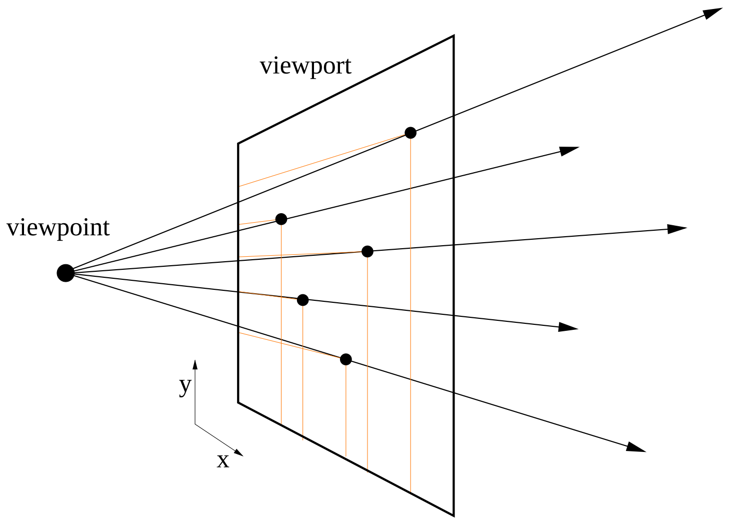


Figure 2.5: Visibility from a point. Lines intersecting a point can be described by a point of intersection with the given viewport.

2.2.7 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.

2.2.8 Summary

The classification of visibility problems according to the dimension of the problem-relevant line set is summarized in Table 2.1. 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 visibility from a region involves non-linear subdivisions of both primal space and line space even for polygonal scenes [Tell92a, Dura99].

According to the classification we can make the following observations:

- Visibility from a region in 2D and visibility from a point in 3D involve a two-dimensional $\mathcal{L}_R$. This suggests that a mapping between these two problems is possible [Bitt01c].
- Solutions to many visibility problems in 2D do not extend easily to 3D since they involve $\mathcal{L}_R$ of different dimensions. This observation conforms with [Dura99].

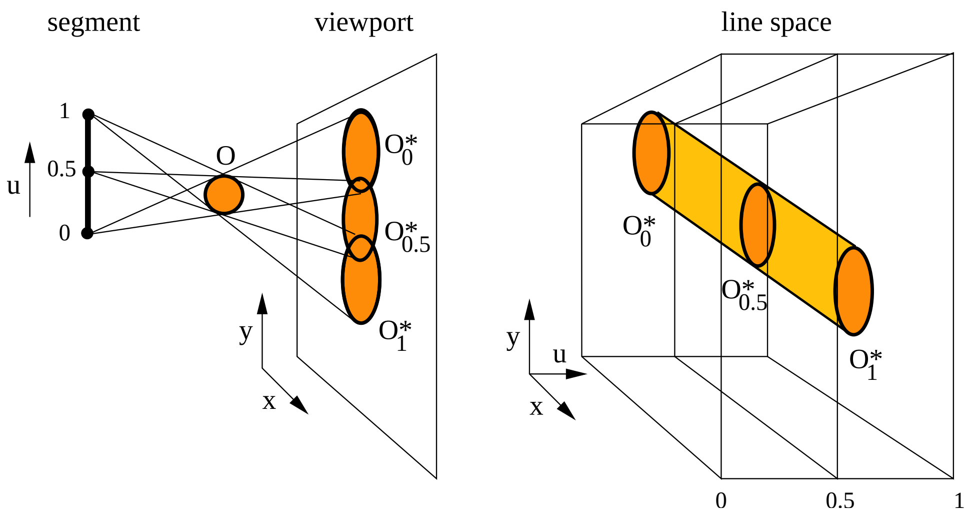


Figure 2.6: Visibility from a line segment. (left) Line segment, a spherical object O, and its projections O_0^*, $O_{0.5}^*$, O_1^* with respect to the three points on the line segment. (right) A possible parametrization of lines that stacks up 2D planes. Each plane corresponds to mappings of lines intersecting a given point on the line segment.

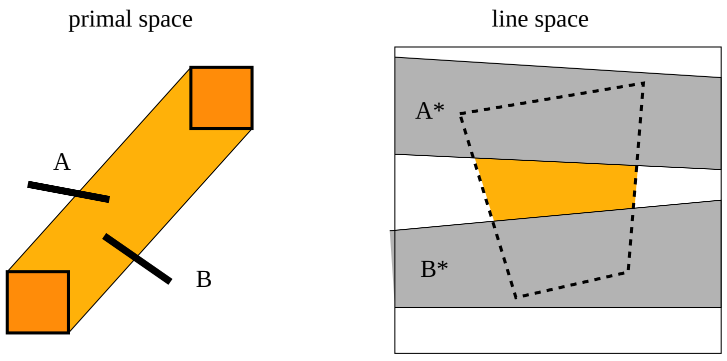


Figure 2.7: Visibility from a region — an example of the region-to-region visibility. Two regions and two occluders A, B in a 2D scene. In line space the region-to-region visibility can be solved by subtracting the sets of lines A^* and B^* intersecting objects A and B from the lines intersecting both regions.

2.3 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.

2.3.1 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

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

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.

2.3.2 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.

2.3.3 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 (these algorithms will be discussed in Section A). The z-buffer [Catm75] is a common example of a discrete algorithm. The Weiler-Atherton algorithm [Weil77] 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.

2.4 Visibility algorithm design

Visibility algorithm design can be decoupled into a series of important design decisions. The first step is to clearly formulate a problem that should be solved by the algorithm. The taxonomy stated above can help to understand the difficulty of solving the given problem and its relationship to other visibility problems in computer graphics. The following sections summarize important steps in the design of a visibility algorithm and discuss some commonly used techniques.

2.4.1 Scene structuring

We discuss two issues dealing with structuring of the scene: identifying occluders and occludees, and spatial data structures for scene description.

Occluders and occludees

Many visibility algorithms restructure the scene description to distinguish between *occluders* and *occludees*. Occluders are objects that cause changes in visibility (occlusion). Occludees are objects that do not cause occlusion, but are sensitive to visibility changes. In other words the algorithm studies visibility of occludees with respect to occluders.

The concept of occluders and occludees is used to increase the performance of the algorithm in both the running time and the accuracy of the algorithm by reducing the number of primitives used for visibility computations (the performance measures of visibility algorithms will be discussed in Section 2.4.3). Typically, the number of occluders and occludees is significantly smaller than the total number of objects in the scene. Additionally, both the occluders and the occludees can be accompanied with a topological (connectivity) information that might be necessary for an efficient functionality of the algorithm.

The concept of occluders is applicable to most visibility algorithms. The concept of occludees is useful for algorithms providing answers (1) and (2) according to the taxonomy of Section 2.1.2. Some visibility algorithms do not distinguish between occluders and occludees at all. For example all traditional visible surface algorithms use all scene objects as both occluders and occludees.

Both the occluders and the occludees can be represented by *virtual objects* constructed from the scene primitives: the occluders as simplified inscribed objects, occludees as simplified circumscribed objects such as bounding boxes. Algorithms can be classified according to the type of occluders they deal with. The classification follows the scene restrictions discussed in Section 2.3.1 and adds classes specific to occluder restrictions:

- vertical prisms,
- axis-aligned polygons,
- axis-aligned rectangles.

The vertical prisms that are specifically important for computing visibility in $2\frac{1}{2}$D scenes. Some visibility algorithms can deal only with axis-aligned polygons or even more restrictive axis-aligned rectangles.

Other important criteria for evaluating algorithms according to occluder restrictions include:

- connectivity information,
- intersecting occluders.

Figure 2.8: Occluders in an urban scene. In urban scenes the occluders can be considered vertical prisms erected above the ground.

The explicit knowledge of the connectivity is crucial for efficient performance of some visibility algorithms (performance measures will be discussed in the Section 2.4.3). Intersecting occluders cannot be handled properly by some visibility algorithms. In such a case the possible occluder intersections should be resolved in preprocessing.

A similar classification can be applied to occludees. However, the visibility algorithms typically pose less restrictions on occludees since they are not used to describe visibility but only to check visibility with respect to the description provided by the occluders.

Scene description

The scene is typically represented by a collection of objects. For purposes of visibility computations it can be advantageous to transform the object centered representation to a spatial representation by means of a spatial data structure. For example the scene can be represented by an octree where full voxels correspond to opaque parts of the scene. Such a data structure is then used as an input to the visibility algorithm. The spatial data structures for the scene description are used for the following reasons:

- *Regularity*. A spatial data structure typically provides a regular description of the scene that avoids complicated configurations or overly detailed input. Furthermore, the representation can be rather independent of the total scene complexity.

- *Efficiency*. The algorithm can be more efficient in both the running time and the accuracy of the result.

Additionally, spatial data structures can be applied to structure the solution space and/or to represent the desired solution. Another application of spatial data structures is the acceleration of the algorithm by providing spatial indexing. These applications of spatial data structures will be discussed in Sections 2.4.2 and 2.4.3. Note that a visibility algorithm can use a single data structure for all three purposes (scene structuring, solution space structuring, and spatial indexing) while another visibility algorithm can use three conceptually different data structures.

2.4.2 Solution space data structures

A solution space data structure is used to maintain an intermediate result during the operation of the algorithm and it is used to generate the result of the algorithm. We distinguish between the following solution space data structures:

- general data structures

  single value (ray shooting), winged edge, active edge table, etc.

- primal space (spatial) data structures

  uniform grid, BSP tree (shadow volumes), bounding volume hierarchy, kD-tree, octree, etc.

- image space data structures

  2D uniform grid (shadow map), 2D BSP tree, quadtree, kD-tree, etc.

- line space data structures

  regular grid, kD-tree, BSP tree, etc.

The image space data structures can be considered a special case of the line space data structures since a point in the image represents a ray through that point (see also Section 2.3.3).

If the dimension of the solution space matches the dimension of the problem-relevant line set, the visibility problem can often be solved with high accuracy by a single sweep through the scene. If the dimensions do not match, the algorithm typically needs more passes to compute a result with satisfying accuracy [Dura00, Wonk00] or neglects some visibility effects altogether [Scha00].

2.4.3 Performance

The performance of a visibility algorithm can be evaluated by measuring the quality of the result, the running time and the memory consumption. In this section we discuss several concepts related to these performance criteria.

Quality of the result

One of the important performance measures of a visibility algorithm is the quality of the result. The quality measure depends on the type of the answer to the problem. Generally, the quality of the result can be expressed as a distance from an exact result in the solution space. Such a quality measure can be seen as a more precise expression of the accuracy of the algorithm discussed in Section 2.3.2.

For example a quality measure of algorithms computing a PVS 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 $\boldsymbol{Q}^A(I)$:

$$\boldsymbol{Q}^A(I) = (Q_o^A(I), Q_u^A(I)), \qquad I \in \mathcal{D} \tag{2.1}$$

$$Q_o^A(I) = \frac{|S^A(I) \setminus S^{\mathcal{E}}(I)|}{|S^{\mathcal{E}}(I)|} \tag{2.2}$$

$$Q_u^A(I) = \frac{|S^{\mathcal{E}}(I) \setminus S^A(I)|}{|S^{\mathcal{E}}(I)|} \tag{2.3}$$

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:

$$\begin{aligned} Q^A &= E[\|\boldsymbol{Q}^A(I)\|] && (2.4) \\ &= \sum_{\forall I \in \mathcal{D}} f(I).\sqrt{Q_o^A(I)^2 + Q_o^A(I)^2} && (2.5) \end{aligned}$$

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 2.3.2:

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
   $\exists I_j, I_k \in \mathcal{D} : Q_o^A(I_j) > 0 \wedge Q_u^A(I_k) > 0$

Scalability

Scalability expresses the ability of the visibility algorithm to cope with larger inputs. A more precise definition of scalability of an algorithm depends on the problem for which the algorithm is designed. The scalability of an algorithm can be studied with respect to the size of the scene (e.g. number of scene objects). Another measure might consider the dependence of the algorithm on the number of only the visible objects. Scalability can also be studied according to the given domain restrictions, e.g. volume of the view cell.

A well designed visibility algorithm should be scalable with respect to the number of structural changes or discontinuities of visibility. Furthermore, its performance should be given by the complexity of the visible part of the scene. These two important measures of scalability of an algorithm are discussed in the next two sections.

Use of coherence

Scenes in computer graphics typically consist of objects whose properties vary smoothly from one part to another. A view of such a scene contains regions of smooth changes (changes in color, depth, texture,etc.) and discontinuities that let us distinguish between objects. The degree to which the scene or its projection exhibit local similarities is called *coherence* [Fole90].

Coherence can be exploited by reusing calculations made for one part of the scene for nearby parts. Algorithms exploiting coherence are typically more efficient than algorithms computing the result from the scratch.

Sutherland et al. [Suth74] identified several different types of coherence in the context of visible surface algorithms. We simplify the classification proposed by Sutherland et al. to reflect general visibility problems and distinguish between the following three types of *visibility coherence*:

- *Spatial coherence*. Visibility of points in space tends to be coherent in the sense that the visible part of the scene consists of compact sets (regions) of visible and invisible points. We can reuse calculations made for a given region for the neighboring regions or its subregions.

- *Line-space coherence*. Sets of similar rays tend to have the same visibility classification, i.e. the rays intersect the same object. We can reuse calculations for the given set of rays for its subsets or the sets of nearby rays.
- *Temporal coherence*. Visibility at two successive moments is likely to be similar despite small changes in the scene or a region/point of interest. Calculations made for one frame can be reused for the next frame in a sequence.

The degree to which the algorithm exploits various types of coherence is one of the major design paradigms in research of new visibility algorithms. The importance of exploiting coherence is emphasized by the large amount of data that need to be processed by the current rendering algorithms.

Output sensitivity

An algorithm is said to be *output-sensitive* if its running time is sensitive to the size of output. In the computer graphics community the term output-sensitive algorithm is used in a broader meaning than in computational geometry [Berg97]. The attention is paid to a practical usage of the algorithm, i.e. to an efficient implementation in terms of the practical average case performance. The algorithms are usually evaluated experimentally using test data and measuring the running time and the size of output of the algorithm. The formal average case analysis is usually not carried out for the following two reasons:

1. *The algorithm is too obscured*. Visibility algorithms exploit data structures that are built according to various heuristics and it is difficult to derive proper bounds even on the expected size of these supporting data structures.
2. *It is difficult to properly model the input data*. In general it is difficult to create a reasonable model that captures properties of real world scenes as well as the probability of occurrence of a particular configuration.

A visibility algorithm can often be divided into the *offline* phase and the *online* phase. The offline phase is also called preprocessing. The preprocessing is often amortized over many executions of the algorithm and therefore it is advantageous to express it separately from the online running time.

For example an ideal output-sensitive visible surface algorithm runs in $O(n \log n + k^2)$, where n is the number of scene polygons (size of input) and k is the number of visible polygons (in the worst case k visible polygons induce $O(k^2)$ visible polygon fragments).

Acceleration data structures

Acceleration data structures are often used to achieve the performance goals of a visibility algorithm. These data structures allow efficient point location, proximity queries, or scene traversal required by many visibility algorithms.

With a few exceptions the acceleration data structures provide a *spatial index* for the scene by means of a spatial data structure. The spatial data structures group scene objects according to the spatial proximity. On the contrary line space data structures group rays according to their proximity in line space.

The common acceleration data structures can be divided into the following categories:

- Spatial data structures
  - *Spatial subdivisions*
    uniform grid, hierarchical grid, kD-tree, BSP tree, octree, quadtree, etc.
  - *Bounding volume hierarchies*
    hierarchy of bounding spheres, hierarchy of bounding boxes, etc.

  - *Hybrid*
    hierarchy of uniform grids, hierarchy of kD-trees, etc.

- Line space data structures
  - *General*
    regular grid, kD-tree, BSP tree, etc.

Use of graphics hardware

Visibility algorithms can be accelerated by exploiting dedicated graphics hardware. The hardware implementation of the z-buffer algorithm that is common even on a low-end graphics hardware can be used to accelerate solutions to other visibility problems. Recall that the z-buffer algorithm solves the visibility from a point problem by providing a discrete approximation of the visible surfaces.

A visibility algorithm can be accelerated by the graphics hardware if it can be decomposed so that the decomposition includes the problem solved by the z-buffer or a series of such problems. Prospectively, the recent features of the graphics hardware, such as the pixel and vertex shaders allow easier application of the graphics hardware for solving specific visibility tasks. The software interface between the graphics hardware and the CPU is usually provided by OpenGL [Moll02].

2.5 Visibility in urban environments

Urban environments constitute an important class of real world scenes computer graphics deals with. The urban environments are the major focus in the design of the algorithms presented in the thesis.

We can identify two fundamental subclasses of urban scenes. Firstly, we consider *outdoor* scenes, i.e. urban scenes as observed from streets, parks, rivers, or a bird's-eye view. Secondly, we consider *indoor* scenes, i.e. urban scenes representing building interiors. In the following two sections we discuss the essential characteristics of visibility in both the outdoor and the indoor scenes. The discussion is followed by summarizing the suitable visibility techniques.

2.5.1 Analysis of visibility in outdoor urban areas

Outdoor urban scenes are viewed using two different scenarios. In a *flyover* scenario the scene is observed from the bird's eye view. A large part of the scene is visible. Visibility is mainly restricted due to the structure of the terrain, atmospheric constraints (fog, clouds) and the finite resolution of human retina. Rendering of the flyover scenarios is usually accelerated using LOD, image-based rendering and terrain visibility algorithms, but there is no significant potential for visibility culling.

In a *walkthrough* scenario the scene is observed from a pedestrians point of view and the visibility is often very restricted. In the remainder of this section we discuss the walkthrough scenario in more detail.

Due to technological and physical restrictions urban scenes viewed from outdoor closely resemble a 2D *height function*, i.e. a function expressing the height of the scene elements above the ground. The height function cannot capture certain objects such as bridges, passages, subways, or detailed objects such as trees. Nevertheless buildings, usually the most important part of the scene, can be captured accurately by the height function in most cases. For the sake of visibility computations the objects that cannot be represented by the height function can be ignored. The resulting scene is then called a *$2\frac{1}{2}D$ scene*.

In a dense urban area with high buildings visibility is very restricted when the scene is viewed from a street (see Figure 2.9-a). Only buildings from nearby streets are visible. Often there are no buildings visible above roofs of buildings close to the viewpoint. In such a case visibility is essentially two-dimensional, i.e. it could be solved accurately using a 2D footprint of the scene and a 2D visibility algorithm. In areas with smaller houses of different shapes visibility is not so severely restricted since some objects can be visible by looking over other objects. The view complexity increases (measured in number of visible objects) and the height structure becomes increasingly important. Complex views with far visibility can be seen also near rivers, squares, and parks (see Figure 2.9-b).

images/foto_street1.pdf

images/foto_castle1.pdf

Figure 2.9: Visibility in outdoor urban areas. (left) In the center of a city visibility is typically restricted to a few nearby streets. (right) Near river banks typically a large part of the city is visible. Note that many distant objects are visible due to the terrain gradation.

In scenes with large differences in terrain height the view complexity is often very high. Many objects can be visible that are situated for example on a hill or on a slope behind a river. Especially in areas with smaller housing visibility is much defined by the terrain itself.

We can summarize the observations as follows (based on Wonka [Wonk01a]) :

- Outdoor urban environments have basically $2\frac{1}{2}$D structure and consequently visibility is restricted accordingly.
- The view is very restricted in certain areas, such as in the city center. However the complexity of the view can vary significantly. It is always not the case that only few objects are visible.
- If there are large height differences in the terrain, many objects are visible for most viewpoints.
- In the same view a close object can be visible next to a very distant one.

In the simplest case the outdoor scene consists only of the terrain populated by a few buildings. Then the visibility can be calculated on the terrain itself with satisfying accuracy [Flor95, Cohe95, Stew97]. Outdoor urban environments have a similar structure as terrains: buildings can be treated as a terrain with *many discontinuities* in the height function (assuming that the buildings do not contain holes or significant variations in their façades). To accurately capture visibility in such an environment specialized algorithms have been developed that compute visibility from a given viewpoint [Down01] or view cell [Wonk00, Kolt01, Bitt01e].

The methods presented later in the thesis make use of the specific structure of the outdoor scenes to efficiently compute a PVS for the given view cell. The key observation is that the PVS for a view cell in a $2\frac{1}{2}$D can be determined by computing visibility from its top boundary edges. This problem becomes a restricted variant of the visibility from a line segment in 3D with $d(\mathcal{L}_R) = 3$.

2.5.2 Analysis of indoor visibility

Building interiors constitute another important class of real world scenes. A typical building consists of rooms, halls, corridors, and stairways. It is possible to see from one room to another through an open door or window. Similarly it is possible to see from one corridor to another one through a door or other connecting structure. In general the scene can be subdivided into cells corresponding to the rooms, halls, corridors, etc., and transparent portals that connect the cells [Aire90, Tell91]. Some portals correspond to the real doors and windows, others provide only a virtual connection between cells. For example an L-shaped corridor can be represented by two cells and one virtual portal connecting them.

Visibility in a building interior is often significantly restricted (see Figure 2.10). We can see the room we are located at and possibly few other rooms visible through open doors. Due to the natural partition of the scene into cells and portals visibility can be solved by determining which cells can be seen through a give set of portals and their sequences. A sequence of portals that we can see through is called *feasible*.

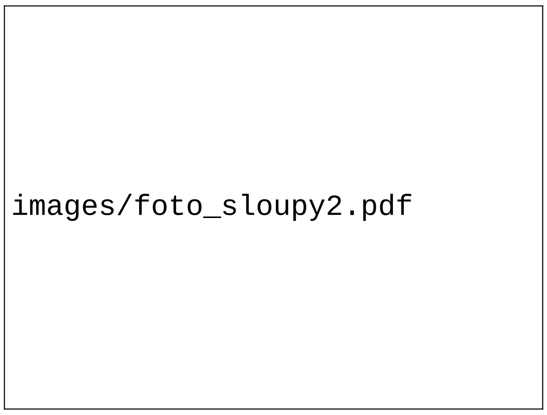


Figure 2.10: Indoor visibility. (left) Visibility in indoor scenes is typically restricted to a few rooms or corridors. (right) In scenes with more complex interior structure visibility gets more complicated.

Many algorithms for computing indoor visibility [Aire90, Tell92b, Lueb95] exploit the cell/portal structure of the scene. The potential problem of this approach is its strong sensitivity to the arrangement of the environment. In a scene with a complicated structure with many portals there are many feasible portal sequences. Imagine a hall with columns arranged on a grid. The number of feasible portal sequences rapidly increases with the distance from the given view cell [Tell92b] if the columns are sufficiently small (see Figure 2.11). Paradoxically most of the scene is visible and there is almost no benefit of using any visibility culling algorithm.

The approach presented later in the thesis partially avoids this problem since it does not rely on finding feasible portal sequences even in the indoor scenes. Instead of determining what *can* be visible through a transparent complement of the scene (portals) the method determines what *cannot* be visible due to the scene objects themselves (occluders). This approach also avoids the explicit enumeration of portals and the construction of the cell/portal graph.

2.6 Summary

Visibility problems and algorithms penetrate a large part of computer graphics research. The proposed taxonomy aims to classify 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 thesis addresses the following classes of visibility problems:

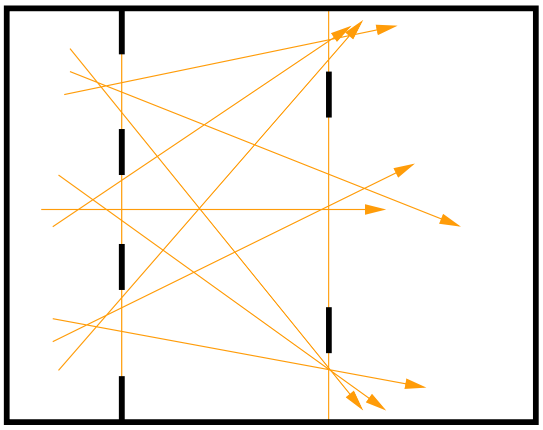


Figure 2.11: In sparsely occluded scenes the cell/portal algorithm can exhibit a combinatorial explosion in number of feasible portal sequences. Paradoxically visibility culling provides almost no benefit in such scenes.

- Visibility from a point in 3D and visibility from region in 2D, $d(\mathcal{L}_R) = 2$.
- Visibility from a region in $2\frac{1}{2}$D, $d(\mathcal{L}_R) = 3$.
- 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 thesis addresses algorithms with the following properties:

- Domain:
  - viewpoint (Chapters 4, 5),
  - line segment or 2D polygon (Chapter 6),
  - vertical trapezoids/prisms (Chapter 7),
  - polygon or polyhedron (Chapter 8)
- Scene restrictions (occluders):
  - convex polygons (Chapters 4, 5 and 8),
  - line segments (Chapter 6),
  - vertical trapezoids (Chapter 7)
- Scene restrictions (group objects):
  - bounding boxes (Chapters 4, 5, Chapter 7 and 8),
  - bounding rectangles (Chapter 6).
- Output:
  - PVS, answers 2-(b), 2-(c) (Chapters 4, 6, 7 and 8)
  - Visible surfaces, answer 3-(d) (Chapters 5, 8)
  - Visibility discontinuities, answer 3-(c) (Chapters 5, 8)

- Accuracy:
  - conservative (Chapters 4, 7)
  - exact (Chapters 4, 5, 6, 7 and 8)
- Solution space:
  - continuous, line space (Chapters 4, 5, 6 and 8)
  - continuous, line space / primal space (Chapter 7)
- Solution space data structure: BSP tree (all methods)
- Use of coherence of visibility:
  - spatial coherence (all methods)
  - line space coherence (all methods)
  - temporal coherence (Chapter 4)
- Output sensitivity: expected in practice (all methods)
- Acceleration data structure: kD-tree (all methods)
- Use of graphics hardware: no

The algorithms described in the rest of the thesis are mainly focused on urban scenes. This chapter also discussed specific issues of visibility in indoor and outdoor urban scenes and outlined appropriate visibility techniques.

Chapter 3

The general concept of a visibility algorithm

This chapter presents a general concept suitable for solution of several visibility problems. Applications of the concept are then presented in Chapters 4, 5, 6, and 8, where it is used for the real-time occlusion culling, computing visibility maps and computing from-region visibility in 2D, $2\frac{1}{2}$D, and 3D scenes. These chapters then thoroughly discuss the details of the method and optimizations for the particular application.

The concept is based on three main ideas: the *approximate occlusion sweep*, the *occlusion tree*, and *hierarchical visibility tests*. The *approximate occlusion sweep* is used to construct an *aggregated occlusion map* (AOM) due to already processed occluders. The AOM is maintained by the *occlusion tree* that represents a union of occluded rays by means of a hierarchical subdivision of the problem-relevant line set. The occlusion tree serves as an abstraction layer allowing to apply the same concept for various visibility problems: the tree can be used to represent visibility with respect to a point, line segment or a region. Additionally, the hierarchical structure of the occlusion tree allows efficient visibility tests making use of visibility coherence. The *hierarchical visibility tests* use the occlusion tree to classify visibility of nodes of the spatial hierarchy. These tests are interleaved with the occlusion sweep to achieve an output-sensitive behavior of the algorithm.

3.1 Related work

The proposed concept extends the ideas of several from-point visibility methods for solving other visibility problems. In particular it builds on the beam tracing [Heck84], the cone tracing [Aman84], the frustum casting [Tell98] and the projection of Binary Space Partitioning (BSP) trees [Nayl92b]. These methods share a common idea of sweeping the scene in the direction defined by a certain set of rays (beam, cone, frustum or the whole viewport). The scene is swept in a front to back order, which leads to an output-sensitive behavior of the algorithm: once an unambiguous solution is found (e.g. the whole beam intersects an object) the algorithm is terminated. Thus the invisible part of the scene need not be processed by the algorithm. Another common principle is the use of visibility coherence. Whole sets of rays are cast at a time. In the case that the rays intersect the same object the algorithm terminates. This contrasts to the classical ray shooting for example, which needs to sample visibility by casting independent rays until sufficient precision is achieved.

3.1.1 Beam tracing

The *beam tracing* was designed by Heckbert and Hanrahan [Heck84] to overcome some problems connected with ray tracing. Rather than shooting a single ray at a time it casts a pyramid (beam) containing infinitely many rays. The resulting algorithm makes better use of coherence of neighboring rays and eliminates some aliasing connected with the classical ray tracing by providing analytic description of visibility.

Beam tracing starts by casting a pyramid corresponding to the whole viewing frustum. A modified Weiler-Atherton algorithm (see Section A.3.2) is used to find intersections of the current pyramid with the scene polygons. The current pyramid is subdivided into frusta (beams) each intersecting a single polygon. The algorithm continues by recursively casting reflected and refracted beams. The drawback of the algorithm is that the beams might become rather complex and the implementation of a robust and fast beam casting algorithm is difficult.

3.1.2 Cone tracing

The *cone tracing* proposed by Amanatides [Aman84] traces a cone of rays at a time instead of a polyhedral beam or a single ray. It was designed to simulate glossy reflections and soft shadows. In contrast to the beam tracing the algorithm does not determine precise boundaries of visibility changes. The cones are intersected with the scene objects and at each intersected object a new cone (or cones) are cast to simulate reflection and refraction. The cone angle can be adjusted based on the surface roughness or fitted to match sizes of areal light sources.

3.1.3 BSP tree projection

Naylor [Nayl92b] developed an elegant and efficient visible surface algorithm for rendering polygonal scenes represented by a BSP tree. The algorithm combines principles of Warnock's algorithm and Weiler-Atherton's algorithm.

The scene is swept in a front to back order using an ordered traversal of the BSP tree. The view of the scene is represented by a 2D BSP tree resulting from projections of the visible scene polygons. The 2D BSP tree is used to clip the invisible polygons using an efficient hierarchical traversal of the tree.

The front to back traversal of the scene BSP tree is interleaved with visibility tests of cells corresponding to interior nodes of the tree. These tests are carried out using a polygonal representation of the cell and the 2D BSP tree representing the current view. The hierarchical visibility tests lead to an output-sensitive behavior of the algorithm. The drawback of the method is that it requires that the whole scene is represented using a BSP tree. This poses a significant problem for large and especially dynamic scenes.

3.1.4 Frustum casting

Teller and Alex [Tell98] characterize visible surface algorithms by a *working set* and *overdraw*. They define the working set as the extent to which, and the order in which, the algorithm accesses virtual memory corresponding to nodes of the spatial index (e.g. kD-tree, object bounding boxes, etc.) or the scene data (e.g. triangles). The working set is closely related to overdraw, i.e. the effort spent by the render to process objects that do not contribute to the final image.

They propose a concept of frustum casting that synthesizes three well-known algorithms: screen space subdivision, beam tracing and accelerated ray shooting using a spatial subdivision. Frustum casting should combine efficient aspects of the three algorithms while overcoming their weaknesses.

Teller and Alex [Tell98] proposed several optimizations that improve the performance of the algorithm, e.g. if all rays hit the same convex object and it is the only object intersecting the current frustum we know that all rays in the frustum intersect the object.

The concept described further in this chapter generalizes the idea of casting a set of rays by casting a general set of rays that need not originate at a single point. It can be seen as an extension of the Naylor's BSP projection algorithm in two aspects: Firstly, instead of a 2D BSP representing a view with respect to a point it uses an occlusion tree representing a set of occluded rays with respect to a point, line segment or a region. Secondly, the concept does not rely on an exact priority order of scene polygons provided by a BSP tree representing the whole scene. Instead, it uses an approximate priority order induced by a spatial hierarchy and resolves the depth priority conflicts by restructuring the occlusion tree.

3.2 Approximate occlusion sweep

Traditional list-priority methods for visible surface determination aim to determine strict priority ordering of the scene polygons. A popular approach is the algorithm using an autopartition BSP tree [Fuch80] that organizes the scene polygons. By simple traversal of the tree the algorithm determines a strict front-to-back or back-to-front order of the polygons with respect to the given viewpoint. There are three main issues with the autopartition BSP algorithm:

- The BSP tree increases the amount of scene polygons due to splitting.
- The tree is not well suited to dynamic scenes since the partitioning planes are aligned with the scene polygons.
- The strict priority order is determined with respect to a single point (viewpoint) not a set of points.

We use a novel concept of approximate priority ordering: *approximate occlusion sweep*. The approximate occlusion sweep processes scene polygons in an approximate front-to-back order: a currently processed polygon can be occluded by k unprocessed polygons. k is typically very small and very often $k = 0$. The main advantage of the method is that almost any common spatial index such as a kD-tree, an octree or a bounding volume hierarchy can be used to establish the approximate front-to-back order. Additionally the approximate occlusion sweep can be applied with respect to a point, line segment or a region.

The approximate occlusion sweep is used to construct the AOM, i.e. a data structure capturing the aggregated occlusion of the processed polygons. To ensure that the computed result is not influenced by the approximate character of the sweep, the AOM *must support* the insertion of polygons in the reverse priority order ($k > 0$). Additional checks are performed during the polygon insertion to accurately determine the position of the currently processed polygon with respect to the already processed ones. Late processing of a polygon with a higher priority that is in front of the already processed polygons, causes a certain performance penalty. The assumption is that such a case is not very frequent in practice and thus the performance penalty is amortized.

Suppose we use a kD-tree to organize the scene polygons. The tree is built by recursive subdivision until certain termination criteria are met. Leaves of the tree contain references to scene polygons. In practice each leaf contains a small number of references (number of objects per leaf is one of the termination criteria of the tree construction algorithm[1]).

Two methods can be used to perform the approximate occlusion sweep using a kD-tree: one corresponding to the depth-first traversal of the tree, the other to the breadth-first traversal.

The depth-first traversal method visits leaves of the tree in a front-to-back order with respect to a given point. The algorithm uses the partitioning planes associated with interior nodes of the tree to establish the front-to-back order as proposed by Fuchs et al. [Fuch80].

The second method is more general. It uses a priority stack for the breadth-first traversal. The priority of the node is inversely proportional to the minimal distance of the hierarchy node from the viewpoint. This approach can also be used for an octree, a bounding volume hierarchy, or a hierarchy of uniform

[1]However, in a pathological case when the scene objects are not separable by an orthogonal plane, there can be as much as $O(n)$ objects per leaf.

grids. The breadth-first traversal is based solely on the priority of the given hierarchy node. This allows to apply the method to dynamic scenes where the structure of the hierarchy changes.

In both traversal methods the polygons associated with a leaf node are processed in random order. Alternatively, a simple runtime depth-ordering of the polygons within a leaf can be used to increase the accuracy of the generated priority order. An illustration of the progress of this method is depicted in Figure 3.1.

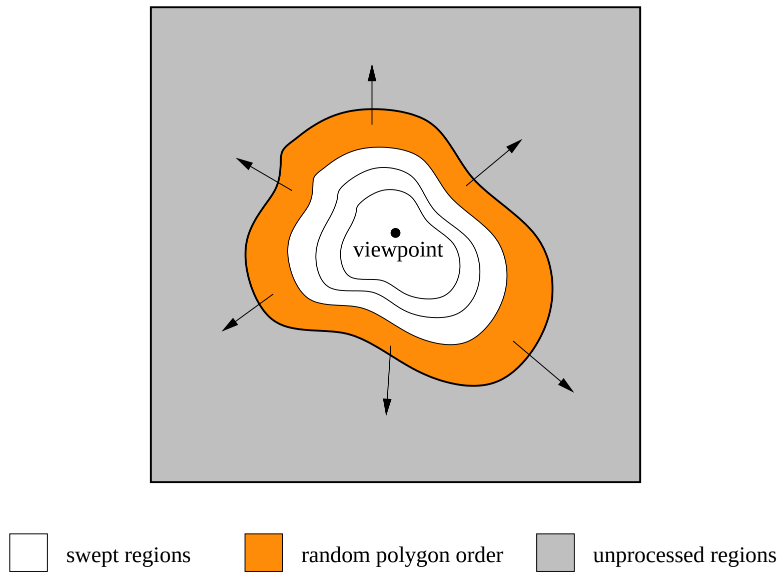


Figure 3.1: Approximate occlusion sweep. The scene is processed in an approximate front-to-back order. The order is defined by the traversal of a spatial hierarchy. On the figure a breadth-first-like traversal is depicted. In a currently visited region polygons are processed in random order.

3.3 Occlusion tree

The occlusion tree is a BSP tree maintaining a line space subdivision. The tree is used to represent the AOM of already processed polygons and depending on the particular visibility problem it captures occluded rays emanating from a point, line segment or a region in the scene that are blocked by processed polygons. For visibility from point in 3D scenes the occlusion tree [Bitt98] is a BSP tree representing a view of the scene. For this case it is conceptually equivalent to the Shadow Volume BSP tree introduced by Chin and Feiner [Chin89] discussed in Section C.1.4. The tree is a BSP representation of the image consisting of already processed polygons. Each point inside a polygon in the image corresponds to a ray blocked by that polygon. Such a BSP representation of the image with depth was also used by Naylor [Nayl92b] (the method described in Section 3.1.3).

The occlusion tree is a generalization of the BSP representation of the view of the scene. The sets of rays blocked by already processed occluders are described as line space polyhedra. The occlusion tree is used to maintain the union of these polyhedra. The intersections of the polyhedra are resolved using an information about the depth of the corresponding occluders.

A detailed description of the structure of the occlusion tree for a specific visibility problem will be discussed in Chapters 4, 5, 6, 7, and 8. This section reviews only the principle of using BSP trees for polyhedra set operations proposed by Thibault and Naylor [Thib87] and Naylor et al. [Nayl90b]. It describes the general structure of the occlusion tree and outlines the associated visibility algorithms.

3.3.1 Polyhedra set operations using BSP trees

BSP trees are often used to organize polygonal scenes. The BSP based visibility algorithm of Fuchs et al. [Fuch80] has been improved by Gordon and Chen [Gord91] to achieve output-sensitivity for a certain type of environment and further developed by Naylor [Nayl92b]. Subramanian and Naylor [Subr97] introduced an algorithm for converting discrete images to 2D BSP trees. Dynamic changes to BSP trees were studied in [Torr90, Chry92]. Other work on BSP trees can be found in [More95, Garr96, Agar97, Huer97, Mars97, Wile97, Berg97, Mura97, Nech96, Tobo99].

In this section we focus on an application of BSP trees for representing a collection polyhedra and performing set operations on the polyhedra [Thib87, Nayl90a, Nayl90b, Nayl92a, Nayl93].

A polyhedron P is represented by recursive subdivision of the whole space by hyperplanes into convex cells. A hyperplane h_v is associated with each interior node v of the tree. Each node v of the tree corresponds to a convex cell R_v. If v is an interior node the hyperplane h_v subdivides R_v into smaller cells R_v^+ and R_v^-. Let h_v^+ be the positive halfspace and h_v^- the negative halfspace bounded by h_v. The cells associated with the left and the right children of v are $R_v \cap h_v^+$ and $R_v \cap h_v^-$, respectively.

The root of the tree corresponds to the whole space and the leaves to elementary cells. Each elementary cell is classified as in or out depending whether it is inside or outside of the polyhedron P.

Set operations on two polyhedra can be performed by *merging* their BSP trees. Merging of two BSP trees can be described as follows:

1. Denote the smaller tree T^A, the bigger one T^B. The algorithm will merge T^A into T^B.
2. Identify a set of *relevant leaves* $\mathcal{N}^A$ of T^A. The notion of a relevant leaf depends on the particular operation.
3. For each leaf L_i^A of $\mathcal{N}^A$ perform the following steps:
   (a) Recursively find all leaves $\mathcal{N}_i^B$ of T^B intersecting the polyhedron P_i^A associated with L_i^A
   (b) Depending on the particular operation update $\mathcal{N}_i^B$ by inserting planes bounding P_i^A. Update classification of the resulting new nodes.

For the sake of representing aggregated occlusion we are interested in the set union operation: at each step of the approximate occlusion sweep we extend the tree by a polyhedra representing rays occluded by the currently processed polygon.

In the case of a set union operation the relevant leaves determined in step 2 of the merging algorithm are all in-leaves of T^A. The step 3-(b) then proceeds as follows:

3.(b) For each leaf $L_j^B \in \mathcal{N}^B$:
   (a) If L_j^B is in-leaf, do nothing.
   (b) If L_j^B is out-leaf, replace it by a subtree representing P_i^A. Use only such planes of P_i^A that intersect the interior of the cell corresponding to L_j^B.

The proposed concept uses a simpler form of the set union operation that merges a single polyhedron into the BSP tree. This case corresponds to merging of two BSP trees T^A and T^B where T^A is an elementary tree with a single in-leaf. See Figure 3.2 for an illustration of the set union operation.

3.3.2 Structure of the occlusion tree

The occlusion tree is a BSP tree that partitions the problem-relevant line set $\mathcal{L}_R$. Each node represents a set of rays $\mathcal{Q}_N \subset \mathcal{L}_R$ emanating from the given point, line segment or a polyhedral region depending

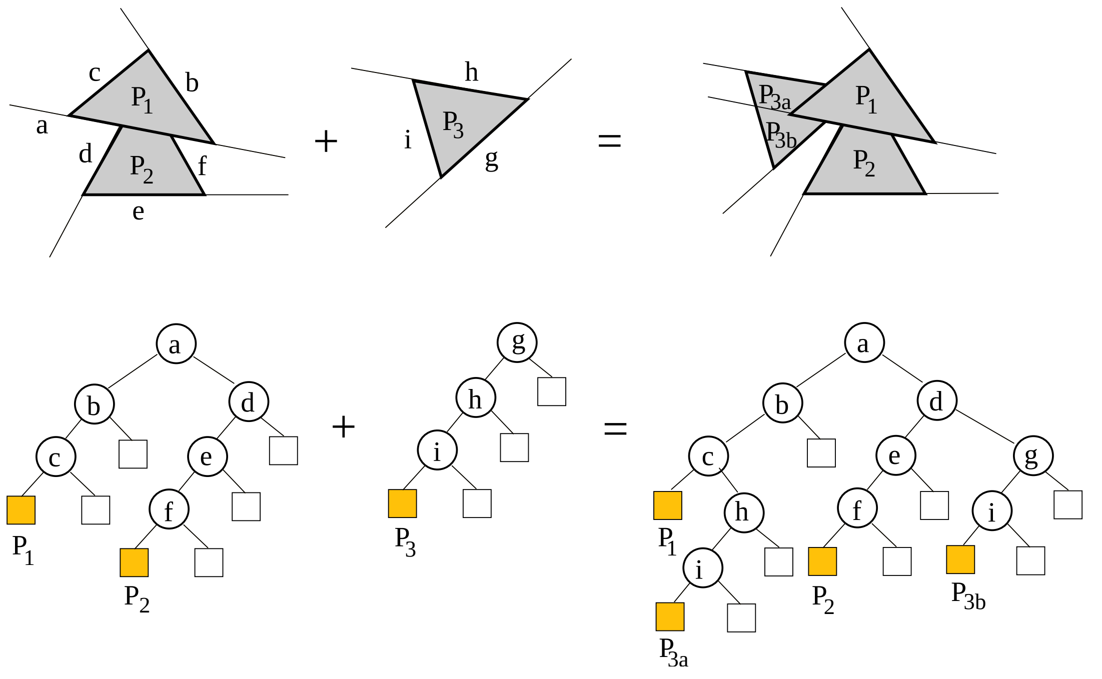


Figure 3.2: A 2D example of the set union operation. Polygon P_3 is merged into the BSP tree representing a union of P_1 and P_2. P_3 is split into two fragments that correspond to the two new in-leaves of the BSP tree.

on the particular visibility problem. The root of the tree represents the whole $\mathcal{L}_R$. Each interior node N is associated with a plane h_N partitioning the set of rays associated with the node. Left child of N represents rays $\mathcal{Q}_N \cap h_N^-$, right child $\mathcal{Q}_N \cap h_N^+$, where h_N^- and h_N^+ are halfspaces induced by h_N. Leaves of the tree are classified in or out. If N is an out-leaf, $\mathcal{Q}_N$ represents unoccluded rays. If N is an in-leaf, it is associated with a closest scene polygon P that is intersected by the corresponding set of rays $\mathcal{Q}_N$.

Probably the most intuitive is the occlusion tree for visibility from point in 3D. To further simplify the understanding of the structure of the tree we can think about the occlusion tree in a projection to a particular 2D viewport. Then the root of the tree corresponds to the whole viewport. Each interior node is associated with a line subdividing the current polygonal region in two parts. Leaves of the tree represent either empty region of the viewport or a fragment of a visible polygon.

Occlusion tree constructed for a single polyhedron P contains interior nodes corresponding to the planes defined by facets of P. We call such a tree *elementary occlusion tree*, denoted e-OT(P) (see Figure 3.3). e-OT(P) contains single in-node corresponding to the interior of P.

3.3.3 Construction of the occlusion tree

The occlusion tree is constructed by incremental insertion of polyhedra. Each polyhedron corresponds to a set of rays blocked by a scene polygon. The insertion order is determined by the approximate occlusion sweep. For visibility from point in 3D the line space polyhedron is represented by the scene polygon itself: each point inside the polygon represents a single ray emanating from the viewpoint and intersecting the polygon. This chapter however discusses the general approach and thus the algorithms are presented in terms of line space polyhedra.

For an occluder O the algorithm inserting a corresponding polyhedron P_O in the tree maintains two variables: the current node N_c and the current polyhedral fragment P_c. Initially N_c is set to the root of the tree and $P_c = P_O$. The insertion of a polyhedron in the tree proceeds as follows: If N_c is an interior node we determine the position of P_c and the plane h_{N_c} associated with N_c. If P_c lies in the negative

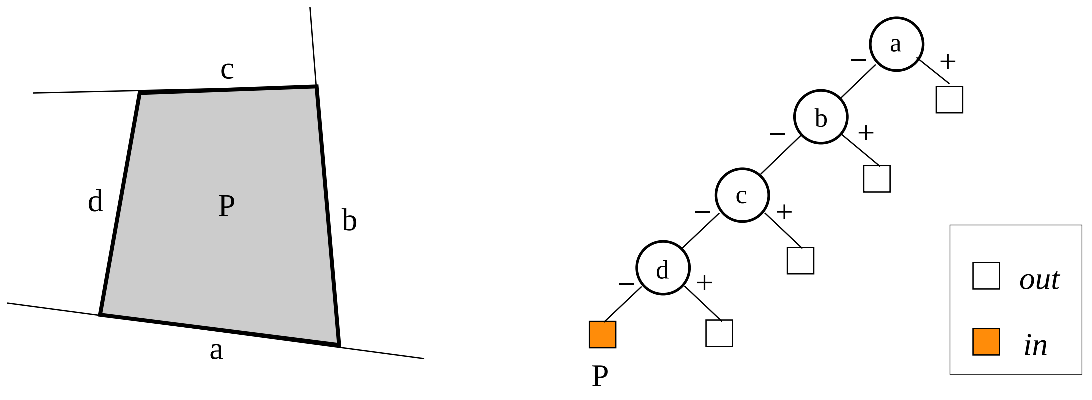


Figure 3.3: 2D example of an elementary occlusion tree. The tree contains four interior nodes corresponding to the edges of the polygon.

halfspace induced by h_{N_c} the algorithm continues in the left subtree. Similarly if P_c lies in the positive halfspace induced by h_{N_c} the algorithm continues in the right subtree. If P_c intersects both halfspaces it is split by h_{N_c} into two parts P_c^- and P_c^+ and the algorithm proceeds in both subtrees of N_c with relevant fragments of P_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 rays corresponding to P_c are unoccluded and N_c is replaced by e-OT(P_c). If N_c is an in-leaf we check the mutual position of the occluder O_{N_c} and the occluder associated with N_c. If O is behind O_{N_c} it is invisible and no modification to the tree necessary. Otherwise N_c is replaced by e-OT(P_c) and the 'old' polyhedron P_{N_c} is merged in the new subtree. Note that this case occurs when the strict front-to-back order of scene polygons is violated. See Figure 3.4 for an example of the occlusion tree for visibility from point in 3D.

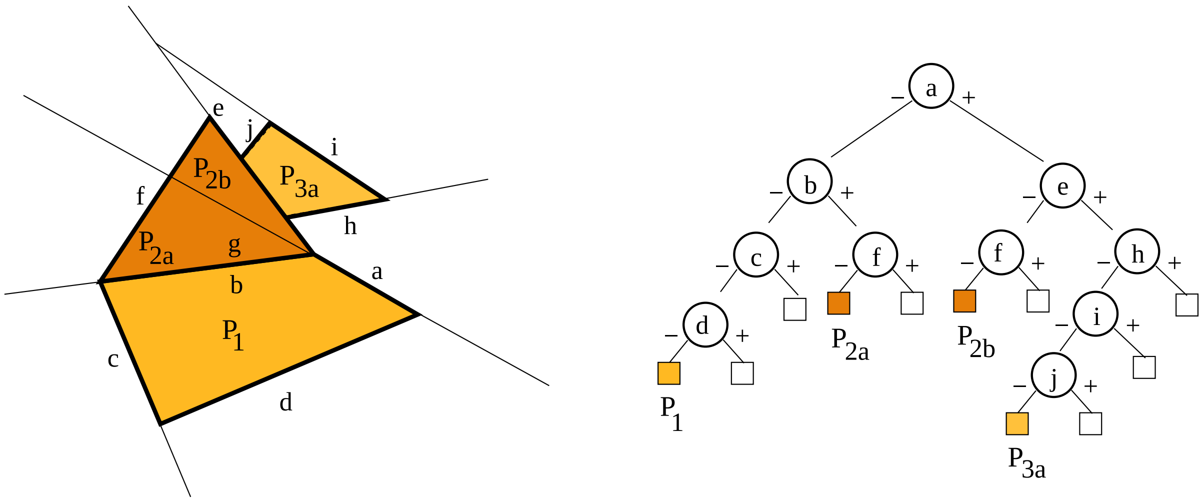


Figure 3.4: An example of the occlusion tree for visibility from point in 3D. Rays intersecting polygonal occluders O_1, O_2, and O_3 are represented by 2D polygons P_1, P_2, and P_3, respectively. Polygon P_2 is split into two visible fragments. P_3 is partially covered by P_2 and so the tree reflects only its visible part P_{3a}.

3.3.4 Visibility test using the occlusion tree

The occlusion tree can be used to test if a given set of rays intersects the already processed polygons. In general the test results in one of the three visibility states: invisible, partially visible, fully visible.

Assume we test visibility of an occluder O that induces a line space polyhedron P_O. The visibility test is performed similarly as the insertion of a polyhedron into the tree. The difference is that no changes to the tree are performed even if the polyhedron P_c corresponds to a set of unoccluded rays. Reaching an out-leaf we classify the corresponding fragment as visible. Reaching an in-leaf the visibility state depends on the mutual position of occluders O and O_{N_c}. If O is in front of O_{N_c}, it is visible. If it is behind O_{N_c}, it is invisible. The visibility classification of P is obtained by combining visibility states of all leaves reached by the traversal of the tree according to the Table 3.1.

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 3.1: Combining visibility states of two fragments.

Whenever the combination results in partial visibility the algorithm can be terminated. If we do not distinguish between the fully visible and partially visible states the algorithm can be terminated as soon as any visible fragment is found. See Figure 3.5 for an illustration of the visibility test.

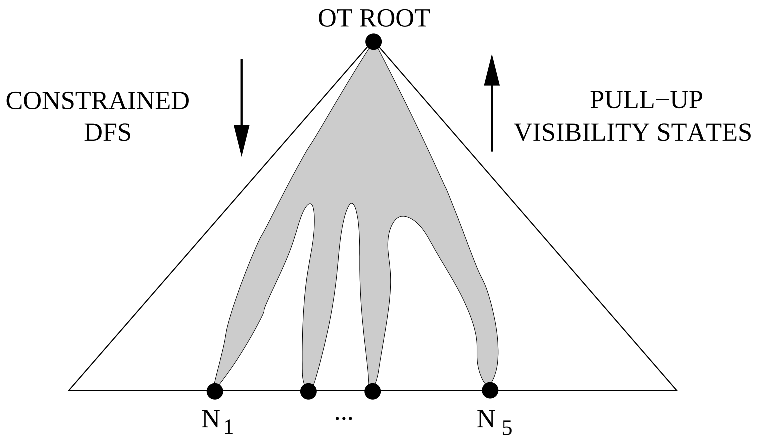


Figure 3.5: Visibility test using an occlusion tree. The algorithm performs a constrained depth first search (DFS) on the tree. The visibility states determined in leaves are pulled up the tree and combined according to the Table 3.1.

3.4 Hierarchical visibility tests

In order to increase efficiency of the algorithm the approximate occlusion sweep can be interleaved with visibility tests applied on the nodes of the spatial hierarchy. If the test determines that the node is invisible, the corresponding subtree and all polygons it contains can be culled.

The visibility of the bounding box of the current kD-tree node is determined using the visibility test described above. The test uses a line space polyhedron representing rays of the problem-relevant line set $\mathcal{L}_R$ that intersect the bounding box. If the test classifies the node as invisible, the whole subtree of the corresponding kD-tree node can be culled. See Figure 3.6 for a 2D example of hierarchical visibility tests for visibility from point.

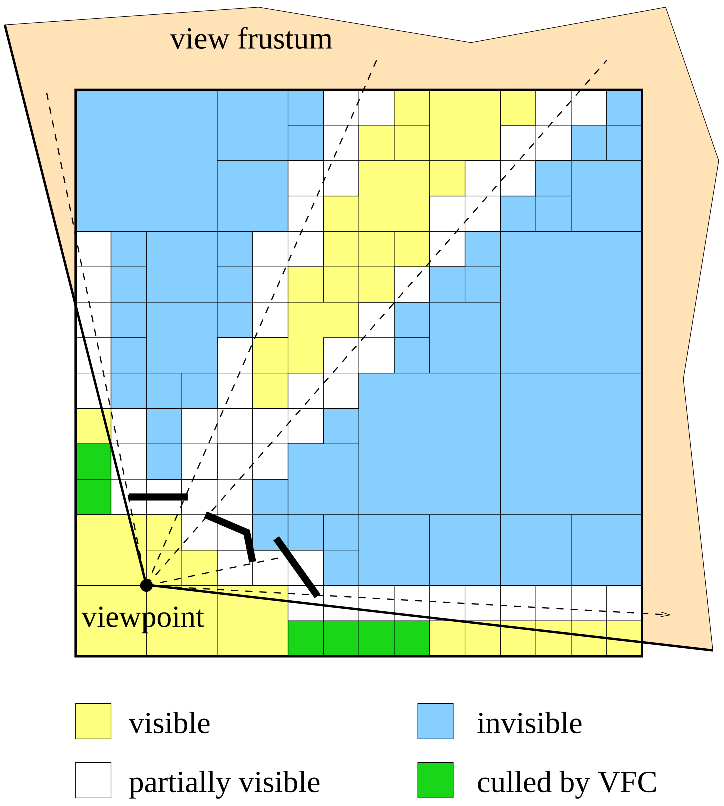


Figure 3.6: Hierarchical visibility tests. The figure depicts resulting visibility classification of kD-tree nodes. These nodes are leaves of the subtree visited by the approximate occlusion sweep.

The hierarchical visibility tests provide a great benefit in densely occluded scenes where many scene objects are invisible. This leads to an output-sensitive behavior of the algorithm in practice. Figure 3.7 depicts a pseudo-code of a hierarchical from-region visibility algorithm.

3.5 Complexity analysis

This section presents an analysis of the time and space complexities of the proposed methods. It is hard to provide meaningful bounds on the expected complexity of the algorithm, since the construction of the supporting structures relies on various heuristics. Therefore, we make several assumptions to simplify the analysis.

3.5.1 kD-tree

The properties of the scene kD-tree significantly influence the behavior of the proposed techniques especially for densely occluded scenes. kD-trees were designed to organize sets of points in $\mathcal{R}^d$ [Bent75, Same90]. The size of a kD-tree for n points is $s_p = 2n - 1$ assuming that each leaf is associated with a single point. A kD-tree of $O(\log n)$ height can be constructed in $O(n \log n)$ time.

```
HierarchicalVisibility( Region $R_S$, kDTree KD ) {
 1:  OT.Init($R_S$)  // initiate OT
 2:  pqueue.Put(KD.root)  // initiate priority queue
 3:  **while** (pqueue is not empty) {
 4:    N ← pqueue.Get() // get next node from the queue
 5:    **if** ($R_N$ intersects $R_S$)
 6:      N.vis ← VISIBLE
 7:    **else**
 8:      N.vis ← OT.TestVisibility($R_N$)
 9:    **if** (N.vis != INVISIBLE) {
10:      **if** (N is leaf)
11:        OT.InsertOccluders( N.occluders )
12:      **else**
13:        pqueue.Put( children **of** N )
14:    }
15:  } // **while**
16: }
```

Figure 3.7: Pseudo-code of the hierarchical from-region visibility algorithm.

Analysis of a kD-tree for a set of polygons is more complicated. First of all a kD-tree with a single polygon per leaf need not exist, if the polygons are not separable by an orthogonal plane. Thus in the worst case for n polygons there can be as much as $O(n)$ polygons per leaf for a kD-tree of size $O(m)$ resulting in total $O(mn)$ space complexity.

In practice we observe that for most real world scenes we can construct a kD-tree with $O(n)$ size, $O(1)$ polygons per leaf and $O(\log n)$ average depth. The tree is typically constructed using the *minimal splits heuristics* [Bitt98] or the *surface area heuristics* [MacD90, Havr00a], which takes $O(n \log n)$ time.

3.5.2 Number of swept nodes

For further analysis it is important to express the number of nodes of the kD-tree visited by the occlusion sweep. If there are k visible polygons in the scene, there are $O(k)$ leaves of the kD-tree classified visible (assuming there is $\Theta(1)$ polygons per leaf). The $O(k)$ leaves induce a subtree of a size between $\Omega(k + \log n)$ and $O(k \log n)$ (assuming the height of the kD-tree is $O(\log n)$). The lower bound holds in the case that the subtree is a compact subtree of height $O(\log k)$, the upper bound holds for visible nodes spread in disjoint branches of the kD-tree. These bounds hold for the total number of nodes visited by the sweep assuming that for each interior node of the spatial hierarchy classified as visible, at least one of its child nodes is visible (this cannot be guaranteed if the visibility test is more accurate for smaller cells).

We can conclude that there are $O(k \log n)$ nodes tested for visibility. The total number of processed polygons is $O(k)$ some of which can be classified invisible due to the approximate polygon ordering.

3.5.3 Size of the occlusion tree

The occlusion tree maintains a subdivision of the problem-relevant line set $\mathcal{L}_R$. The dimension of $\mathcal{L}_R$ induces bounds on the size of the tree. Assume the tree captures $O(k)$ blocker polyhedra, where k corresponds to the number of visible occluders and each blocker polyhedron consist of $O(1)$ faces. The trivial upper bound of the tree size is $O(k^d)$ where d is the dimension of the problem-relevant line set. This bound corresponds to the size of the arrangement of hyperplanes in $\mathcal{R}^d$ [Dobk97]. The actual size of the tree highly depends on the mutual positions of the blocker polyhedra, their volume and the order of their insertion into the tree.

For the case of $d = 2$ we can derive a more accurate upper bound: the expected size of the tree is $O(k \log k)$ for k random non-intersecting polygons. This bound follows from the analysis of size of a 2D BSP tree for a set of random line segments constructed using a simple randomized algorithm [Berg97]. For higher dimensions ($d > 2$) however a similar randomized algorithm yields a $O(k^d)$ upper bound.

In Chapters 5, 6, and 8 we will present measurements of the size of the occlusion tree for various types of input data. We can observe that the $O(k^d)$ upper bound is rather pessimistic and the size of the occlusion tree is typically much smaller in practice.

3.5.4 Analysis of a visibility test

A visibility test using an occlusion tree of size s with $O(\log s)$ height takes $O(\log s)$ time at the best case and $O(s)$ time at the worst case. The best case corresponds to the situation that the test terminates immediately after reaching a leaf node. The test can be terminated as soon as an out-leaf is reached since we know that the tested region is at least partially visible.

The worst case holds for the case that we check visibility of a very large invisible region that covers all rays captured by the tree. Note that if a large region is found invisible all its subregions are culled, which amortizes the total time taken by visibility tests.

3.6 Summary

This chapter presented a general concept suitable for solving from-point, from-segment and from-region visibility. The concept is based on the idea of sweeping the scene and constructing an aggregated occlusion map. To abstract from the particular visibility problem we propose to use an occlusion tree, i.e. a BSP tree maintaining the occlusion map using a hierarchical partitioning of the problem-relevant line set. The tree represents a set of rays blocked by the already processed polygons. The hierarchical structure of the occlusion tree is used for efficient updates and visibility tests. The visibility tests are applied on the nodes of the spatial hierarchy during the occlusion sweep. These tests allow to quickly cull invisible parts of the scene, which leads to the output-sensitive behavior of the algorithm. The proposed approximate occlusion sweep provides a compromise between computationally costly strict ordering of the scene polygons and a fast ordering that does not require additional data structures and calculations.

Chapter 4

Real-time visibility culling

This chapter presents an algorithm for real-time visibility culling for the acceleration of walkthroughs of large scenes. The proposed technique belongs to the class of visibility from point problems. According to the classification from Chapter 2 it is an online, continuous, and output-sensitive from-point visibility algorithm that makes use of visibility coherence (object space, line space, and temporal).

4.1 Problem statement

The goal of the real-time visibility culling can be defined as follows: given a scene and a viewpoint quickly determine a PVS, i.e. a set of objects potentially visible from the viewpoint. These objects are then sent to the graphics pipeline; the rest of the scene is culled. The final visibility is typically resolved using z-buffer. The opposite criteria posed on the real-time visibility culling are the speed of the algorithm versus the accuracy of the resulting visibility classification. The algorithm should take only a fraction of the total frame time but the visibility classification should be precise enough so that most invisible objects are culled.

4.2 Related work

In computational geometry visible surface determination was studied theoretically by de Berg [Berg93b, Berg93a], Mulmuley [Mulm89] and Sharir [Shar92]. Grant [Gran92] presented a survey of practical visibility algorithms for computer graphics concentrating on visible surface and hard shadow algorithms.

Visible surface determination is commonly solved using the z-buffer algorithm [Catm75] or BSP trees [Fuch80]. Both these algorithms are not output-sensitive since they can spend significant time by processing objects that are actually invisible. The efficiency of these methods can be improved by applying some culling techniques such as back face culling [Zhan97a, Kuma96b, Kuma96a] or view frustum culling [Clar76, Assa00] (see Chapter B for more details). Another possibility is a parallelization of the visible surface algorithm [Geor95, Fran90]. In the rest of this section we review work on visibility culling methods.

A lot of research has been devoted to the concept of *potentially visible sets* (PVS) in architectural environments that can be decomposed into cells connected by transparent portals [Aire90, Tell91]. The cell/portal methods belong to the from-region visibility problems that will be discussed in Chapters 6, 7 and 8. An exception is the algorithm of Luebke and Georges [Lueb95] that applies the cell/portal based culling for each viewpoint in real time. The *hierarchical z-buffer* of Greene [Gree93] uses a discrete *z-pyramid* to represent the occlusion map with respect to the viewpoint. It exploits spatial coherence of visibility by processing the spatial hierarchy through the z-pyramid (see Section B.3.1 for more details). The hierarchical z-buffer is a promising approach for hardware implementation, but the simulation of the z-pyramid in software causes a significant overhead. Greene [Gree94a] combines hierarchical visibility culling with antialiasing techniques to avoid visual artifacts for scenes with many small polygons. Another method of Greene [Gree96], suitable for high resolution rendering, uses a hierarchy of discrete coverage masks to represent the occlusion map (see Section B.3.2). An algorithm that uses discrete image space representation of the occlusion map was introduced by Zhang et.al [Zhan97b] (see Section B.3.3). This technique exploits graphics hardware to create a hierarchical occlusion map using texture filtering operation. While taking advantage of hardware rendering, this method suffers if the rendering support is insufficient or the frame buffer read-back is slow. Bartz et al. [Bart98] proposed a simple extension of the graphics hardware to support occlusion queries.

The method presented in this chapter is closely related to object space visibility culling algorithms presented by Hudson et.al [Huds97] and Coorg [Coor96a, Coor96b, Coor97]. Hudson et al. [Huds97] create a *shadow frustum* for each selected occluder. These frusta are used to classify visibility regions corresponding to nodes of the spatial hierarchy. The method does account for occluder fusion: a region is tested against each frustum independently and the occlusion caused by multiple unconnected occluders is not discovered (see Section B.3.4 for more details). Coorg and Teller [Coor96a, Coor96b, Coor97] construct supporting and separating planes for an occluder and a box corresponding to a node of the spatial hierarchy. The algorithm tests the position of the viewpoint with respect to the umbra and penumbra induced by the supporting and separating planes. By caching these planes the algorithm makes use of temporal coherence. The method does not account for occluder fusion since occlusion due to multiple occluders is not discovered (see Section B.3.5 for more details).

4.3 Algorithm overview

The algorithm presented in this chapter is designed to assist z-buffer to achieve an output-sensitive behavior. The proposed method is conservative, i.e. for a given viewpoint it determines a superset of objects visible from the viewpoint. The algorithm uses the concept introduced in Chapter 3: the approximate occlusion sweep, the occlusion tree, and hierarchical visibility tests. In preprocessing we build a kD-tree to organize the scene polygons. The polygons larger than a given threshold are classified as potential occluders. For the current viewpoint the kD-tree is used to sweep the scene in an approximate front-to-back order. The potential occluders encountered by the sweep are inserted in the *occlusion tree*. The occlusion tree represents all rays blocked by the inserted occluders and so it efficiently merges their occlusion volumes. When a new node of the kD-tree is processed, it is first tested for visibility using the current occlusion tree. If the node is invisible, the whole corresponding subtree is culled. The insertion of occluders into the occlusion tree stops once we have inserted a predefined number of occluders. Then the tree is only used for visibility classification of the remaining part of the scene using the given occluder subset.

The kD-tree is used to obtain the approximate front-to-back order and to exploit spatial coherence of visibility. Starting at the root of the kD-tree the algorithm determines visibility using the current occlusion tree. If a node is found visible, all its descendants are visible. Similarly, if a node is found invisible, all its descendants are invisible. Descendants of nodes classified as partially visible must be further tested to refine their visibility. When the visibility of all leaves is known, objects from fully visible and partially visible leaves are gathered. These objects form the desired PVS and they are rendered using the z-buffer. See Figure 3.6 for an illustration of the from-point hierarchical visibility culling.

The rest of the chapter is organized as follows: Section 4.4 focuses on the spatial hierarchy. The occluder selection is outlined in Section 4.5. In Section 4.6 we discuss the motivation for building a unique data structure that represents the occlusion map. Section 4.7 presents two algorithms for visibility tests using the occlusion tree. Section 4.10 describes a fast conservative visibility test. Section 4.11 presents several techniques that exploit spatial and temporal coherence in the scope of the hierarchical visibility algorithms. Section 4.12 summarizes the results of the implementation of the proposed algorithms. Finally, Section 4.13 concludes.

4.4 Spatial hierarchy

The scene polygons are organized using a kD-tree [Kapl85]. kD-trees are highly flexible and are simple to construct and traverse. The most important step during the construction of the kD-tree is the choice of the partitioning plane. A partitioning plane is selected that subdivides the current region into two smaller ones and two descendants of the current node are created. Objects are distributed into the descendants according to their position with respect to the partitioning plane. Initially, the root node of the kD-tree corresponds to the bounding box of the whole scene. The whole kD-tree is built by recursive application of the subdivision algorithm. The recursion is terminated when the number of objects in the current node falls under a user defined threshold or a specified maximum depth of the hierarchy is reached. Alternatively, more advanced automatic termination criteria can be applied [Havr02].

For visibility culling purposes we want to minimize the number of object references in leaves while keeping a well balanced tree. To achieve this goal we have used the following heuristics: For the current node we identify an axis x with the largest extent of the corresponding cell. Only planes perpendicular to the axis x are considered. The algorithm identifies boundaries of object bounding boxes located within a certain distance from the spatial median of the cell. Each identified boundary induces a *boundary plane*. We evaluate the number of objects that are split by each boundary plane. The boundary plane with the lowest number of splits is selected to partition the current cell. Another option is to use the surface area heuristics developed by MacDonald and Booth [MacD90] for raytracing acceleration. Bartz et al. [Bart99] have shown that this heuristics is well suited for visibility culling as well.

4.5 Occluder selection

The goal of the occluder selection is to determine a specified number of polygonal occluders, given a viewpoint and a viewing direction. To estimate the quality of an occluder the algorithm uses the *area-angle* heuristics [Coor96a] that approximates the solid-angle spanned by the occluder. The area-angle is expressed as:

$$M = \frac{-A(\vec{N} \cdot \vec{V})}{\|\vec{D}\|^2} \qquad [sr] \tag{4.1}$$

where A is the area of the occluder, $\vec{N}$ denotes the occluder normal, $\vec{V}$ the viewing direction and $\vec{D}$ corresponds to the vector from the viewpoint to the center of the occluder ($\|\vec{N}\| = \|\vec{V}\| = 1$). The dynamic occluder selection is performed after each change of the viewpoint or the viewing direction. The algorithm evaluates area-angle of occluders associated with visible nodes of the spatial hierarchy. Therefore the occluders selected in the current frame are used for construction of the occlusion tree in the next frame.

The algorithm proceeds as follows: We identify all visible and partially visible leaves of the hierarchy, that correspond to regions located within a certain distance λ from the viewpoint. For each potential occluder referred in these leaves the area-angle is computed. These values are used to select k occluders with the largest area-angle that form the desired occluder set for the next frame.

The distance λ has an impact on the time spent by the occluder selection. In our implementation it is a user specified constant defined as a multiple of the observer's step size. The number of desired occluders (k) influences the size of the occlusion tree and in turn the time of visibility determination and its accuracy. In our experiments we have used k between 6 and 256. Usually the more occluders, the more objects are culled at the cost of increased time for the visibility culling (results of the measurements for various settings will be presented in Section 4.12).

4.6 Representation of the aggregated occlusion map

The crucial part of the visibility culling algorithm is the representation of the aggregated occlusion map (AOM). Unlike previous continuous methods for visibility culling, we build an auxiliary continuous data structure representing the AOM — the occlusion tree. This approach has two main advantages compared with methods that treat occlusion volumes separately [Coor97, Huds97]:

- Visibility culling is *more accurate* since the occlusion tree accounts for *occluder fusion*.
- Visibility culling is *faster* since it exploits *visibility coherence*.

The occlusion tree efficiently merges occlusion volumes of the selected occluders. This allows to discover occlusion caused by multiple connected occluders and even occluders at different depths that overlap in the image plane. Merging occlusion volumes requires an additional time to build the appropriate data structure. On the contrary such a data structure allows efficient insertions and queries. We have observed that for typical densely occluded scenes with large occluders the additional time is easily recovered by savings due to visibility culling.

The occlusion tree for the from-point visibility culling is equivalent to the *shadow volume BSP* (SVBSP) tree of Chin and Feiner [Chin89]. The SVBSP tree was designed for fast rendering of shadows with respect to a point light source in a polygonal scene (see Section C.1.4 for more details).

4.7 Occlusion tree

An occlusion tree for visibility from point is a BSP tree built with respect to a set of occluders and a viewpoint. The occlusion tree is constructed by processing occluders in approximate front-to-back order and enlarging the tree by the corresponding shadow frusta. The insertion of an occluder proceeds as the SVBSP construction algorithm described in Section C.1.4.

Each in-leaf is linked to a fragment occluding the frustum that corresponds to this leaf. These links are used in the visibility algorithm to determine if a polyhedron tested for visibility lies behind the occluder.

If we use an approximate front-to-back order to insert the selected occluders into the occlusion tree, the tree construction algorithm must allow to insert an occluder lying in front of some already inserted occluder(s). If the closer occluder was not inserted, the occlusion tree would contain a conservative depth information. Alternatively, since the set of selected occluders is small, the occluders can be used to construct an autopartition BSP tree at each frame. This tree allows to establish a strict front-to-back order of occluders.

Visibility of a closed polyhedral region can be determined by combining visibility classifications of its polygonal faces. The next section describes how to classify visibility of a convex polygon. Further we present a visibility algorithm for a convex polyhedron. Both these algorithms classify the visibility *exactly* with respect to the selected occluders. Since these occluders form only a subset of scene objects the resulting visibility classification is only conservative.

4.8 Visibility of a polygon

Visibility of a polygon is determined using a constrained depth first search of the occlusion tree. In each internal node of the tree we test the position of the polygon with respect to the plane referred in the node. If the polygon lies completely in the negative/positive halfspace defined by the plane, the algorithm recursively continues in the left/right child of the current node. Otherwise the polygon is split in two fragments and the algorithm is applied on both children using appropriate fragments.

When a leaf is reached, the visibility of the current fragment of the polygon is classified as follows: Reaching an out-leaf the fragment is fully visible. Reaching an in-leaf visibility is determined by checking the depth of the processed polygon and the occluder fragment associated with the leaf (this test will be described later in this section). The visibility classification is propagated up the occlusion tree. Visibility of an internal node is computed by combining visibility states of node's children according to Table 3.1. Visibility state of the root corresponds to the visibility of the tested polygon.

The tree traversal can be terminated whenever a fragment is found partially visible. This follows from the fact that if a fragment of the polygon is partially visible, the polygon itself must be partially visible (see Table 3.1). This constraint can significantly accelerate the visibility algorithm, particularly for large polygons, which are likely to be partially visible.

Since the polygon tested for visibility need not lie behind all occluders, reaching an in-leaf we need to check the depth of the polygon fragment with respect to the occluder associated with the leaf. If the fragment is completely in front of the supporting plane of the occluder, it is fully visible. If it is completely on the back side of the plane, it is invisible. Otherwise, it lies on both sides of the plane and it is partially visible. The pseudo-code of the polygon visibility algorithm is presented in Figure 4.1.

```
Algorithm Visibility(Node, Polygon)
 1: begin
 2:   if Node is leaf then
 3:     if Node is out-leaf then
 4:       Visibility ← VISIBLE
 5:     else
 6:       Visibility ← visibility state based on
 7:       FragmentIntersection(Node.Fragment, Polygon);
 8:   else
 9:     case Split(Polygon, Node.Splitter, Back, Front) of
10:       FRONT : (* pass the polygon to the front subtree *)
11:           Visibility ← Visibility(Node.FrontChild, Polygon);
12:       BACK  : (* pass the polygon to the back subtree *)
13:           Visibility ← Visibility(Node.BackChild, Polygon);
14:       SPLIT : (* pass fragments to apropriate subtrees *)
15:           Visibility ← Visibility(Node.FrontChild, Front)
16:           if Visibility <> PARTIALLY then
17:           begin
18:             aux ← Visibility(Node.BackChild, Back)
19:             Visibility ← CombineVisibility(aux, Visibility);
20:           end
21:     end
22: end
```

Figure 4.1: An algorithm determining visibility of a polygon with respect to an occlusion tree.

4.9 Visibility of a polyhedron

The above presented polygon visibility algorithm can be applied to determine visibility of a convex polyhedron by testing visibility of its boundary faces. Visibility states of the faces are combined according to Table 3.1. Whenever the combination results the partially visible state, the algorithm terminates. Otherwise, it proceeds with the next face until all faces have been processed.

In the case of a spatial hierarchy based on a kD-tree, the cells corresponding to the hierarchy nodes are parallelepipeds. To determine visibility of such a cell at most three front-facing rectangular polygons must be tested for visibility. These polygons can be determined by a table lookup.

4.10 Conservative occlusion tree traversal

The described visibility algorithm is used extensively in the scope of the complete hierarchical visibility algorithm. The elementary operation taking place in both the tree construction and the visibility algorithms is the polygon splitting. The splitting introduces a computational overhead due to the fragment allocation when the polygon gets split. Moreover, the splitting operation prevents to apply the algorithm from Section 4.8 for polyhedra due to the complexity of polyhedron splitting.

Motivated by the idea of the visibility algorithm without the necessity of splitting, we have designed a fast conservative variant of the algorithm. It is based on the observation that the occlusion tree can be traversed without splitting while still obtaining an accurate visibility classification.

4.10.1 Occlusion tree for the conservative visibility algorithm

To provide accurate results of the conservative visibility algorithm the occlusion tree construction is modified to exclude the *redundant planes* from the tree. Firstly, the redundant planes increase the size of the tree unnecessarily, secondly, these planes would lead to overly conservative behavior of the algorithm.

The polygon splitting operation used during the construction of the tree is enriched by marking edges of the polygon embedded in any plane on the path from the root. If the polygon is split, the edges of both new fragments lying in the plane that splits the polygon are marked as well. When an out-leaf is reached only non-marked polygon edges are used to create planes enlarging the occlusion tree. Planes that would have been created by marked edges are already present in the tree (otherwise the edges would not be marked). If an out-leaf is reached and all edges of the filtered fragment are marked the out-leaf is replaced by an in-leaf associated with the given fragment. The difference between occlusion trees constructed without and with edge-marking is illustrated in Figure 4.2.

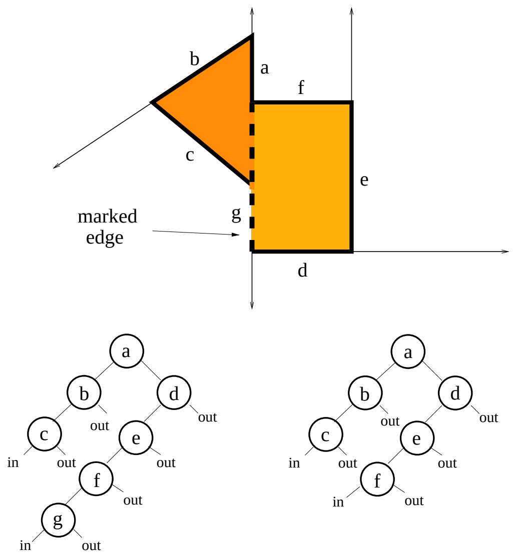


Figure 4.2: The difference between occlusion trees constructed without (left) and with the edge-marking (right). Both trees are constructed with respect to the same occluders. The occluders are shown as seen from the viewpoint. Node g is not present in the tree on the right, since the corresponding occluder edge was marked.

In the next section we present a conservative visibility algorithm that determines the visibility of regions of various shapes “directly”, i.e. without the decomposition into boundary faces and testing visibility of each face. The splitting operation is replaced by the test of the position of the polyhedron with respect to a plane.

4.10.2 Conservative visibility of a region

The conservative visibility method uses a *positional test* for a given region with respect to a plane. The positional test determines if the region lies in negative (back), positive (front), or both half spaces induced by the plane. The occlusion tree is traversed similarly to the algorithm described in Section 4.8. At each internal node of the tree we determine the position of the region with respect to the associated plane and apply the algorithm recursively on appropriate subtrees. The region is not split even if it lies on both sides of the plane.

Reaching an in-leaf the region is tested for position with respect to the occluder fragment F associated with the leaf. If the region lies on both sides of the supporting plane of F, it is classified partially visible. Since the region was not split it is possible that the part of the region crossing the supporting plane actually does not intersect the frustum corresponding to the fragment F. In this case the region is conservatively classified as partially visible although in fact it can be invisible.

The conservative behavior of the algorithm can be eliminated by testing the region for an intersection with the fragment F instead of using only its supporting plane. If F and the region do not intersect, the reached in-leaf is classified invisible. Note that the region can still be found partially visible when the visibility of all leaves reached by the visibility algorithm is combined. In the case of box shaped regions a fast algorithm can be used for the box/polygon intersection [Gree94b]. In practice however the improvement in the accuracy of the algorithm often does not pay off the time spent by the additional intersection test (see Section 4.12 for more details).

The conservative nature of the described algorithm implies that the visibility classification of leaves can vary compared with the previously mentioned exact algorithms. A situation when an invisible region is misclassified as partially visible is depicted in Figure 4.3. The algorithm is likely to give an imprecise (conservative) result if the angle α between the planes a and e gets larger. The measurements presented in Section 4.12 indicate that in practice the accuracy of the algorithm is very close to the exact one.

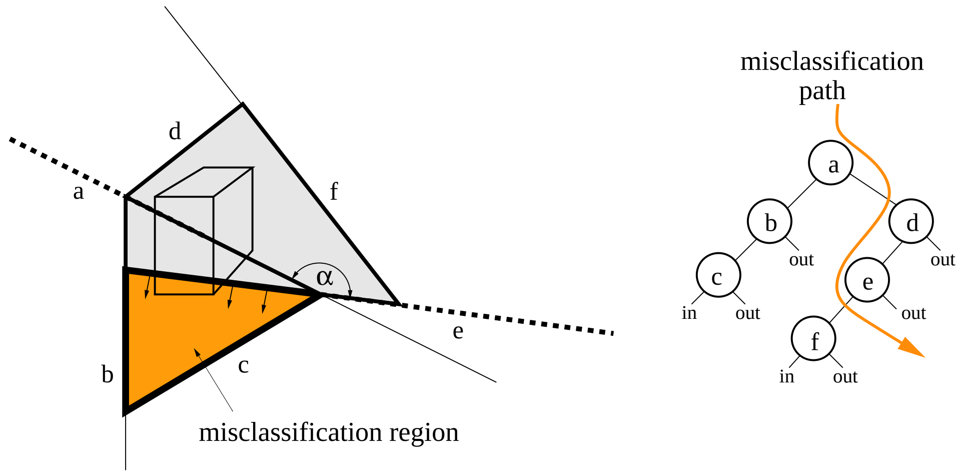


Figure 4.3: An example of a disadvantageous configuration of occluders. The planes corresponding to nodes of the occlusion tree are shown by thin lines. If the polyhedron intersects both the shadow plane a and the darker (orange) area bounded by plane e, it is classified as partially visible although it is invisible.

4.11 Exploiting temporal and spatial coherence

A typical hierarchical visibility algorithm uses a *visibility test*, that classifies a node of the spatial hierarchy as completely visible, partially visible or invisible depending on the visibility of the spatial region corresponding to that node. The visibility test is applied recursively starting at the root node. As soon as a node is found completely visible or invisible, the current branch of the traversal can be terminated, since visibility of all nodes in the current subtree is imposed by the visibility of the current node. In this section we do not focus on the amount of image space or temporal coherence, that may be exploited by the visibility test itself. Instead we suggest a more general framework that is independent of the particular visibility algorithm.

Traditional hierarchical visibility culling algorithms traverse the spatial hierarchy starting at the root node. Firstly, we propose a method, that saves up to half of the visibility tests by skipping certain interior nodes of the hierarchy (assuming the spatial hierarchy corresponds to a binary tree). The skipping is guided by visibility classifications obtained during the previous invocation of the visibility algorithm. Secondly, we describe an algorithm that increases the amount of spatial coherence exploited. It reuses visibility classifications of hierarchy nodes already processed in the current pass of the algorithm. The nodes are processed in the front-to-back order and the algorithm tries to determine visibility of the region corresponding to the current node by combining visibility states of neighboring regions. If it fails, the usual visibility test is applied. Finally, we propose a conservative method, that aims to avoid repeated visibility tests of nodes that probably remain visible.

4.11.1 Related work

Some visibility algorithms exploit temporal coherence in a specialized way. Greene et al. [Gree93] uses the set of visible objects from one frame to initialize the z*-pyramid* in the next frame and so reduces "overdraw" of the hierarchical z-buffer. Coorg and Teller [Coor96b] developed an algorithm that uses *relevant planes* which form a subset of visual events. They restrict the hierarchy traversal to nodes corresponding to planes that were crossed between successive viewpoint positions. Another method of Coorg and Teller [Coor97] exploits temporal coherence by caching occlusion relationships.

Chrysanthou and Slater have proposed a probabilistic scheme for view frustum culling [Slat97]. They partition objects into groups, which are sampled according to their distance from the view frustum. It is difficult to generalize this method for visibility algorithms, since the visible volumes can be very complex, and usually they are not explicitly reconstructed. Moreover, this method is not conservative unless changes in the viewing direction and the position of the viewpoint are restricted. Recently, Wonka et al. [Wonk01b] proposed a method that exploits temporal coherence of visibility by computing a PVS valid for a small neighborhood of the given viewpoint.

The methods presented here can be used to make use of temporal coherence in the scope of existing visibility algorithms, that utilize a spatial hierarchy. Examples of these are algorithms based on hierarchical occlusion maps [Zhan97b], coverage masks [Gree96], shadow frusta [Huds97], and occlusion trees [Bitt98].

4.11.2 Classical approach

An elementary step of the hierarchical visibility culling is the *node visibility test*, i.e., visibility classification of a single node of the hierarchy using certain occlusion map. Given a viewpoint and a viewing direction the visibility algorithm classifies visibility of the node as *completely visible*, *partially visible*, or *invisible*. Further in this chapter we assume that the algorithms from Sections 4.9 and 4.10.2 are used to resolve the node visibility test.

The classical hierarchical visibility culling proceeds as follows: Starting from the root node of the hierarchy, the view frustum culling is applied on the current node [Rohl94, Assa00, Moll02]. If the node is outside the view frustum, it is classified invisible. Otherwise, the node visibility test is performed. If the node is found visible all its descendants are visible. Similarly, if the node is invisible all its children are invisible. Descendants of nodes classified as partially visible are tested further to refine their visibility (see Figure 3.6). When visibility of all leaves is known, objects from fully visible and partially visible leaves can be gathered and rendered using a low level exact visibility solver such as z-buffer. A simple improvement can be used to avoid visibility tests of hierarchy nodes that contain only few objects and so the estimated cost of rendering the objects is lower than the cost of the visibility determination. In such a case the node can be simply classified as visible.

4.11.3 Modifications overview

In order to give an overview of the proposed modifications we first show how they are exploited in the scope of the hierarchical visibility algorithm (see Figure 4.4). The *hierarchy updating* test is applied first. This test eventually decides to skip all the remaining steps and to continue determining visibility of descendants of the current node. The *view frustum* culling can report the node as invisible if it is outside the view frustum. Otherwise, the *visibility propagation* is applied that can succeed classifying the node as visible or invisible. The *conservative hierarchy updating* classifies some nodes as visible with certain probability. If all previous steps failed in determining node's visibility, the node visibility test is applied. Note that the steps are applied in order of increasing computational cost, which reflects the general idea of culling: use a more complicated tests only when the simple test fails to find a solution.

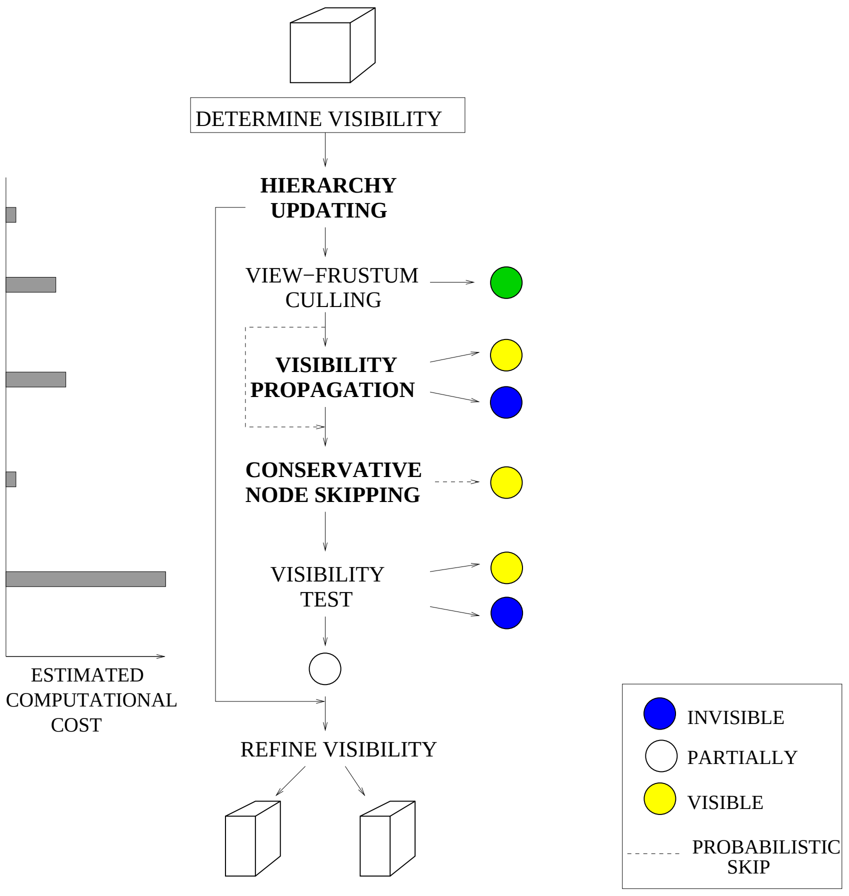


Figure 4.4: Series of steps determining visibility of a node of the hierarchy. The novel methods are highlighted.

4.11.4 Hierarchy updating

The hierarchical visibility algorithm can be seen as a traversal of the hierarchy, that is terminated either at leaves or nodes classified either as visible or invisible. Let us call such nodes the *termination nodes* and nodes that have been classified partially visible the *opened nodes*. Denote sets of termination and opened nodes in the i-th frame $\mathcal{T}_i$ and $\mathcal{O}_i$, respectively. In the classical approach $\mathcal{T}_i \cup \mathcal{O}_i = \mathcal{V}_i$, where $\mathcal{V}_i$ is the set of all nodes visited in the i-th rendering frame.

Imagine the viewpoint is fixed. Visibility of all nodes of the hierarchy does not change and the sets $\mathcal{T}_i$, $\mathcal{O}_i$, and $\mathcal{V}_i$ are fixed as well. Nevertheless, the classical algorithm repeatedly tests visibility of all nodes $\mathcal{V}_i$. The hierarchy updating is a modification that aims to eliminate the repeated visibility tests of the set of opened nodes from the previous frame. It skips all nodes of $\mathcal{O}_{i-1}$ and applies node visibility tests only on nodes of $\mathcal{T}_{i-1}$. In order to propagate eventual changes in visibility up into the hierarchy the visibility states determined at the termination nodes are pulled up according to the following rule: The visibility state of the node is updated as visible or invisible, if all its children have been classified as visible or invisible, respectively. Otherwise, it remains partially visible and thus opened. The pseudo-code of the hierarchical visibility algorithm with hierarchy updating is outlined in Figure 4.5. Note that the set of termination nodes is not maintained explicitly. Instead, each node contains its previous visibility classification. The *frame variable* is associated with each node that is used to identify nodes below the current termination nodes.

```
Algorithm HierarchicalVisibility( NODE )
begin
  if NODE is leaf or NODE.visibility $\neq$ PARTIALLY
     (* termination nodes *)
     or NODE.frame $<$ frame-1 then
  begin
     NODE.visibility $\leftarrow$ TestVisibility( NODE );
     NODE.frame $\leftarrow$ frame;
  end
  case NODE.visibility of
     VISIBLE   : Render subtree of NODE;
     PARTIALLY :
       if NODE is leaf then Render NODE;
       else
         for all children $\mathcal{C}$ of NODE do
           HierarchicalVisibility( $\mathcal{C}$);
       (* pull-up *)
       if visibilty of all children equals $\mathcal{V}$ then
         begin
         NODE.visibility $\leftarrow$ $\mathcal{V}$;
         NODE.frame $\leftarrow$ frame;
         end
     INVISIBLE : (* terminate the DFS *)
   end
 end
```

Figure 4.5: Pseudo-code of the hierarchical visibility culling with hierarchy updating.

The hierarchy updating provides the same visibility classification as the classical approach. The behavior of the modified hierarchical visibility algorithm is illustrated in Figure 4.6. Note that if the pull up did not take place the algorithm could end up with the termination nodes being all leaves of the hierarchy. Hence, it would loose advantages of the hierarchical algorithm. For kD-trees $|\mathcal{O}_i| = |\mathcal{T}_i| - 1$ since the set of visited nodes is a binary subtree of the kD-tree. Thus the hierarchy updating can save up to a half of the visibility tests that would be applied on the interior nodes of the hierarchy.

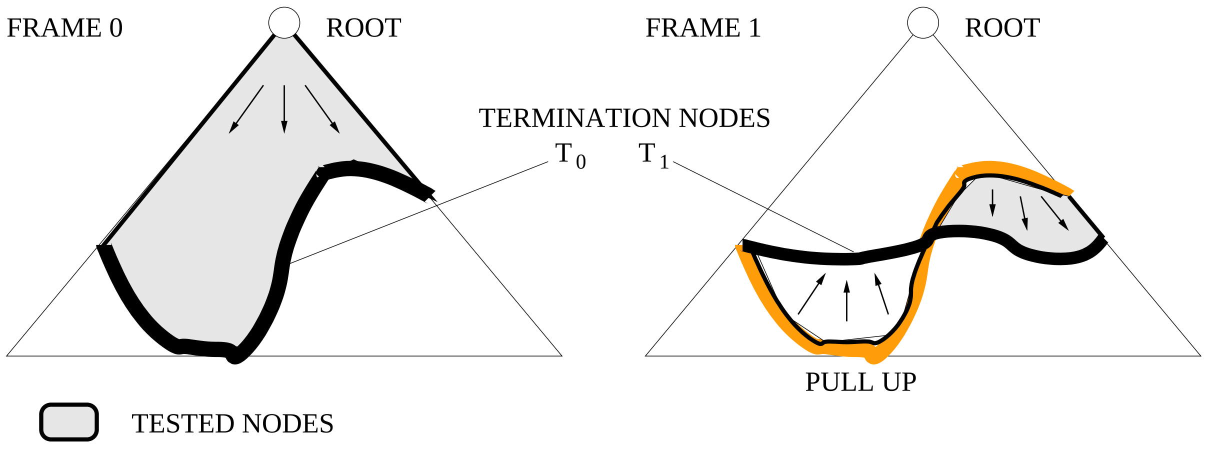


Figure 4.6: Illustration of the hierarchy updating. Initially the algorithm proceeds starting at the root of the hierarchy (left). In the second frame the opened nodes $\mathcal{O}_0$ are skipped and the visibility tests are applied on the termination nodes $\mathcal{T}_0$ (and eventually below). Visibility changes are propagated up to the hierarchy and the new set of termination nodes $\mathcal{T}_1$ is established.

4.11.5 Conservative hierarchy updating

The hierarchy updating method ensures that on each path to a leaf node of the hierarchy at least one node is tested for visibility. We can further reduce the expected number of node visibility tests at the cost of the conservative behavior of the modified algorithm. The conservative hierarchy updating produces a superset of visible nodes determined by the hierarchy updating alone.

Due to the complexity of the occlusion volume it is difficult to predict changes in visibility unless a specialized visibility algorithm is involved [Coor96b]. To keep the conservative behavior of the algorithm we cannot classify a node as invisible without testing its visibility. Nevertheless, assuming visibility does not change significantly over successive frames, visibility states of visible and partially visible nodes do not have to be updated in each frame.

We propose a simple method for conservative visibility updates that uses a probabilistic sampling scheme. Visibility of a termination node that was classified visible or partially visible in the last frame is updated with probability $1 - p_{skip}$. With probability p_{skip} the node visibility test is skipped and the node is classified as visible. This method reduces the number of visibility tests applied on visible nodes of the hierarchy, but it does not immediately capture all changes in visibility. In such cases more nodes are classified as visible and consequently more objects are rendered compared with the nonconservative hierarchy updating. The results presented in Section 4.12 show that for the tested scenes and corresponding walkthrough paths we could determine such a p_{skip} that the total frame time was minimized.

4.11.6 Visibility propagation

The hierarchical visibility culling already makes use of spatial coherence by utilizing a spatial hierarchy (kD-tree). However, we can further increase the amount of coherence exploited by reusing visibility information computed for neighboring regions.

Suppose that the nodes of the spatial hierarchy are processed in the front-to-back order with respect to the viewpoint. Using kD-tree this ordering is determined easily [Fuch80]. First, the visibility propagation tries to determine visibility of the currently processed node by combining visibility classifications of its relevant neighbors. If the combination fails, it reverts to the node visibility test.

Let us denote the cell corresponding to node N as B_N. The visibility of N can be determined combining visibility of its front-facing faces $\mathcal{F}_{B_N}$ of B_N ($|\mathcal{F}_{B_N}| \leq 3$). Visibility of a face $F \in \mathcal{F}_{B_N}$ can be determined by combining visibility of appropriate *neighbor nodes*. If all faces of $\mathcal{F}_{B_N}$ are invisible the node N is invisible. Similarly, if all faces of $\mathcal{F}_{B_N}$ are visible and there is no occluder intersecting B_N, N can be classified as completely visible. Otherwise, the visibility propagation fails and the usual node visibility test must be applied. An example of a node that can be classified as invisible is depicted in Figure 4.7.

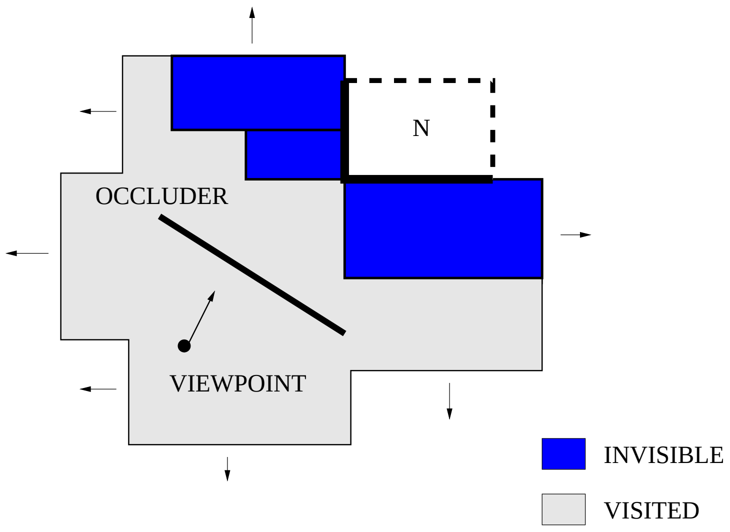


Figure 4.7: Node N can be classified invisible since all its appropriate neighbors are invisible.

A neighbor node of N on a face F is a node U of the kD-tree with B_U laying in the opposite halfspace (induced by F) than B_N and having non-empty intersection with F. Instead of keeping a list of neighbor nodes for each face we have used neighbor links (ropes) for kD-trees [Havr98a] that have low memory requirements and allow hierarchical visibility propagation.

Within each face F we associate a link to a neighbor node U that corresponds to a smallest cell containing the face completely ($F \cap B_U = F$). When determining visibility of a face F there are three possible cases:

1. the link points to a node that is visible/invisible,
2. the link points to a node that is partially visible,
3. the link points to a node that has not been visited in the current frame.

The first case is trivial; the visibility of the face can be set immediately. In the second case we perform a constrained depth first search and combine visibility of reached nodes. The search is constrained to nodes having non-empty intersection with the face F and terminates at the termination nodes $\mathcal{T}_i$. This process is illustrated in Figure 4.8. The visibility combination is performed using the same rule as in the pull up pass of the hierarchy updating (Table 3.1). We can terminate the search whenever the combination results in partial visibility.

The third case is solved by a lazy propagation of visibility classification as follows: If the link is pointing to a node that has not been visited in the current frame, there must be some termination node on the path to the root. This path is followed until the termination node is reached (see Figure 4.9). Note that if the visibility states were propagated into subtrees of the termination nodes, the third case would never occur.

The visibility propagation does not always succeed to determine visibility of the processed node. In such a case it introduces an additional overhead into the visibility determination. However, we can use information obtained in the previous frame to guide the algorithm in the current frame. Firstly, we can avoid visibility propagation on nodes that we expect to remain partially visible and thus the

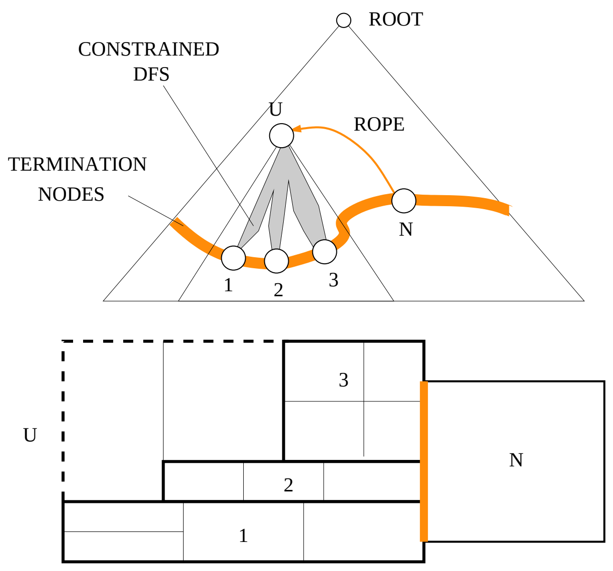


Figure 4.8: The hierarchical visibility propagation using neighbor links (ropes).

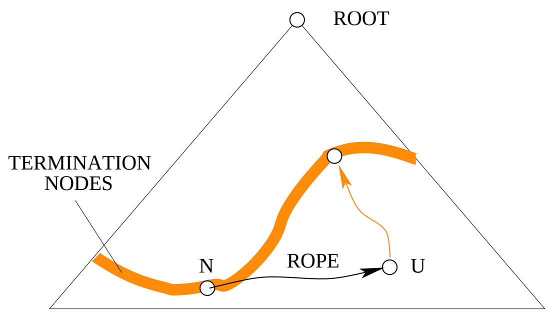


Figure 4.9: Lazy propagation of the visibility classification.

visibility propagation would probably fail. To achieve this, visibility propagation is applied only on nodes that have not been classified as partially visible in the previous frame. Secondly, if for a given node the visibility propagation succeeded in the previous frame, it is applied in the current frame as well. Otherwise, it is applied with a user specified probability $p_{vp} < 1$.

4.12 Results

In this section we evaluate the behavior of the visibility culling algorithm using occlusion trees. We compare the proposed algorithms with the hierarchical view frustum culling [Clar76, Rohl94, Assa00]. Further in this section we evaluate the asset of the methods making use of temporal and spatial coherence.

4.12.1 Visibility culling with occlusion trees

We evaluated the efficiency of the proposed algorithms on walkthroughs of three indoor scenes. The first scene (soda-5) is a model of the interior of the fifth floor of the Soda-Hall [1] (see Figure 4.17). The second scene (big-7) is a computer generated maze (see Figures 4.15-(a), 4.16). The third scene is a represents the whole building of the Soda-Hall (see Figure 4.15-(b)).

[1]http://graphics.lcs.mit.edu/˜becca/research/SodaHall

As a reference algorithm we used the hierarchical view frustum culling algorithm. To increase scene complexity 100 virtual plants were spread randomly in the scene. Each plant consisted of 644 polygons. The *minimum cost* of a node to be tested for visibility was set to 50 (the cost of a node expresses the number of polygons associated with the node).

scene	method	occluders [—]	rendered polygons [—]	overhead [ms]	frame time [ms]	speedup [—]
	F	—	18192	—	276.5	1.00
	FME	16	7390	5.9	139.0	1.99
soda–5	FME	24	6537	7.9	116.0	2.38
	FME	32	5941	9.8	109.8	2.52
total # of polygons = 66085	FSE	16	6512	12.7	120.5	2.29
# of occluders = 1685	FSE	24	5569	16.0	110.5	2.50
# of kD–tree nodes = 2527	FSE	32	4725	19.1	100.1	2.76
	FMC	16	7988	5.4	135.9	2.03
	FMC	24	7362	7.3	129.3	2.14
	F	—	12587	—	182.0	1.00
	FME	16	3641	8.8	71.5	2.55
big–7	FME	24	2286	10.2	54.6	3.33
	FME	32	1818	11.6	48.5	3.75
total # of polygons = 66876	FSE	24	2214	19.9	63.4	2.87
# of occluders = 2476	FMC	16	3640	8.4	74.3	2.44
# of kD–tree nodes = 3439	FMC	24	2263	9.7	51.4	3.54
soda						
total # of polygons = 73529	F	—	15297	—	315.2	1.00
# of occluders = 9129	FME	24	4744	10.1	119.7	2.63
# of kD–tree nodes = 10475	FSE	24	4388	23.7	128.0	2.46

F	– view frustum culling
S	– OT + exact visibility algorithm
M	– OT + conservative visibility algorithm
E	– exact occluder-fragment/parallelepiped intersection test
C	– conservative occluder-fragment/parallelepiped intersection test

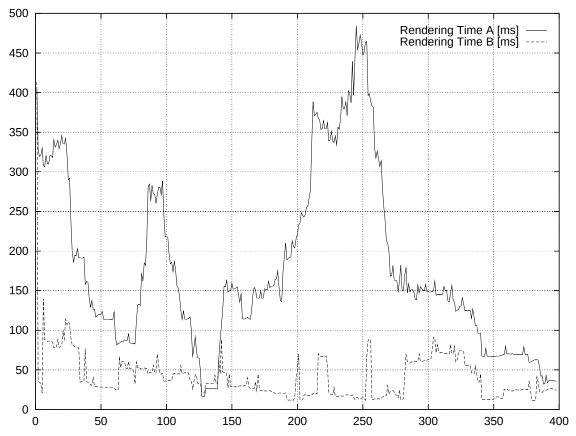
Table 4.1: Results of the hierarchical visibility culling. The table shows the average number of polygons rendered, the average frame time and the speedup over the view frustum culling for different scenes and methods of the visibility culling. Measured on SGI O^2 with 128MB RAM.

The results are summarized in Table 4.1. Each line in the table corresponds to values averaged per one frame of the walkthrough. The *frame time* field is the average frame time. The *overhead* field depicts the additional overhead of the visibility culling algorithms. The overhead includes the dynamic occluder selection, building the occlusion tree, and the hierarchical visibility culling. The *speedup* is a ratio of the frame time with visibility culling and the frame time using only view frustum culling. The average number of polygons rendered in one frame is shown in the *rendered polygons* field. The last two fields in the table are user specified constants: the *occluders* field expresses the number of occluders used to build the occlusion tree and the *method* field identifies the type of the algorithm used.

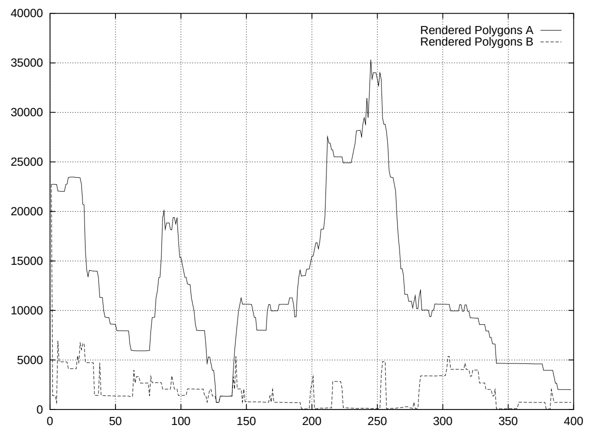
The achieved speedup varies between 1.75 and 3.75. The average speedup was not linearly proportional to the number of occluders used for the visibility culling: it increased until a sufficient number of occluders were used that caused significant occlusion.

An important property of the algorithm is that the overhead of visibility culling increased sub-linearly in dependence on the number of selected occluders. This behavior can be explained by the following two observations: Firstly, if the occlusion tree contains occluders that occlude a large portion of the view, it is of high probability that another inserted occluder is found invisible. Consequently, the effective occluders in the occlusion tree correspond only to visible occluders since the invisible occluders are eliminated. Secondly, the occlusion tree provides logarithmic search ability for the node visibility test. The traversal of tree is restricted only to nodes that are relevant for the tested region.

Table 4.1 indicates that the best results were achieved by the FME method, i.e. the conservative visibility algorithm with the fragment/cell intersection test applied in leaves of occlusion tree. In some cases the FSE method provided greater speedup than FME (see rows 4 and 7 of Table 4.1). This happens when the time saved for faster visibility tests using the FME method is less significant than the additional time spent on rendering invisible objects conservatively classified as visible. Plots of frame times and numbers of rendered polygons measured during a walkthrough of the *big-7* scene are shown in Figures 4.10-a and 4.10-b.



(a)



(b)

Figure 4.10: Evaluation of a sequence of 400 frames during walkthrough of big-7 scene. (a) Rendering times. Curve A corresponds to view frustum culling; curve B includes hierarchical visibility culling using 32 occluders (FME). (b) The amount of rendered polygons. Curve A corresponds to view frustum culling; curve B includes hierarchical visibility culling using 32 occluders (FME).

4.12.2 Temporal and spatial coherence

We evaluated the proposed algorithms on two test scenes. The first scene (scene I) is a model of the fifth floor of the Soda-Hall, the second scene (scene II) is a building interior with a precomputed lighting (see Figure 4.18). The measurements were conducted using SGI O^2 with 128MB memory.

The following methods were evaluated:

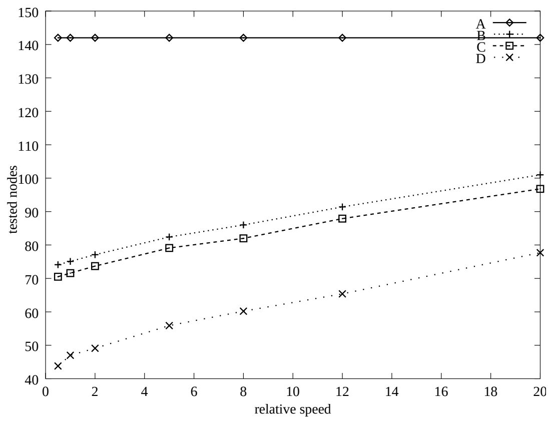
A — the classical approach,

B — hierarchy updating applied,

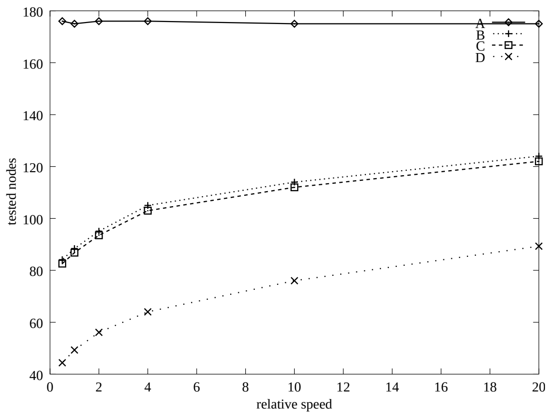
C — hierarchy updating + visibility propagation with probability $p_{vp} = 0.5$,

D — as **C** + conservative hierarchy updating with probability $p_{skip} = 0.5$.

The constructed kD-tree consisted of 1187 nodes for scene I, and of 1605 nodes for scene II. For each position of the viewpoint 16 occluders were identified and used to build the occlusion tree during walkthrough of scene I. For the scene II, we used 32 occluders, since the scene contained smaller patches resulting from the radiosity precomputed lighting. In both scenes a predetermined walkthrough path was followed for each measurement (see Figures 4.17 and 4.18 for scene snapshots). If not stated differently, all presented values are averaged per one frame of the walkthrough.

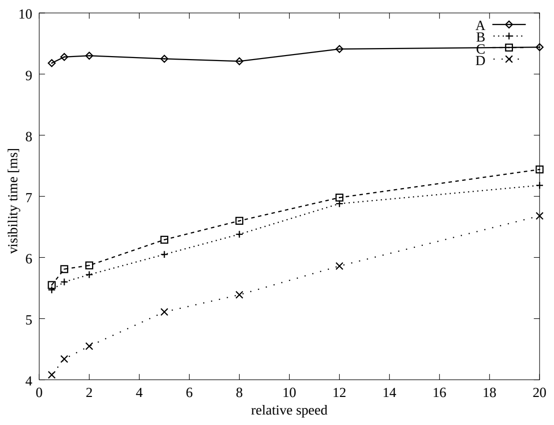


(a)

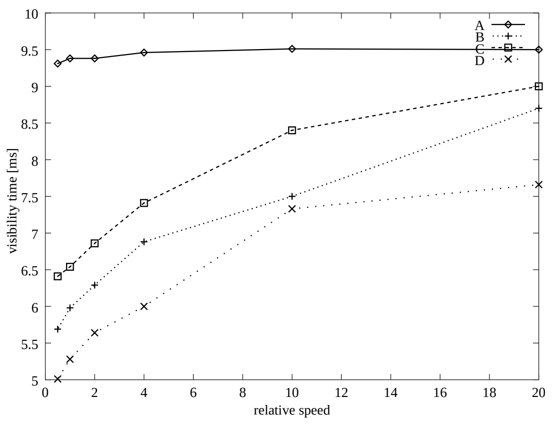


(b)

Figure 4.11: Dependence of the number of node visibility tests on the relative speed of the walk for scene I (a) and scene II (b).



(a)



(b)

Figure 4.12: Average time spent by the hierarchical visibility algorithm for scene I (a) and scene II (b).

The first six plots illustrate the dependence of the algorithms on the relative speed of the walk (Figures 4.11, 4.12, and 4.13). A unit relative speed roughly corresponds to the usual walking speed. We have measured the number of node visibility tests, the time spent by the hierarchical visibility determination, and the total frame time.

All evaluated methods exhibit a very slow growth of the number of necessary node visibility tests. For a walk of relative speed 1.0 the following savings in average number of node visibility tests were achieved (compared to **A**):

- **scene I** — method **B** – 47%, method **C** – 50%, and method **D** – 67%.
- **scene II** — method **B** – 49%, method **C** – 51%, and method **D** – 72%.

The hierarchy updating (method **B**) saves almost half of the node visibility tests as expected. We have observed that the visibility propagation (method **C**) succeeds in determining visibility of only few nodes that usually correspond to rather large regions. The **D** method significantly decreases the number of node visibility tests. This is paid by a higher number of nodes classified as visible or partially visible (details follow further in the text).

Figures 4.12-a,b show that the time spent by the hierarchical visibility culling was roughly proportional to the number of node visibility tests. Nevertheless, we can observe that the time spent by the

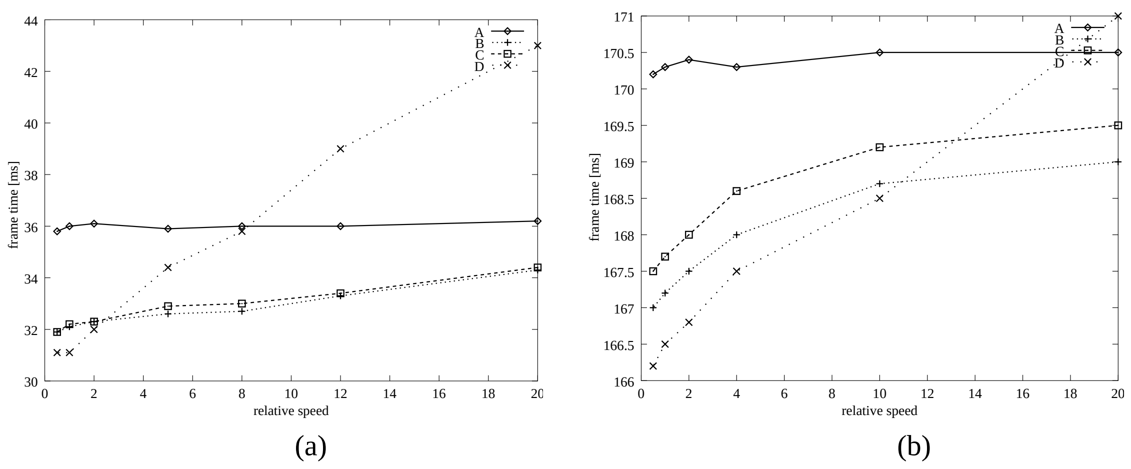


Figure 4.13: Average frame time in dependence on the relative speed of the walk for scene I (a) and scene II (b).

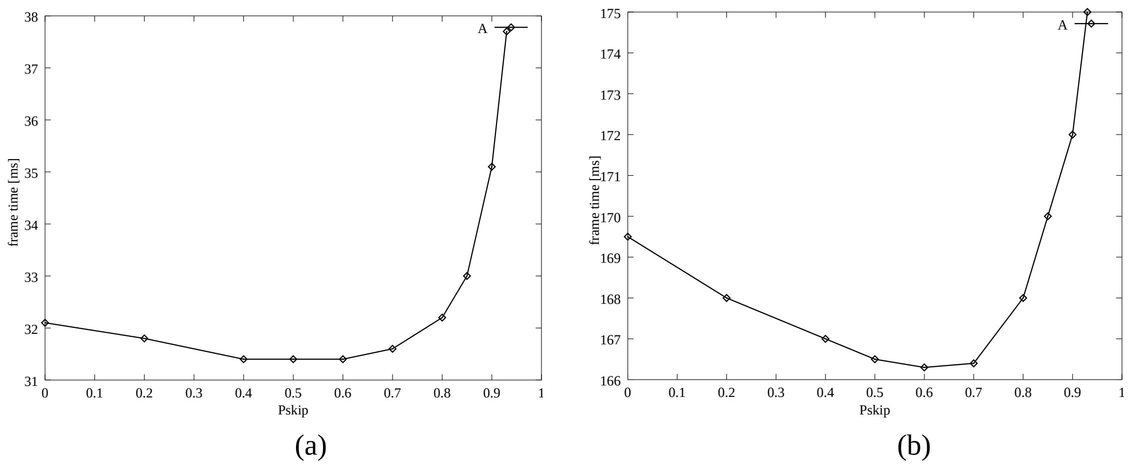


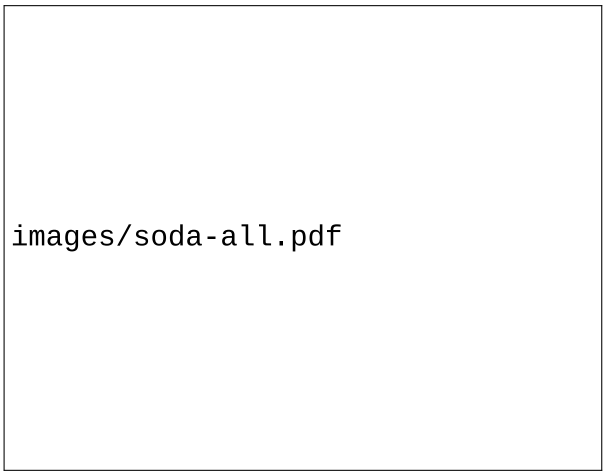
Figure 4.14: Dependence of the average frame time on the probability p_{skip} using the conservative hierarchy updating for scene I (a) and scene II (b).

visibility propagation (method **C**) is not recovered by the savings in number of node visibility tests. In particular, this follows from the fact that the node visibility test using the occlusion tree is almost as fast as the visibility propagation.

In Figures 4.13-a,b we can observe the conservative behavior of method **D**. When the viewpoint moves slowly, the method achieves better frame times than the other ones. As the relative speed of the walk increases the visibility states of many nodes change quickly. Hence reusing some previously visible nodes leads to a larger set of nodes to render and the frame time is increased.

Finally, we measured the behavior of the conservative hierarchy updating algorithm in dependence on the probability p_{skip} (Figures 4.14-a,b). We can observe local minima in the average frame time at $p_{skip} = 0.5$ for scene I and $p_{skip} = 0.6$ for scene II. For probabilities greater than this minimum savings in visibility classification do not recover the time necessary for rendering otherwise invisible objects.

It is worth mentioning that the aim of the measurements presented in this section was not to evaluate the visibility algorithm itself, but rather to document the impact of the proposed methods. If the visibility algorithm was more demanding, the proposed methods would decrease the total frame time more significantly.



(a) **(b)**

Figure 4.15: (a) A bird's eye view of the *big-7* scene. (b) The *soda* scene.

4.13 Summary

In this section we described an algorithm for real-time visibility culling based on the concept proposed in Chapter 3. The algorithm uses an occlusion tree constructed using selected occluders nearby the viewpoint. The occlusion tree is then used to classify visibility of the spatial hierarchy. We have described three algorithms for visibility tests. The first two algorithms determine an exact visibility of a polygon and a polyhedron with respect to the selected occluders. The third algorithm is a fast conservative visibility test suitable for regions of more general shapes. The results indicate that the conservative algorithm performs superior compared to the exact one for typical walkthroughs. The proposed method was evaluated on walkthroughs of complex architectural scenes. It was shown that the occlusion trees can be used to quickly identify invisible parts of the scene that need to be rendered. For the tested scenes the savings in rendering time provided speedup between 1.8 and 3.8.

Further, we have presented a series of modifications of the classical hierarchical visibility culling for rendering acceleration. The hierarchy updating proved to perform well in practice as it saves almost half of the visibility tests that would have to be applied using the classical approach. The savings would be less remarkable for hierarchies with higher branching factors, but empirical results indicate that kD-trees with arbitrary positioned partitioning planes are more efficient for visibility computation than octrees or bounding volume hierarchies [Havr00a]. We have shown that the conservative hierarchy updating can improve the overall frame time for certain settings. The visibility propagation saves only few visibility tests. This documents that the spatial coherence is already exploited well in the classical approach.

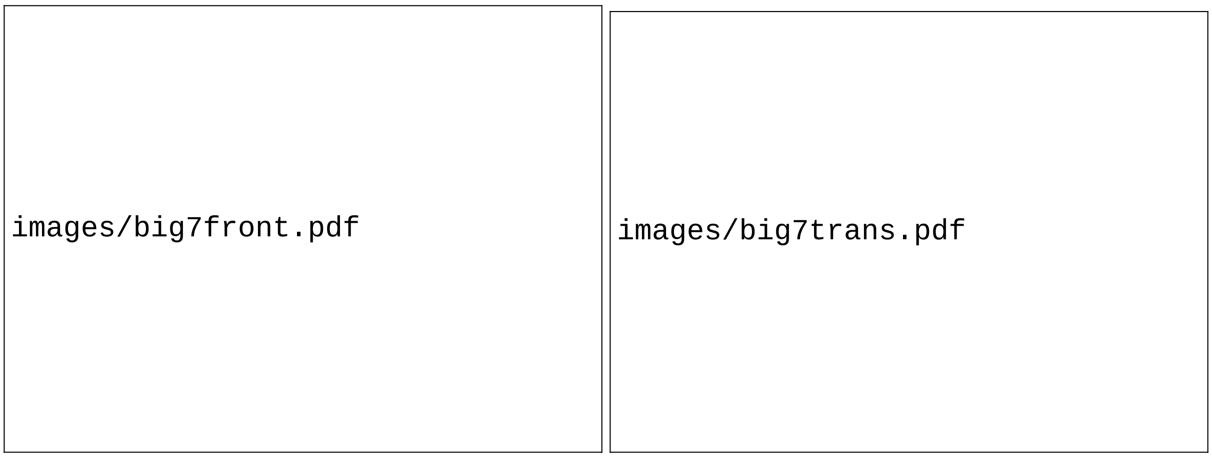


Figure 4.16: (a) A view in the big-7 scene. (b) The same view rendered in wireframe mode. Note the high depth-complexity of the scene.

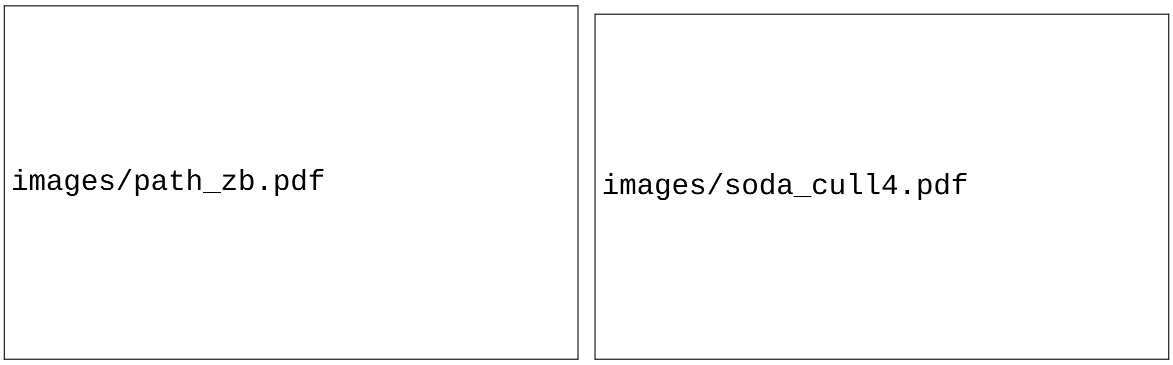


Figure 4.17: (a) The path used for a walk through the model of the scene I. For relative speed of the walk equal to 1.0 the walk consists of 980 steps. (b) An example of the hierarchical visibility culling. The green regions are outside of the view frustum. The few yellow regions in the viewing direction are completely visible. Invisible regions are shown in dark blue. The light blue regions were found invisible by the visibility propagation algorithm. Partially visible regions are shown transparent.

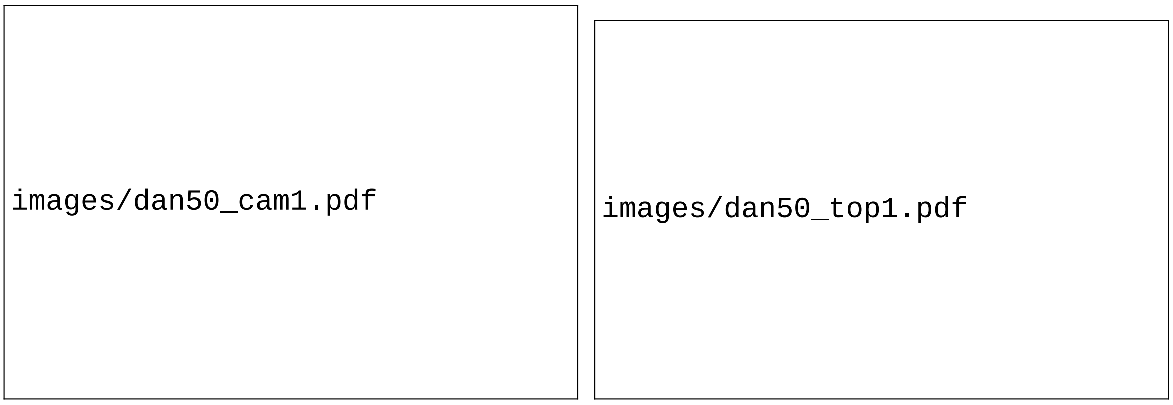


Figure 4.18: (a) A camera view on the test path through scene II. (b) Top view showing the part of the scene classified invisible (gray).

Chapter 5

Construction of visibility maps

This chapter presents an algorithm that constructs a visibility map for a given view of a 3D polygonal scene. The view is represented by an occlusion tree and the visibility map is obtained by postprocessing of the occlusion tree. The scene is organized in a kD-tree that is used to perform an approximate occlusion sweep. The occlusion sweep is interleaved with hierarchical visibility tests. We evaluate the implementation of the algorithm using several test scenes and outline its application to discontinuity meshing.

5.1 Problem statement

The computation of visibility maps is related to the problem of visible surface determination. Visible surface algorithms aim to determine a collection of visible surfaces for a given view of the scene. A visibility map contains more information — it captures also the topology of the view. The visibility map is a planar graph representing the view of a polygonal scene where vertices, edges, and faces are associated with vertices, edges, and polygons of the scene [Stew98a, Gras99]. Each element of the visibility map holds an information about its adjacent elements. See Figure 5.1 for an example of a visibility map.

(a) (b) (c)

Figure 5.1: (a) A view of a scene with 556 polygons. (b) The view with the corresponding visibility map. Color coding of the visibility map vertices corresponds to the number of adjacent edges. Edges are colored depending on their semantic classification. (c) A sweep through the visibility map allows to identify edges corresponding to “corners” (black) or edges forming “shadow” boundaries (red).

Visibility maps can be used to construct an approximate discontinuity mesh [Stew98a, Heck92] in the context of the radiosity global illumination algorithm. Another application is an efficient antialiasing for high resolution rendering [Gras99]. Visibility maps can also guide occluder preprocessing for real-time visibility culling [Coor97, Huds97, Bitt98]. Furthermore, a visibility map provides cues that can help a user to understand the view of the scene [Gras99].

The proposed method handles a view of 360^o that spans the whole spatial angle since it does not rely on a single projection plane. The algorithm is exact in the sense that it does not use a discrete representation of the view. The view is represented hierarchically, which allows its efficient construction and postprocessing. The algorithm uses an approximate occlusion sweep interleaved with hierarchical visibility tests. This concept results in an output-sensitive behavior of the algorithm in practice without the necessity of a specialized data structure for obtaining the exact priority order of scene polygons.

5.2 Related work

Computation of visibility maps is related to the visible surface determination [Gran92]. Some traditional visible surface algorithms such as the algorithm of Watkins, Weiler-Atherton, Warnock [Fole90] provide a continuous output that can be used for the construction of visibility maps. Unfortunately these methods do not scale very well to large scenes with dense occlusion.

Visible surface algorithms are nowadays dominated by the z-buffer algorithm that is often implemented in hardware. Nevertheless it is rather difficult to reconstruct a visibility map from the discretized image obtained by the z-buffer algorithm [Stew98a]. Another drawback of the z-buffer is the lack of output sensitivity of the algorithm. Therefore many recent techniques aim to increase efficiency of the z-buffered rendering by visibility culling [Coor97, Huds97, Bitt98].

Recently, computation of visibility maps was studied by Stewart and Karkanis [Stew98a]. They propose an algorithm for the construction of approximate visibility maps using dedicated graphics hardware. They first render the scene in the *item buffer*. Then they construct a rectilinear graph that is *relaxed* to match the edges and vertices of visible scene polygons. The drawback of the algorithm is that it can fail to correctly relax all features of the visibility map. Grasset et al. [Gras99] presented a paper dealing with some theoretical operations on visibility maps and their applications in computer graphics.

5.3 Algorithm overview

The proposed algorithm consists of two main steps: Firstly, an occlusion tree is constructed for a given view of the scene. The occlusion tree is built with respect to the viewpoint and a set of polygons and it can be seen as a BSP tree representing the view. Secondly, the visibility map is constructed by postprocessing of the occlusion tree. The hierarchical structure of the occlusion tree is used for efficient lookups of adjacent elements during the construction.

The occlusion tree is constructed using an approximate occlusion sweep with respect to the given viewpoint. Scene polygons are swept in an approximate front-to-back order that is established using a kD-tree. The order is approximate in the sense that a currently processed polygon can be occluded by a constant number of unprocessed polygons. The occlusion tree is constructed incrementally by inserting the currently processed polygon. At each step the occlusion tree represents the view of the scene consisting of already processed polygons. The traversal of the kD-tree is interleaved with hierarchical visibility tests applied on its nodes. The visibility test uses the current occlusion tree to determine visibility of a region corresponding to the given node of the kD-tree. If the region is invisible the corresponding node and its whole subtree are culled.

When the occlusion tree represents the complete view it is used to construct the visibility map. Each non-empty leaf of the occlusion tree corresponds to a fragment of a visible polygon. Visibility map is constructed by inserting the visible fragments and updating adjacency links to the fragments already processed. For each fragment the occlusion tree is used to efficiently locate its neighbor fragments. All subsequent operations are restricted to the located neighbors. When the construction of the visibility map is finished a simple sweep through the map can classify its edges and vertices into several categories. Based on this classification the visibility map can be pruned depending on the particular application.

The rest of this chapter is organized as follows: Section 5.4 discusses the structure of the occlusion tree for this particular application. Section 5.5 briefly discusses the use of the occlusion tree for hierarchical visibility tests. In Section 5.6 we discuss the construction of visibility map by postprocessing of the occlusion tree. Section 5.7 contains an evaluation of the implementation of the proposed method. Finally, Section 5.8 summarizes the chapter.

5.4 Occlusion tree

We briefly review the structure of the occlusion tree for the from-point visibility in 3D and the algorithm of its construction.

5.4.1 Structure of the occlusion tree

The occlusion tree is a BSP tree where each node represents a set of rays $\mathcal{Q}_N$ emanating from the viewpoint. The root of the tree represents the whole view. Each interior node N is associated with a plane h_N passing through the viewpoint. The right child of N represents rays $\mathcal{Q}_N \cap h_N^+$, the left child $\mathcal{Q}_N \cap h_N^-$, where h_N^+ and h_N^- are halfspaces induced by h_N.

Leaves of the tree are classified in or out. If N is an out-leaf, $\mathcal{Q}_N$ represents unoccluded rays. If N is an in-leaf, it is associated with a closest scene polygon O that is intersected by the corresponding set of rays $\mathcal{Q}_N$. Further N stores a fragment P_N that is an intersection of the polygon O and $\mathcal{Q}_N$.

It is easier to think about the occlusion tree in a restricted projection to a particular 2D viewport. The root of the tree corresponds to the whole viewport. Each interior node is associated with a line subdividing the current polygonal region in two parts. Leaves of the tree represent either empty region of the viewport or a fragment of a visible polygon. An elementary occlusion tree e-OT(P) constructed for a single polygon P contains interior nodes corresponding to the planes defined by edges of P and the viewpoint (see Figure 3.3).

5.4.2 Construction of the occlusion tree

The occlusion tree is constructed incrementally by inserting scene polygons in the order given by the approximate occlusion sweep. The algorithm inserting a polygon O in the tree maintains two variables — the current node N_c and the current polygon fragment P_c. Initially N_c is set to the root of the tree and P_c equals to O.

The insertion of a polygon in the tree proceeds as follows: If N_c is an interior node we determine the position of P_c and the plane h_{N_c} associated with N_c. If P_c lies in the positive halfspace induced by h_{N_c}, the algorithm continues in the right subtree. Similarly, if P_c lies in the negative halfspace induced by h_{N_c}, the algorithm continues in the left subtree. If P_c intersects both halfspaces, it is split by h_{N_c} into two parts P_c^+ and P_c^- and the algorithm proceeds in both subtrees of N_c with appropriate fragments of P_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 P_c is visible and N_c is replaced by e-OT(P_c). If N_c is an in-leaf, the mutual position of P_c and fragment P_{N_c} associated with N_c is determined. If P_c is behind P_{N_c} it is invisible and no modification to the tree is necessary. Otherwise, N_c is replaced by e-OT(P_c) and the old fragment P_{N_c} is inserted in the new subtree e-OT(P_c) using the just described polygon insertion algorithm. The nodes corresponding to the edges of the old fragment are kept in the tree. Consequently the tree is slightly larger than in the case of a strict front-to-back order of input polygons. An example of an occlusion tree for three polygons is depicted in Figure 3.4.

5.5 Hierarchical visibility tests

To increase efficiency of the algorithm the traversal of the scene kD-tree is interleaved with visibility tests applied on its nodes. If the test determines that the node is invisible, the corresponding subtree and all polygons it contains are culled.

We use a conservative visibility test described in Section 4.10. The test uses a constrained depth first search on the occlusion tree using a bounding box corresponding to the given kD-tree node. Starting at the root of the occlusion tree the position of the box and the plane associated with the root. If the box intersects only positive halfspace defined by the plane, the algorithm recursively continues in the right subtree. Similarly, if the box intersects only negative halfspace, the algorithm proceeds in the left subtree. If the box spans both halfspaces, both subtrees are processed recursively. Reaching an out-leaf the algorithm identifies the node as visible and terminates. Reaching an in-leaf the position of the box and the fragment associated with the leaf is determined. If the box lies at least partially in front of the fragment, it is visible and the algorithm terminates. If the search does not find any visible part of the box, the corresponding node can be culled.

5.6 Construction of the visibility map

The visibility map is constructed by postprocessing of the occlusion tree. The advantage of this approach is that only visible polygons are considered for the construction of the map. Each in-leaf of the tree corresponds to a visible fragment and each such fragment is incrementally inserted in the visibility map.

The visibility map consists of the following elements:

- *vm-vertex* — a vm-vertex corresponds to a vertex of a scene polygon or an apparent vertex that results from an intersection of edges in the view.
- *vm-polygon* — a vm-polygon corresponds to a fragment associated with a leaf of the occlusion tree.
- *vm-edge* — a vm-edge corresponds to an edge or a part of an edge of a scene polygon. A *silhouette* vm-edge is associated with a single vm-polygon, other vm-edges are associated with two polygons, each on one side of the edge.

The elements of the visibility map contain the following connectivity information:

- *vm-vertex*
  list of adjacent vm-edges,
  list of adjacent vm-polygons.
- *vm-edge*
  the two vm-vertices it connects,
  the two vm-polygons that share this vm-edge (one is possibly empty).
- *vm-polygon*
  list of vm-edges that bound the polygon,
  list of adjacent vm-vertices.

There is some redundancy in the above described representation, but the redundant information provides more efficient lookups.

The visibility map is linked with the occlusion tree so that each in-leaf of the tree contains a link to the corresponding vm-polygon. In the following sections we describe how the visibility map is constructed by incrementally inserting visible fragments.

5.6.1 Neighbor location

The construction of the visibility map proceeds by incrementally inserting polygonal fragments associated with the leaves of the occlusion tree. Assume we process a fragment P_{N_c} associated with a leaf N_c. The first step of the insertion of a fragment P_{N_c} creates a new vm-polygon V_{N_c}. V_{N_c} is associated both with P_{N_c} and the corresponding in-leaf N_c. Then the algorithm locates all already processed *neighbor* vm-polygons that share a boundary with P_{N_c}. The algorithm performs a constrained search on the occlusion tree pruning subtrees that have no intersection with P_{N_c} (see Figure 5.2).

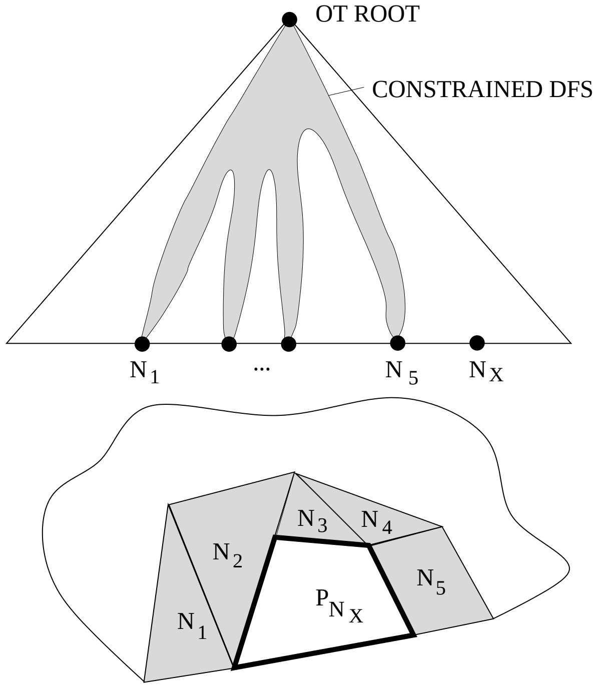


Figure 5.2: Neighbor location is performed using constrained depth first search on the occlusion tree. The figure depicts a fragment, its five neighbors and symbolic illustration of the search through the occlusion tree.

For each vertex of P_{N_c} we check if it was already inserted in the map by comparing it with vm-vertices associated with the neighbor vm-polygons. If the corresponding vm-vertex is not found, we insert a new vm-vertex in the map. The either found or newly created vm-vertex is then is associated with V_{N_c}.

5.6.2 Inserting fragment edges

The crucial step of the visibility map construction is the insertion of edges of the currently processed fragment P_{N_c}. For each edge E_i the following steps are performed:

1. Locate all vm-edges of the neighbor vm-polygons that intersect the E_i. Denote the set of such edges $\mathcal{E}$.
2. Create links from these edges of $\mathcal{E}$ that completely overlap E_i to the new vm-polygon V_{N_c}.
3. Create new vm-edges for the part of E_i that is not covered by any edge from $\mathcal{E}$. These edges are associated with V_{N_c} and contain an empty link to the other vm-polygon.
4. Subdivide and update edges of $\mathcal{E}$ that partially overlap E_i.

An illustration of the insertion of a new edge into the visibility map is depicted in Figure 5.3.

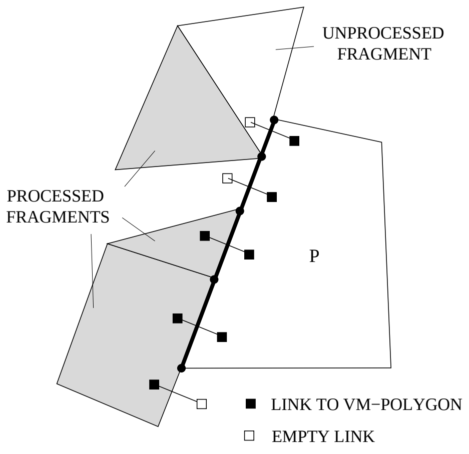


Figure 5.3: Insertion of a new edge in the visibility map. The figure depicts links corresponding to the updated or newly created vm-edges.

5.6.3 Classification of edges and vertices

The elements of the visibility map can be classified into several categories depending on the configuration of elements from their neighborhood. We used the following 5 categories (see Figure 5.4):

- *silhouette edge* — an edge that is associated with a single vm-polygon. It forms a part of the silhouette of the view.
- *shadow edge* — an edge that forms a “shadow” boundary. The two associated polygons do not share the corresponding edge in 3D, i.e. there is a depth discontinuity on the edge. The closer vm-polygon associated with the edge is the *occluder* the farther is the *occludee*.
- *corner edge* —- an edge in a corner or on the rim of an object. The edge is shared by two connected vm-polygons that form an angle greater than the predefined *crease angle*.
- *flat edge* —- the edge is shared by two connected vm-polygons that form an angle smaller than the predefined crease angle.
- *bsp edge* —- a flat edge shared by vm-polygons that result from splitting of the same scene polygon in the process of construction of the occlusion tree.

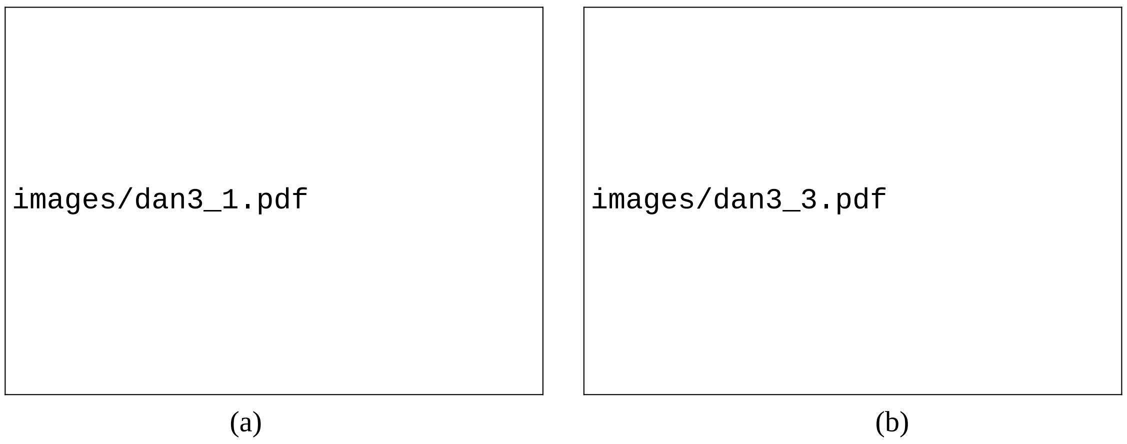


Figure 5.4: (a) The classification of the edges of the visibility map. (b) The visibility map after removal of flat and bsp edges.

The classification enables to better understand the structure of the view. Depending on the application only edges of certain classes are considered. For example in the context of discontinuity meshing the visibility maps can be constructed with respect to each vertex of a given light source. Then the shadow edges define a subset of vertex-edge (VE) discontinuities due to the light source.

5.7 Results

We have implemented the proposed algorithm in C++ and evaluated the construction of visibility maps using three types of scenes:

- *rad* — a building interior with some detailed objects and finer meshes resulting from radiosity computation. See Figures 5.1, 5.4.
- *soda* — a building interior with large walls, see Figure 5.5-a.
- *random* — random triangles, see Figure 5.5-b.

The measurements were conducted on a PC with 500MHz CPU, 256MB RAM, running Linux. For each scene we selected several viewpoints and measured the following parameters: the total time for construction of the visibility map, the time for construction of the occlusion tree only, the number of visibility map vertices, edges and polygons, and the percentage of leaves of the scene kD-tree culled by the hierarchical visibility tests. The measurements are summarized in Table 5.1.

scene	polygons [−]	kD-tree nodes [−]	view No.	total time [s]	OT time [s]	vm-vertices [−]	vm-edges [−]	vm-polygons [−]	culled kD-leaves [%]
rad	26526	8809	I	0.24	0.15	642	1229	567	99.6
			II	0.14	0.07	555	1029	474	99.5
			III	1.15	0.69	3312	6307	2906	94.0
soda	1685	4735	I	0.02	0.01	39	62	24	98.8
			II	0.04	0.03	109	188	78	98.0
			III	0.09	0.07	201	368	163	95.1
random	10000	47139	I	0.10	0.08	275	472	197	99.9
			II	1.24	0.99	2269	3758	1472	95.0
			III	4.38	2.98	12016	20718	8533	84.1

Table 5.1: Summary of the results. The table contains the scene name, the number of scene polygons, the number of kD-tree nodes, the total time for construction of the visibility map, the time for construction of the occlusion tree, the number of visibility map polygons, vertices and edges, and the percentage of leaves of the scene kD-tree culled by the hierarchical visibility tests.

The first scene contains many polygons that result from meshing due to computation of global illumination using the radiosity method. For a view with very restricted visibility (view No. I and II) the computation of the visibility map is very fast and the majority of the scene (99%) is culled by the hierarchical visibility tests. With less restricted visibility the computational time increases approximately linearly with the number of resulting vm-polygons.

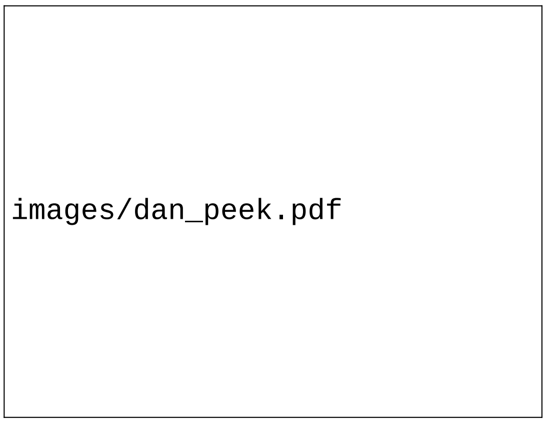
The second scene is a building interior consisting of large polygons. The resulting visibility map is much simpler than for the *rad* scene and computational times are proportionally faster. A view of the *soda* scene is depicted in Figure 5.5-a.

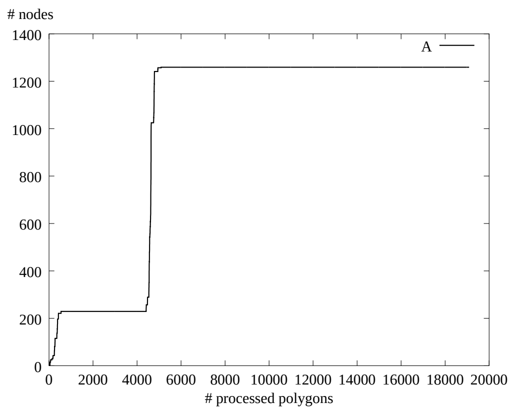
The third scene contains 10000 randomly generated triangles. Due to the lack of a regular structure the resulting visibility map is complex. The complexity of the view is also increased due to mutual triangle intersections. The first view (random-I) corresponds to a viewpoint located inside of the cluster of triangles. The triangles appear larger and block visibility of many other triangles and consequently the computation is significantly faster in comparison with the other views. The second view (random-II) is depicted in Figure 5.5-b.

(a) (b)

Figure 5.5: (a) The visibility map corresponding to the view soda-III in Table 5.1. (b) The visibility map corresponding to the view random-II in Table 5.1.

Further we studied the growth of the occlusion tree during its construction. We measured the size of the tree after processing each scene polygon using the approximate occlusion sweep. To better understand the results hierarchical visibility tests were not applied for this test. On the measured curve (Figure 5.6-b) we can identify two big steps of a sudden increase of the tree size. The first step corresponds to the insertion polygons near the viewpoint, the second to farther polygons visible through the door (see Figure 5.6-a). We can observe that once the occlusion tree contains all visible polygons its size does not increase.





(a) (b)

Figure 5.6: (a) A view of the *rad* scene with shadow vm-edges depicted. (b) The size of the occlusion depending on the number of inserted polygons. Note the two steps corresponding to a sudden increase of the tree size. The first corresponds to polygons close to the viewpoint, the second to the farther polygons visible through the door. No hierarchical visibility tests were applied.

Figure 5.7-(a) shows a visibility map from the bird's-eye view. We can see that the proposed algorithm efficiently culls invisible part of the scene during the construction of the occlusion tree. The visibility map is then built using visible polygons only.

Figure 5.7-(b) shows a subset of a discontinuity mesh computed using four visibility maps. The four maps were constructed for views centered at the vertices of the rectangular light source. Shadow vm-edges were then projected on the associated occludees. The remaining discontinuity edges could be identified by searching through vertices of the four visibility maps and matching the information about associated shadow edges [Stew98a].

(a)

(b)

Figure 5.7: (a) The computation of the visibility map from the bird's perspective. The algorithm efficiently culls invisible regions (shown in gray) and considers only the visible part of the scene for visibility map construction. Computation time: 0.3s. (b) A subset of a discontinuity mesh constructed using four visibility maps. The picture shows shadow edges of four visibility maps projected on occludees. The visibility maps correspond to views from the four vertices of the rectangular light source. Computation time: 1.2s.

5.8 Summary

We have presented a new algorithm that efficiently constructs a visibility map for a given view of the scene. The method does not rely on a single projection plane and easily handles views that span the whole spatial angle. The visibility map is constructed by a two stage algorithm: construction of a hierarchical representation of the view and its postprocessing.

The view is represented by the occlusion tree. The tree is constructed using an approximate occlusion sweep that allows efficient incremental construction without complicated data structures for exact priority orders. The occlusion sweep is interleaved with hierarchical visibility tests, which results in output-sensitive behavior of the algorithm in practice. The visibility map is constructed by simple post-processing of the occlusion tree. We presented a classification of the elements of the visibility map that helps to better understand the structure of the view. The implementation of the technique was evaluated on several non trivial scenes.

Chapter 6

From-region visibility in 2D scenes

In this chapter we present a new method for an exact and output-sensitive determination of visibility from a polygonal region in the plane. The technique is based on a hierarchical partitioning of line space maintained by the occlusion tree. It provides a comprehensive description of visibility for a set of occluders and it is suitable for computing potentially visible sets in large scenes with various visibility characteristics. The proposed algorithm is expected to exhibit output-sensitive behavior in practice, i.e. its running time is proportional to the number of visible objects. The method is suitable for large and degenerate inputs and it can be applied to spatial regions of arbitrary size, without under-sampling artifacts [Wonk00]. The algorithm implicitly solves all types of occluder fusion [Dura00, Scha00]. The technique described herein serves as a basis for a $2\frac{1}{2}$D from-region visibility algorithm that will be described in the next chapter.

6.1 Problem Statement

We restrict the discussion to 2D scenes consisting of line segments and the regions of interest (view cells) to convex polygons. Given a polygonal view cell the goal is to determine:

(a) The set of potentially visible line segments.

(b) The visible fragments of the visible line segments.

(c) Which fragments are visible from which parts of the view cell.

The above states goals are presented in order of increasing accuracy of the computed result. The proposed technique can be used to resolve all three goals and yields a comprehensive description of visibility from a given region in the plane. An illustration of the from-region visibility is depicted in Figure 6.1.

6.2 Related work

In this section we briefly discuss related 2D visibility algorithms, the algorithms for more general $2\frac{1}{2}$D and 3D scenes will be discussed in Chapters 7 and 8.

2D visibility was studied intensively in computational geometry [Asan00]. The visibility graph [Welz85] is a well known structure for capturing visibility in 2D scenes. Vegter introduced the visibility diagram [Vegt90] that contains more information than the visibility graph. The visibility complex is a similar structure introduced by Pochiolla and Vegter [Pocc93]. The visibility complex for polygonal scenes was studied by Riviere [Rivi97a] and it has been applied to solve various visibility problems in the plane [Rivi95, Rivi97b, Orti96]. Hinkenjann and Müller [Hink96] proposed *hierarchical blocker*

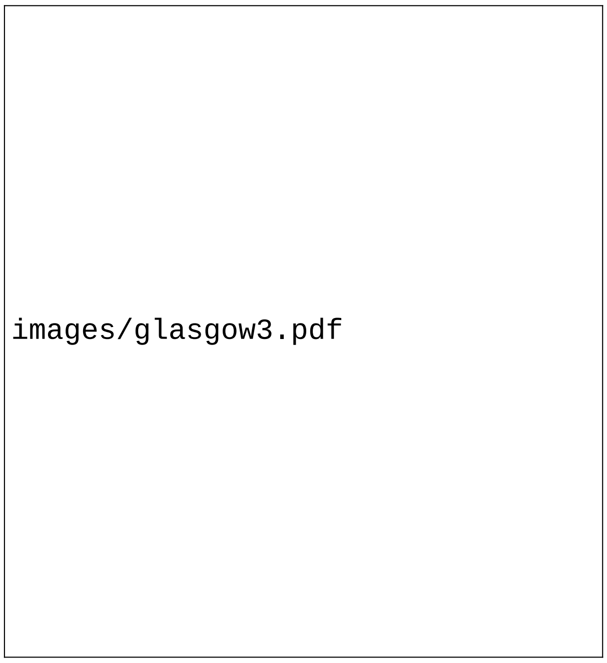


Figure 6.1: Illustration of the from-region visibility in 2D. The scene represents 25km^2 of Glasgow city. The 2D projection of the scene contains 94460 line segments. Parts of occluders visible from the given region are shown in red. Yellow line segments depict extremal lines that correspond to changes in visibility. Visibility computation took 700ms.

trees — a discrete structure similar to the visibility complex. Ghali and Stewart [Ghal96] use duality for maintaining visibility with respect to a moving point in the plane. This approach was further elaborated by Nechvíle and Tobola [Nech99].

Unlike most methods developed in computational geometry, the method discussed in this chapter focuses on *practical usability* of solving the from-region visibility problem for large 2D scenes.

6.3 Algorithm overview

Consider a scene consisting of a set of line segments that we call *occluders*. Given a convex polygonal *view cell* R_S, the task is to determine visibility from R_S. We incrementally build the *occlusion tree* that associates with each ray emerging from R_S an occluder that it first intersects or a label describing that the ray is unoccluded. The occluders are organized in a kD-tree. Each leaf of the tree stores a list of occluders that intersect the corresponding spatial region.

The kD-tree is processed using an approximate occlusion sweep with respect to R_S. At each leaf of the tree the corresponding occluders are processed in random order. For every occluder, a line space *blocker polygon* is constructed that represents rays intersecting the view cell and the occluder. The blocker polygon is then inserted in the occlusion tree. The insertion yields parts of the occluder that are currently visible. If the occluder is invisible, the tree remains unmodified. The traversal of the kD-tree is interleaved with testing visibility of regions corresponding to its nodes using the current occlusion tree. If a node is classified invisible, its whole subtree and the corresponding occluders are culled.

The proposed technique is mostly related to the visibility diagram and the visibility complex. In terminology of Vegter [Vegt90] the occlusion tree for from-region visibility in the plane is a hierarchical representation of the *visibility function*. The visibility complex is a 2D cellular complex [Good97] immersed in 3D. The occlusion tree can be seen as a hierarchical representation of a 2D cross-section of the visibility complex involving objects visible from the given view cell.

The rest of the chapter is organized as follows: In Section 6.4 we discuss the correspondence between *primal space* and *line space*. Section 6.5 describes the occlusion for the from-region visibility in the plane and presents algorithms for its construction and traversal. Section 6.6 outlines the complete hierarchical visibility algorithm. In Section 6.7 we evaluate the implementation of the proposed methods. Finally, Section 6.8 concludes the chapter.

6.4 Line space

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 [Pocc93].

We use a 2D projection of Plücker coordinates [Stol91] 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 that will be discussed in Chapter 8: 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 6.2-(a) and 6.2-(b). Further in this chapter we will mostly use "projected" 2D illustrations of line space (such as in Figure 6.2-(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.

6.4.1 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 6.2). Points in p^*_+ correspond to oriented lines passing

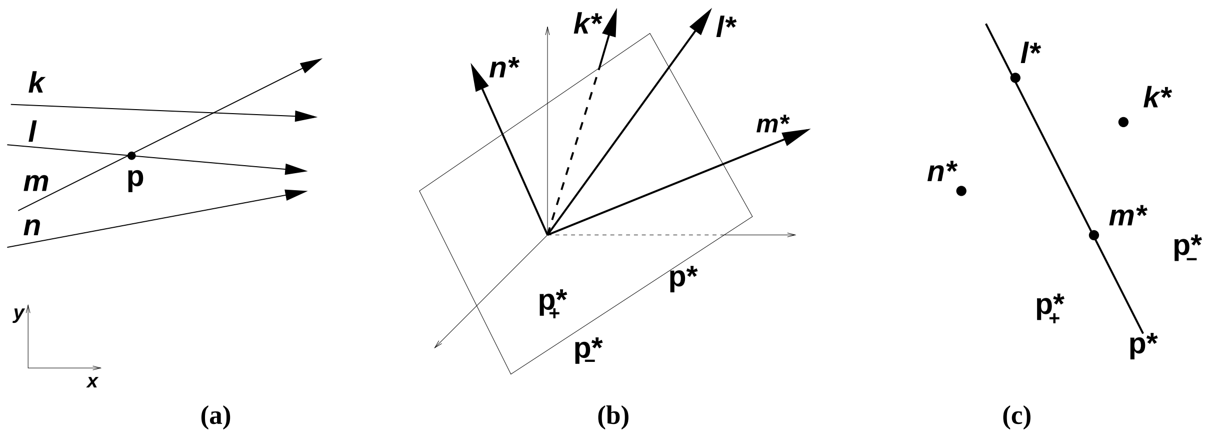


Figure 6.2: **(a)** Four oriented lines in primal space. **(b)** Mappings of the four lines and point p. Lines intersecting p map to plane p^*. Lines passing clockwise (counterclockwise) around p, map to p^*_- (p^*_+). **(c)** The situation after projection to a plane perpendicular to p^*.

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:

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

6.4.2 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 6.3-(a,b)).

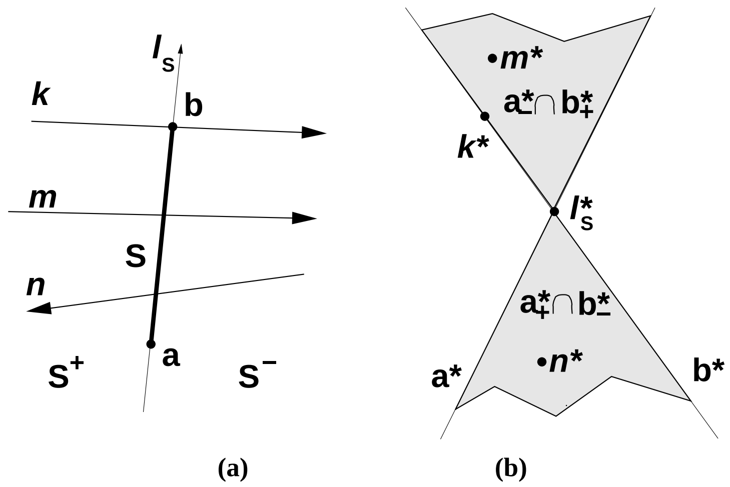


Figure 6.3: **(a)** A line segment S and three oriented lines that intersect S. **(b)** The situation in line space: the projection of two wedges corresponding to lines intersecting S. Mappings of supporting line l_S of S are two halflines that project to point l^*_S. Line k intersects point b and therefore maps to plane b^*. Lines m and n map to the wedge corresponding to their orientation.

6.4.3 Lines passing through two line segments

Consider two disjoint line segments such as those depicted in Figure 6.4-(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 polygon* $B(S, O)$ (see Figure 6.4-(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 $\mathcal{L}$.

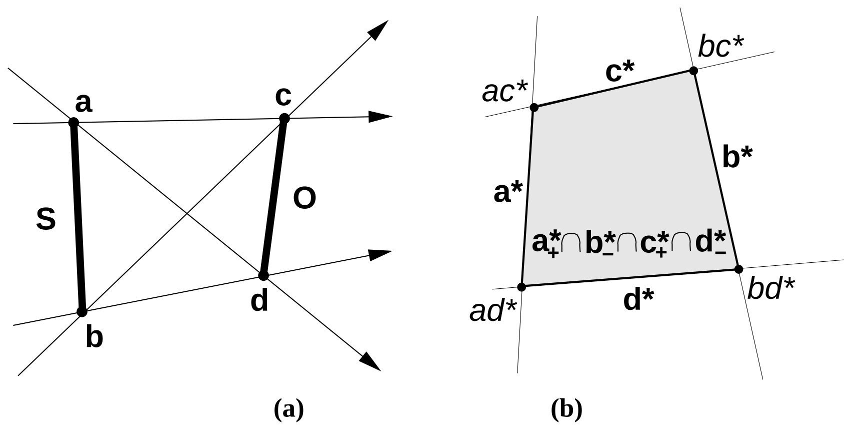


Figure 6.4: **(a)** Two line segments and corresponding four extremal lines oriented from S to O. Separating lines ad and bc bound region of partial visibility of S behind O (penumbra). Supporting lines ac and bd bound region where S is invisible (umbra). **(b)** Blocker polygon $B(S, O)$ representing pyramid $\mathcal{P}(S, O)$.

In Figure 6.5-(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 6.5-(b).

6.4.4 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 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}

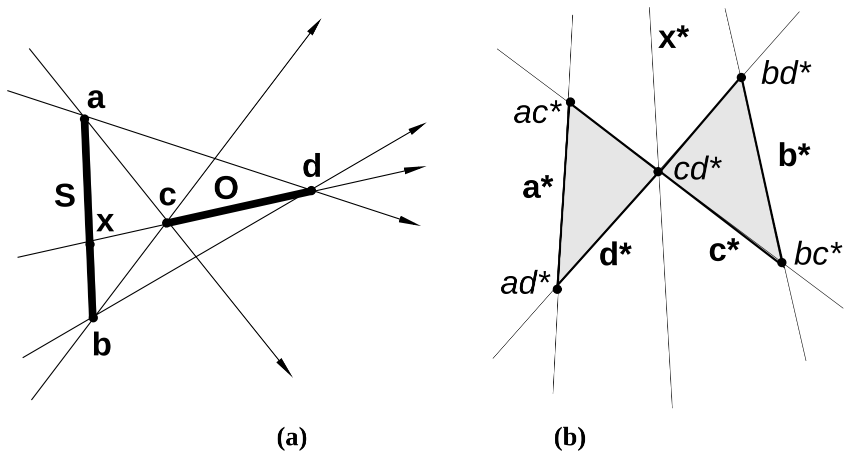


Figure 6.5: **(a)** Degenerate configuration of line segments: the supporting line of cd intersects ab at point x. There are five extremal lines. Note, that there is no umbra region. **(b)** In line space the configuration yields two pyramids sharing a boundary that is a mapping of the oriented line cd.

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 6.6-(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 $\mathcal{P}(S, O_k)$.

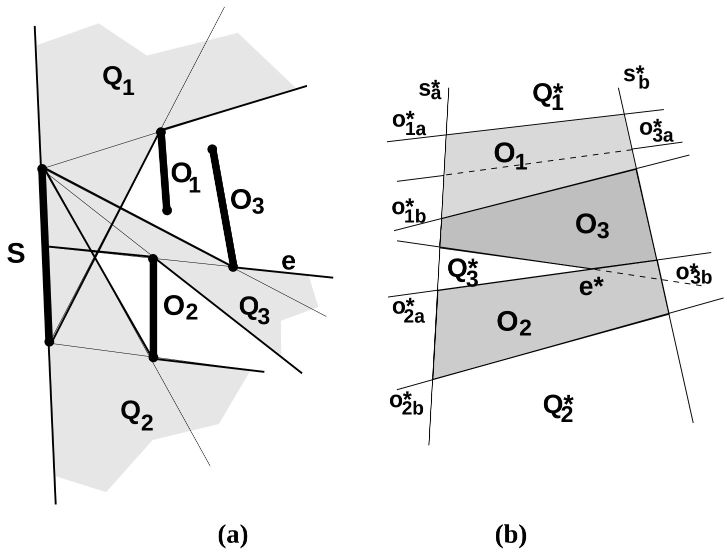


Figure 6.6: **(a)** The source line segment S and three occluders. Q_{1-3} denote unoccluded funnels. **(b)** The line space subdivision. For each cell, the corresponding occluder-sequence is depicted. Note the cells Q_1^*, Q_2^* and Q_3^* corresponding to unoccluded funnels.

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 2. 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.

6.5 Occlusion tree

The occlusion tree for visibility from region in 2D is a BSP tree maintaining the arrangement $\mathcal{A}_k$. Each node N of the tree represents certain subset of line space $\mathcal{Q}^*_N$. The root of the tree represents the whole $\mathcal{L}$. If N is an interior node, it is associated with a plane h_N. The left child of N represents $\mathcal{Q}^*_N \cap h^-_N$, the right child $\mathcal{Q}^*_N \cap h^+_N$, where h^-_N and h^+_N are halfspaces induced by h_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. If N is an in-leaf, it is associated with an occluder O_N that blocks the corresponding set of rays $\mathcal{Q}^*_N$. Further N stores an intersection of the blocker polygon $B(S, O_N)$ and $\mathcal{Q}^*_N$, denoted B_N. $\mathcal{Q}^*_N$ represents a *funnel* $\mathcal{Q}_N$ in primal space that is bound by points corresponding to planes of the occlusion tree on the path from root to N. Node N also contains a line segment I_N that is an intersection of O_N and $\mathcal{Q}_N$.

Consider a source and a single occluder such as in Figure 6.4-(a). The corresponding tree has four interior nodes that represent endpoints of the two line segments. In this case the occlusion tree is actually a BSP tree representation of the pyramid $\mathcal{P}(S, O)$ (see Figure 6.7-(a)). We call this tree elementary occlusion tree for a blocker polygon $B(S, O)$, denoted e-OT($B(S, O)$). A more complex situation is depicted in Figure 6.7-(b).

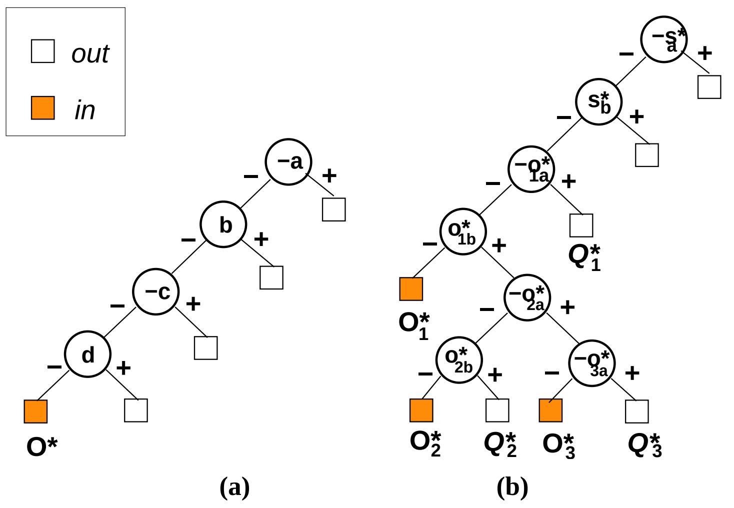


Figure 6.7: **(a)** Elementary occlusion tree for a configuration shown in Figure 6.4-(a). **(b)** Occlusion tree for the three occluders depicted in Figure 6.6-(a). The three in-leaves correspond to rays blocked by the relevant occluders. The out-leaves Q_i represent three funnels of unoccluded rays emerging from S in the direction of occluders.

We need to perform polyhedra set operations on the arrangement $\mathcal{A}_k$ to obtain both the arrangement $\mathcal{A}_{k+1}$, and $\mathcal{V}(S, O_{k+1})$ that describes visibility of occluder O_{k+1}. In particular we must determine the set difference of $\mathcal{P}(S, O_{k+1})$ and $\mathcal{A}_k$ to obtain $\mathcal{V}(S, O_{k+1})$ and the union of $\mathcal{A}_k$ and $\mathcal{V}(S, O_{k+1})$ to obtain $\mathcal{A}_{k+1}$. The set difference operation can be performed easily using BSP tree by "filtering" $\mathcal{P}(S, O_{k+1})$ down the tree as outlined in Section 3.3.3. The filtering identifies the desired set $\mathcal{V}(S, O_k + 1)$ that is then used to extend the tree so that it represents the arrangement $\mathcal{A}_{k+1}$.

6.5.1 Occlusion tree construction

Assume that we can determine a strict front-to-back order of occluders with respect to the source S. When we process occluder O_k all occluders O_j $(1 \leq j < k)$ that can block visibility of O_k have been already processed before. On the other hand O_k cannot block any occluder O_j $(1 \leq j < k)$.

The occlusion tree construction algorithm can be outlined as follows: For an occluder we construct a pyramid $\mathcal{P}(S, O_k)$ corresponding to rays blocked by this occluder. The pyramid is represented by a blocker polygon $B(S, O_k)$, that is then filtered down the tree. The algorithm maintains two variables — the current node N_c and the current blocker polygon B_c. Initially N_c is set to the root of the tree and B_c equals to $B(S, O_k)$.

If N_c is not a leaf we determine the position of B_c and the plane h_{N_c} associated with N_c. If B_c lies in the negative halfspace induced by h_{N_c} the algorithm continues in the left subtree. Similarly, if B_c lies in the positive halfspace induced by h_{N_c} the algorithm continues in the right subtree. If B_c intersects both halfspaces it is split by h_{N_c} into two parts B_c^- and B_c^+ and the algorithm proceeds in both subtrees of N_c with appropriate 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 in-leaf all rays represented by B_c are blocked by the occluder O_{N_c} referred in N_c. If N_c is an out-leaf then all rays represented B_c are unoccluded. Thus B_c is a part of $\mathcal{V}(S, O_k)$. We replace the N_c by e-OT(B_c) representing the pyramid induced by B_c that contains planes defined by the origin of $\mathcal{L}$ and edges of B_c.

Until now we have assumed that a front-to-back order of occluders with respect to the source is determined. The approximate occlusion sweep determines only an approximate front-to-back order of occluders. Thus we cannot guarantee the position of the currently processed occluder O_k with respect to the already processed occluders. In the remainder of this section we describe the necessary modification of the occlusion tree construction algorithm.

The modification involves the behavior of the algorithm when reaching the in-leaves of the tree. Previously we have assumed that all lines represented by blocker polygon B_c are blocked by an occluder O_{N_c}. Now we have to check the position of O_k and O_{N_c} with respect to the source. We use the line segments I_k and I_{N_c} that are intersections of O_k and O_{N_c} with the funnel $\mathcal{Q}_{N_c}$ (see I_2 and I_1 on Figure 6.8-(a)). Denote an open halfspace defined by I_{N_c} that contains the source H^+ and the opposite open halfspace H^-.

The two line segments can be in four mutual positions:

1. I_k behind I_{N_c}: I_k lies completely in H^-.
2. I_k in front of I_{N_c}: I_k lies completely in H^+.
3. I_k intersects I_{N_c}: I_k lies in both H^+ and H^-.
4. I_k on I_{N_c}: I_k does not intersect H^+ nor H^-.

In the first case all lines represented by B_c are blocked by I_{N_c}. Thus O_k is not visible through this set of lines and no modification to the tree is necessary.

In the second case all lines represented by B_c are not blocked by I_{N_c}. Thus O_k is visible through all lines of B_c and contrary O_{N_c} is not visible through these lines. We construct e-OT(B_c) and filter the "old" blocker polygon B_{N_c} down this tree (see Figure 6.8). Finally, we replace the N_c by the constructed tree. Note that it is possible that the e-OT(B_c) consists of a single in-leaf node.

In the third case some lines of B_c are blocked by I_{N_c} and some block I_{N_c}. We determine the intersection X of line segments I_k and I_{N_c}. Point X maps to plane h_X in line space. We split B_c by h_X obtaining B_c^+ and B_c^-. Depending on the position of the endpoints of I_k we decide which of the two fragments corresponds to lines blocked by I_{N_c}. This fragment can be deleted as in the case 1. The other fragment is treated as in the case 2.

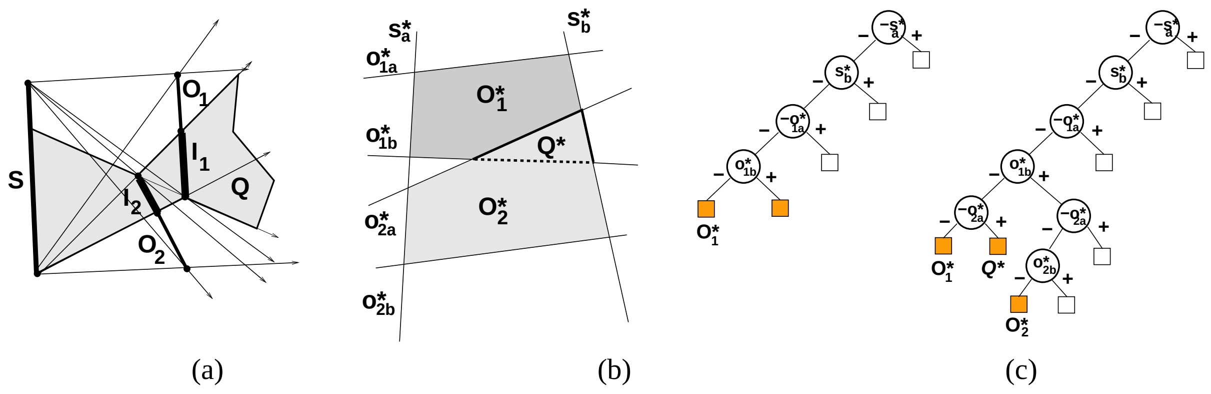


Figure 6.8: (a) Occluder O_2 partially hides occluder O_1 that is already in the tree. The funnel $\mathcal{Q}$ corresponds to the lines through which O_2 hides O_1. Note that O_1 is still completely visible from some points on S. (b) Situation in line space. (c) The original occlusion tree and the tree after inserting O_2. Note, the lower left node has both descendants classified as in.

The solution of the fourth case depends on the application. For the sake of simplicity we assume that O_k is invisible through lines of B_c.

6.5.2 Visibility from a region

Visibility from a convex polygonal region can be determined by computing visibility from its boundaries. A separate occlusion tree can be built to capture visibility from each boundary line segment S_i of the view cell R_S. Alternatively we can build a single tree that captures visibility from all boundaries. The latter approach is more consistent since it keeps a single hierarchical structure for representation of visibility from the view cell. Inserting rays blocked by occluder O_k into the tree involves insertion of blocker polygons $B(S_i, O_k)$. The blocker polygon $B(S_i, O_k)$ represents rays emerging from i-th boundary of R_S that is facing the occluder O_k.

6.5.3 Visibility of a line segment

Suppose we are interested only in determining visibility of a line segment G, but we do not want to include rays intersected by G in the occlusion tree. An example of such situation is determining visibility of a polygonal region that is bound by a set of 'virtual' line segments, that should not be used as occluders. We use the algorithm from Section 6.5.1, without performing any modifications to the tree. We only collect the set $\mathcal{V}(S,G)$ of visible fragments of the blocker polygon $B(S,G)$. If this set is empty, L is not visible from the view cell. Otherwise, it is visible through the set of rays that correspond to $\mathcal{V}(S,G)$.

6.5.4 Fast conservative visibility of a region

Given an occlusion tree and a query region R_X the *conservative visibility test* determines conservatively if R_X is invisible or at least partially visible. It can eventually classify an invisible region as visible, but it never classifies a visible region as invisible.

The conservative visibility test proceeds as follows: given a view cell and a polygonal region R_X we find supporting and separating lines of the view cell R_S and R_X (see Figure 6.9-(a,b)). We construct a blocker polygon $B(R_S, R_X)$ using mappings of the supporting and separating lines as its vertices. This blocker polygon generally represents a superset of rays intersecting R_S and R_X (see Figure 6.10-

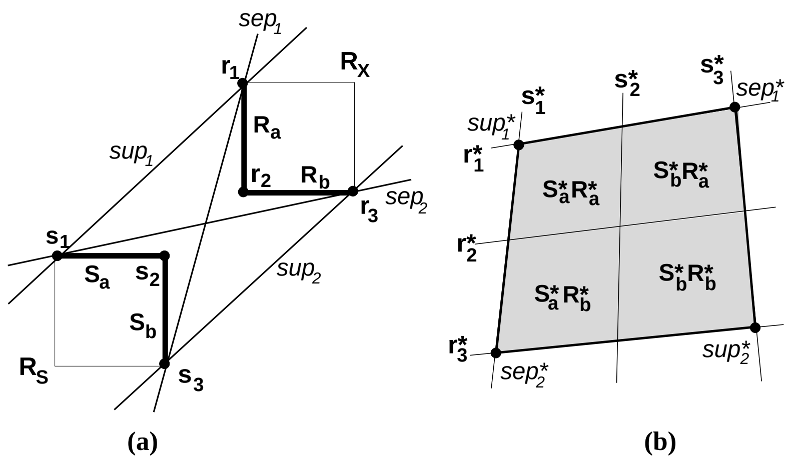


Figure 6.9: **(a)** Two rectangular regions and corresponding extremal lines. **(b)** In line space there are four blocker polygons corresponding to four combinations of the mutually visible boundaries of R_S and R_X. Note that the vertices of the union of these blocker polygons correspond to supporting (sup$_1$, sup$_2$) and separating (sep$_1$, sep$_2$) lines of R_S and R_X.

(a,b)). $B(R_S, R_X)$ is filtered down the tree as described in Section 6.5.1. Reaching an out-leaf we can conclude that at least part of R_X is visible and terminate the algorithm.

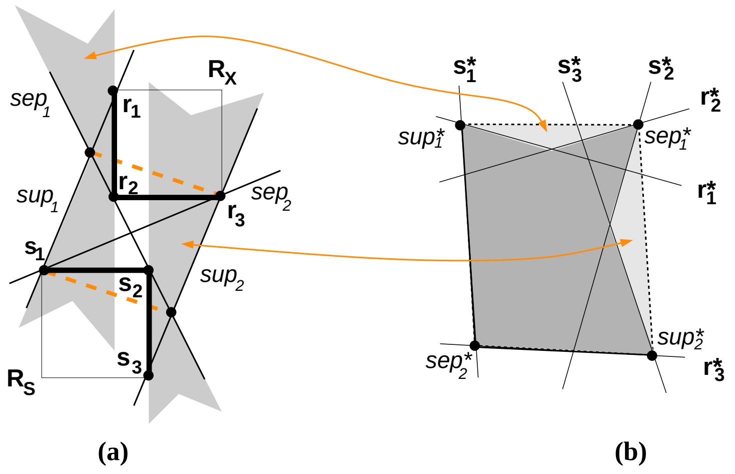


Figure 6.10: **(a)** A configuration of R_S and R_X that leads to a conservative blocker polygon. The blocker polygon represents lines intersecting the 'virtual' line segments show dashed. **(b)** The blocker polygon constructed from mappings of supporting and separating lines represents a superset of the rays intersecting R_S and R_X.

If the filtering procedure reaches an in-leaf L we determine the relative position of O_L and R_X. Denote an open halfspace defined by O_L that contains the source H^+ and the opposite halfspace H^-. Visibility of R_X in L is classified as follows:

1. R_X is visible, if R_X lies completely in H^+.
2. R_X is invisible, if R_X lies completely in H^-.
3. R_X is partially visible, if R_X intersects both H^+ and H^-.

The decision in the third case is conservative. An exact visibility test of R_X would require computation of the intersection of R_X and the funnel $\mathcal{Q}_L$ corresponding to leaf L, followed by the intersection test with I_L.

Further speedup of the visibility test can be achieved for an additional decrease of its accuracy. The filtering procedure can be modified so that the blocker polygon B_c being processed is not split by planes of the occlusion tree that intersect B_c. At each interior node N_c only the position of B_c with respect to the plane h_{N_c} is evaluated. The algorithm then continues in children of N_c that correspond to halfspaces intersected by B_c. Using this approach B_c can be filtered down to leaves that correspond to pyramids that actually does not intersect B_c. Nevertheless this case occurs rather seldom in practice. Similar modification was used for the real-time visibility culling as described in Section 4.10.

6.5.5 Maximal visibility distance

Each node N of the occlusion tree can be associated with a *maximal visibility distance* (MVD) in the corresponding funnel $\mathcal{Q}_N$. Denote d_N the MVD of node N. If N is an out-leaf, $d_N = \infty$. If N is an in-leaf, d_N is a maximal distance of the view cell R_S and the relevant part I_N of occluder O_N. If N is an interior node of the tree, d_N is a maximum of MVDs of its children. MVDs are updated by propagating their changes up the tree after each insertion of an occluder. MVD of node N can be used to quickly determine that an occluder O_x is invisible with respect to N if the minimal distance of O_x from R_S is greater than d_N.

Consider a situation that all oriented lines emerging from R_S are blocked after inserting k occluders. The visibility distances of nodes near the root of the tree will correspond to distance of a farthest visible occluders in the corresponding funnels. An occluder or a region farther from R_S is immediately culled by the distance comparison at nodes near the root.

The maximal visibility distances can be seen as a hierarchical piecewise constant representation of the depth map with respect to the given view cell. In leaves of the tree the depth is represented exactly by the associated occluder fragments I_N.

6.6 The complete hierarchical visibility algorithm

The above mentioned methods are used within a hierarchical visibility algorithm. Occluders are organized in a kD-tree, in which a node N corresponds to rectangular region R_N. The root of the tree corresponds to the bounding box of the whole scene. Leaves of the kD-tree contain links to the occluders that intersect the corresponding cells. The kD-tree is used for two main purposes: Firstly, it allows to determine the approximate front-to-back order of occluders efficiently. Secondly, pruning of the kD-tree by hierarchical visibility tests leads to the expected output-sensitive behavior of the algorithm.

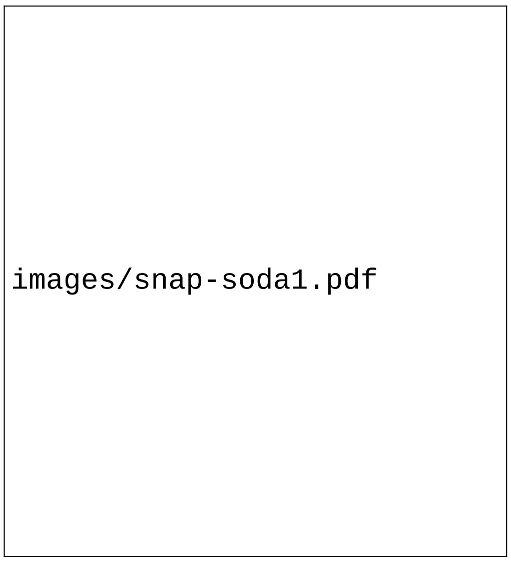
The scene is processed using an approximate occlusion sweep with respect to the view cell. We have used two algorithms for the occlusion sweep. The first processes the kD-tree using a strict front-to-back ordering of kD-tree nodes with respect to the center of the view cell. The second uses a priority queue, where the priority of node N is inversely proportional to the minimal distance of R_N from the view cell. In both cases occluders stored within a leaf node are processed in a random order. The occlusion tree is constructed incrementally by inserting blocker polygons $B(S_i, O_k)$ corresponding to the currently processed occluder O_k and the i-th boundary of the view cell that faces O_k.

The occlusion tree construction is interleaved with visibility tests of region R_N corresponding to the currently processed node N. If R_N intersects the view cell R_S, it is classified visible. Otherwise the visibility test classifies visibility of R_N with respect to the already processed occluders. If R_N is invisible, the subtree of N and all occluders it contains are culled. If it is visible, we continue testing visibility of its descendants that are processed recursively using the approximate occlusion sweep.

Hierarchical visibility tests provide a useful information about visibility of the whole scene from the given view cell. Nevertheless the visibility classification of hierarchy nodes as described above is only conservative. Due to the approximate depth ordering of processed nodes it is possible that there is an unprocessed region R_l containing occluders which might occlude (or at least partially occlude) the currently processed region R_k. R_k can be classified visible, although in fact it is invisible due to the influence of unprocessed occluders from R_l. If an exact classification of hierarchy nodes is required, we have to perform additional visibility tests on all leaves of the kD-tree that were previously classified visible. These tests eventually decide that the tested regions are invisible if all relevant occluders are considered. The visibility changes can be propagated up the kD-tree and the coarse visibility information can then be stored as a list of visible nodes at the highest level of the kD-tree.

6.7 Results

We have evaluated the presented algorithms on three types of scenes: a city plan, a building interior, and random line segments. The three types of scenes are depicted in Figures 6.1, 6.11-(a) and Figure 6.11-(b), respectively. On each figure the blue rectangle represents the view cell. Yellow lines correspond to extremal lines that bound the funnels $\mathcal{Q}_{L_x}$. Red line segments depict visible parts of occluders I_{L_x}. Gray regions were culled by the hierarchical node visibility test. All measurements were performed on a PC, equipped with an Athlon 950MHz CPU, 256MB RAM and Linux OS. Measurements for several selected regions and various termination criteria for the construction of the kD-tree are summarized in Table 6.1.



(a)

(b)

Figure 6.11: **(a)** The ground plan of the building of the Soda hall (873 occluders). 159 occluders are visible, the occlusion tree has 745 nodes. Computation took 190ms. **(b)** A scene with 1000 random line segments. 540 segments are visible, the occlusion tree has 8607 nodes. The green regions were reclassified invisible by the second pass of the visibility tests. The computation took $1.6s$.

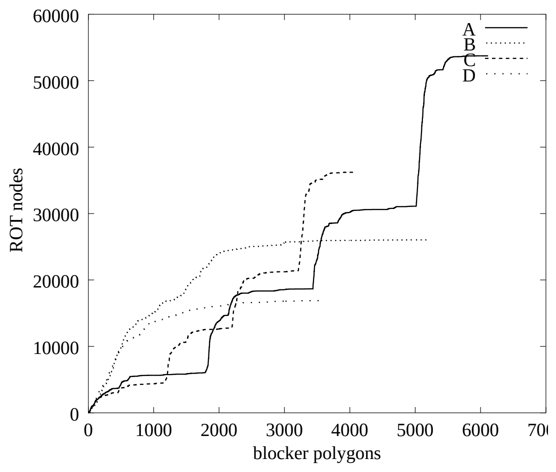
Figure 6.1 depicts a 2D cut through the scene representing a city of Glasgow[1]. For the selected view cell the majority of the scene was culled by the hierarchical visibility test and only few occluders corresponding to the neighboring streets were found visible. Figure 6.11-(a) depicts a ground plan of the Soda Hall of the University of Berkeley and Figure 6.12-(b) shows the corresponding blocker polygons.

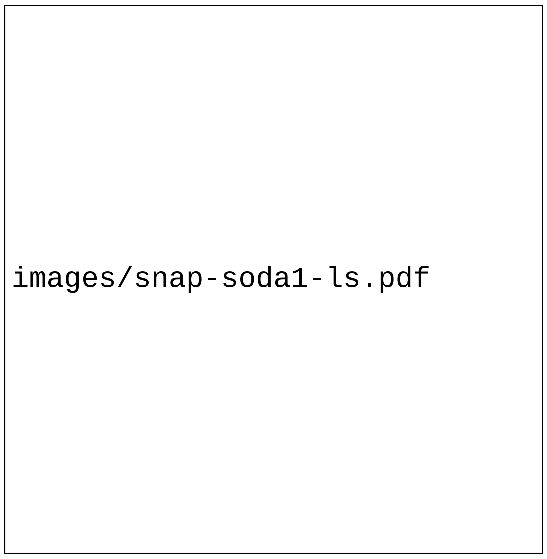
Scenes consisting of random line segments allow to study the dependence of the algorithm on the complexity of the scene as well as its *visibility complexity*. The size of the occlusion tree is proportional to the visibility complexity of the given view cell. More precisely the size of the tree corresponds to the total number of funnels $\mathcal{Q}_{N_x}$ through which occluders are visible from the view cell.

[1]`http://www.vrglasgow.co.uk`

Scene	Occluders $[-]$	kD-tree nodes $[-]$	R_S size $[\%]$	Tested kD-tree nodes $[\%]$	Blocker polygons $[-]$	OT nodes $[-]$	Visible occluders $[\%]$	Time $[s]$
Glasgow	94460	13711	0.017	4.2	4400	1371	0.32	1.2
		13711	0.21	6.8	6919	3233	0.75	3.5
		13711	0.64	16	13142	7545	1.5	9.2
Soda	873	335	2.5	50	759	1045	19	0.31
		2169	2.5	34	619	671	19	0.63
		2169	6	65	1078	2605	44	2.2
Random	1000	61	5.1	62	1682	8639	54	1.3
		1767	5.1	51	1018	4385	54	1.3
Random	20000	7003	0.21	3.3	964	3185	2	0.61
		7003	3.4	8.2	2602	19179	13	5.2

Table 6.1: Summary of the results of the visibility algorithm. The table contains the total number of kD-tree nodes, relative size of the view cell with respect to the bounding box of the scene, the number of nodes of the kD-tree on which the hierarchical visibility test was applied, the total number of blocker polygons constructed for occluders, the total number of occlusion tree nodes, the percentage of occluders that are visible, and the total running time of the visibility algorithm.





(a) **(b)**

Figure 6.12: **(a)** Dependency of the number of occlusion tree nodes on the number of processed blocker polygons for various settings of the algorithm (see Section 6.7). **(b)** Visualization of blocker polygons of processed occluders. Note the spiky triangles on the top-left. These triangles are blocker polygons of line segments supporting line of which intersects R_S. The chains of blue vertices lay in planes corresponding to mappings of vertices of R_S.

The behavior of the algorithm also depends on the properties of the kD-tree used to organize occluders. The more occluders are associated with each leaf of the kD-tree the less accurate depth order is determined. Consequently, the occlusion tree is slightly larger due to late insertions of visible occluders. Additionally more occluders are inserted in the occlusion tree, although they could be culled by the hierarchical visibility test if a more precise front-to-back order was determined.

Figure 6.12-(a) depicts the dependence of the size of occlusion tree on the number of inserted blocker polygons. All curves were measured for a scene with 10000 random line segments. Curves A and B

were measured for the kD-tree with 50 occluders per leaf. Curves C and D correspond to the kD-tree with 5 occluders per leaf. For curves A and C the occlusion sweep with respect to the center of R_S was used, whereas for curves B and D the priority queue based occlusion sweep was applied. Note the "stairs" in graphs A and C that occur due to the fact that the regions of the kD-tree are processed in depth-first-like search. It follows from the graphs that the best results were obtained using method D — finer kD-tree and the priority queue sweep. The corresponding curve also shows that once the occlusion tree represents "enough" blocked rays its size does not grow.

6.8 Summary

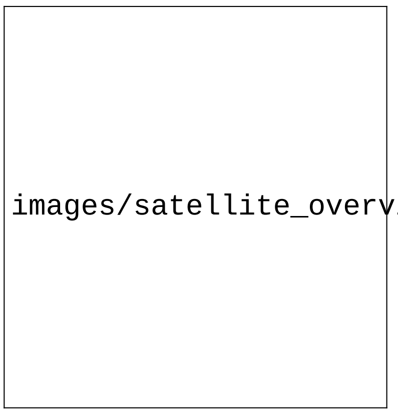
We have described an algorithm that computes visibility from a given region in a 2D scene consisting of a set of line segments. The method uses the concept of the approximate occlusion sweep, the occlusion tree, and hierarchical visibility tests discussed in Chapter 3.

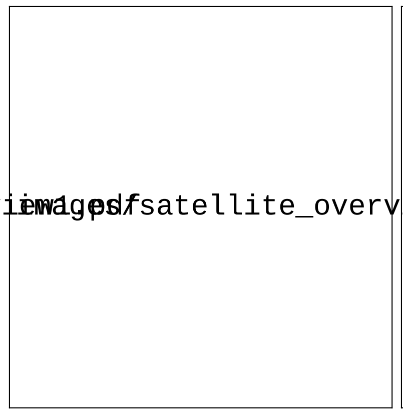
The algorithm is exact and accounts for all types of occluder fusion. We have demonstrated that the method is suitable for visibility preprocessing of large scenes by applying it to a scene representing a footprint of a large part of a city. Coherence of visibility is exploited by using two hierarchical structures: the occlusion tree for partitioning of the problem-relevant line set and the kD-tree for organizing occluders. The algorithm exhibits output-sensitive behavior for all tested scenes. The proposed method requires implementation of only few geometrical algorithms, namely partitioning of a polygon by a plane. The presented technique extends to $2\frac{1}{2}$D and it was used as a core of a visibility preprocessing algorithm for urban scenes [Bitt01e] discussed in the next chapter.

Chapter 7

From-region visibility in $2\frac{1}{2}$D scenes

In this chapter we present two algorithms for computing from-region visibility in $2\frac{1}{2}$D scenes. The proposed techniques are designed for visibility preprocessing for real-time walkthroughs in urban environments. The first algorithm uses the occlusion tree to maintain a subdivision of line space and the conservative *funnel visibility tests* to calculate a potentially visible set for a given view cell. The second algorithm computes an exact analytic solution to from-region visibility by using the occlusion tree and the *stabbing line computation*. See Figure 7.1 for an illustration of the from-region visibility in an urban scene.


(a)


(b)

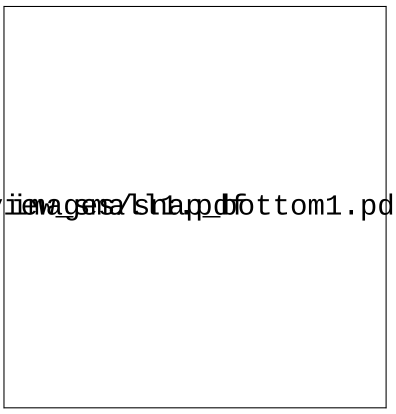

(c)

Figure 7.1: (a) Selected view cell in the scene representing the city of Vienna and the corresponding PVS. The dark regions were culled by hierarchical visibility tests. (b) A closeup of the view cell and its PVS. (c) Snapshot of an observer's view from a viewpoint inside the view cell.

We present a detailed evaluation of the methods including a comparison to another recently published visibility preprocessing algorithm by Wonka et al. [Wonk00]. The methods presented herein are specifically targeted for computing from-region visibility in an urban scene. Calculating visibility for a 3D spatial region is a complex problem. Previous methods (Schaufler et al. [Scha00] and Durand et al. [Dura00]) rely on several simplifications to cope with the computational complexity of 3D visibility. While these algorithms can handle a large variety of scenes, they only consider a subset of possible occluder interactions (occluder fusion) and require comparatively high calculation times.

It is useful to develop visibility algorithms building on simplifications that match the demands of specific types of scenes. Wonka et al. [Wonk00] observed that urban environments can be seen as $2\frac{1}{2}$D scenes. Although they propose an algorithm that handles all types of occluder fusion and treats occlusion systematically, the algorithm does not scale very well to large scenes and large view cells.

The algorithms described in this chapter significantly improve the *scalability* of the previous methods for the from-region visibility in $2\frac{1}{2}$D scenes. The algorithms exhibit *output-sensitive* behavior and therefore they are especially useful for large scenes and large view cells, both of which cannot be easily handled by previous techniques with an exceptions of the method of Koltun et al. [Kolt01]. We will demonstrate on a test model of Vienna that the proposed methods find a *tighter PVS* than the method proposed by Wonka et al. [Wonk00], while requiring lower calculation times.

7.1 Problem statement

The proposed algorithm is designed for scenes of $2\frac{1}{2}$D nature such as urban scenes (see Section 2.5.1). The scene can contain arbitrary geometry, but the *occluders* are restricted to be vertical trapezoids connected with the ground (typically building façades or roof ridges).

Given a view cell, the task is to determine the potentially visible set (PVS) of objects with respect to the view cell. The view cell is defined as a convex hull of its *vertical faces*. In order to compute the PVS in a $2\frac{1}{2}$D scene it is sufficient to consider only the *top edges* of the view cell faces, occluders, and object bounding boxes [Wonk00]. The PVS can be determined by solving visibility from each *top edge* bounding the view cell. The resulting PVS is then a union of PVSs computed for all top edges of the given view cell.

7.2 Related work

In this section we briefly discuss related methods for visibility preprocessing suitable for an application to urban scenes.

Airey et al. [Aire90] applied visibility preprocessing to architectural models. Visibility in indoor scenes was further studied by Teller and Séquin [Tell91]. Both methods partition the scene into cells and portals. For each cell they identify objects visible through sequences of portals. These objects form a *potentially visible set* for each cell. These methods are restricted to indoor scenes with a particular structure. Recently, several techniques for visibility preprocessing were introduced that are suited to urban environments. Cohen-Or et al. [Cohe98a] used ray shooting to sample occlusion due to single convex occluder. Schaufler et al. [Scha00] used *blocker extensions* to handle occluder fusion. Durand et al. [Dura00] proposed *extended occluder projections* and occlusion sweep to handle occluder fusion. Wonka et al. [Wonk00] used *cull maps* for visibility preprocessing in $2\frac{1}{2}$D scenes. Recently Koltun et al. [Kolt01] proposed a conservative algorithm that computes from-region visibility using mapping to line space and the z-buffer algorithm. To test visibility of an object the algorithm solves the 2D from-region visibility in the supporting plane of the object and the view cell. The from-region visibility for terrains was studied by Stewart [Stew97], and Cohen-Or and Shaked [Cohe95].

7.3 Algorithm overview

The main idea of the presented algorithms is to combine a solution to 2D visibility using the occlusion tree with a solution in *primal space* for the remaining “half dimension”. The algorithm organizes the scene in a kD-tree. For each top edge of a given view cell, it processes the occluders using an approximate occlusion sweep and incrementally builds the line space occlusion tree that represents the currently visible parts of the scene with respect to the already processed occluders.

For each occluder, we perform the following steps:

- Construct a line space *blocker polygon* from its 2D footprint on the ground plane. The blocker polygon is a special case of the *blocker polyhedron* discussed in Section 3.3.3.

- Calculate intersections with already processed blocker polygons. As a result, the blocker polygon is split into several fragments. Each fragment represents a set of rays that can be blocked by the same sequence of occluders. This set of rays forms a *funnel* in primal space. The intersection of the occluder and the funnel is called *occluder fragment*.
- For each blocker polygon fragment, we test visibility of the corresponding occluder fragment by mapping the problem back to primal space. We describe two techniques for computing visibility in the given primal space funnel:
  - A conservative algorithm using linear approximations of visibility events.
  - An exact algorithm using a stabbing line computation in Plücker coordinates.
- If an occluder fragment is found visible, the line space structure is updated accordingly.

The rest of the chapter is organized as follows: In Section 7.4 we discuss the description of $2\frac{1}{2}$D visibility in line space. In Section 7.5 we describe the conservative *funnel visibility test* used to determine if an occluder fragment is visible with respect to a given set of rays. In Section 7.6 we present the exact funnel visibility test. Section 7.7 outlines the complete hierarchical visibility algorithm. In Sections 7.8 and 7.9 we evaluate and discuss our implementation of the proposed methods.

7.4 $2\frac{1}{2}$D visibility and line space

The proposed visibility algorithm operates mainly on a 2D projection of the scene. The "heights" of scene entities are considered only when necessary (see Section 2.5.1 for a discussion of visibility in urban scenes). In order to solve the underlying 2D visibility problem, we use a mapping of oriented 2D lines to points in 2D oriented projective space – *line space* [Stol91]. This mapping was discussed in the previous chapter. To denote entities in line space, we use the asterisk notation, e.g. line l maps to l^*. The mapping of the problem to line space allows us to represent 2D bundles of rays (which carry the crucial part of visibility information) by simple polygons.

7.4.1 Basic definitions and terminology

We call the 2D projection of the $2\frac{1}{2}$D scene the *ground plan*. Each *2D ray* in the ground plan represents a vertical plane in the scene. The 2D ray corresponds to infinitely many *3D rays* that are rays lying in the corresponding vertical plane.

We call an *extended visibility function* a mapping that assigns each 2D ray a sequence of occluders, in which each occluder is visible by at least one 3D ray induced by the given 2D ray. Note, that the sequence might be empty.

7.4.2 Blocker polygon

This section reviews the correspondence of occluders in primal space and blocker polygons in line space. A detailed description was already presented in the previous chapter. A blocker polygon carries the 2D visibility information induced by an occluder and an edge of the given view cell. More specifically, it represents all 2D rays that emanate from the view cell edge and intersect the occluder. This set of rays is bounded by four *extremal lines*, forming an hourglass shaped region that we call a *2D funnel* or simply *funnel*. The four extremal lines map to points in line space and these points define the corresponding line space *blocker polygon*. Conversely, starting from a blocker polygon a corresponding funnel in primal space can be constructed by inverse mapping of the vertices of the blocker polygon to oriented lines in primal space (see Figure 6.4 in the previous chapter). We call a *3D funnel* a vertical extension of the 2D funnel that represents all 3D rays projecting to the 2D funnel.

If only 2D visibility was required, the blocker polygons could be used to solve the visibility problem in the following way: process occluders in front-to-back order. To determine whether a newly added occluder is visible, it suffices to test whether its associated blocker polygon is completely covered by other blocker polygons in line space. In such a case, the occluder is invisible: any 2D ray through which the new occluder could be visible is occluded by the already processed occluders.

7.4.3 Subdivision of line space

Mapping several occluders to line space induces a subdivision of line space into polygonal cells. Each cell contains a sequence of blocker polygons ordered by the distance of the occluders from the given view cell edge. A line space subdivision induced by three occluders is depicted in Figure 6.6 of the previous chapter.

The line space subdivision holds an important visibility information. Each cell corresponds to a funnel of 2D rays that intersect the same sequence of occluders. Thus for each each cell we have a sequence of occluders that are *potentially visible* through 3D rays corresponding to the cell. We are looking for a subdivision in which each cell corresponds to a sequence of blocker polygons that are *really visible* through the corresponding 3D rays. All blocker polygons corresponding to invisible occluders should be eliminated. We therefore construct the subdivision of line space incrementally and determine whether the currently processed occluder O is visible with respect to the occluders already processed.

The algorithm for constructing the line space subdivision proceeds as follows: for each occluder O, we identify the cells of the subdivision that are intersected by the corresponding blocker polygon. For each such cell, visibility of O is tested in primal space using occluders associated with this cell. This test is called the *funnel visibility test*. If O is found visible, it is inserted into the sequence of occluders for the cell, or the cell is further subdivided, depending on the height structure of occluders. If O is occluded, no changes are necessary.

For a conservative funnel visibility test discussed in Section 7.5 the resulting line space subdivision is a conservative representation of the extended visibility function. The exact funnel visibility test presented in Section 7.6 provides a subdivision representing the extended visibility function exactly.

7.4.4 Occlusion tree

The *occlusion tree* maintains a subdivision of line space similarly as described in the previous chapter for the case of 2D from-region visibility. The occlusion tree for the $2\frac{1}{2}$D visibility captures also occluders visible above closer occluders that would not be visible considering only their projections to the ground plan.

The occlusion tree is a BSP tree, in which each node N represents a cell $\mathcal{Q}^*_N$ of the line space subdivision. The root of the tree represents the whole problem-relevant line set. If N is an interior node, it is associated with a plane h_N. Left child of N represents $\mathcal{Q}^*_N \cap h^-_N$, right child $\mathcal{Q}^*_N \cap h^+_N$, where h^-_N and h^+_N are halfspaces induced by h_N. Leaves of the tree are classified in or out. If N is an out-leaf, $\mathcal{Q}^*_N$ represents unoccluded rays emerging from the source. If N is an in-leaf, it is associated with a sequence of occluders $\mathcal{S}_N$ that intersect the corresponding set of 2D rays $\mathcal{Q}_N$. N is also associated with a blocker polygon that represents the corresponding cell of line space subdivision.

We use the occlusion tree to identify cells intersected by the currently processed blocker polygon. The tree is then used to efficiently update the line space subdivision if we conclude that the currently processed occluder is visible.

7.5 Conservative funnel visibility test

This section describes the conservative *funnel visibility test* that classifies visibility of a new occluder O_n with respect to a cell $\mathcal{Q}_L^*$ of the current line space subdivision. The test is carried out in primal space within a funnel $\mathcal{Q}_L$ which the cell $\mathcal{Q}_L^*$ maps to. The solution proposed in this section is conservative since it approximates occlusion due to EEE event surfaces by planes.

We want to determine if a fragment of a new occluder O_n is visible with respect to the sequence of occluders $\mathcal{S}_L$ associated with the funnel $\mathcal{Q}_L$. The funnel visibility test can have three possible results:

- O_n is completely occluded by occluders $\mathcal{S}_L$ associated with $\mathcal{Q}_L^*$. In this case, O_n does not contribute to this cell.
- The top edge of O_n is visible across the whole funnel. In this case, it is simply added to the sequence $\mathcal{S}_L$ of relevant occluders for $\mathcal{Q}_L^*$.
- O_n is visible in only a part of the funnel. This means that a change of visibility occurs inside $\mathcal{Q}_L$, and the line space subdivision needs to be updated.

7.5.1 Extended umbra

The visible part of O_n is computed as follows: We select all occluders stored in the cell $\mathcal{Q}_L^*$ which lie in front of O_n in primal space. For each such occluder O, we calculate two *shadow planes*, against which we clip the top edge of O_n. The first plane, π_o, is defined by the top edge of O and a vertex of the top edge of the view cell face S, such that S and O lie on the same side of π_o. The second plane, π_s, is defined by an edge of S and a vertex of the top edge of O, such that S and the top edge of O lie on opposite sides of π_s (see Figures 7.2 and 7.3).

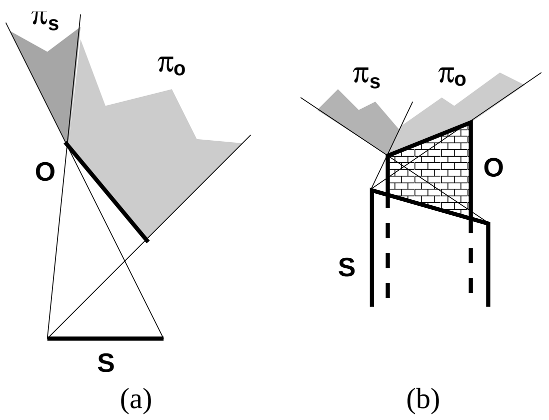


Figure 7.2: (a) Projections of the two shadow planes due to an occluder O and a view cell face S. (b) Front view of the view cell face, occluder and the two shadow planes.

The two planes π_o and π_s define an *extended umbra* of an occluder O within a funnel: any point inside the extended umbra is occluded by O considering all rays from the funnel. Note that the extended umbra of an occluder contains also points that can be visible from S through the rays that do not belong to the given funnel. Nevertheless, for the given configuration of the source and occluder the extended umbra properly captures all points reachable by all 3D rays that project to the given funnel defined by a top edge S of the view cell and a top edge of occluder O. If a smaller fragment S' of S is considered, the extended umbra of S and O is a conservative approximation of the extended umbra of S' and O (see Figure 7.4).

images/model_top1.pdf

(a)

images/model_front1.pdf

(b)

Figure 7.3: A simple model for studying the lines intersecting two occluders and their relation to the shadow planes. (a) A top view of the model. (b) A front view of the model.

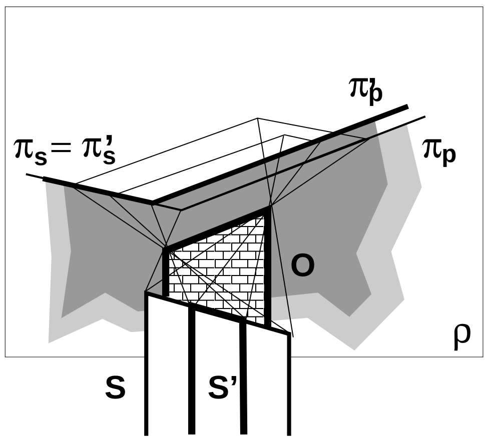


Figure 7.4: Extended umbra for a fragment S' of the view cell face S. The original extended umbra of S is a subset of the extended umbra of S'.

7.5.2 Occlusion tree updates

The visibility test in a funnel is performed by clipping a top edge of the inserted occluder by shadow planes of occluders associated with the funnel. Due to the configuration of the shadow planes the clipping results in at most one visible fragment. After clipping by shadow planes of all occluders in front of O_n, there is either a single fragment of the top edge left, or O_n is considered occluded.

If we found a visible fragment, each of its endpoints lying inside the funnel is mapped to an edge in line space. The resulting line space edge(s) are used to subdivide the original line space cell $\mathcal{Q}_L^*$ into at most three new cells. In exactly one of these cells, a fragment of O_n is visible and it is added to the associated sequence of occluders. The update of the occlusion tree is performed by replacing the leaf of the tree corresponding to $\mathcal{Q}_L^*$ by a subtree induced by the subdivision edges. One of the new leaves of the tree corresponds to a funnel in which O_n is visible. The other leaves (if any) correspond to cells of the line space subdivision where O_n is invisible. These leaves associated with the same sequence of occluders as the original cell $\mathcal{Q}_L^*$.

To see what this means in primal space, consider the two or three newly established funnels corresponding to the new cells. The *primary funnel* includes the visible part of O_n. The *secondary funnels* correspond to parts of the original funnel where O_n is invisible.

Figure 7.5 depicts a funnel $\mathcal{Q}$ containing three occluders. The new occluder O_n is partially visible with respect to the funnel. Funnel $\mathcal{Q}$ is split into three new funnels. The primary funnel $\mathcal{Q}_p$ contains the occluder sequence O_1, O_2, O_n, O_3. The first secondary funnel $\mathcal{Q}_{s1}$ is induced by the part of O_n hidden by the shadow plane of O_2. It contains the occluder sequence O_1, O_2, O_3. The other secondary funnel $\mathcal{Q}_{s2}$ corresponds to the set of 2D rays that do not intersect O_n. Figure 7.6 depicts the three blocker polygons corresponding to funnels $\mathcal{Q}_p$, $\mathcal{Q}_{s1}$ and $\mathcal{Q}_{s2}$.

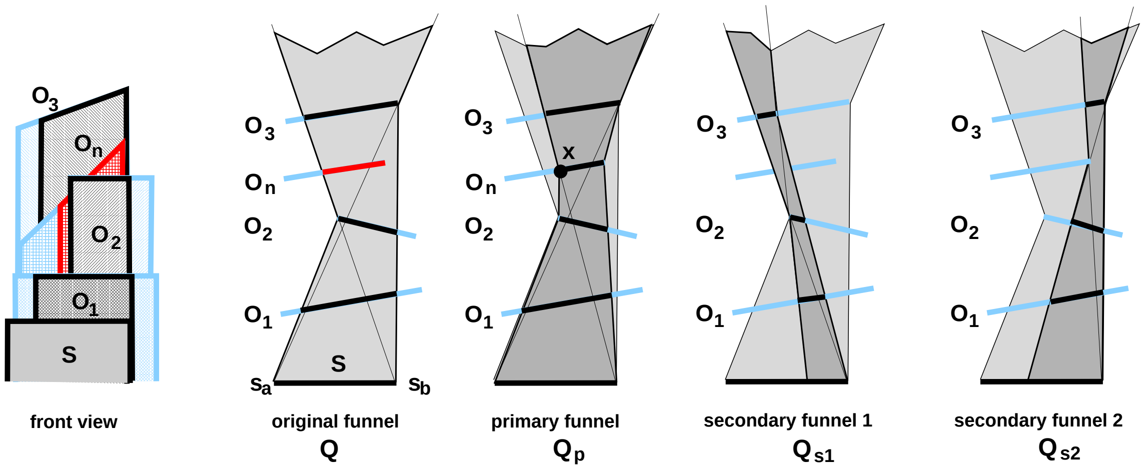


Figure 7.5: Adding a new occluder O_n into a funnel induced by three occluders can yield at most three new funnels.

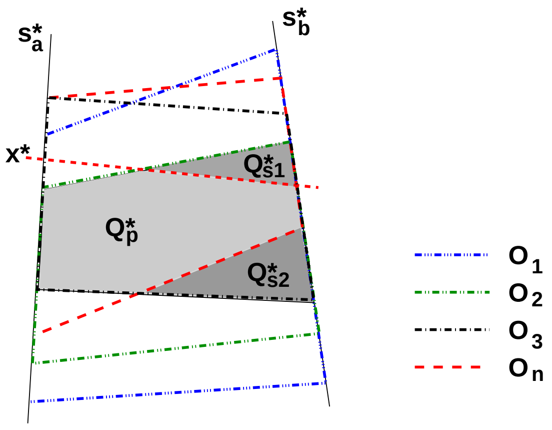


Figure 7.6: Three blocker polygons corresponding to funnels Q_p, Q_{s1}, Q_{s2} from Figure 7.5. x^* is the mapping of point x introduced by clipping occluder O_n by a shadow plane.

7.6 Exact funnel visibility

This section presents an exact funnel visibility test that provides an exact analytic solution to from-region visibility in $2\frac{1}{2}$D scenes. We use the set of occluders associated with a given funnel to construct a sequence of *virtual portals* through which the new occluder O_n might be visible from a given view cell. The algorithm then applies a *stabbing line* computation on a sequence of polygonal portals [Tell92b]. If there is a stabbing line that pierces O_n, the view cell, and all virtual portals, O_n is classified visible. Then we determine which part of O_n is visible to update the line space subdivision accordingly.

7.6.1 Stabbing line computation

For each occluder in the sequence associated with $\mathcal{Q}_L^*$ that lies in front of O_n we construct a portal that is bound by a top edge of the occluder and two vertical halflines directed upwards. Then we construct two portals that correspond to unbounded vertical trapezoids defined by O_n and a given view cell face. A stabbing line algorithm applied on the constructed portal sequence tests an existence of a ray leaving the view cell and intersecting O_n that is not blocked by any occluder (see Figure 7.7). If such ray exists, O_n is visible with respect to the given funnel. To determine the visible part of O_n in the funnel we need to reconstruct the *antipenumbra* [Tell92a] induced by the given set of portals and intersect it with O_n.

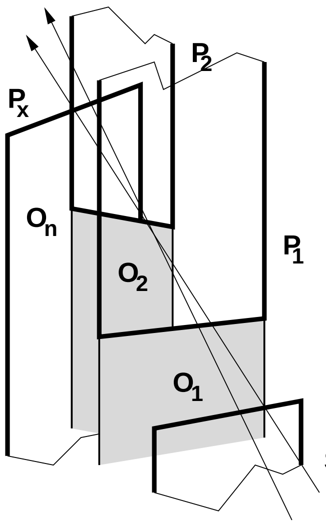


Figure 7.7: View cell and three occluders. Visibility of O_n is determined by stabbing line computation applied on four portals.

The stabbing line computation is carried out by halfspace intersection in 5D [Tell92a]. The 5D half-spaces are defined by hyperplanes that correspond to Plücker coefficients of lines bounding the virtual portals (more details on Plücker coordinates and associated operations will be presented in Chapter 8). Lines that stab all the portals correspond to 5D points on the Plücker quadric [Tell92a] that lie inside the intersection of these halfspaces. If there is no such point, there is no stabbing line and hence O_n is invisible with respect to the given occluder sequence.

Due to the $2\frac{1}{2}$D nature of the scene we can simplify the computation considering only a subset of feasible stabbing lines, namely the lines that intersect both the top edge of the occluder O_n and the top edge of the view cell. Consequently, we do not compute a full-dimensional 5D polyhedron B_{5D} of feasible stabbing lines, but rather its three-dimensional subset B_{3D} resulting from an intersection of B_{5D} with the hyperplanes defined by the top edges of the view cell and the occluder O_n. The polyhedron B_{3D} can be computed by treating the hyperplanes defined by the view cell and the occluder as equalities instead of treating them as halfspace constraints. B_{3D} has lower combinatorial complexity [Good97] and thus it can be computed faster than B_{5D} .

The polyhedron corresponding to feasible stabbing lines can be computed by linear programming in 5D [Tell92a]. We evaluated several implementations [Fuku02, Avis02] and finally we used a floating point version of the reverse search polyhedron enumeration algorithm [Avis96, Bitt97].

7.6.2 Computing visible fragments

The existence of a stabbing line only proves that the occluder O_n is visible in the funnel $\mathcal{Q}_L$. To determine the visible part of O_n we could reconstruct the *antipenumbra* [Tell92a] of the given portal sequence and intersect it with O_n. As stated above in $2\frac{1}{2}$D scenes we need to consider only a subset of antipenumbra consisting of lines that intersect the top edge of the view cell and the occluder.

To compute the antipenumbra we first compute the polyhedron B_{3D} as an intersection of the 5D halfspaces corresponding to portal edges. By intersecting the 1D skeleton of B_{3D} with the Plücker quadric we obtain a set of *extremal stabbing lines* [Tell92a]. The visible part of the occluder O_n is then given by the convex hull of the intersections of the extremal stabbing lines and the occluder O_n.

7.6.3 Acceleration of the funnel visibility test

The 5D halfspace intersection is a rather costly algorithm with asymptotic time complexity $O(n^2)$, where n is the number of halfspaces [Tell92a]. Fortunately, we can apply early termination criteria that decide if the new occluder is either definitely visible or definitely invisible. As a result the higher-dimensional algorithm is invoked relatively seldom in practice.

We select all occluders associated with the cell $\mathcal{Q}_L^*$ which lie in front of O_n in primal space. For each occluder we compute four shadow planes that bound the wedge defined by its top edge and a given top edge of the view cell. This wedge bounds a part of penumbra due to the occluder (see Figure 7.8). Note that the "lower" shadow planes correspond to the planes defining an extended umbra discussed in Section 7.5.1.

To determine if O_n is definitely invisible with respect to $\mathcal{Q}_L^*$ we clip it against the two lower shadow planes of each relevant occluder. If there is no fragment of O_n that lies above all lower shadow planes, O_n is definitely invisible. Otherwise, we clip O_n against the "upper" shadow planes. If there is no fragment of O_n that lies below, all these planes the top edge of O_n is completely visible with respect to $\mathcal{Q}_L^*$.

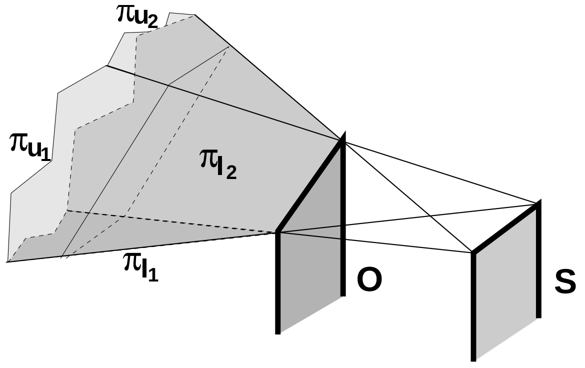


Figure 7.8: Four shadow planes bounding the penumbra wedge due to a top edge of occluder O and view cell face S. The lower shadow planes are denoted π_{l_1} and π_{l_2}, the upper ones π_{u_1} and π_{u_2}.

In all other cases we evaluate the intersection of the occluder top edge with all wedges (penumbras). If the result intersects only one penumbra, we can conclude that it is visible. If the result of the clipping lies in several wedges (penumbras), the merged umbra can be bound by quadratic EEE event surface (event surfaces will be discussed in Section 8.5). Since the EEE event surfaces are not handled by the conservative funnel visibility test we invoke the stabbing line computation.

7.7 Hierarchical visibility algorithm

The hierarchical visibility algorithm traverses the scene kD-tree using an approximate occlusion sweep with respect to the given view cell edge. The order is established using a priority queue, in which the priority of a node is inversely proportional to the minimal distance of the node from the view cell. Occluders stored within a leaf node are processed in random order.

The occlusion tree is constructed incrementally by inserting blocker polygons corresponding to the currently processed occluder. The occlusion tree construction is interleaved with visibility tests of the currently processed kD-tree node. The visibility test classifies visibility of the node with respect to the already processed occluders. If the node is invisible, the subtree rooted at the node and all occluders it contains are culled. If it is visible, the algorithm recursively continues testing visibility of its descendants. In the special case of a node intersecting the view cell edge, it is classified visible.

Visibility classification of scene objects is carried out after the complete occlusion tree has been constructed. If an object intersects a visible kD-tree leaf, we check its visibility using the occlusion tree and the funnel visibility test.

7.8 Results

We have evaluated the proposed methods using a scene representing a large part of the city of Vienna. First, we made a comparison of the conservative algorithm with the discrete hardware-accelerated approach by Wonka et al. [Wonk00]. Then the conservative algorithm was compared to the exact method. In the comparisons, we call the new methods the *line space subdivision* methods, denoted c-LSS (conservative) and e-LSS (exact). The *discrete cull map* method of Wonka et al. [Wonk00] is denoted DCM.

7.8.1 c-LSS vs. DCM

For evaluation of the c-LSS we used a PC equipped with a 950MHz Athlon CPU, and 256MB RAM. The DCM was evaluated on a PC with 650MHz Pentium III, 512MB RAM, and a GeForce DDR graphics card.

The tested scene represents 8 km^2 of the city of Vienna and consists of approximately 8 million triangles. The triangles are grouped into 17854 objects that are used for visibility classification. We automatically synthesized 15243 larger polygons to be used as occluders. Most occluders correspond to building façades.

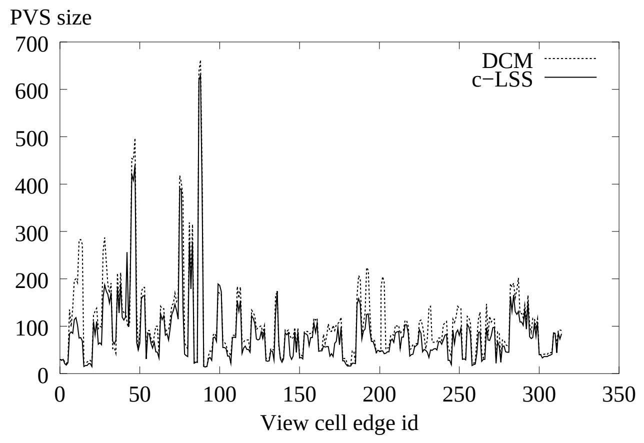
We conducted three different tests. In the first test, we randomly selected 105 out of 16447 view cells in the whole scene. For each view cell edge we computed a PVS. Figure 7.9 shows two plots depicting the sizes of the PVS and the running times of the two methods for each processed view cell edge.

The second test was designed to test the scalability of the two methods with respect to view cell size. We manually placed 10 larger view cells (with perimeter between 600 and 800 meters) and applied the algorithms on the corresponding edges. Table 7.1 summarizes the results for both the first and the second tests.

Test	Method	Avg. PVS size $[-]$	Avg. time $[ms]$
Test I	DCM	105.2	202.1
	c-LSS	**84.7**	**44.6**
Test II	DCM	274.0	4304.8
	c-LSS	**236.8**	**211.9**

Table 7.1: The average number of visible objects and the corresponding computational times for the first and the second tests.

The last test was carried out only for c-LSS. The goal was to test the scalability of the method with respect to the size of the scene and verify its output-sensitive behavior. We replicated the original scene on grids of size 2x2, 4x4 and 6x6. We selected a few view cells that provided the same size of the



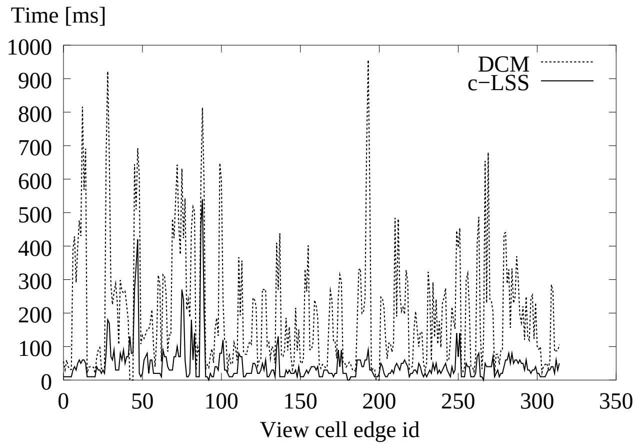


Figure 7.9: (top) The number of potentially visible objects for 305 view cell edges. (bottom) The running times of the two tested methods.

resulting PVS for each of the tested scenes. Table 7.2 depicts the size of the kD-tree and computation times for the original scene and the three larger replicated scenes.

7.8.2 c-LSS vs. e-LSS

For the comparison of the conservative and the exact LSS methods we have used a 1GHz Pentium III based PC with 384MB RAM. The tests were conducted using the same scene and view cells as described in the previous section.

Table 7.3 summarizes the results of the comparison. We can observe that the PVS computed by the e-LSS method is slightly smaller than the one computed by the c-LSS at the cost of increased computational time.

More detailed plot of the size of the PVS and the computational time for the Test II is depicted in Figure 7.10. The plot highlights a problem with the numerical accuracy of the two algorithms: for two view cell edges the conservative method produces a PVS one object smaller than the exact one, which contradicts the definition of the exact and conservative algorithm. This problem occurs due to the numerical inaccuracies of floating point computations in some degenerate configurations of the occluders and the objects. In particular when a top edge of an object is aligned with a shadow plane of some occluders the decision about its visibility depends on the ϵ-thresholds used for comparison of floating point values. This behavior can be partially solved by carefully tuning the ϵ-thresholds and applying them in the same solution space domain.

Grid	Area $[km^2]$	kD-tree nodes $[-]$	Avg. time $[ms]$
1x1	8	1609	90
2x2	32	6511	92
4x4	128	26191	101
6x6	288	57935	105

Table 7.2: Average PVS computation times for different scene sizes.

Test	Method	Avg. PVS size [-]	Avg. time $[ms]$
Test I	c-LSS	84.7	49.2
	e-LSS	84.2	200.1
Test II	c-LSS	236.8	206.1
	e-LSS	233.9	575.7

Table 7.3: Conservative vs. exact LSS. The average number of visible objects and the corresponding computational times for the first and second tests.

7.9 Discussion

In this section, we give an interpretation of the results. We discuss the suitability of the method for real-time rendering, the importance of large view cells, the output-sensitivity of the method, the exact vs. the conservative algorithm and the relation of the method to the algorithm of Koltun et al. [Kolt01].

7.9.1 Real-time rendering

The first test shows the calculation times and the PVS sizes for smaller view cells. We observe that both methods produce comparable results. The view cell size for this test was chosen so as to give reasonably-sized PVSs that would allow for walkthroughs with high frame rates [Wonk00]. The LSS methods produce a tighter PVS, because they do not rely on discretization and occluder shrinking.

7.9.2 Large view cells

The second test shows the scalability of the method for larger view cells. Although smaller view cells are more interesting for real-time rendering applications, larger view cells can be very useful. If we consider a very simple model, for example, where each façade is just one large flat polygon, it can be sufficient to calculate a solution for a rather large view cell. Another very important application is the hierarchical precalculation of visibility information. Similar to previous methods [Dura00], the visibility calculation could start with a subdivision of the view space into larger view cells. Smaller view cells are only calculated when necessary (e.g., when the size of the PVS is too large or when a heuristic determines large changes in visibility within the view cell). For urban environments, this hierarchical approach can be efficiently combined with a priori knowledge about the scene structure. If we use street sections as view cells, we can observe that visibility within one street section hardly changes (see Figure 7.11). In this context, we also want to emphasize that our view cells are not restricted to axis-aligned boxes.

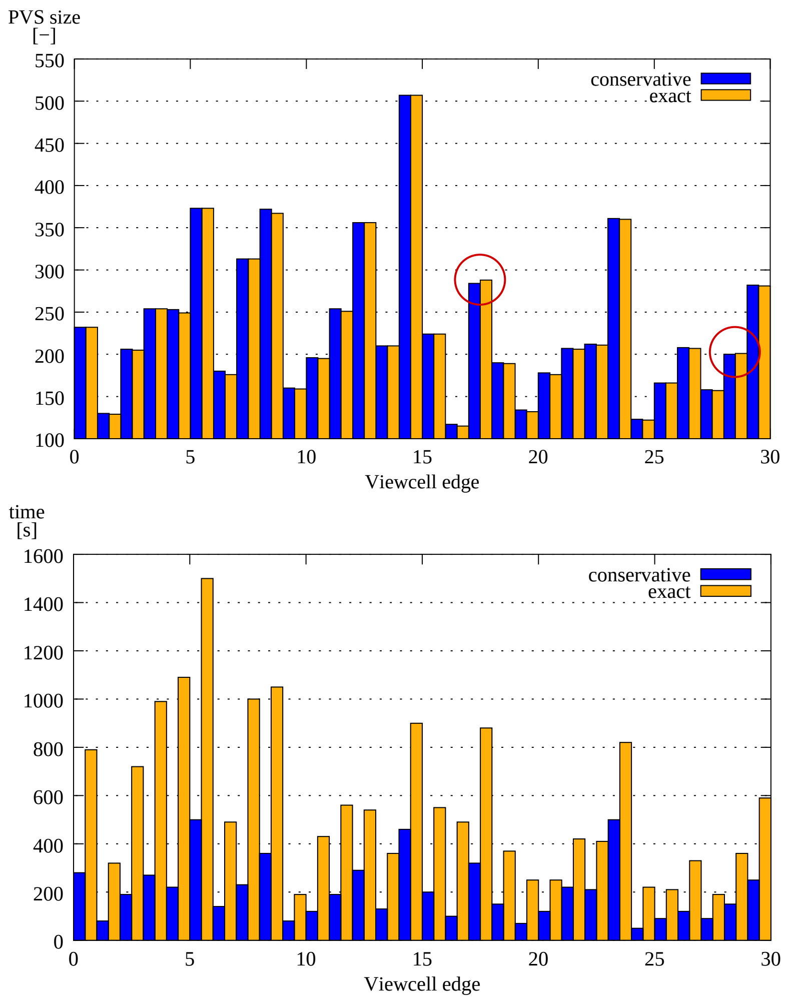


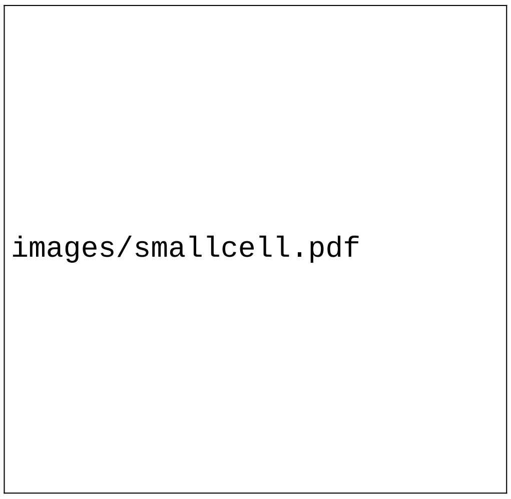
Figure 7.10: The size of the PVS and the running times of the c-LSS and the e-LSS methods for the 30 view cell edges used in Test II. The two red circles mark the edges where the PVS computation produces misleading results due to numerical inaccuracies of floating operations.

7.9.3 Output sensitivity

The third test shows the scalability of the method to larger scenes. It is a desired property of a visibility algorithm that the computation time mainly depends on the size of the PVS (=output) and not on the size of the scene (=input). The results strongly indicate output sensitivity of the algorithm in practice: the calculation times hardly change when the size of the scene is increased. Such a behavior can not be achieved easily by previous methods [Wonk00, Dura00, Scha00].

7.9.4 Exact vs. conservative

In the measurements the exact method e-LSS provided only slightly smaller PVS than the conservative one. This is an important result that shows that the solution obtained by the conservative method is very close the exact one. Due to the optimizations used in the e-LSS method (described in Section 7.6.3) the computational time of is not restrictive even for the large scenes. Nevertheless according to the results the effort required to implement the higher-dimensional polyhedra enumeration is not really recovered by the small improvement in the size of the computed PVS.



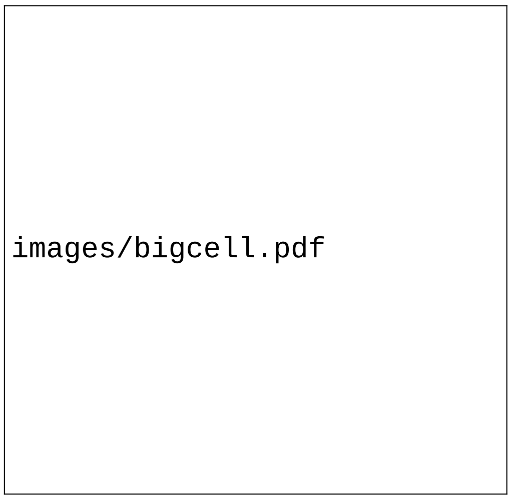


Figure 7.11: (left) A small view cell and its PVS. (right) The PVS for a larger view cell can be very similar.

7.9.5 Comparison with the method of Koltun et. al

Our method and the method proposed by Koltun et al. [Kolt01] share the idea of transforming the problem to line space. For each occluder Koltun at el. compute an intersection of the supporting plane of the occluder and the view cell with all other occluders. Visibility of the given occluder is then determined by solving a from-region visibility problem in that plane. The 2D from-region visibility is solved in line space using the z-buffer algorithm.

In contrast to the method of Koltun et. al. our method incrementally constructs a data structure capturing visibility from the view cell. This data structure consists from the occlusion tree and the shadow planes or the virtual portals. Our approach provides the following benefits:

- Higher accuracy.

  Given an occluder the method Koltun et al. uses a single plane in which the 2D from-region visibility is solved. Our conservative method captures the height structure of already processed polygons by locating a number of shadow planes withing each funnel of the line space subdivision. Additionally due to the continuous description of visibility our method does not rely on the discretization resolution. The exact method is more accurate by definition.

- Better use of coherence.

  Once an occluder is processed all visibility interactions with already processed occluders are identified and stored in the occlusion tree. When processing farther occluders the computed visibility interactions are reused.

- Generality.

  Maintaining visibility information in a general continuous data structure allows to associate various auxiliary information with the funnels, extremal lines or occluders.

A possible drawback of our method is its sensitivity to overly detailed inputs, whereas the computational complexity of the method of Koltun et al. is predominantly given by the resolution of the discretization.

7.10 Summary

This chapter presented two algorithms that determine visibility from a given view cell in a $2\frac{1}{2}$D scene. The algorithms are targeted at computing PVS in an outdoor urban environment. The first, conservative, method combines an exact solution to the 2D visibility problem with a tight conservative solution for the remaining "half dimension". The second, exact, method applies a stabbing line computation on a set of virtual portals determined by the solution of the 2D visibility problem. Both methods exploit visibility coherence by using a hierarchical subdivision of line space as well as a hierarchical organization of the scene. The algorithms achieve output-sensitive behavior by combining ordered processing of occluders and hierarchical visibility tests.

The methods are suitable for visibility preprocessing of large scenes, which was shown by applying it to a scene representing a large part of the city of Vienna. The proposed methods compare favorably with the previously published algorithm of Wonka et al. [Wonk00].

Chapter 8

From-region visibility in 3D scenes

This chapter presents an algorithm for computing exact from-region visibility in 3D scenes. From-region visibility problems were disregarded by the research community for a long time [Chaz96]. The 4D nature of these problems makes them difficult to solve. A discrete solution typically leads to a huge amount of samples that must be taken to obtain a reasonably precise solution. The storage of visibility in a discrete 4D or 5D data structure can lead to prohibitive memory consumption for the given sampling density [Arvo87, Simi94, Chry98a]. A continuous solution potentially allows a better use of visibility coherence, but its robust implementation is difficult [Dret94b, Stew94, Dura96, Dura97, Dugu02, Nire02]. The computational complexity of exact continuous methods is bounded by $O(n^4 \log n)$ in the worst case [Pell97] and thus these methods are sensitive to overly detailed inputs.

The algorithm presented in this chapter provides an analytic solution to the from-region visibility problem in 3D that is based on the concept presented in Chapter 3. The key idea is the mapping from primal space to a 5D line space using Plücker coordinates. The arrangement of 5D blocker polyhedra is maintained using a 5D occlusion tree. The occlusion tree facilitates efficient set operations on the blocker polyhedra and improves robustness of the method by providing a consistent BSP representation of the union of polyhedra [Nayl90b]. The construction of the tree is based on the operation of splitting a polyhedron by a hyperplane. Thus the efficiency and robustness of the solution to the from-region visibility problem is primarily determined by the efficiency and robustness of the polyhedron splitting algorithm. The principle of the method is simple and in contrast to other analytic from-region visibility algorithms [Dret94b, Stew94, Dura97] the proposed method treats all visibility events in a unified manner.

8.1 Problem statement

This chapter addresses the following from-region visibility problems:

I. Given a polygonal scene and two polygons determine:
   (a) Are the two polygons visible ?
   (b) Which fragments of the two polygons are mutually visible ?
   (c) Visibility events between the two polygons.

II. Given a polygonal scene and a polyhedral view cell determine:
   (a) A set of potentially visible polygons with respect to the view cell.
   (b) Fragments of potentially visible polygons with respect to the view cell.
   (c) Mutually visible fragments of the visible polygons and the view cell.
   (d) Visual events with respect to the view cell.

The I-(a), I-(b), and I-(c) problems address elementary from-region visibility problems in polygonal scenes. They reflect the case of a localized visibility computation, i.e. the visibility computation that is restricted by a set of rays between the two polygons. The other three problems (II-(a), II-(b), and II-(c)) address the PVS computation. They are presented in the order of increasing accuracy of the computed result. The II-(d) problem addresses a precise computation of shadows with respect to an areal light source. The proposed method provides an exact analytic solution to all above mentioned problems.

8.2 Related work

Below we briefly discuss the related work on from-region visibility in several application areas.

8.2.1 Aspect graph

The first algorithms dealing with from-region visibility belong to the area of computer vision. The *aspect graph* [Gigu90, Plan90, Sojk95] 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 viewspace. A *scale space* aspect graph [Egge92, Shim93] improves robustness of the method by merging similar features according to the given scale.

8.2.2 Potentially visible sets

In the computer graphics community Airey [Aire90] 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. [Tell92b, Tell91] 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 [Tell92a] and a simpler and more robust conservative solution [Tell92b]. 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 [Tell92b]. Yagel and Ray [Yage95] 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 [Plan93]. The construction of viewspace partitioning was further studied by Chrysanthou et al. [Chry98b], Cohen-Or et al. [Cohe98a] and Sadagic [Sada00]. Sudarsky and Gotsman [Suda96] proposed an output-sensitive visibility algorithm for dynamic scenes. Cohen-Or et al. [Cohe98c] 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. [Dura00] and Schaufler et al. [Scha00] make use of several simplifying assumptions to avoid the usage of 4D data structures. Wang et al. [Wang98] 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 [Simi94] 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. [Wonk00], Koltun et al. [Kolt01], and Bittner et al. [Bitt01e].

The method presented in the thesis was first outlined in [Bitt99]. Recently, a similar exact algorithm for PVS computation was developed by Nirenstein et al. [Nire02]. This algorithm uses Plücker coordinates to compute visibility in shafts defined by each polygon in the scene.

8.2.3 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 [Chin92], Loscos and Drettakis [Losc97], and Chrysanthou [Chry96] and Chrysanthou and Slater [Chry97]. Discrete solutions have been proposed by Nishita [Nish85], Brotman and Badler [Brot84], and Soler and Sillion [Sole98]. An exact algorithm computing an antipenumbra of an areal light source was developed by Teller [Tell92a].

8.2.4 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 [Camp90, Camp91], Lischinski [Lisc92], and Heckbert [Heck92]. More elaborate methods were developed by Drettakis [Dret94a, Dret94b], and Stewart and Ghali [Stew93, Stew94]. 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 [Schr93]. The visibility computation is a crucial step in the form factor evaluation [Tell93b, Hain94, Tell94, Nech96, Teic99]. Similar visibility computation takes place in the scope of hierarchical radiosity algorithms [Sole96, Dret97, Daub97].

8.2.5 Global visibility

The aim of *global visibility* computations is to capture and describe visibility in the whole scene [Dura96]. 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 [Pocc93] that describes global visibility in 2D scenes. The visibility complex has been applied to solve various 2D visibility problems [Rivi95, Rivi97b, Rivi97a, Orti96]. The approach was generalized to 3D by Durand et al. [Dura96]. Nevertheless, no implementation of the 3D visibility complex is currently known. Durand et al. [Dura97] 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 [Dugu02] 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. [Chry98a] and Blais and Poulin [Blai98]. These data structures are closely related to the *lumigraph* [Gort96, Bueh01] or *light field* [Levo96]. An interesting discrete hierarchical visibility algorithm for two-dimensional scenes was developed by Hinkenjann and Müller [Hink96]. 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 [Caza97b] provide a useful estimate on the required sampling density and the size of the solution space data structure.

8.2.6 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 [Gran85]. The from-region visibility can also be used in the context of simulation of the sound propagation [Funk98]. 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.

8.3 Algorithm overview

The method presented in this chapter follows the concept introduced in Chapter 3. The algorithm incrementally constructs an *occlusion tree* for a given source polygon P_S. Visibility from a polyhedral view cell is determined by computing visibility from each face of the view cell.

The scene polygons are processed using an approximate occlusion sweep with respect to P_S. At each step the occlusion tree represents the set of lines blocked by the already processed polygons. Additionally the tree captures a complete description of visibility from P_S including all visual events that may occur when viewing the scene from P_S.

The key idea is the description of a set of lines intersecting the source polygon P_S and a given scene polygon by a 5D blocker polyhedron in Plücker coordinates. Visibility of all scene polygons is then evaluated by set theoretical operations on the corresponding polyhedra maintained by the occlusion tree.

As discussed in Chapter 2 the problem-relevant line set for the from-region visibility in 3D is four-dimensional. As we shall see later we "add" one more dimension, which allows to describe even non-linear visual events by means of hyperplanes and their intersections. To obtain the final solution the 5D subdivision is intersected with a 4D hypersurface.

The rest of the chapter is organized as follows: Section 8.4 discusses the Plücker coordinates of lines in 3D, Section 8.5 describes visibility events in polygonal scenes and their relation to the Plücker coordinates. Section 8.6 discusses the description of the set of lines intersecting a single polygon, Section 8.7 discusses lines between two polygons. Section 8.8 describes the occlusion tree, Section 8.9 presents the algorithm of its construction. Section 8.10 presents algorithms for both conservative and exact visibility tests using the occlusion tree. Section 8.11 describes several optimizations. Section 8.12 mentions possible applications of the method. Section 8.13 discusses the implementation issues. Section 8.14 summarizes the measured results. Finally, Section 8.15 concludes the chapter.

8.4 Plücker coordinates of lines

We will use a mapping that describes an oriented 3D line as a point in a projective 5D space [Bois98] by means of Plücker coordinates [Tell92b, Pell97, Yama97, 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:

$$l = \begin{pmatrix} u_x & u_y & u_z & u_w \\ v_x & v_y & v_z & v_w \end{pmatrix} \tag{8.1}$$

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

$$\begin{aligned} \boldsymbol{\pi}_l &= (\pi_{l0}, \pi_{l1}, \pi_{l2}, \pi_{l3}, \pi_{l4}, \pi_{l5}) = \\ &= (\xi_{wx}, \xi_{wy}, \xi_{wz}, \xi_{yz}, \xi_{zx}, \xi_{xy}), \end{aligned} \tag{8.2}$$

where

$$\xi_{rs} = det \begin{pmatrix} u_r & u_s \\ v_r & v_s \end{pmatrix}. \tag{8.3}$$

Substituting $u_w = 1$ and $v_w = 1$ into Eq. 8.2 enumerates to:

$$\begin{aligned} \pi_{l0} &= v_x - u_x \\ \pi_{l1} &= v_y - u_y \\ \pi_{l2} &= v_z - u_z \\ \pi_{l3} &= u_y v_z - u_z v_y \\ \pi_{l4} &= u_z v_x - u_x v_z \\ \pi_{l5} &= u_x v_y - u_y v_x \end{aligned} \tag{8.4}$$

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:

$$\begin{aligned} \boldsymbol{\omega}_l &= (\omega_{l0}, \omega_{l1}, \omega_{l2}, \omega_{l3}, \omega_{l4}, \omega_{l5}) = \\ &= (\xi_{yz}, \xi_{zx}, \xi_{xy}, \xi_{wx}, \xi_{wy}, \xi_{wz}) \end{aligned} \tag{8.5}$$

Substituting Eq. 8.4 into Eq. 8.5 we get:

$$\begin{aligned} \omega_{l0} &= \pi_{l3} \\ \omega_{l1} &= \pi_{l4} \\ \omega_{l2} &= \pi_{l5} \\ \omega_{l3} &= \pi_{l0} \\ \omega_{l4} &= \pi_{l1} \\ \omega_{l5} &= \pi_{l2} \end{aligned} \tag{8.6}$$

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:

$$\hat{\boldsymbol{\omega}}_l = \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\omega}_l \odot \boldsymbol{\pi} = 0\} \tag{8.7}$$

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

$$\begin{aligned} \hat{\boldsymbol{\omega}}_l^+ &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\omega}_l \odot \boldsymbol{\pi} \geq 0\} \\ \hat{\boldsymbol{\omega}}_l^- &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\omega}_l \odot \boldsymbol{\pi} \leq 0\} \end{aligned} \tag{8.8}$$

These definitions of Plücker coordinates and coefficients follow the "traditional" convention [Pu98]. They differ from the definitions used by Teller [Tell92b] who used a permuted order of the coordinates. The traditional convention provides an elegant interpretation of Plücker coordinates that will be discussed in Section 8.4.1.

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 $\boldsymbol{\pi}_a \times \boldsymbol{\pi}_b$:

$$\begin{aligned} side(a,b) &= \boldsymbol{\omega}_a \odot \boldsymbol{\pi}_b = \\ &= \omega_{a0}\pi_{b0} + \omega_{a1}\pi_{b1} + \omega_{a2}\pi_{b2} + \omega_{a3}\pi_{b3} + \omega_{a4}\pi_{b4} + \omega_{a5}\pi_{b5} = \\ &= \boldsymbol{\pi}_a \times \boldsymbol{\pi}_b = \\ &= \pi_{a0}\pi_{b3} + \pi_{a1}\pi_{b4} + \pi_{a2}\pi_{b5} + \pi_{a3}\pi_{b0} + \pi_{a4}\pi_{b1} + \pi_{a5}\pi_{b2} \end{aligned} \tag{8.9}$$

This relation can be interpreted with the right-hand rule (Figure 8.1). 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.

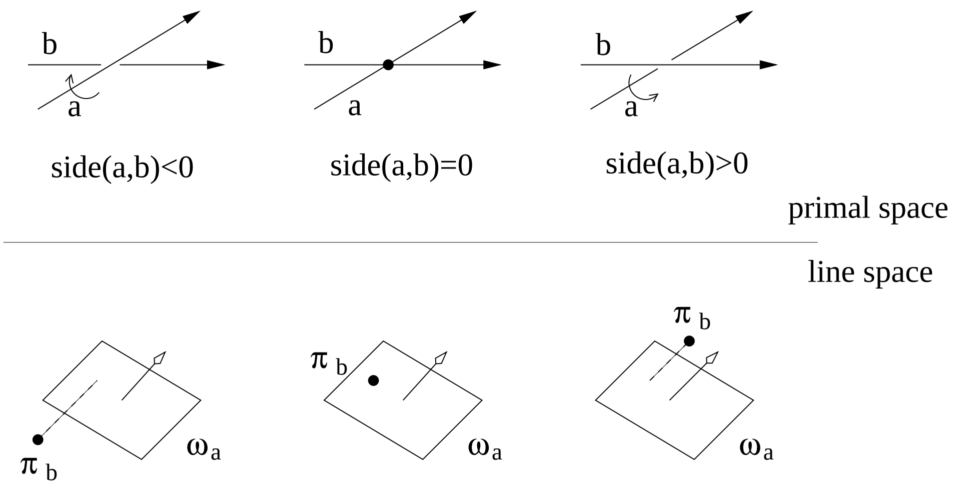


Figure 8.1: The $side(a,b)$, interpreted as the right-hand rule.

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

$$\boldsymbol{\pi} \odot \boldsymbol{\pi} = 0 \qquad \equiv \qquad \pi_0\pi_3 + \pi_1\pi_4 + \pi_2\pi_5 = 0, \tag{8.10}$$

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. 8.10 forms a 4D hyperboloid of one sheet called the *Plücker quadric*, also known as the *Klein quadric* or the *Grassman manifold* (see Figure 8.2).

The six Plücker coordinates of a real line are not independent. Firstly, they describe an oriented projective space, secondly, they must satisfy the equation 8.10. Thus there are four degrees of freedom in the description of a 3D line, which conforms with the classification from Chapter 2.

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 8.10 any line always meets itself.

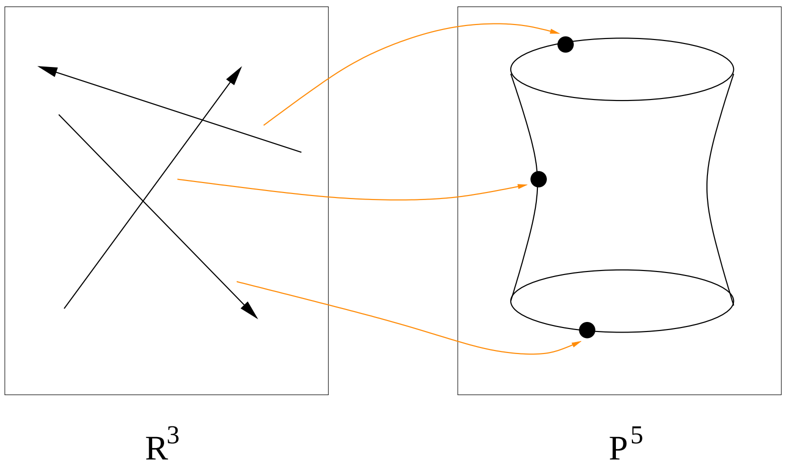


Figure 8.2: Real lines map on points on the Plücker quadric.

8.4.1 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:

$$\begin{aligned} \boldsymbol{d}_l &= (\pi_{l0}, \pi_{l1}, \pi_{l2}), \\ \boldsymbol{l}_l &= (\pi_{l3}, \pi_{l4}, \pi_{l5}), \end{aligned} \tag{8.11}$$

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:

$$\begin{aligned} \boldsymbol{\pi}_l &= [\boldsymbol{d}_l; \boldsymbol{l}_l], \\ \boldsymbol{\omega}_l &= [\boldsymbol{l}_l; \boldsymbol{d}_l]. \end{aligned} \tag{8.12}$$

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. 8.12). The anchor point $\boldsymbol{a}_l$ can be computed as:

$$\boldsymbol{a}_l = (a_x, a_y, a_z) = \frac{\boldsymbol{d}_l \times \boldsymbol{l}_l}{\| \boldsymbol{d}_l \|^2}. \tag{8.13}$$

Computing the distance

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

$$dist(l, l') = \frac{|(\boldsymbol{a}_l - \boldsymbol{a}_{l'}) \cdot (\boldsymbol{d}_l \times \boldsymbol{d}_{l'})|}{\| \boldsymbol{d}_l \times \boldsymbol{d}_{l'} \|}. \tag{8.14}$$

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

8.5 Visual events

This section discusses visual events occurring in polygonal scenes [Gigu90]. 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 [Gigu90]. 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 [Gigu90]:

- *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 8.3).

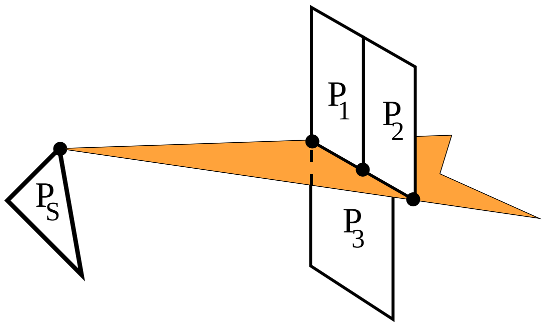


Figure 8.3: Degenerated VE event. The VE event is induced by a vertex and three edges of scene polygons.

The intersections of event surfaces correspond to *extremal lines* [Tell92b]. 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 8.4-(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 8.4-(b) depicts an 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.

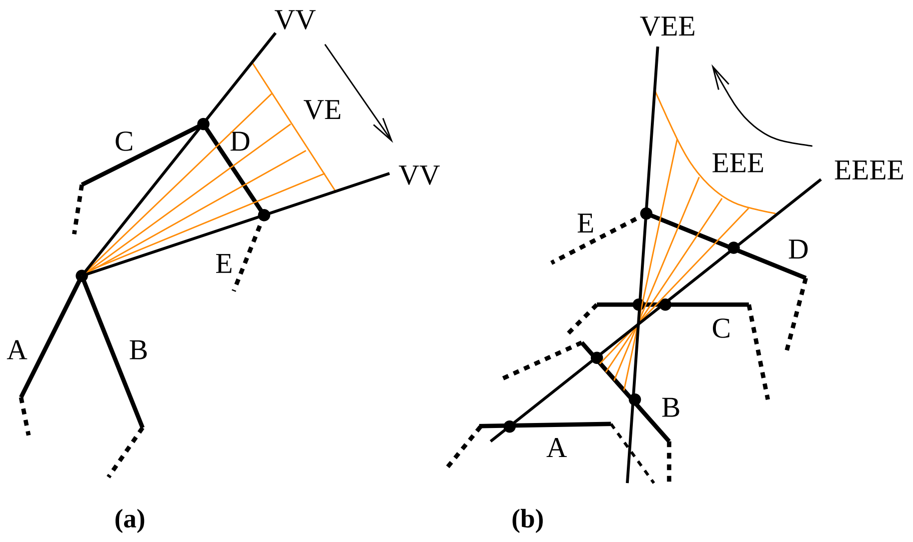


Figure 8.4: Swaths of event surfaces. (a) VE swath. (b) EEE swath.

8.5.1 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 [Good97] 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 [Tell92b, Pell97, Pu98].

An extremal line l intersects four generator edges. Consequently, the corresponding Plücker point $\hat{\pi}_l$ 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 8.4. 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 8.5).

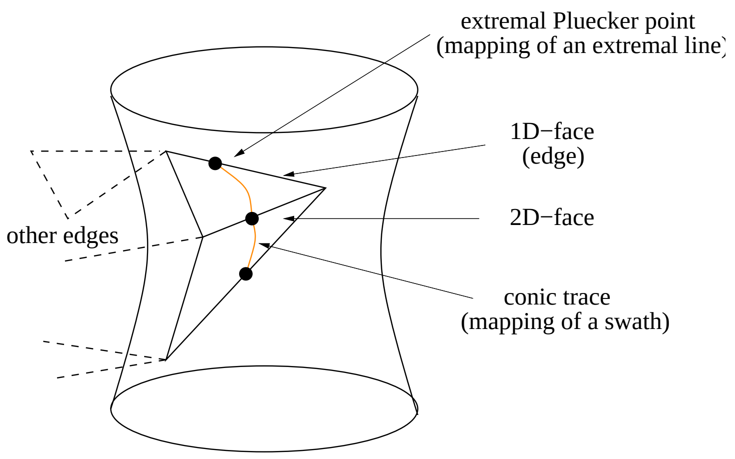


Figure 8.5: 3D swaths correspond to conics on the Plücker quadric.

8.6 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 counterclockwise. The set of lines $\mathcal{L}_P$ intersecting the polygon that are oriented in the direction of the polygon's normal satisfies:

$$\mathcal{L}_P = \{l | l \in (R^3, R^3), side(\boldsymbol{\pi}_l, \boldsymbol{\pi}_{e_i}) \leq 0, \forall i \in \langle 0, n)\}, \tag{8.15}$$

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. 8.9 and rewriting the equation in terms of a set of Plücker points we get:

$$\begin{aligned} \mathcal{F}_P &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\pi} \times \boldsymbol{\pi}_{e_i} \leq 0, \forall i \in \langle 0, n)\} = \\ &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\pi} \odot \boldsymbol{\omega}_{e_i} \leq 0, \forall i \in \langle 0, n)\}, \end{aligned} \tag{8.16}$$

where $\mathcal{F}_P$ is a set of *feasible Plücker points* for polygon P. Substituting Eq. 8.8 into 8.16 we obtain:

$$\mathcal{F}_P = \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \boldsymbol{\pi} \in \hat{\boldsymbol{\omega}}_{e_i}^-, \forall i \in \langle 0, n)\} \tag{8.17}$$

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:

$$\mathcal{S}_P = \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{F}_P, \boldsymbol{\pi} \odot \boldsymbol{\pi} = 0\}. \tag{8.18}$$

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:

$$\begin{aligned} \mathcal{F}_P^- &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{P}^5, \hat{\boldsymbol{\pi}} \in \hat{\boldsymbol{\omega}}_{e_i}^+, \forall i \in \langle 0, n)\} \\ \mathcal{S}_P^- &= \{\hat{\boldsymbol{\pi}} | \hat{\boldsymbol{\pi}} \in \mathcal{F}_P^-, \boldsymbol{\pi} \odot \boldsymbol{\pi} = 0\}. \end{aligned} \tag{8.19}$$

8.7 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:

$$\mathcal{F}_{P_S P_j} = \mathcal{F}_{P_S} \cap \mathcal{F}_{P_j} \tag{8.20}$$

and thus

$$\mathcal{S}_{P_S P_j} = \mathcal{S}_{P_S} \cap \mathcal{S}_{P_j}. \tag{8.21}$$

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 in Chapters 6 and 8 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 6.4.3). 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 $\mathcal{S}_{P_S P_j}$.

8.7.1 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. 8.10). A robust algorithm for computing this intersection was developed by Teller [Tell93a].

Intersecting all edges of the blocker polyhedron with the Plücker quadric yields all extremal lines of $\mathcal{S}_{P_S P_j}$ [Tell92a, Pu98]. See Figure 8.6 for an example of extremal lines computed for the given source polygon and a set of 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 $\mathcal{S}_{P_S P_j}$ [Tell92b]. 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

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

Figure 8.6: Extremal lines for the given source polygon (yellow) and three occluders.

extremal lines [Tell92b]. 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.

For solution of some from-region visibility problems (e.g. PVS computation, region-to-region visibility) the event swaths need not be reconstructed. For example the visibility culling algorithm that will be discussed in Section 8.12.2 only computes extremal Plücker points and uses them to test an existence of a set of stabbers of a given size.

8.7.2 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 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 8.7.1.

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 8.7.

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.

8.8 Occlusion tree

The occlusion tree for the visibility from region problem is a 5D BSP tree maintaining a collection of the 5D blocker polyhedra. 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.

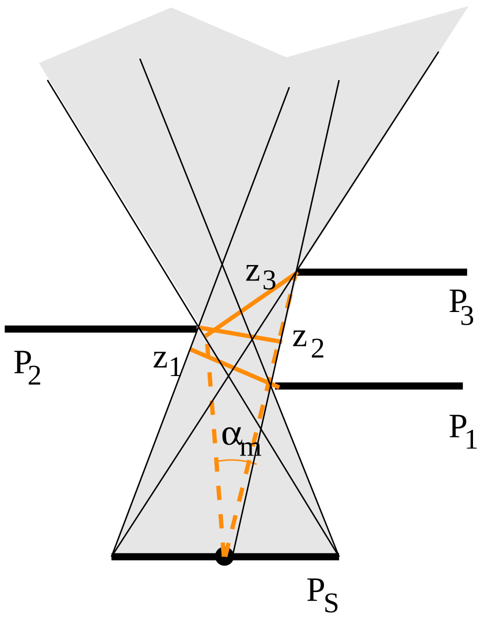


Figure 8.7: A 2D example of evaluation of the size of a set of lines. The three line segments z_1, z_2 and z_3 maximize the distance of the corresponding occluder edges from the extremal lines. The line segment z_3 spans a minimal angle α_m with respect to the center of the source polygon P_S.

8.9 Occlusion tree construction

The occlusion tree is constructed incrementally by inserting blocker polyhedra in the order given by the approximate occlusion sweep of the scene 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.

8.9.1 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, the mutual position of the currently processed polygon B_j and the polygon B_{N_c} associated with N_c is determined. This test will be described in Section 8.9.3. If B_c is behind B_{N_c} it is invisible 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.

8.9.2 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[2]. 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 [Fuku02, Avis02].

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 [Fuku96, Avis96]. 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.

8.9.3 Polygon positional test

The polygon positional test aims to determine the order of two polygons P_i and P_j with respect to the source polygon P_S [Chry96]. We assume that polygons P_i and P_j do not intersect, i.e. all potential polygon intersections were resolved in preprocessing. The test is applied to determine which of the two polygons is closer to P_S with respect to the given set of rays intersecting both polygons. The test proceeds as follows:

[2]Such a traversal was also used in Section 4.10 in the context of the from-point visibility culling.

1. If P_i lays in the positive/negative halfspace defined by P_j, it is before/behind P_j. Otherwise, proceed with the next step.
2. If P_j lays in the positive/negative halfspace defined by P_i, it is before/behind P_i. Otherwise, proceed with the next step.
3. Find a ray from the given set of rays that intersects both polygons. Compute an intersection of the ray with P_i and P_j and determine the position of the polygons according to the order of intersection points along the ray.

The first two steps aim to determine the absolute priority of one of the polygons. If these steps fail, the order of the polygons is determined using a sample ray in step 3.

8.10 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.

8.10.1 Exact visibility test

The exact visibility 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 8.9, 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 is computed using a combination of visibility states of its boundary faces according to Table 3.1.

8.10.2 Conservative visibility test

The conservative visibility test provides a fast estimation of visibility of the given region since it does not require the 5D polyhedra enumeration. Visibility of the given face of the region is determined by a traversal of the occlusion tree and testing the position of the corresponding blocker polyhedron with respect to the encountered hyperplanes as described in Section 8.9.2. If the blocker polyhedron reaches only in-leaves and the face is behind the polygon(s) associated with the leaves, the face is classified invisible . Otherwise, it is conservatively classified as visible. The visibility of the whole region is determined using a combination of visibility of its faces as mentioned in the previous section.

8.11 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.

8.11.1 Shaft culling

The algorithm as described computes and maintains visibility of all polygons facing the given source polygon. In complex scenes the description of visibility from the source polygon can reach $O(n^4)$ complexity in the worse case [Tell92b, Pell97, Dura99]. The size of the occlusion tree is then bound by $O(n^5)$. Although such a configuration occurs rather rare in practice we need to apply some general technique to avoid the worst-case memory complexity. If the algorithm should provide an exact analytic solution to the problem, we cannot avoid the $O(n^4 \log n)$ worst-case computational complexity, but the computation can be decoupled to reduce memory requirements.

We propose to use the *shaft culling* [Hain94] method that divides the computation into series of from-region visibility problems restricted by a given shaft of lines. Ideally the shafts are determined so that the *visibility complexity* in the shaft is bound by a given constant. This is however the from-region visibility problem itself. We can use an estimate based on a number of polygons in the given shaft. First, the hemicube is erected over the source polygon and it is used to construct five shafts corresponding to the five faces of the hemicube. The shafts are then subdivided as follows: if a given shaft intersects more than a predefined number of polygons the corresponding hemicube face is split in two and the new shafts are processed recursively. Visibility in the shaft is evaluated using all polygons intersecting the shaft. When the computation for the shaft is finished the occlusion tree is destroyed. The algorithm then proceeds with the next shaft until all generated shafts have been processed. See Figure 8.8 for an example of a regular subdivision of the problem-relevant line set into 80 shafts.

Figure 8.8: An example of the shaft culling for from-region visibility. Visibility computation is carried out in each of the 80 shafts defined by the source polygon (the yellow rectangle) and the rectangles on the hemicube erected over the source polygon.

This technique shares a similarity with the algorithm recently published by Nirenstein et al. [Nire02] that uses shafts between the source polygon and each polygon in the scene. Our technique provides the following benefits:

- The shaft can contain thousands of polygons, which allows to better exploit the coherence of visibility. The hierarchical line space subdivision allows efficient searches and updates.
- The method is applicable even in scenes with big differences in polygon sizes. Unlike the method of Nirenstein et al. [Nire02], the proposed technique generates shafts to find balance between the use of coherence and the memory requirements. Due to the hierarchical subdivision of the problem-relevant line set our method can handle the case when there is a huge number of polygons in a single shaft between two large scene polygons.
- Visibility in the whole shaft is described in a unified data structure, which can serve for further computations (e.g. extraction of event surfaces, hierarchical representation of visibility, etc.).

8.11.2 Occluder sorting

Occluder sorting aims to increase the accuracy of the front-to-back ordering determined by the approximate occlusion sweep. Higher accuracy of the ordering decreases the number of the late merging of blocker polyhedra in the tree. Recall that the merging extends the occlusion tree by a blocker polyhedron corresponding to an occluder that hides an already processed occluder.

The occluder sorting is applied when reaching a leaf node of the spatial hierarchy during the approximate occlusion sweep. Given a leaf node N the occluder sorting proceeds as follows:

1. Determine a ray r piercing the center of the source polygon P_S and the node N.
2. Compute intersections of r and all supporting planes of the polygons associated with N.
3. Sort the polygons according to the front-to-back order of the corresponding intersection points along r.

The occluder sorting provides an exact priority order within the given leaf in the case that the polygons are pierced by the computed ray r and they do not intersect.

8.11.3 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.

8.11.4 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.

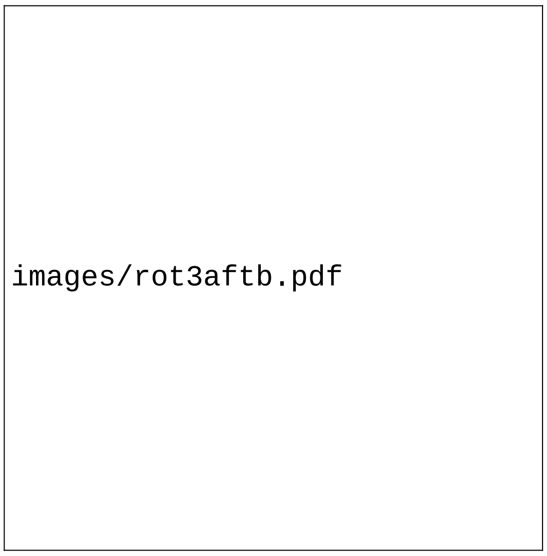
The algorithm classifies an interior node of the occlusion tree as occluded (in) if the following conditions hold:

- 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.

8.11.5 Hierarchical visibility

As described in the previous chapters the hierarchical visibility tests applied on the nodes of the spatial hierarchy can be used to cull whole sets of invisible polygons. A hierarchical visibility test is applied using a bounding box of a node of the spatial hierarchy. Visibility of the box is determined by testing visibility of its boundary faces. See Figure 8.9 for an illustration of the hierarchical visibility tests.



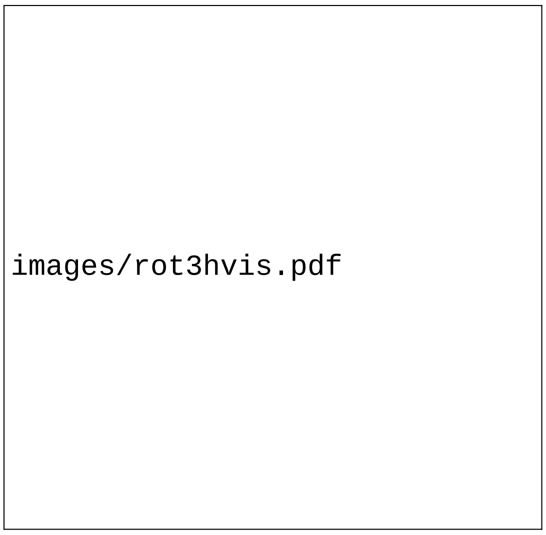


Figure 8.9: Illustration of the hierarchical visibility culling in a scene with 10,000 random triangles. (left) Polygons in the selected shaft that were processed by the visibility algorithm. The wireframe polygons were then found invisible. (right) Cells of the kD-tree culled by the hierarchical visibility tests. Green nodes were outside of the given shaft, blue nodes were classified invisible due to the already processed occluders.

8.12 Applications

The proposed method can be used in the context of various applications of from-region visibility. The occlusion tree captures a complete description of visibility from the given source polygon. Depending on the application the visibility information stored in the tree is either refined (the topology of visibility changes is extracted) or simplified (a PVS is generated).

8.12.1 Discontinuity meshing

We briefly discuss the application of the method to discontinuity meshing, which illustrates the relation of the structure of the line space subdivision to the visibility events in primal space.

The construction of the discontinuity mesh using the occlusion tree proceeds similarly to the construction of the visibility map described in Chapter 5. The occlusion tree for a given polygonal light source is constructed using all polygons in the scene. The fragments of the blocker polyhedra stored in in-leaves of the tree correspond to sets of rays through which the associated polygons are visible. The intersection of the 2D-faces of the polyhedra with the Plücker quadric then corresponds to a superset of all event surfaces with respect to the light source.

Vertices of the polyhedra correspond to the extremal lines that induce a superset of vertices of the discontinuity mesh (DM). Some extremal lines correspond to the *apparent* DM vertices. These vertices occur due to the “global impact” of the splitting hyperplanes used for binary space partitioning of line space. To determine if an extremal line corresponds to an apparent vertex we test it for an intersection with polygon edges (line segments) defining the corresponding extremal line (recall that at least four polygon edges, i.e. Plücker hyperplanes are associated with an extremal line). If the extremal line intersects at least four polygon edges it is a real extremal line inducing a real vertex of the discontinuity mesh. Otherwise it is an apparent extremal line corresponding to an apparent vertex of the discontinuity mesh.

The intersections of the 2D-faces of the polyhedra with the Plücker quadric correspond to the superset of swaths of event surfaces. The computed swath induces a real visual event, if it separates regions where a change in the topology of the backprojection of the light source occurs. In line space this means that the polyhedra fragments sharing the given 2D-face are associated with different scene polygons. Otherwise the swath induces an apparent visual event that is analogical to the BSP edge discussed in Section 5.6.3.

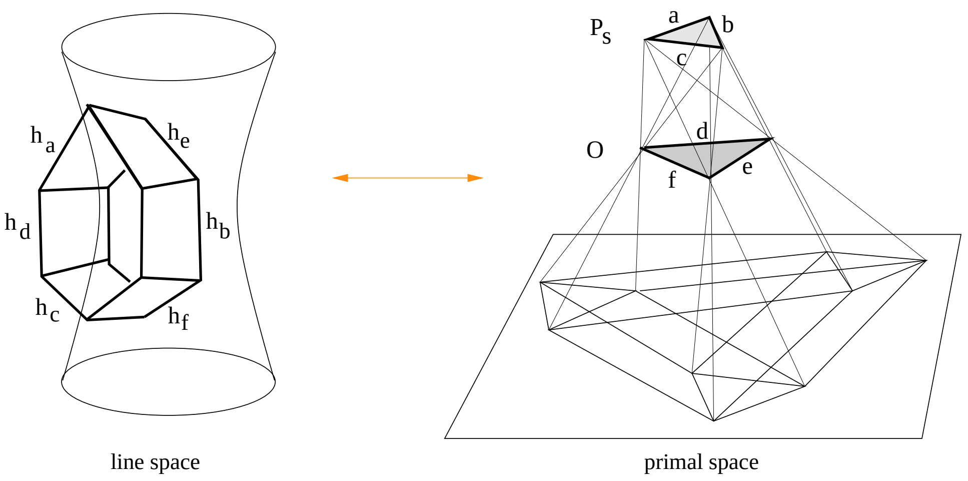


Figure 8.10: The intersection of the blocker polyhedron and the Plücker quadric yields all extremal lines between a source polygon and an occluder. The intersection of the 2D-faces of the blocker polygon with the Plücker quadric corresponds to the swaths of event surfaces. These swaths induce edges of the discontinuity mesh.

8.12.2 Visibility culling

The algorithm presented above is well suited for visibility culling, namely computing a PVS with respect to the given view cell. The occlusion tree is constructed with respect to all boundary faces of the given view cell using all occluders in the scene. The occlusion tree can be used to determine the PVS as a union of all occluders associated with in-leaves. In the case that the occluders do not directly correspond to the scene objects the occlusion tree is used to test visibility of the objects as described in Section 8.10.

Since we are interested only in the set of visible objects/polygons we can reduce the memory requirements of the algorithm by deallocating unnecessary data structures as soon as possible during the computation. For example we can deallocate the blocker polyhedra as soon as we determine its intersection with the Plücker quadric and evaluate the size measure. If we use the shaft culling technique described in Section 8.11.1 we can deallocate the whole occlusion tree as soon as we sweep all polygons in the given shaft.

8.12.3 Occluder synthesis

The occlusion tree captures a complete information about visibility and occlusion from the given source polygon P_S. The tree can be seen as a from-region visibility map with a depth information. By extracting and simplifying the information stored in the tree we can construct virtual occluders valid for viewing the scene from P_S. The virtual occluders can be computed by extracting *silhouette edges* of the set of occluders. A silhouette edge is an edge that defines a boundary between the sets of occluded and unoccluded rays. The silhouette edges describe a virtual occluder resulting from the fusion of possibly a large number of small occluders at different depths. The depth of the virtual occluder can be described using a coarse depth subdivision. In the simplest case the virtual occluder can be constructed for a depth corresponding to a plane parallel to the P_S at a specified distance.

8.13 Implementation

The proposed method was implemented in C++ under Linux. The implementation consists of operations with the Plücker coordinates, 5D BSP tree construction and maintenance, 5D polyhedra splitting and enumeration and supporting routines such as shaft culling, kD-tree construction etc.

The crucial part of the implementation is the enumeration of the 5D polyhedron described as an intersection of 5D halfspaces. We evaluated three different implementations of this algorithm: the CDD library of Fukuda [Fuku02], the LRS library of Avis [Avis02], and the floating point implementation of the reverse search algorithm [Avis96, Bitt97]. The CDD and LRS libraries use a multi-precision number representation for accurate computations on rational numbers.

The polyhedron enumeration algorithms can also be used to perform the polyhedron splitting by adding a halfspace corresponding to the splitting plane and enumerating the resulting polyhedron(s): The fragment on the negative/positive side of the splitting planes is obtained by adding the negative/positive halfspace to the list of halfspaces defining the polyhedron and invoking the polyhedron enumeration algorithm.

Additionally we implemented the polyhedron splitting algorithm of Bajaj and Pascucci [Baja96]. This algorithm uses a polyhedron lattice to classify positions of polyhedra faces with respect to the splitting plane and to resolve possible inconsistencies in the classification. The implementation algorithm however exhibited problems with numerical stability in some degenerate cases and thus it was not used for further evaluation of the method. However, a robust efficient implementation of the direct polyhedron splitting could substantially improve the total running time of the visibility algorithm.

After evaluating the advantages and disadvantages of the different implementations of the polyhedron enumeration and splitting we selected our floating point implementation of the reverse search as a core for all further measurements [Bitt97]. The drawback of the method is that the floating point implementation is prone to numerical inaccuracies. The advantage is the speed of the method: it is about 50 times faster than the methods based on multi-precision rational numbers.

In the implementation we treat numerical inaccuracies by normalizing the input of the algorithm (set of linear inequalities corresponding to the halfspaces bounding the polyhedron) and carefully using an epsilon-threshold for number comparisons. On the application level the inaccuracies are treated by using the size measure for the computed sets of lines (see Section 8.7.2).

8.14 Results

The implementation of the proposed algorithms was evaluated on a portable PC with 1GHz Pentium III and 384MB RAM under Linux operating system. We measured the behavior of the algorithm on the scenes consisting of random triangles, structured scenes consisting of regular patterns of quads, and a real-world scene representing a part of the city of Graz.

8.14.1 Random triangles

The scenes consisting of random triangles were used to study the behavior of the algorithm with respect to the number of triangles, their size and the size of the source polygon. For all measurements we used the decomposition of the computation into 20 shafts. Hierarchical visibility tests were applied only on kD-tree nodes containing more than 200 polygons. The measured results are summarized in Table 8.1, two of the tested configurations are depicted in Figure 8.11.

Figure 8.11: Computing PVS in scenes consisting of random triangles. (left) Test R1-a: 100 big triangles and a small source polygon. (right) Test R9-d: 30000 small triangles and a larger source polygon.

The results show that the method is sensitive to the area of the source polygon, the number of visible triangles, and the configuration (density) of visible triangles. The greater the area of the source polygon the larger is the problem-relevant line set $\mathcal{L}_R$ and consequently the more complex is the line space subdivision maintained by the occlusion tree. The size of the occlusion tree is given by the number of visible triangles and the structure of their *visibility interactions*: a visibility interaction of two triangles is reflected as an intersection of the corresponding blocker polyhedra in line space.

We can observe that the method is not very sensitive to the number of triangles, but rather to their configuration. This can be seen at the $R8$ and $R9$ tests that correspond to small triangles and a comparatively larger source polygon. In these tests increasing the number of triangles increased the amount of occlusion due to the occluder fusion. Consequently, less triangles were visible and the size of the occlusion tree as well as the computational time were decreased.

Figure 8.12 depicts the dependence of the average size of the occlusion tree per shaft and the running time of the algorithm on the number of visible triangles for R1, R4, and R7 tests. These tests correspond to a small source polygon and different sizes of the scene triangles. We can observe that decreasing the size of the triangles slightly increases the average size of the occlusion tree. The total running time of the algorithm exhibits a slower growth with respect to the number of visible triangles. This behavior can be explained by the fact that testing visibility of a small triangle is more efficient than testing visibility of a larger one. A smaller triangle induces a smaller blocker polyhedron that reaches less leaves during the traversal of the occlusion tree.

8.14.2 Structured scenes

The structured scenes were used to study the behavior in cases when the visible part of the scene remains constant. We used two types of test scenes. The first type (used in tests S1–S3) were scenes consisting of

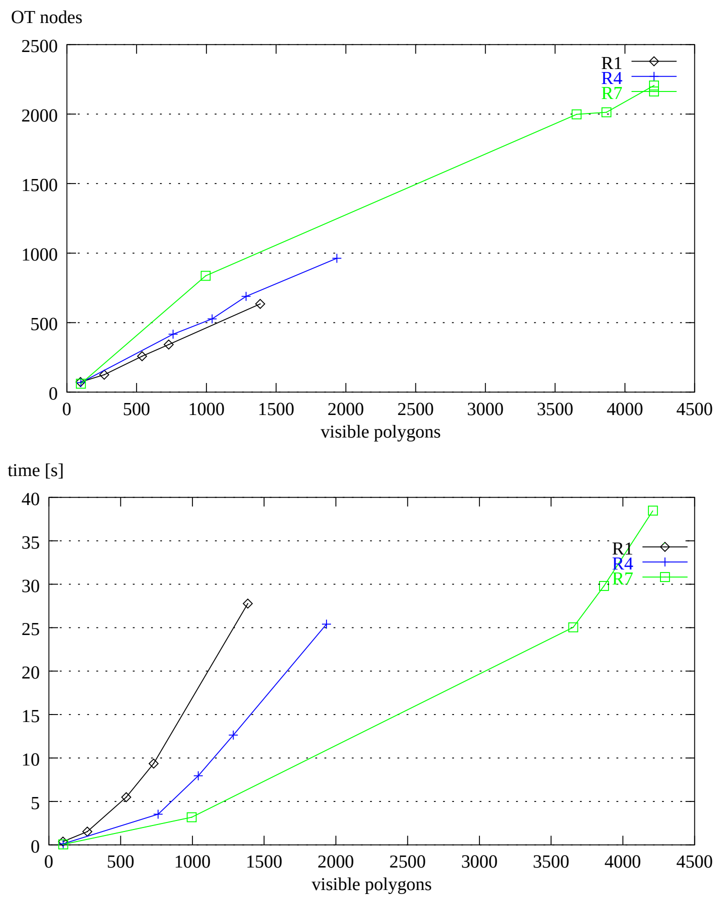


Figure 8.12: The dependence of the size of the average size of the occlusion tree and the running time of the algorithm on the number of visible triangles for R1, R4, and R7 tests.

several layers of “walls” where each wall consisted of 20x20 quads. The second type (used in tests S4–S6) were scenes with layers of quads aligned in a chessboard pattern (see Figure 8.13). The tests were applied using different size of the source polygon and different number of quad layers (20,100,1000). The results are summarized in Table 8.2.

We can observe that the increase of the number of quads in the scene has practically no influence on the running time of the method. Paradoxically, in some cases (tests S1 and S4) the running time even decreased for scenes with more quads. This behavior can be explained by the fact that the structure of the kD-tree was more efficient for the particular configuration of the source polygon and the quads. In other words the hierarchical visibility tests culled more nodes at the higher level of the spatial hierarchy than for the scenes with less quads.

Increasing the size of the source polygon always led to the increase of the size of the occlusion tree and the running time, although the number of visible quads remained constant (tests S1–S3) or slightly increased (tests S4–S6). This behavior can be explained by the increase of the complexity of the line space subdivision in the regions of line space that do not correspond to real lines, i.e. the regions in the neighborhood of the Plücker quadric. The arrangement of the blocker polyhedra is more complex although its intersection with the Plücker quadric remains constant.

images/rot3_filled.pdf

images/rot3_check.pdf

Figure 8.13: Computing PVS in scenes consisting of regularly structured quads. (left) The configuration used in test S3-a. (right) The configuration used in test S6-a.

8.14.3 A real-world scene

For evaluation of the method on a real-world scene we used a model of the city of Graz[3] depicted in Figure 8.14. The model consists of 12235 triangles. Some triangles represent simplified building facades (typically two triangles per facade), the other triangles represent the streets, parks and the terrain. The constructed kD-tree consisted of 17167 nodes.

images/graz_top1.pdf

Figure 8.14: An overview of the Graz scene.

We computed a PVS for 100 randomly selected view cells. The view cells were defined as prisms with a triangular base erected 2 meters over the terrain. The PVS computation was carried out by computing a PVS for each vertical face and a top face of a given view cell. For all view cells the computation was initiated to use a single shaft per view cell face, the hierarchical visibility tests were applied on nodes associated with more than 100 polygons. The results summarized in Table 8.3.

The algorithm computed a PVS that consisted of 1% of scene objects on average, the hierarchical visibility tests culled 93% of polygons per view cell face. The computation took 23s per viewcell on average. Figure 8.15 depicts the plots of the PVS size and the running time for each view cell in a logarithmic scale.

We can observe that the computation times for several view cells (number 20, 30, and 40) were substantially higher than for the other ones, although the size of the PVS for these view cells is only slightly bigger. By studying the PVS for these view cells we could see that these cases correspond to a degenerate configuration of the view cell and the occluders, in which the viewcell intersects the occluders or

[3]Courtesy of VRVis Graz.

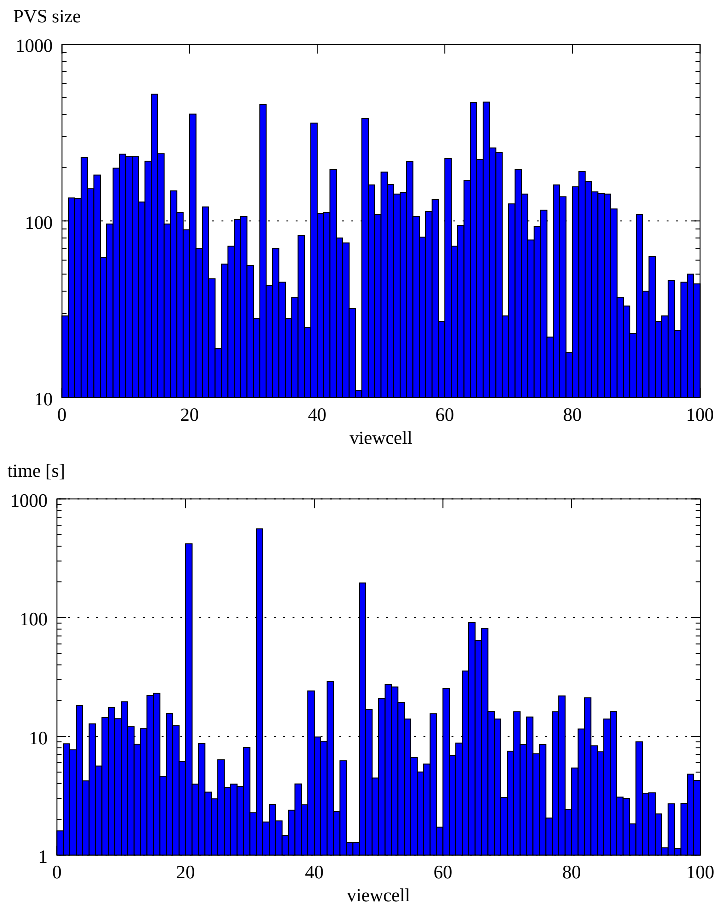


Figure 8.15: Plots of the results for the Graz scene. The plots show the size of the PVS and the computation time for each view cell in logarithmic scale.

there are little gaps between the occluders in the model (see Figure 8.16). Such a configuration led to a significantly bigger occlusion tree and consequently the computation was automatically restarted using 36 shafts to avoid restrictive memory consumption. An a priori handling of such a degenerate cases in the implementation of the method remains a topic for future work.

8.14.4 Silhouette edges extraction

Finally, we have tested the algorithm for extracting silhouette edges with respect to the source polygon outlined in Section 8.12.3. The algorithm computed silhouette edges with respect to a rectangular source polygon in a scene consisting of 10000 randomly generated triangles. Recall that a silhouette edge is an edge that defines a boundary between a set of occluded and unoccluded rays emerging from the source polygon. The result is depicted in Figure 8.17.

There were 521 silhouette edges near the boundary of the cluster of triangles when viewed from the source polygon. Note that due to the occluder fusion, there are no silhouette edges in the center of the cluster. The result illustrates that the silhouette extraction algorithm is a good candidate for a synthesis of virtual occluders in scenes containing many spatially disconnected occluders.

8.15 Summary

This chapter presented a new method for computing from-region visibility in polygonal scenes. The method is based on the concept of line space subdivision, approximate occlusion sweep and hierarchical visibility tests. 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 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. The chapter presented 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 chapter discussed three possible applications of the method: discontinuity meshing, visibility culling, and occluder synthesis.

The implementation of the method was evaluated on scenes consisting of randomly generated triangles, structured scenes, and a real-world urban model. The evaluation was focused on the application method for PVS computation. By computing a PVS in a model of the city of Graz it was shown that the approach is suitable for visibility preprocessing in urban scenes. The principal advantage of the method 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.

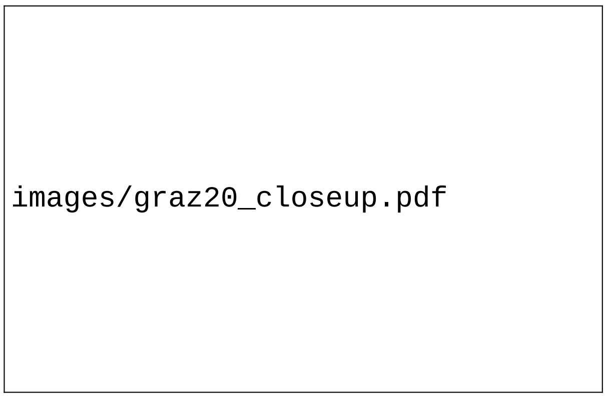


Figure 8.16: Inconsistencies in the input data. (left) A PVS computed for a view cell that intersects the occluders (view cell no. 20 in the test). (right) A closeup of the view cell. We can also see gaps between the occluder walls and the terrain.

Figure 8.17: Extraction of silhouette edges in a scene consisting of 10000 random triangles. The figure depicts a source polygon, the visible triangles (shown in color), the processed but invisible triangles (wire-frame) and the computed silhouette edges (red). There were 1300 visible triangles and 521 silhouette edges. The computation took 40 seconds.

Test	source area $[m^2]$	polygon area $[m^2]$	polygons $[-]$	kD nodes [-]	avg. processed polygons [-]	avg. OT nodes [-]	visible polygons [-]	time [s]
	2.00	200	100	25	9	73	98	0.38
	2.00	200	1000	1075	84	125	268	1.52
R1	2.00	200	10000	18355	421	258	539	5.49
	2.00	200	30000	34459	868	342	730	9.36
	2.00	200	100000	48733	2614	635	1386	27.77
	50.00	200	100	25	9	100	99	0.53
	50.00	200	1000	1075	90	189	296	2.48
R2	50.00	200	10000	18355	486	391	582	8.06
	50.00	200	30000	34459	1005	589	805	15.21
	50.00	200	100000	48733	2988	1139	1578	47.99
	200.00	200	100	25	10	148	100	0.85
	200.00	200	1000	1075	101	307	334	4.48
R3	200.00	200	10000	18355	568	650	617	13.98
	200.00	200	30000	34459	1176	974	876	25.23
	200.00	200	100000	48733	3479	2315	1811	96.23
	2.00	50	100	21	7	68	100	0.12
	2.00	50	1000	329	74	417	761	3.53
R4	2.00	50	10000	9479	379	527	1041	7.96
	2.00	50	30000	31293	644	689	1285	12.64
	2.00	50	100000	56619	1564	963	1935	25.40
	50.00	50	100	21	8	99	100	0.20
	50.00	50	1000	329	80	993	910	10.06
R5	50.00	50	10000	9479	460	930	1166	15.60
	50.00	50	30000	31293	781	1180	1383	22.88
	50.00	50	100000	56619	1933	1722	2130	46.46
	200.00	50	100	21	9	152	100	0.34
	200.00	50	1000	329	88	2442	975	31.97
R6	200.00	50	10000	9479	573	1746	1296	34.85
	200.00	50	30000	31293	969	2178	1492	47.70
	200.00	50	100000	56619	2404	3480	2293	103.08
	2.00	12.5	100	21	6	60	100	0.06
	2.00	12.5	1000	231	64	837	995	3.18
R7	2.00	12.5	10000	3913	605	1998	3654	25.04
	2.00	12.5	30000	14919	787	2013	3868	29.81
	2.00	12.5	100000	42929	1325	2206	4209	38.48
	50.00	12.5	100	21	7	85	100	0.08
	50.00	12.5	1000	231	70	2452	1000	13.81
R8	50.00	12.5	10000	3913	703	6953	5276	131.37
	50.00	12.5	30000	14919	1120	4802	4757	93.98
	50.00	12.5	100000	42929	1840	4431	4663	95.44
	200.00	12.5	100	21	8	131	100	0.14
	200.00	12.5	1000	231	80	6450	1000	64.00
R9	200.00	12.5	10000	3913	791	22086	6730	846.57
	200.00	12.5	30000	14919	1553	12187	5758	419.70
	200.00	12.5	100000	42929	2458	9490	5196	293.98

Table 8.1: Results for randomly generated triangles. The table shows the name of the test, the area of the source polygon, the area of each triangle, the number of triangles, the number of kD-tree nodes, the average number of processed triangles per shaft, the average number of OT nodes per shaft, the total number of visible triangles (i.e. the size of the PVS), and the total running time of the visibility algorithm.

Test	source area $[m^2]$	polygon area $[m^2]$	polygons $[-]$	kD nodes [-]	avg. processed polygons [-]	avg. OT nodes [-]	visible polygons [-]	time [s]
S1	2.00	4.0804	8000	31459	1799	1737	400	0.88
	2.00	4.0804	40000	64067	1893	1751	400	0.90
	2.00	4.0804	400000	104691	1200	1801	400	0.60
S2	50.00	4.0804	8000	31459	1799	1829	400	1.20
	50.00	4.0804	40000	64067	1923	1979	400	1.34
	50.00	4.0804	400000	104691	1370	2531	400	1.36
S3	200.00	4.0804	8000	31459	1808	1927	400	1.63
	200.00	4.0804	40000	64067	1923	2373	400	2.11
	200.00	4.0804	400000	104691	1370	3539	400	2.47
S4	2.00	4.0804	4000	17777	1792	7325	924	3.66
	2.00	4.0804	20000	43083	1716	7451	924	3.76
	2.00	4.0804	200000	83801	1990	7459	924	3.64
S5	50.00	4.0804	4000	17777	2138	11481	1132	7.65
	50.00	4.0804	20000	43083	2337	11605	1132	8.39
	50.00	4.0804	200000	83801	2545	11613	1132	8.77
S6	200.00	4.0804	4000	17777	2866	19917	1430	28.91
	200.00	4.0804	20000	43083	2831	20233	1430	28.98
	200.00	4.0804	200000	83801	3001	20251	1430	31.86

Table 8.2: Results for quads aligned in a regular pattern. The table contains the name of the test, the area of the source polygon, the area of each quad, the number of triangles, the number of kD-tree nodes, the average number of processed quads per shaft, the average number of OT nodes per shaft, the total number of visible quads (i.e. the size of the PVS), and the total running time of the visibility algorithm.

avg. processed polygons [-]	avg. OT nodes [-]	avg. visible polygons [-]	avg. time [s]
855.98	2400.89	132.47	23.02

Table 8.3: Results for the Graz scene. The table depicts the average number of processed polygons per shaft, the average number of occlusion tree nodes per shaft, the average number of visible polygons per view cell (i.e. the size of the PVS) and the average computational time per view cell.

Chapter 9

Conclusions

This chapter summarizes the results of the thesis and discusses possible directions for future work.

9.1 Summary of results

The introductory part of the thesis provides a taxonomy of visibility problems and algorithms, and discusses important steps in the design of a visibility algorithm. The taxonomy is based on restricting the domain of visibility problems and specifying the type of the answer to the problem. The domain of a visibility problem is characterized by the dimension of the problem-relevant line set, which groups together problems of similar complexity independently of their application area. The dimension of the problem-relevant line set $d(\mathcal{L}_R)$ varies from 0 to 4. The classification highlighted the fact the from-region visibility in 2D and the from-point visibility in 3D involve problem-relevant line set of dimension $d(\mathcal{L}_R) = 2$.

The answers to visibility problems are divided in three classes: visibility classification, subset of the input, and the constructed data structure. The type of the answer restricts the domain of the output of an algorithm solving a particular visibility problem. Visibility problems with a "coarser" type of answer (e.g. visibility classification) are simpler to solve than problems requiring a precise description of visibility.

The taxonomy helps to understand the nature of the given problem and assists in finding relationships between visibility problems and algorithms in different application areas. It should also help algorithm designers to transfer existing concepts for solving other visibility problems.

9.1.1 The concept of a visibility algorithm

Chapter 3 presented a general concept of a visibility algorithm suitable for several visibility problems. The concept builds on the top of existing from-point visibility algorithms and it generalizes their ideas for solving other visibility problems. It aims to capture the essence of efficient solution to visibility problems localized to a given point or a region. Solving different visibility problems using the same concept allows to share the experience with the behavior of the method and reuse parts of the implementation.

The concept is based on three main ideas: the *approximate occlusion sweep*, the *occlusion tree*, and *hierarchical visibility tests*. The approximate occlusion sweep is used to generate an approximate depth order of scene polygons. It represents a compromise between a computationally costly strict ordering of the scene polygons and a fast ordering that need not be supported by additional data structures and calculations. The occlusion sweep is used to construct the occlusion tree representing a subdivision of the problem-relevant line set into sets of occluded and unoccluded lines. The occlusion tree serves as an abstraction layer making the approach applicable to several visibility problems. The tree maintains a hierarchical binary space partitioning of the problem-relevant line set, which provides the following benefits:

- The occlusion tree is applicable to problem-relevant line sets of different dimensions.
- The occlusion tree is adaptable to the input data. It reflects the boundaries of visibility events and consequently it provides a coherent representation of visibility.
- The hierarchical structure of the occlusion tree provides efficient searches and updates.
- The occlusion provides a consistent representation of visibility, what improves the robustness of the method.
- The construction of the occlusion tree requires only a few geometrical operations. In particular the tree is constructed by partitioning of a polyhedron by a plane in the dimension given by dimension the problem-relevant line set.

The occlusion tree can be used to test visibility of an object or a whole spatial region with respect to the already processed polygons. The hierarchical visibility tests complement the proposed concept by providing a general means of achieving output-sensitivity of the visibility algorithm. If applied on the nodes of the spatial hierarchy these tests allow to quickly cull whole groups of invisible scene polygons. The proposed concept exploits coherence of visibility by using two hierarchical structures: the occlusion tree for partitioning of the problem-relevant line set and the spatial hierarchy for organizing the scene.

In the next sections we summarize the results of applications of the concept discussed in Chapters 4, 5, 6, 7 and 8.

9.1.2 From-point visibility culling

Chapter 4 presented an application of the proposed concept to the from-point visibility culling. The algorithm is designed to accelerate real-time rendering of large densely occluded scenes with large occluders.

The occlusion tree is constructed using selected occluders close to the viewpoint. Then the tree is used to classify visibility of nodes of the spatial hierarchy. We have described the algorithm for constructing the occlusion and three algorithms for visibility tests. The first two algorithms determine an exact visibility of a polygon and a polyhedron with respect to the selected occluders. The third one is a fast conservative visibility test suitable for regions of a general shape.

Unlike previous continuous visibility culling methods [Huds97, Coor97] the proposed technique accounts for occluder fusion and it is suitable for the use with a higher number of occluders. In complex densely occluded scenes the occluder fusion is very important since it can significantly restrict the size of the PVS computed for the given viewpoint. The lower sensitivity of the method to the number of selected occluders is given by the hierarchical structure of the occlusion tree. In practice it is difficult to select an optimal set of occluders since the occluder selection is a visible surface problem itself. Thus the lower sensitivity of the algorithm to the number of selected occluders significantly improves its practical usability.

The proposed method was evaluated on walkthroughs of complex architectural scenes. The results indicate that the conservative visibility test performs superior compared to the exact one for a typical walkthrough scenario. The savings in rendering time achieved for the tested scenes provided speedup between 1.8 and 3.8.

The second part of Chapter 4 presented a series of modifications of the classical hierarchical visibility culling aiming at exploiting temporal and spatial coherence. The proposed modifications are applicable to most existing hierarchical visibility culling algorithms. The modifications assume a subsequent application of a visibility test on nodes of a spatial hierarchy in the scope of a scene walkthrough.

The hierarchy updating proved to perform well in practice as it saves almost half of the visibility tests that would have to be applied using the classical approach. The conservative hierarchy updating improved the overall frame time for certain settings. The visibility propagation saves only few visibility tests. This however documents that the spatial coherence is already exploited well in the classical approach.

9.1.3 Visibility maps

Chapter 5 presented an algorithm that constructs a visibility map for a given view of the scene. The advantage of the method is that it does not rely on a single projection plane and easily handles views that span the whole spatial angle. The visibility map is constructed by a two stage algorithm: the construction of a hierarchical representation of the view and its postprocessing. The occlusion tree can be used to semantically classify elements of the visibility map, which helps to understand the structure of the given view. The implementation of the technique was evaluated on several non trivial scenes.

The implementation proved the applicability of the proposed concept to the visible surface problem in 3D. Additionally, it showed that the structure of the occlusion tree can be used to extract various additional information about visibility that can potentially serve as a starting point for another visibility algorithm (e.g. discontinuity meshing, occluder synthesis).

9.1.4 From-region visibility in 2D

Chapter 6 described an algorithm that computes visibility from a given region in a 2D scene. The algorithm is exact and it accounts for all types of occluder fusion. The method transforms the problem from primal space to line space. Visibility is solved by hierarchical partitioning of line space maintained by the occlusion tree. The algorithm requires an implementation of only a few geometrical algorithms, namely partitioning of a polygon by a plane. The robustness of the method is thus determined by the robustness of the polygon splitting algorithm.

The proposed method was tested using randomly generated scenes as well as a real-world scene representing a footprint of a large part of a city. The results indicate that the method is suitable for the 2D visibility preprocessing of large scenes and exhibits output-sensitive behavior in practice. An application of the 2D method in the 3D computer graphics is possible by transforming the input data to the plane and a correct interpretation of the results. For example we can preprocess visibility in a building interior by using a (conservative) ground plan of the given building floor.

9.1.5 From-region visibility in $2\frac{1}{2}$D

Chapter 7 presented two methods for computing from-region visibility in $2\frac{1}{2}$D scenes. Both methods are targeted at the application in urban scenes. They build on the observation that the majority of visibility interactions can be described by solving a 2D visibility problem. The first method combines an exact solution to the 2D visibility problem with a tight conservative solution for the remaining “half dimension”. The second method provides an exact analytic solution to the from-region $2\frac{1}{2}$D visibility problem. It uses a stabbing line computation by means of a polyhedra enumeration in Plücker coordinates. The exact method is complemented with optimization techniques that make it suitable for visibility computation in large scenes.

The from-region visibility algorithms for $2\frac{1}{2}$D scenes were evaluated by computing PVS in a scene representing the center of the city of Vienna. The results compare favorably with the previously published method of Wonka et al. [Wonk00, Wonk01a] in both the running time and the size of the computed PVS. The proposed methods efficiently handle large view cells and achieve output-sensitive behavior in practice. Their suitability to large view cells is given by the fact that the computational complexity method is given by the complexity of the visible part of the scene. Increasing the size of the view cell need not result in the proportional increase of the visibility complexity and thus the computational time is not altered. The conservative algorithm computed a PVS that consisted of less that 1% of total number of scene objects on average. The PVS computed by the conservative method was 19.5% smaller that the PVS computed by the algorithm of Wonka et al. [Wonk00] for small view cells and 13.6% smaller for large viewcells. The exact method decreased the size of the PVS by 0.6% for small view cells and 1.2% for large view cells. The results indicate that the conservative method is very close to the exact result and the effort required to implement the exact method is probably not worth the minor improvement of the accuracy of the PVS. The tests also highlighted the problem of numerical accuracy of the floating point implementation. For several tested view cells the exact method reported a PVS containing an object that was classified as invisible by the conservative method. These cases occurred due to the finite precision of the floating point computations and were emphasized by the fact that computations were performed in different solution space domains (primal space for the conservative method, Plücker coordinates for the exact method). The conclusion taken from this observation is that the visibility algorithms implemented with floating point arithmetics (including the exact ones) should apply a unified ϵ-thresholding in the domain reflecting to the target application (e.g. z-buffered rendering).

9.1.6 From-region visibility in 3D

Chapter 8 presented a new method 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. The Plücker coordinates allow to perform operations on sets of lines by means of set theoretical operations on 5D polyhedra in Plücker coordinates. A union of the polyhedra is maintained by the occlusion tree that represents lines occluded from the given region (polygon).

The chapter discussed the relation of sets of lines in 3D and the polyhedra in Plücker coordinates. It 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. Two algorithms for the construction of the occlusion tree were described. The first uses a polyhedron splitting, the second is based on the polyhedron enumeration. The occlusion tree was used to test visibility of a given polygon or region with respect to the source polygon/region using either exact or conservative visibility tests. Several optimizations of the method were proposed that make the approach applicable for large scenes. Three possible applications of the method were discussed: discontinuity meshing, visibility culling and occluder synthesis.

The implementation of the method was evaluated on scenes consisting of randomly generated triangles, structured scenes, and a real-world urban model. We focused on the application method for the PVS computation. The artificial scenes allowed to study the behavior of the method, computing a PVS in the urban model explored the applicability of the method for visibility preprocessing in real-world scenes. The measured results correspond to the theoretical expectations: the computational and memory complexities of the method are primarily determined by the size and the structure of the visible part of the scene. Increasing the number of scene objects need not lead to the increase of the computational time if visibility remains constant. The results indicated the importance of occluder fusion for scenes consisting of many small and disconnected polygons: in such scenes increasing the number of polygons led to the increase in occlusion and consequently to the decrease of the number visible polygons and the running time of the algorithm.

Although the method is based on a simple straightforward concept, an efficient and robust implementation of the method became rather complicated. We evaluated several implementations of the polyhedron enumeration and an implementation of polyhedron splitting. Finally, we used a floating point implementation of the reverse search polyhedra enumeration algorithm [Avis96, Bitt97] combined with ϵ-thresholding based on the proposed size measure for PVS computation. This combination proved robust in practice, but it still draws a potential for acceleration of the method. The drawback of the current implementation is that the polyhedron enumeration is executed independently of each other although the computed polyhedra might be very coherent.

The principal advantage of the method is its generality. The method does not rely on many tuning parameters or significant restrictions of the input characterizing many conservative or approximate algorithms [Dura00, Scha00, Wonk00].

9.2 Suggestions for further research

This section discusses suggestions for further research on the topics discussed in the thesis.

9.2.1 The concept of a visibility algorithm

The concept of a visibility algorithm discussed in Chapter 3 was targeted at computing visibility from a given point, line segment, or region. The concept could potentially be extended for computing global visibility in an adaptive fashion: the global visibility computation could balance a localized from-region visibility with the merging of the computed results in a global visibility data structure. Another topic is employing solutions to the simpler from-point or from-segment visibility problems to guide the global visibility computation and to increase its efficiency.

9.2.2 From-point visibility culling

The visibility culling algorithm presented in Chapter 4 assumes that the occluders are sufficiently large convex polygons. If this assumption is not satisfied we should apply some occluder synthesis techniques such as the method discussed in Section 8.12.3. Finding a balance between the accuracy and computational demands of the occluder synthesis and the accuracy and the computational demands of real-time visibility culling is an open problem.

The occlusion tree for the from-point visibility culling is built from scratch in each frame of a walkthrough although the trees constructed for subsequent frames are highly coherent. By reusing the tree within more frames the tree construction time could be amortized. Consequently, a more complex occlusion tree might be used to achieve higher culling efficiency.

The conservative hierarchy updating and the probabilistic modification of the visibility propagation algorithm are based on a user-specified constants. More elaborate methods might be applied that automatically adjust these probabilities according to the history of visibility changes. The algorithm as described determines, whether a node is visible or invisible. It could be extended to estimate, whether a node that was partially visible in the previous frame remains partially visible. This modification would benefit in sparsely occluded environments where many small regions are classified as partially visible.

9.2.3 Visibility maps

A challenging topic is the construction of an approximate visibility map using the proposed method. The approximate variant of the algorithm could overcome the problem of overly detailed output of the exact algorithm for a large number of tiny polygons. Another interesting topic is the application of occlusion maps in occluder synthesis. Although the visibility map is constructed for a single viewpoint it provides a lot of structural information that can potentially be exploited in solutions to from-region or from-segment visibility problems.

9.2.4 From-region visibility in 2D and $2\frac{1}{2}$D

The subdivision of the problem-relevant line set maintained by the occlusion tree could be replaced by a regular subdivision using a discrete occlusion map. The discrete aggregated occlusion map could be implemented on a dedicated graphics hardware similarly to the recent method of Koltun et al. [Kolt01].

9.2.5 From-region visibility in 3D

The solution of the from-region visibility problem can be used to accelerate algorithms solving problems of visibility along a line, visibility from point or visibility from a line segment. There is a great potential in optimizing the proposed technique by using a more efficient polyhedron splitting/enumeration algorithm.

One of the most promising applications of the method is the computation and the description of virtual occluders. Another important topic concerns a construction of a viewspace subdivision driven by the visibility events determined by the from-region visibility algorithm. The method could potentially be applied in the context of an adaptive global visibility computation that would balance the accuracy of the visibility computation with the memory and computational demands, considering an estimate of the benefit achieved by providing a more accurate description visibility.

Appendix

Appendix A

Traditional visibility algorithms

Traditional visibility algorithms were designed to solve the *visible surface determination*. Nowadays the visible surface determination is commonly carried out by the z-buffer algorithm implemented in the graphics hardware. Nevertheless, many recent techniques exploit concepts provided by the classical algorithms. In the following text we briefly discuss the most important visible surface algorithms.

A.1 Z-buffer

Z-buffer, introduced by Catmull [Catm75], is one of the simplest visible surface algorithms. Its simplicity allows efficient hardware implementation and therefore the z-buffer is nowadays commonly available even on a low cost graphics hardware. Z-buffer is an image precision algorithm that associates a depth information with each pixel of the frame buffer. The depths are initiated to zero, representing the depth at the farthest clipping plane. The largest depth value that can be stored in the buffer corresponds to the near clipping plane. Polygons are rasterized in an arbitrary order. For each pixel covered by a polygon the algorithm computes the depth of the polygon at the center of the pixel. The depth is compared to the corresponding entry in the z-buffer. If the new value is greater, the corresponding polygon fragment is closer to the viewpoint and both the frame buffer and the z-buffer are updated. Otherwise the new fragment is discarded since it is hidden by some previously processed polygon.

Z-buffer uses discrete samples of object depths and therefore it is prone to aliasing. This problem is addressed by the A-buffer algorithm [Carp84]. Another source of aliasing is the finite precision of the depth values stored in the z-buffer (see Figure A.1).

Z-buffer algorithm can be accelerated by processing objects in the front to back order. Such an order minimizes the number of updates of both the frame and the depth buffers. The z-buffer algorithm is not output-sensitive since it needs to rasterize all scene objects even if many objects are invisible. The lack of output-sensitivity is not restrictive for scenes where most of the scene elements are visible, such as a single albeit complex object. For large scenes with high *depth complexity*, the processing of invisible objects causes significant “overdraw”. This problem is addressed by visibility culling methods that are discussed in Appendix B.

A.2 List priority algorithms

List priority algorithms determine an ordering of scene objects making sure that the correct image is produced if the objects are rendered in this order. Typically objects are processed in a back to front order: closer objects are painted over the farther ones in the frame buffer. There are several approaches to determine the appropriate order of scene objects. In some cases no suitable order exists due to *cyclic overlaps*. In such a case some object(s) are split. List priority algorithms differ according to which objects get split and when the splitting occurs.

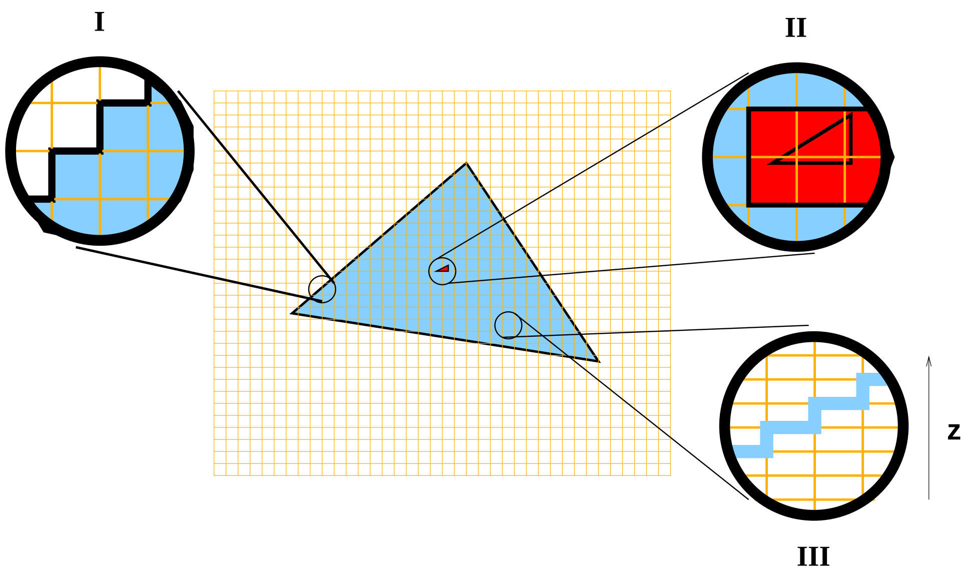


Figure A.1: Aliasing in the z-buffer algorithm. The finite precision of the image plane causes alias at the boundaries (I) and overestimation of small features (II). The finite precision of z-values causes alias in depth (III).

A.2.1 Depth sort

The basic idea of the *depth sort* algorithm by Newell et al. [Newe72] is to render polygons in the frame buffer in order of decreasing distance from the viewpoint. The following three steps are performed:

1. Sort all polygons according to the farthest z-coordinate.

2. Resolve any polygon dependency cycles that possibly occur when the polygons' z-extents overlap. In such a case some polygons are split.

3. Rasterize each polygon in ascending order of the smallest z-coordinates, i.e. in the back-to-front order.

A simplified variant of this algorithm that ignores step (2) is called the *painter's algorithm* due to the similarity to the way a painter paints closer objects over distant ones.

To resolve the step 2 up to five tests are applied to determine if polygons overlap in the projection plane [Newe72, Fole90]. These tests are performed in the order of increasing complexity. If all five tests fail we now that the polygons overlap and at most two additional tests are applied that aim determine the correct order of the two polygons. If these fail, one of the polygons must be split.

A.2.2 BSP trees

The *Binary Space Partitioning* (BSP) tree introduced by Fuchs, Kedem, and Naylor [Fuch80] allows efficient calculation of visibility ordering among static polygons. A BSP tree is constructed in preprocessing. At this step all potential dependency cycles are broken and the corresponding polygons are split. The size of the BSP tree corresponds to the number of resulting polygon fragments and it is heavily dependent on the particular BSP construction algorithm. The rendering is performed by an ordered traversal of the BSP tree in a linear time with respect to the number of tree nodes.

The BSP tree algorithm is based on the work of Schumacker et al. [Schu69] who manually partitioned the scene into collection of clusters. If clusters can be separated by a plane then the cluster on the same side of the plane as the viewpoint cannot be obscured by the clusters on the other side of the plane. If such separating planes exist, the clusters can be subdivided recursively. The resulting subdivision of the scene can be represented by a binary tree where each interior node corresponds to a plane separating a given set of clusters and each leaf corresponds to one cluster. The clusters are determined so that there is a viewpoint independent priority ordering of all polygons within a cluster.

The BSP tree algorithm is a generalization of the approach of Schumacher et al. [Schu69] that automatically solves the problem of polygon priority without the necessity of manual construction of clusters and their priorities. The BSP tree is constructed recursively as follows:

1. Let the current set of polygons $\mathcal{S}$ be all polygons in the scene.
2. If $|\mathcal{S}| \leq 1$ create a leaf node with a reference to the single polygon from $\mathcal{S}$ (if any) and terminate this branch of the algorithm.
3. Select a plane P that partitions $\mathcal{S}$ into $\mathcal{S}^-$ and $\mathcal{S}^+$ corresponding to polygons in the negative and positive halfspace induced by P. Polygons straddling plane P are split.
4. Construct a new interior node N with a reference to P. Associate all polygons embedded in plane P with N.
5. Construct child nodes of N by recursively repeating steps 2 to 5 using $\mathcal{S}^-$ and $\mathcal{S}^+$ for left and right child nodes respectively.

The crucial step of the BSP tree construction is the selection of the splitting plane (3). The usual technique is to restrict splitting planes to the planes aligned with the scene polygons. The resulting tree is called *autopartition* [Berg97]. To construct the autopartition we have a finite number of choices. The order in which the splitting planes are selected is very important. Several heuristics for splitting plane selection have been proposed that aim to minimize the number of polygons splits and thus the size of the constructed tree. For purposes of rendering the tree need not be balanced. In fact a tree degenerated to a linear list would exhibit the same rendering performance as a perfectly balanced tree. Nevertheless the construction time is increased if the tree is heavily unbalanced. The construction of a balanced tree is also advantageous for other tasks the tree might be used for such as point location.

In the rendering phase the tree is used to establish a back-to-front order of scene polygons. Visibility is solved in image space by overwriting farther polygons by the closer ones. Given a viewpoint the rendering using a BSP tree proceeds as follows:

1. Let current node C be the root.
2. If C is a leaf rasterize the associated polygon and terminate this branch of the algorithm.
3. Otherwise determine the position of the viewpoint with respect to the plane P associated with C.
4. Recursively repeat steps 2-4: First, process the child corresponding to the opposite side of P than the viewpoint. Second, rasterize P. Third, process the child on the same side as the viewpoint.

A similar procedure can be used to generate a front-to-back order of polygons. The difference is in the processing order of child nodes in step 4. Algorithms that use a front-to-back order and a screen space data structure for correct image updates [Gord91, Nayl92b] can achieve output-sensitive behavior. An 2D example of a BSP tree and its front-to-back traversal is depicted in Figure A.2.

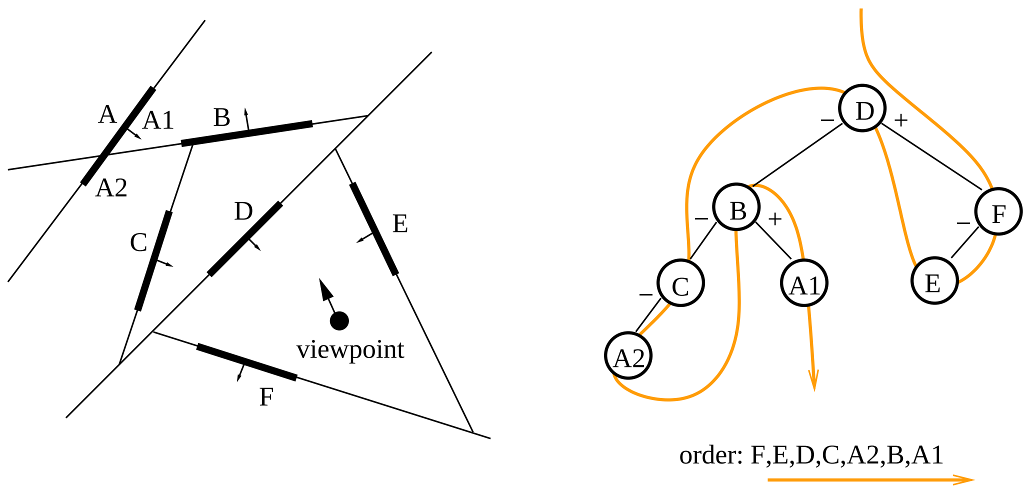


Figure A.2: An example of a 2D BSP tree. The scene consist of 6 line segments. The tree contains 7 nodes (segment A is split). On the right a front-to-back traversal order of the BSP tree is depicted.

A.3 Area subdivision algorithms

Area subdivision algorithms use the *divide and conquer* strategy that subdivides the projection plane. At a time an area of the image is examined. If it is easy to decide which objects are visible in the area the objects are rendered. Otherwise the algorithm subdivides the current area in smaller ones and continues recursively. As the areas become smaller less objects need to be considered and the visibility decision becomes simpler.

A.3.1 Warnock's algorithm

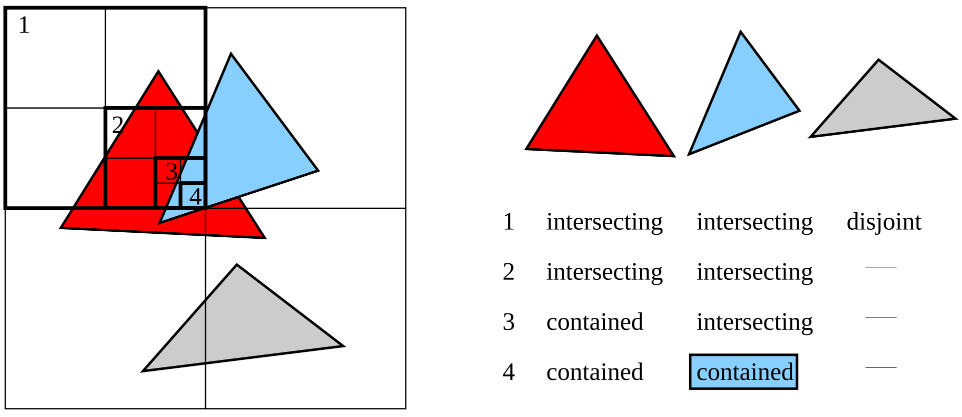
Warnock [Warn69] developed an area subdivision algorithm that subdivides a given image area into four equal rectangles. At each step the algorithm identifies the position of each polygon with respect to the current image area as follows:

1. *Surrounding* polygon – it completely contains the area.
2. *Intersecting* polygon – partially intersects the area.
3. *Contained* polygon – it is completely inside the area.
4. *Disjoint* polygon – it is completely outside the area.

In the following four cases the visibility within the current area can be solved easily and the area does need to be subdivided:

1. All polygons are *disjoint*. The background color is displayed.
2. There is only one *intersecting* or *contained* polygon. This polygon is rasterized (an intersecting polygon is first clipped).
3. There is only one *surrounding* polygon. The area is filled with the polygon color.
4. There is a *surrounding* polygon that is in front of all other intersecting, contained or surrounding polygons. The area is filled by the polygon color. Determining if the polygon is in front of all other is carried out by comparing z-coordinates of supporting planes of the polygons at the four corners of the area.

If all these four tests fail the area is subdivided. The subdivision is terminated when the area matches the pixel size. Alternatively sub-pixel subdivision can be used to remove aliasing. An illustration of the Warnock's algorithm is shown in Figure A.3.



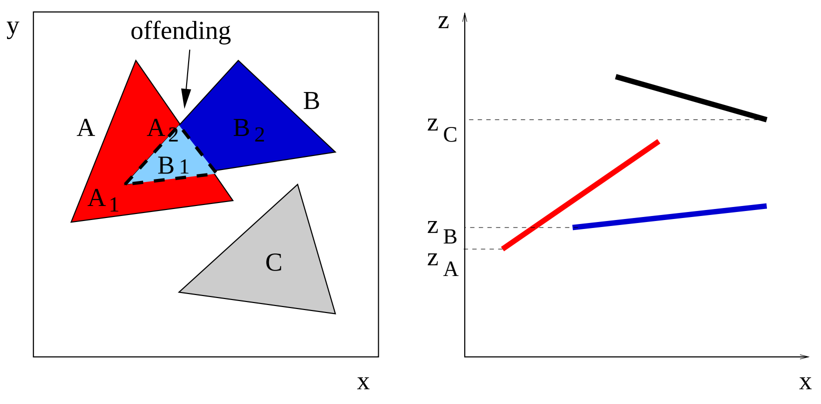
	🟥	🟦	⬜
1	intersecting	intersecting	disjoint
2	intersecting	intersecting	—
3	contained	intersecting	—
4	contained	contained	—

Figure A.3: An illustration of the Warnock's algorithm. The screen is recursively subdivided into rectangles. The right part shows a classification of the polygons with respect to the four depicted screen rectangles.

A.3.2 The Weiler-Atherton algorithm

The algorithm developed by Weiler and Atherton [Weil77] subdivides the image plane using the actual polygon boundaries, rather than rectangles used in Warnock's algorithm. The Weiler-Atherton algorithm does not rely on a pixel resolution and typically needs a lower number of subdivision steps. On the other hand, it requires robust polygon clipping algorithm.

The polygons are sorted according to their nearest z-coordinate (this step is mandatory but improves efficiency). The polygon closest to the viewer is selected and used clip all other polygons in two sets: polygons inside or outside with respect to the "shadow" of the selected polygon. All inside polygons that are behind the selected polygon can be deleted since they are invisible. If an inside polygon is closer than the selected one it is called *offending*. An offending polygon is used to clip all remaining offending polygons. All remaining outside polygons are processed recursively. The algorithm uses a stack to handle cyclic overlaps. An illustration of the algorithm is depicted in Figure A.4.



1. A	B −> B_1 + B_2	
	B_1	offending
	B_2	outside
	C	outside
1b. B_1	A −> A_1 + A_2	
	A_1	outside
	A_2	inside
2. B_2	C	outside
3. C		

Figure A.4: An illustration of the Weiler–Atherton algorithm. Four steps are necessary to resolve visibility of the three depicted polygons.

A.4 Ray casting

Ray casting [Appe68] is a visible surface algorithm that solves the visibility from the viewpoint by by *ray shooting*, i.e. shooting rays through each pixel in the image. The drawback of ray casting in comparison with the other visible surface algorithms is that it does not exploit coherence of neighboring rays unless specialized techniques are used [Havr00b]. On the other hand if ray casting is implemented using a variant of the ray-walking technique proposed by Glassner [Glas84] it results in an output-sensitive algorithm.

A naive implementation of ray shooting would test each object in the for an intersection with a given ray and then select the one closest to the viewpoint. The resulting $O(n)$ time complexity for each ray is not suitable for scenes with more than a few objects. To improve the time complexity of the ray shooting query many acceleration techniques have been designed. Most of them use a spatial subdivision of the scene that allows to restrict the set of objects tested for an intersection only to those laying in proximity of the given ray (see Figure A.5). Ray shooting will be discussed in more detail in Section C.5.

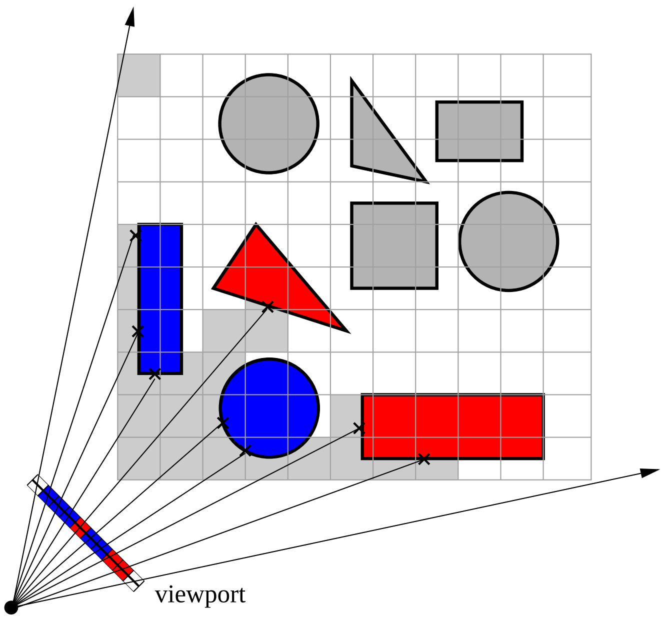


Figure A.5: A 2D illustration of ray casting. Ray casting is performed using ray shooting supported by a uniform grid. Note the output-sensitive behavior of such ray casting algorithm: only visible objects are touched by the algorithm.

A.5 Scan line algorithms

The first scan-line algorithms introduced in late 60's and early 70's are summarized in Foley et al. [Fole90]. The basic approach is an extension of a scan conversion of a single polygon. The algorithm constructs an edge table of all non horizontal edges of the scene polygons. Entries in the edge table are sorted into buckets based on the smaller y-coordinate of each edge. Within each bucket edges are ordered by increasing x-coordinate of their lower endpoint.

The algorithm maintains an active edge table and an in-out flag for each polygon indicating if we are currently inside our outside of the polygon. If there is more polygons with an in-out flag set to inside a depth comparison is used to determine the closest one to the viewpoint. The coherence is exploited using incremental updates of the active edge table similarly as with the scan conversion of a single polygon [Fole90].

Appendix B

Visibility in real-time rendering

This chapter discusses the use of visibility algorithms for real-time rendering acceleration. It also presents other important techniques for rendering acceleration that are typically used in combination with the visibility algorithms.

The are at least three performance goals for real-time rendering: more frames per second, higher resolution and more realistic scenes [Moll02]. A frame rate of 60-72Hz is typically considered sufficient. Optimally the frame rate should match the monitor refresh rate [Wonk01a]. There is also a bound on resolution – 1600x1200 seams sufficient for typical size of the display device. Nevertheless there is no natural upper limit on the scene complexity. For example a reasonably detailed aircraft model consists include 500M polygons [Moll02], a model of a city with a few building details consists of 8M polygons [Wonk01a, Bitt01e]. The general conclusion is that the user demands and expectations on the scene complexity grow even faster than the computer processing power. This draws a great potential for acceleration techniques that aim to reduce the amount of data sent to the rendering pipeline [Schm97, Rohl94, Bish98, West97, Deva97, Cham96].

B.1 Backface culling

Back face culling aims to avoid rasterization of surfaces that are not facing the viewpoint. For an orthographic projection about 50% is of the scene polygons is expected to be culled on average. For the perspective projection the percentage of backfaces increases as the viewpoint moves closer to the scene objects.

Backface culling on a per polygon basis is often supported in hardware [nVID02, ATi 02]. The problem is that it is applied rather late in the rendering pipeline. Additional speedups can be achieved if a whole group of polygons is identified as backfacing at a single test. This can be achieved by the *clustered backface culling* [Kuma96b, Kuma96a, Zhan97a].

B.2 View frustum culling

View frustum culling (VFC) eliminates rendering of objects that are outside of the viewing volume. The viewing volume is represented by a frustum of a pyramid called *view frustum*. Objects that do not intersect the view frustum cannot be seen and therefore need not be rendered.

The VFC is often supported by the current graphics hardware [nVID02, ATi 02]. Similarly to the backface culling, considerable savings can be achieved if a whole group of objects is culled before sending the objects to the graphics pipeline. This is a goal of the *hierarchical view frustum culling* (HVFC). HVFC uses either a modeling hierarchy (scene graph) or a spatial hierarchy. Starting at the root node of the hierarchy, the algorithm computes the intersection of the volume corresponding the current node and the view frustum. If the volume does not intersect the view frustum the corresponding

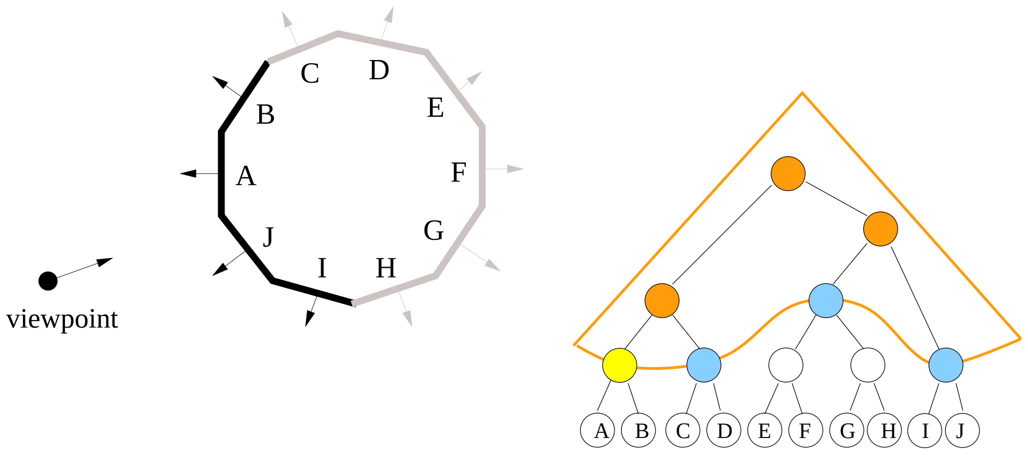


Figure B.1: Clustered backface culling. Polygons are grouped according to their supporting planes. Pruning of the resulting hierarchy results in culling whole groups of backfacing polygons: orange nodes have ambiguous visibility classification, the blue nodes are invisible, yellow node is visible.

subtree is culled. If it partially intersects the view frustum the algorithm is applied recursively on the children of the current node. Otherwise the volume is completely included in the view frustum and objects from the whole subtree of the current node are sent to the graphics pipeline. Note that in this case the screen space clipping can be disabled for all objects fully contained in the volume.

Often a spatial hierarchy is used for HVFC instead of the scene graph. The spatial hierarchy follows the idea of *spatialization*, i.e. it is built according to the spatial proximity of objects. On the contrary the scene graph is constructed according to the modeling rules that need not group close objects together.

A popular spatial hierarchy for HVFC is the *bounding volume hierarchy* (BVH) [Clar76]. BVH consists of bounding spheres, axis aligned bounding boxes (AABB), or oriented bounding boxes (OBB). Often a mixture of bounding primitives is used in the same BVH. The choice of the bounding primitive to use depends on the depth of the given node. For nodes closer to the root only the coarser bounding sphere or AABB intersection test is applied, for deeper nodes the OBBs are used.

Volumes of the bounding volume hierarchy can overlap significantly, which makes the HVFC algorithm inefficient. This problem is eliminated by using spatial subdivisions at the cost of lower flexibility for dynamic scenes. Spatial subdivisions commonly used for HVFC are octrees, kD-trees, and BSP trees. The requirements for construction of spatial hierarchies for VFC are similar to construction of hierarchies for visibility culling and ray shooting [Ione98]. An illustration of the hierarchical view frustum culling is depicted in Figure B.2.

Several optimizations of the HVFC algorithm were proposed by Assarsson and Möller [Assa00]. Some of these techniques aim to exploit temporal coherence, other concern efficient implementation of the intersection tests. Slater and Chrysanthou [Slat97] presented a probabilistic scheme for acceleration of the view frustum culling. Visibility status of an object is updated according to its estimated distance from the view frustum.

B.3 From-point visibility culling

View frustum culling eliminates objects outside the view frustum, but there might still be many objects invisible due to the occlusion. Techniques that aim to avoid rendering of occluded objects are called *visibility culling* (also called *occlusion culling*).

Visibility culling can be applied with respect to a single point or a whole region. Typically the from-point visibility culling is applied *online*, i.e. visibility is recomputed after each change of the viewpoint. On the contrary the from-region visibility culling can be applied *offline*, i.e. it precomputes visibility

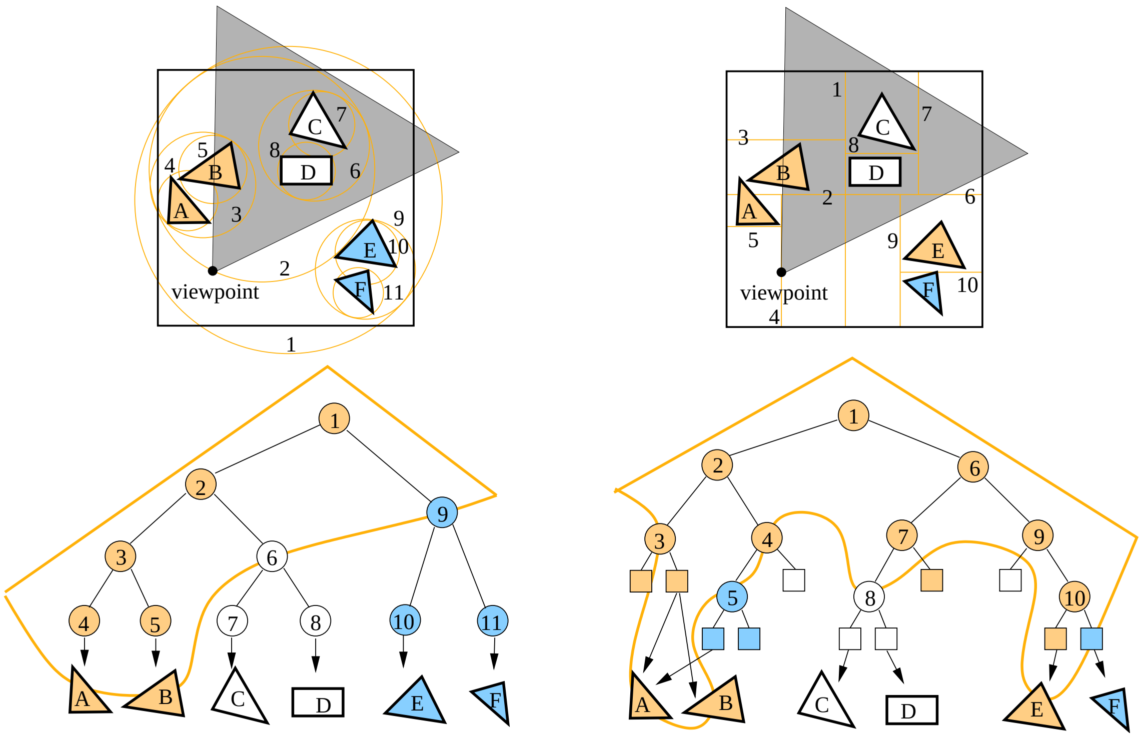


Figure B.2: Hierarchical view frustum culling using hierarchy of bounding spheres (left) and a kD-tree (right). The white nodes are completely inside the view frustum, orange nodes partially intersect the view frustum and the blue nodes are culled. The objects are colored according to their visibility classification. The orange line depicts a cut of the hierarchy at which the HVFC algorithm was terminated.

in preprocessing. The from-region visibility culling will be discussed in Section B.4. Here we outline several methods for the from-point visibility culling designed for an online application. We first review the algorithms that use a discrete representation of occlusion and then the algorithms using a continuous representation.

B.3.1 Hierarchical z-buffer

A general technique for the from-point visibility culling is the *hierarchical z-buffer* introduced by Greene et al. [Gree93]. It uses a z-pyramid to perform fast visibility queries. Each level of the pyramid corresponds to a z-buffer with a down-sampled resolution. Level 0 corresponds to the traditional z-buffer, level 1 to a z-buffer with half resolution, etc. Each entry at level n+1 contains the farthest of the four corresponding entries from level n.

The z-pyramid is used in combination with a scene octree. The octree nodes are recursively tested for visibility using the corresponding bounding boxes. For each face of the box its depth is first compared to the coarsest level of the pyramid. If the face is farther it is hidden and thus invisible. Otherwise the test proceeds using relevant entries at the finer level of the z-pyramid. See Figure B.3 for an illustration of the hierarchical z-buffer algorithm. The z-pyramid is maintained by propagating changes at the level 0 z-buffer up into the hierarchy. Figure B.4 depicts a conservative depth propagation in the z-pyramid.

To make the algorithm suitable for the current graphics hardware Greene et al. [Gree93] proposed a heuristics based on the temporal coherence. The two pass algorithm can be outlined as follows: First, previously visible objects are rendered using traditional z-buffering. The z-buffer is then read back from the hardware and the z-pyramid is built in software. Second, the octree test is performed in software, but it skips nodes that have already been rendered in the first pass. Finally, the list of visible objects is updated for the use in the next frame.

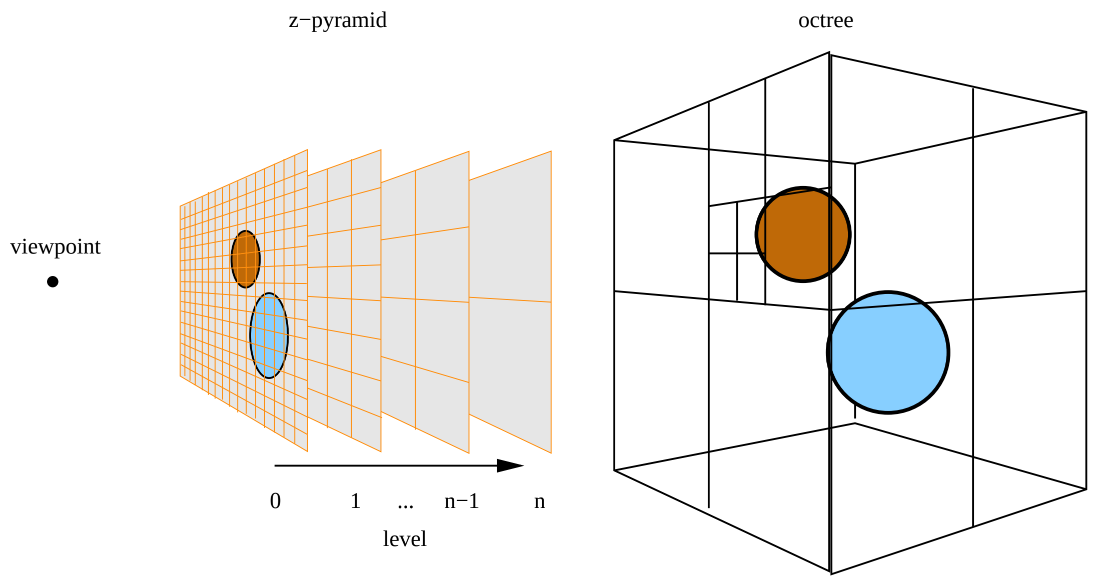


Figure B.3: The hierarchical z-buffer algorithm. The z-pyramid is used to cull invisible nodes of the octree. Objects are rasterized into the level 0 z-buffer.

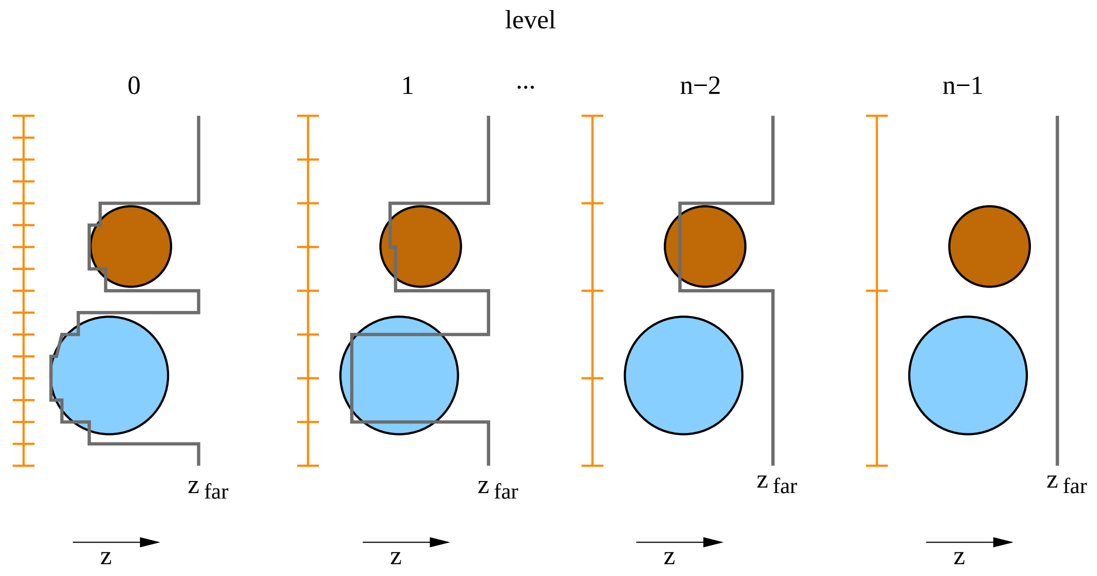


Figure B.4: The propagation of depths in the z-pyramid. The depths are conservatively propagated up the z-pyramid: each entry at level n+1 contains the farthest of the four corresponding entries from level n.

B.3.2 Hierarchical polygon tiling

The hierarchical z-buffer algorithm applies the occlusion test on the octree nodes. Greene [Gree96] proposed *hierarchical polygon tiling* algorithm operating on a per polygon basis. It exploits *coverage masks* to accelerate operations on the z-pyramid. Additionally if a front-to-back order of polygons is established, the z-pyramid can be replaced by a *coverage pyramid*. The coverage pyramid contains only two state values indicating if the corresponding part of screen is occluded. The depth tests are eliminated completely. The hierarchical polygon tiling with coverage masks is well suited for rendering large scenes in high resolution without a dedicated graphics hardware.

B.3.3 Hierarchical occlusion maps

Zhang et al. [Zhan97b] described *hierarchical occlusion maps* (HOM) that use similar idea as the hierarchical z-buffer and exploit the standard graphics hardware.

Occluder selection

The HOM algorithm assumes that a set of *good potential occluders* is identified in preprocessing. Ideally for each viewpoint we pick only objects that are visible and cause significant occlusion. This however is the visible surface problem itself. The solution is to preprocess the scene and identify set of promising occluders for a given range of viewpoints.

Occlusion map

The algorithm uses a pyramid of occlusion maps. Each entry of the occlusion map contains a opacity value: 1 corresponds to full occlusion, 0 to no occlusion. Values between 0 and 1 indicate partial occlusion. Level 0 occlusion map is obtained by rendering selected occluders with lighting and depth tests disabled. The resolution of the level 0 occlusion map is typically much lower than one of the rendered image (e.g. 256x256 pixels). Pixels occupied by some occluder contain value 1. Level 1 occlusion map with half resolution is formed by down-sampling: each pixel is an average of the four corresponding pixels from level 0. The complete hierarchy of occlusion maps is built in the same manner. The down-sampling can be performed efficiently with the help of the texture filtering operations implemented in the graphics hardware.

Depth estimation map

Occlusion maps do not contain any depth information. The depth of occluders is represented by the *depth estimation map* (DEM). Each entry of DEM corresponds to a rectangular region on the rendered image (the DEM resolution is typically only 64x64). The DEM entry contains a maximum distance of occluders that project to the corresponding rectangle. For simplicity only bounding boxes of occluders are used for the depth estimation (see Figure B.5).

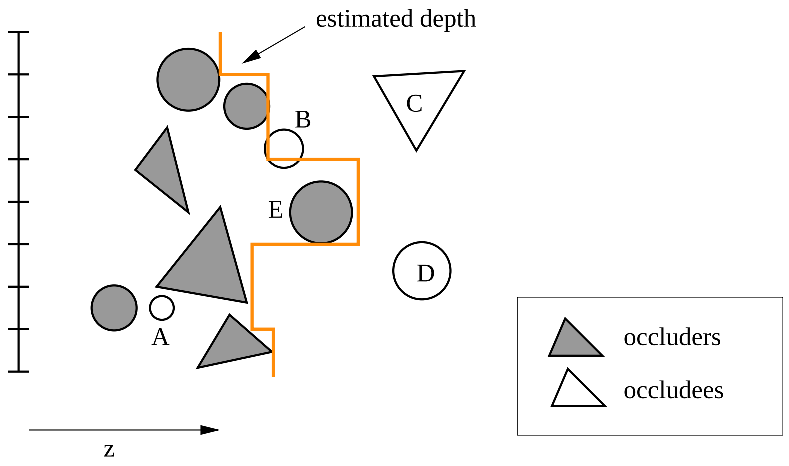


Figure B.5: DEM for the hierarchical occlusion map. Note that occludees A and B pass the depth test although they are invisible with respect to the selected occluders.

HOM culling algorithm

The HOM culling algorithm can be outlined as follows: First, render the occluders to the level 0 occlusion map and form the HOM pyramid. Second, construct the DEM. Third, starting at the root node of the spatial hierarchy traverse the scene while performing occlusion tests. If a leaf of the hierarchy is found visible render the associated objects.

The occlusion test is performed as follows: For each node's bounding box test if the corresponding pixels are occluded with respect to the coarsest level occlusion map. If they are not occluded (opacity is lower than 1) descend to the higher resolution occlusion map. The procedure can be easily modified to perform an *approximate* visibility culling. For details on the algorithm see [Zhan98].

B.3.4 Shadow frusta

The principle of shadow frusta culling of Hudson et al. [Huds97] is rather simple: Identify few good polygonal occluders and use their shadow volumes to cull invisible objects by hierarchical visibility tests. A visibility test is performed against each frustum independently using a spatial hierarchy. The test determines if a bounding box is completely contained in any shadow frustum. If the box is in shadow the corresponding hierarchy node and its whole subtree are culled. The drawback of the method is that it does not account for occluder fusion and its time complexity grows linearly with the number of selected occluders (see Figure B.6).

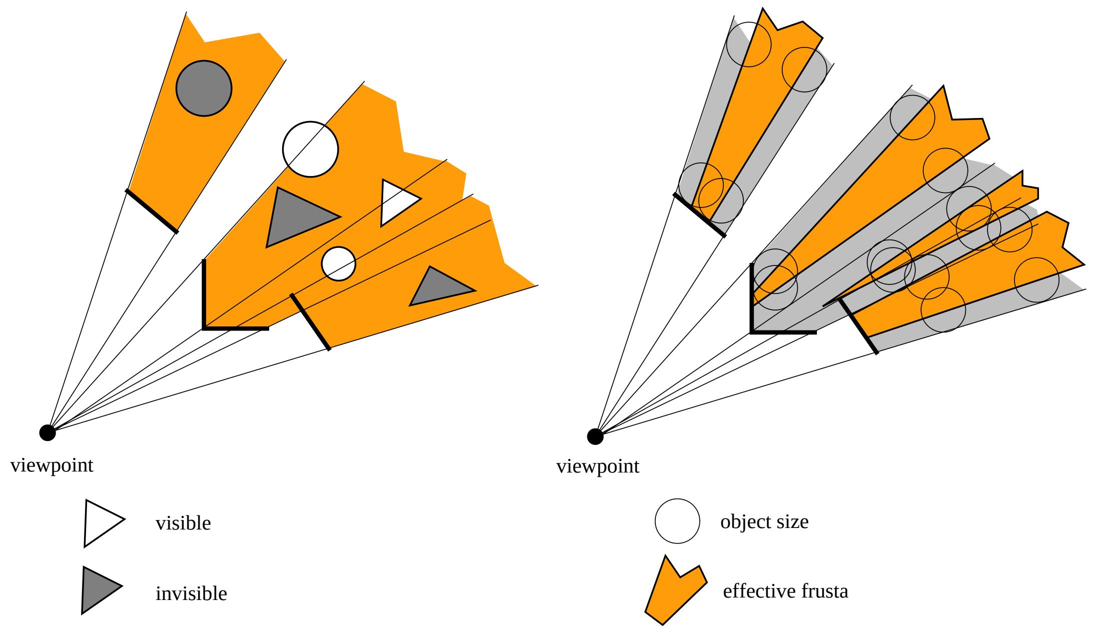


Figure B.6: (left) Visibility culling using shadow frusta. Four shadow frusta are constructed for the four depicted occluders. The gray objects are classified invisible. The white objects are classified visible. (right) Effective shadow frusta for the depicted objects. The algorithm classifies the object as invisible if the center of the object is located inside an effective shadow frustum.

B.3.5 Visual events

The previously mentioned methods use some representation of the occluded volume with respect to the given viewpoint. Coorg and Teller [Coor97] developed a method that is based on a different idea. For a given spatial region they identify an umbra volume from which the region is invisible with respect to a given occluder. If the viewpoint is located in the umbra the region is culled. The penumbra of the region and occluder is a volume from which the region is partially visible. The umbra is bound by the supporting planes of the region and occluder, the penumbra is bound by the separating planes.

For each viewpoint a few promising occluders are identified as in the shadow frusta algorithm of Hudson et al. [Huds97]. The culling is performed hierarchically starting at the root node of the spatial hierarchy (Coorg and Teller use a kD-tree). For the current node supporting and separating planes are constructed for the corresponding box and each occluder. The algorithm tests the position of the viewpoint with respect to the computed umbras and penumbras. The test classifies the node as fully visible, partially visible or invisible. Invisible nodes are culled, fully visible nodes are rendered. Visibility of partially visible nodes is recursively refined. To exploit temporal coherence the algorithm maintains a cache of supporting and separating planes. See Figure B.7 for an illustration of the algorithm.

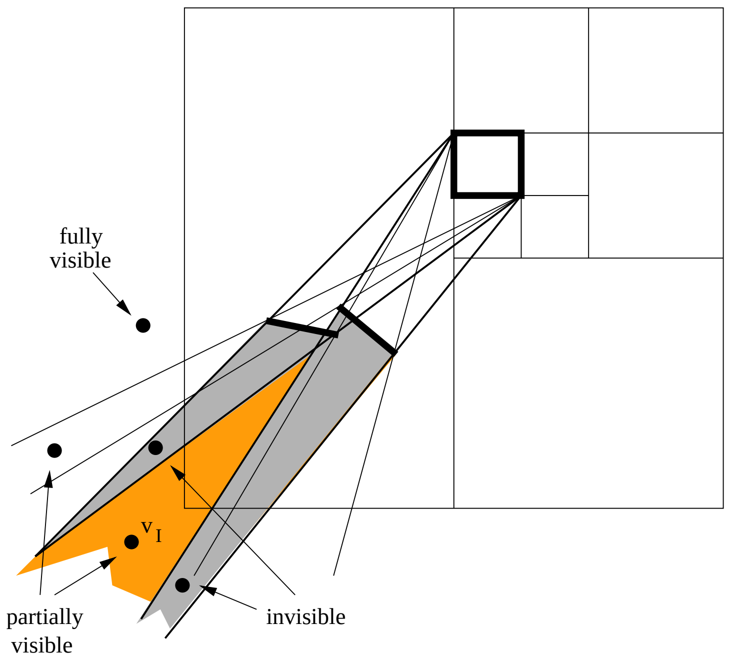


Figure B.7: A 2D example of supporting and separating planes for two occluders and an octree node. There are two umbra regions and four partially overlapping penumbras. The figure depicts the visibility classification of the node from the five depicted viewpoints. Note that the algorithm conservatively classifies viewpoint v_I as partially visible since it does not account for the *occluder fusion*.

B.3.6 Cells and portals

Jones [Jone71] proposed to subdivide the scene into cells and portals to solve the hidden line removal. He manually subdivided the model into convex polyhedral cells and transparent polygonal portals connecting these cells. Every polygon of the scene must be embedded in some face of one or more cells.

The scene is traversed starting at the cell containing the viewpoint. All faces of the current cells are rendered. Then the algorithm recursively continues by processing cells visible through portals of the current cell. The portal sequence through which any cell is reached forms a convex mask that is used to clip the faces of the cell. If the mask is empty the current branch of the algorithm is terminated. This method is complementary to the culling algorithms mentioned so far in the way how the invisible objects are identified. The culling is not based on identifying that an object is *occluded by* other objects (occluders), but an object is culled if it is not *visible through* a portal or a sequence of portals.

The method of Luebke and Georges [Lueb95] is a simplification of the method Jones that uses rectangular masks. The rectangular mask is a conservative overestimation of the polygonal mask. This approach enables simple and fast real-time intersection of the masks. For each viewpoint cells are processed in the front-to-back order and the opened portals are represented by a set of rectangles. The algorithm checks if a given cell is visible through at least one opened portal using rectangle overlap tests. If the cell is visible its geometry is rendered and the portals of the cell are intersected with the portals through which the cell is visible (see Figure B.8).

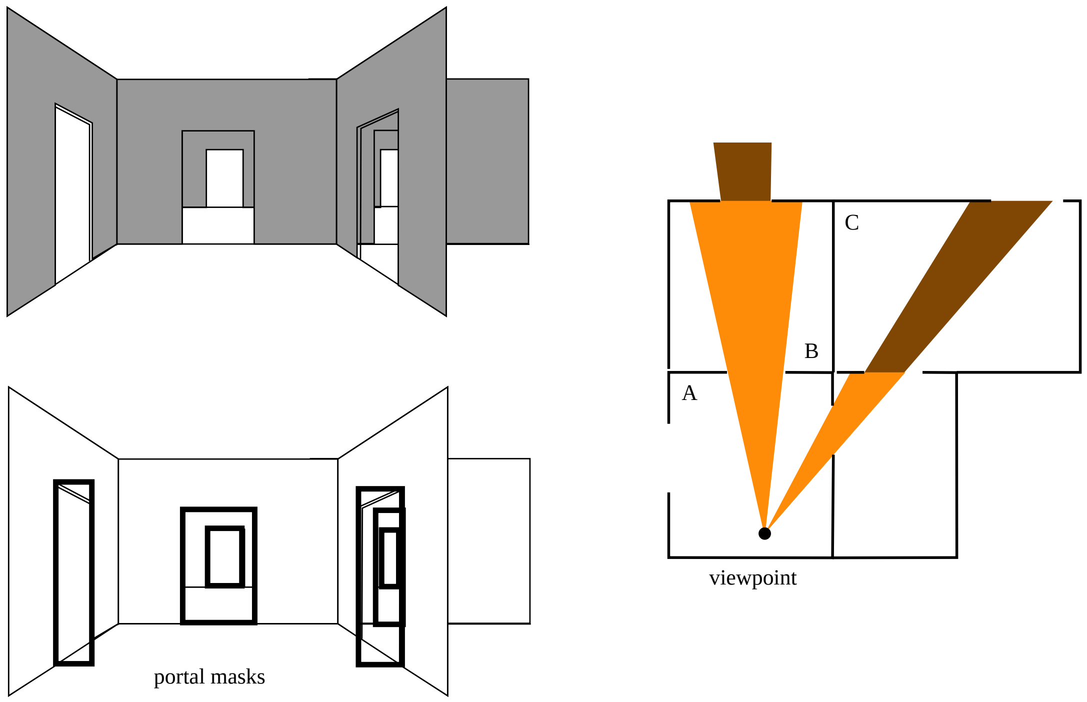


Figure B.8: Cells and portals for the from-point visibility culling. The portals are used to create rectangular masks. The masks are intersected and the algorithm traverses the cells until the masks intersection is non-empty.

These two approaches are similar to the visibility preprocessing algorithm of Teller and Séquin [Tell91] that will be discussed in Section B.4.1. The cell/portal methods are well suited for indoor scenes where cells roughly correspond to rooms and portals to doors. The drawback of these methods is that in general scenes there is no natural subdivision into cells and portals. In such a case their efficiency drops both in the running time and in the percentage of the culled objects.

B.3.7 Occlusion horizons

Occlusion horizons of Downs et al. [Down01] exploit the $2\frac{1}{2}$D nature of outdoor urban areas (see Section 2.5.1) to perform a fast and efficient online visibility culling.

The occluded volume is represented by a conservative piecewise constant *horizon* that is maintained by a balanced binary tree. The scene is swept in a front-to-back order and the horizon is updated by the processed objects (or their relevant subset). This process is interleaved with the hierarchical visibility tests. The visibility test checks if the bounding box corresponding to a hierarchy node is below the horizon. In such a case the node is culled.

The strength of the method is that it accounts for the occluder fusion. Additionally it can handle significantly more occluders than the other continuous methods.

B.3.8 Occluder shadows for visibility from point

Wonka et al. [Wonk99] proposed the *occluder shadows* for the from-point visibility culling in $2\frac{1}{2}$D scenes. The algorithm uses orthographic projection and the z-buffer algorithm to merge occlusion due to a set of occluders. *Occluder shrinking* is used to guarantee the conservative behavior of the algorithm.

For a given viewpoint the algorithm determines a set of relevant occluders. Each occluder induces an occluded volume represented by a *shadow polygon*. The shadow polygon is a truncated wedge defined by the viewpoint and the top edge of the shrank occluder.

The algorithm creates a discrete *cullmap* that is formed by rasterizing shadow polygons. An entry of the cullmap corresponds to the highest point that is guarantied to be occluded from the viewpoint. The size of necessary occluder shrinking is given by the resolution of the cullmap. See Figure B.9 for an illustration of the algorithm.

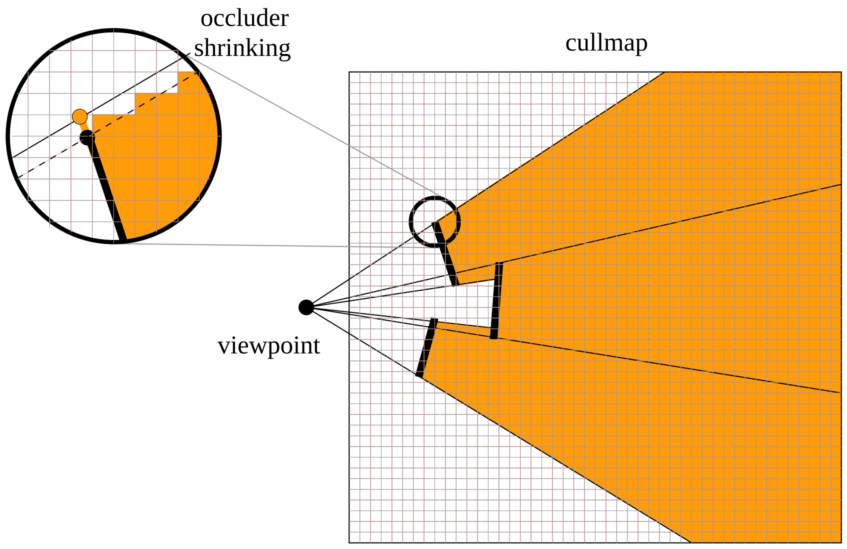


Figure B.9: Discrete cullmaps for visibility from point. Shadow polygons polygons are rasterized into the cullmap. To guarantee a conservative behavior of the algorithm the occluders are shrank according to the resolution of the cullmap.

B.4 From-region visibility culling

The from-point visibility culling requires recomputation of visibility for each change of the viewpoint. On the contrary, the from-region visibility culling techniques typically precompute a superset of objects visible from any point inside a given view cell.

B.4.1 Cells and portals

Airey et al. [Aire90] proposed the concept of *potentially visible set* (PVS). The model of indoor architectural environment is subdivided into cells and portals. For each cell the algorithm identifies other cells that can be visible through associated portals or their sequences. The identified cells form a PVS that is associated with the cell. A more precise solution is to calculate a PVS consisting of visible objects rather than visible cells.

In real time the algorithm identifies the cell in which the observer is located. Only objects from the cell's PVS are rendered. This can deliver speedups of orders of magnitude for large indoor scenes with restricted visibility.

Airey et al. used ray shooting to find a PVS of the given cell. This method is only approximate since some objects can be missed by the sample rays. Another drawback is that huge amount of rays must be cast in order to obtain sufficiently precise results. Visibility in indoor scenes was further studied by Teller and Séquin [Tell91]. Teller [Tell92b] proposed an exact analytic method to compute PVS in scenes with natural cell/portal structure. The algorithm uses five-dimensional halfspace intersection to test the existence of a stabbing line for a sequence of polygonal portals. Teller also proposed a simpler conservative algorithm based on clipping planes. The cells and portals methods are restricted to indoor scenes with a particular structure. Next, we present more general methods suitable for other types of scenes.

B.4.2 Single occluder ray shooting

Cohen-Or et al. [Cohe98a] use ray shooting to sample occlusion due to a single convex occluder. They cast rays that bound a shaft between the view cell and the region in question. If all rays hit the same convex occluder the region must be invisible (see Figure B.10). Cohen-Or et al. provide an analysis of visibility in densely occluded random scenes and derive bounds on the expected efficiency of the method. They show that in very large densely occluded random scenes the size of computed PVS is small in comparison to the total size of the scene. Nevertheless, the method generally produces significantly larger PVS than methods that handle occluder fusion.

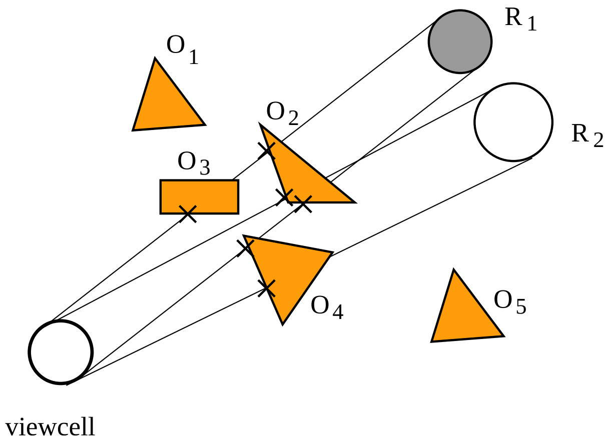


Figure B.10: Ray shooting based visibility preprocessing. The rays bounding the shaft between the given region and the view cell must intersect the same convex object. The region R_2 is conservatively classified as visible since the method does not account for occluder fusion.

B.4.3 Volumetric occluder fusion

Schaufler et al. [Scha00] use *blocker extensions* to handle occluder fusion. They use a discrete subdivision of the scene into transparent and opaque regions. Given a view cell the occluded part of the scene is determined by extending the opaque regions using a set of simple rules. All objects fully contained in the occluded regions are excluded from the PVS. The method finds a discrete conservative approximation of occluded volumes. Although the technique handles occluder fusion it fails to resolve all configurations of occluders.

The principle of discrete volumetric visibility is also used in the method of Yagel and Ray [Yage95] for computing visibility in caves. The cave system is represented by a volumetric model and visibility is determined by finding connected sequences of unoccluded voxels. This method can be seen as an inverse of the method by Schaufler et al. [Scha00], i.e. it extends the visible part of the scene instead of the occluded one.

B.4.4 Extended projections

Durand et al. [Dura00] proposed *extended occluder projections* and *occlusion sweep* to handle the occluder fusion. For a given view cell nearby occluders are projected on a projection plane using extended projection operators. The resulting occlusion map is swept through the scene using reprojection and merging using extended projections of encountered occluders. The set of projection planes forms a discrete conservative approximation of occluded volumes. Graphics hardware can be used to accelerate both the extended projection and the reprojection operators.

B.4.5 Occluder shadows

Wonka et al. [Wonk00] extended the occluder shadows method for the from-region visibility in $2\frac{1}{2}$D scenes. The algorithm uses orthographic projection and point sampling to merge occlusion due to a set of occluders. Occluder shrinking is used to guarantee conservative behavior.

The cullmap is formed by sampling visibility from points on boundaries of the given view cell. For a given view cell the algorithm determines a set of relevant occluders. The cullmap is formed by an intersection of occluded volumes defined by the occluders and the sample points on the view cell boundary. For each sample point occluder shadow polygons are rendered into the cullmap. The shadow polygon is a wedge formed by the sample point and the top edge of the shrank occluder. The size of necessary occluder shrinking is given by the distance between the sample points and the resolution of the cullmap.

Wonka et al. [Wonk01b] proposed an application of the occluder shadows method in an online algorithm. The PVS is computed for a small neighborhood of the given viewpoint. This PVS is valid for several frames assuming that the viewpoint moves smoothly. In this way the time spent on computing PVS is amortized over several frames.

B.4.6 Dual ray space

Koltun et al. [Kolt01] proposed a conservative algorithm for computing from-region visibility in $2\frac{1}{2}$D scenes using *dual ray space*. Given a view cell and a region the algorithm computes a plane that is tangent to the view cell and the region. Visibility is then resolved in 2D using intersections of this plane and the scene occluders. The 2D visibility is solved using a mapping of rays to the discrete dual space. The algorithm exploits graphics hardware to perform operations on the duals of the occluded rays.

B.4.7 Hardware occlusion test

Newer graphics hardware provides an occlusion test for bounding boxes. The problem of this occlusion test is that the results of such an occlusion query are not readily available. A straightforward algorithm would therefore cause many unnecessary delays (pipeline stalls) in the rendering. The focus of research has now shifted to finding ways of ordering the scene traversal to interleave rendering and visibility queries in an efficient manner [Hey01], [Klos01].

B.5 Visibility in terrains

Visibility algorithms for terrains are important for applications such as flight simulation, urban visualization, path planning or telecommunications. Similarly to urban scenes (Section 2.5.1) the terrain can be considered a 2D height function. In comparison to the outdoor urban scenes the terrain is typically much smoother, i.e. it does not contain many height discontinuities. Furthermore the description of the terrain is typically much more regular (grid, octree) than description of an urban scene (general objects). As a result the terrain visibility methods are being developed independently from the algorithms for general and urban scenes. A survey of terrain visibility problems and algorithms was published by Nagy [Nagy94]. De Floriani and Magillo present a survey of visibility algorithms for triangulated terrains [Flor94].

The most common visibility problem in terrains is computing the *horizon* or the *viewshed* for a given viewpoint. Stewart [Stew96, Stew98b] proposed an algorithm for computing horizon for a given viewpoint. Cabral et al. [Cabr87] and Max [Max88] proposed approximate methods designed specifically for *bump maps*. Cohen-Or and Shaked [Cohe95] used sampling to determine visible and invisible parts of the terrain with respect to the given viewpoint. De Floriani and Magillo [Flor95] developed an algorithm computing horizon at different resolutions that can be updated at different levels of detail.

Stewart [Stew97] proposed a from-region $2\frac{1}{2}$D algorithm for visibility preprocessing in terrains. He maintains a multi-resolution representation of the terrain in a quadtree. For each quad the algorithm computes a conservative discrete *horizon*. The horizon for a quad is divided into sectors and for each sector an ordered list of vertices is determined that represent an *occlusion region* from which the quad is not visible. In real time the algorithm processes the quadtree recursively. If the viewpoint lies in an occlusion region of the current quad the corresponding subtree is culled.

The results from the computational geometry show that to compute a horizon of a point is equivalent to computation of an upper envelope of $O(n)$ segments for a terrain mesh consisting of n points [Atal83]. The horizon has $O(n\alpha(n))$[1] complexity and can be computed in $O(n \log n)$ time. Cole and Sharir [Cole89] and Bern et al. [Bern94] studied visibility in terrains in order to efficiently answer ray shooting queries.

B.6 Other acceleration techniques

Visibility culling is only one of many acceleration techniques for real-time rendering. It can yield a significant speedup for scenes with significant occlusion, but for example it will fail in outdoor scenes with unrestricted visibility. Visibility culling should be used in combination with other acceleration techniques to achieve an optimal performance. For example results of visibility computation can be used to drive the levels of detail [Carl00] or image based rendering [Lueb95]. We review the most important acceleration techniques and briefly discuss their links to the visibility computations where appropriate.

B.6.1 Geometry structuring

A general technique to increase rendering performance is to structure the geometry in a way convenient for the graphics hardware. Optimized geometry structuring saves calculations of transformation and lighting and reduces the bandwidth between the CPU and the graphics hardware. Geometry structuring can be successfully applied on most polygonal data and it is rather independent of visibility calculations.

Triangle strips and fans

Any polygonal model can be converted to a set of triangles. Connected triangles share an edge and two vertices. If the triangles are sent to the graphics hardware independently, there are redundant computations performed for the shared vertices. Additionally the same data is sent twice.

One way to reduce these overheads is to form strips of connected triangles [Evan96b]. Initially we sent the vertices of the first triangle in the strip and continue by sending one vertex per triangle. The algorithm uses a buffer for two vertices to which the current vertex should be connected. By default the triangle is connected to the edge formed by the last two vertices. If the connecting rule should be changed a special command indicating a *swap* is issued or a vertex is retransmitted. The *Iris GL* [Neid93] uses an additional bit for each new vertex indicating to which edge the new triangle should be connected. Triangle fan can be seen as a degenerated triangle strip where the opposite default connectivity rule is used. All triangles in the fan share a common vertex (see Figure B.11).

A generalization of triangle strips and fans for use with vertex buffers of more than two entries is called *generalized triangle mesh*. It was shown that even small buffer sizes can substantially improve the total rendering performance [Evan96b]. Construction of optimal triangle strips is a NP-complete problem [Evan96a, Evan96b]. Several heuristics for constructing good triangle strips have been proposed [Evan96b, Belm01, Stew01].

[1]$\alpha(n)$ is an inverse of the Ackermann function that can be considered a constant for any practical n.

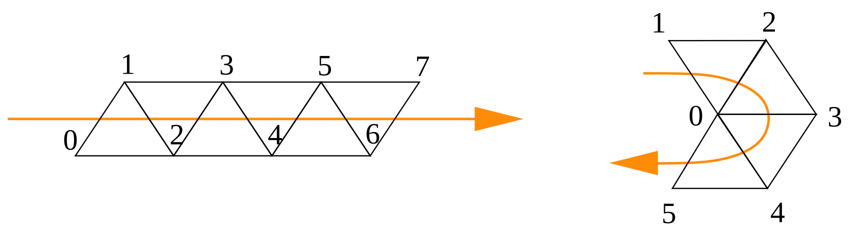


Figure B.11: A triangle strip and a triangle fan.

Indexed face sets

Indexed face sets provide a flexible representation of polygonal meshes [Fole90]. The mesh is stored as a set of vertices and their attributes, and a set of vertex and attribute indices for each face of the mesh. The typical attributes include normals per vertex, colors per vertex and texture coordinates per vertex. Colors and normals can also be specified on a per face basis.

Such a representation allows to eliminate the repeated transformation and lighting calculation for shared vertices. Nevertheless, care must be taken if this representation is used with the graphics hardware. All attributes and their transformed values should then be stored in the graphics hardware so that efficient indexing can take place. Incoherent access to the attributes can cause significant cache penalties.

B.6.2 Levels of detail

Complex objects can contain many small details that cannot be seen when observed from a distance. In real-time applications we often sacrifice the details in order to achieve high frame rates and a pleasant feedback. Complex objects can be simplified using *levels of detail* (LOD). The lower the level of detail the less polygons are used to represent the object. The choice of the LOD used in run-time is typically based on the distance of the object from the viewpoint, its semantic importance or the desired overall rendering time [Funk93]. Visibility computations can be used to guide the LOD selection algorithm [Carl00].

Discrete LOD

An object can be simplified using several LODs with progressively smaller amount of polygons [Clar76, Pupp97]. Distracting artifacts called *popping* can occur when switching between the LODs. Popping can be eliminated by continuous blending between the LODs at the cost of additional computations in real time [Cohe98b, Zach02].

Continuous LOD

The idea of continuous LOD is to perform a series of elementary simplifications of an object in real time. The simplifications can adapt to the current viewing conditions. This ability is important namely in large terrains, where the terrain close to the observer should be finely tessellated whereas the farther parts can use rather coarse representation.

The typical elementary simplification is the *edge collapse* [Hopp96, Hopp97]. A triangle mesh gets progressively simplified by a series of edge collapses. At each edge collapse the nearby vertices are shifted to better express the shape of the object. By using a history of modifications the original shape can be restored. The *geomorph* LODs use smooth vertex transformations to avoid popping when the edge is collapsed or expanded [Hopp98, Grab01].

B.6.3 Image-based rendering

Image based rendering has become an alternative for rendering complex environments in real time [Shad96, Scha96, Alia97, Scha97]. Many image based rendering techniques use impostors: an impostor represents a complex geometry by a combination of simple geometry and a texture. The impostor can be reused until it is no longer a satisfying representation of the view of the represented geometry. The associated rendering cost mostly depends on the number of pixels in the final image and not on the scene complexity.

Probably the simplest impostor is a quadrilateral with texture representing a rather flat or distant object. The texture includes opacity component to capture boundaries of the object. Such impostors have long been used in computer games for trees. The simple quadrilateral impostor often appears very flat and unnatural. Many techniques have been designed to improve the quality of representation as well as to allow an automatic construction and updating of impostors [Wimm01, Jesc02]. Many recent techniques exploit the idea of *layered depth images* to capture occluded parts of the model in an image based representation [Shad98, Pope98, Alia99]. McMillan [McMi97] proposed an algorithm for warping images with depth information from one viewpoint to another. The algorithm resolves visibility by a correct occlusion compatible traversal of the input image.

Nailboards

Nailboards proposed by Schaufler [Scha97] are impostors with a texture including a depth component instead of opacity. This depth information is used to obtain correct depth values in the framework of z-buffered rendering. The depth stored in the nailboard is an offset from the nailboard polygon using only a small number of bits.

3D image cache

The 3D image cache proposed independently by Schaufler et al. [Scha96] and Shade et al. [Shad96] allows an automatic generation of impostors suitable for walkthroughs and flyovers of large 3D scenes with unrestricted visibility.

A spatial hierarchy is built over the whole scene. For each sufficiently distant node of the hierarchy an impostor is generated that represents all geometry contained in that node. For leaf nodes the impostor is generated from the geometry, impostors for intermediate nodes are regenerated from impostors lower in the hierarchy. For obtaining the final image the hierarchy is traversed in a front to back order and the impostors are drawn yielding a correct visibility.

B.6.4 Point-based rendering

Point-based rendering is another alternative to the conventional polygon-based rendering. It is useful for highly complex models, which would otherwise require a huge number of triangles. Most point-based rendering algorithms project points on the screen using *splatting* and apply some *filtering* technique to render the resulting image. Splatting implicitly reconstructs the local topology of the model to avoid gaps in the image and to resolve visibility of projected points.

Pfister et al. [Pfis00] use software visibility splatting to identify holes in the image. Most other techniques use a hardware accelerated z-buffer. Grossman and Dally [Gros98] use a variant of the hierarchical z-buffer [Gree93] to resolve visibility. Rusinkiewicz and Levoy [Rusi00] use splats of different shapes to improve the quality of the image, particularly of the object silhouette. The randomized z-buffer algorithm proposed by Wand et al. [Wand01] samples a large triangle mesh to accelerate its rendering. The mesh is preprocessed and the algorithm selects sufficient number of sample points so that each pixel receives at least one sample point from a visible triangle with high probability.

Appendix C

Visibility in realistic rendering

The goal of realistic rendering techniques is to generate a (photo)realistic image of a virtual scene using an accurate simulation of the light propagation. In this chapter we first review methods that aim to increase realism of the synthesized image by including shadows and discuss how these methods make use of visibility computations. We first review methods computing *hard shadows*, i.e. shadows due to point light sources. Then we discuss algorithms for *soft shadows*, i.e. shadows due to areal light sources. Then we discuss visibility algorithms for the *radiosity* method and review work on efficient ray shooting that serves as a core of most recent *global illumination* algorithms. Finally, we discuss several *global visibility* algorithms.

C.1 Hard shadows

The presence of shadows in a computer generated image significantly increases its realism [Woo90b, Slat92, Moll02]. Shadows provide important visual cues about positions and sizes of objects in the scene. Traditionally shadow algorithms have been the domain of realistic rendering and they were too slow for the use in real time. Nowadays some algorithms, particularly these exploiting graphics hardware, are often suitable for a real-time application.

C.1.1 Ray tracing

The ray tracing [Whit79] algorithm simulates the light propagation by tracing rays from the eye back to the scene. At each intersection with an object a shadow ray is cast to each point light source. This point-to-point visibility query determines if the given point is in shadow (light source is invisible) or lit (light source is visible). Ray tracing does not explicitly reconstruct shadow volumes. It samples points on the light path for inclusion in the shadow independently of each other.

Tracing of shadow rays can be significantly accelerated by using the *light buffer* introduced by Haines and Greenberg [Hain86]. The light buffer is an array corresponding to the projection of the scene centered at the light source. Each element of the array forms a list of objects that need to be tested for an intersection if the query point projects to this element. Another speedup can be achieved by precomputing shadowed regions in the scene and storing this information within the spatial subdivision [Woo90a]. More details on the ray shooting acceleration techniques will be presented in Section C.5.

C.1.2 Shadow maps

A shadow map proposed by Williams [Will78] is a discrete representation of shadows due to a single point light source. It is a 2D array corresponding to a projection of the scene centered at the light source. Each element of the array contains the depth of the closest object that projects to that element.

The algorithm first constructs a shadow map by rendering the scene using the z-buffer algorithm from the light source point of view [Sega92, Heid99]. Then the scene is rendered using a given view and projecting visible points to the shadow map. The depth of the point is compared to the value stored at the shadow map. If the point is farther than the stored value it is in shadow. This algorithm can be accelerated significantly using extensions of the current graphics hardware [Sega92].

Similarly to ray tracing, shadow maps can represent shadow due to objects defined by complex surfaces, i.e. any object that can be rasterized into the shadow map is suitable. In contrast to ray tracing the shadow map approach explicitly reconstructs the shadow volume. The shadow volume is represented in a discrete form as a set of rays defined by non empty elements of the shadow map. The disadvantage of the approach is that the discrete representation of the shadow volume can lead to severe aliasing. Many techniques have been proposed to reduce the aliasing effects [Gran92].

C.1.3 Shadow volumes

The shadow volume of a polygon with respect to a point is a semi-infinite frustum. The intersection of the frustum with the scene bounding box can be explicitly reconstructed and represented by a set of polygons bounding the frustum (see Figure C.1). Crow [Crow77] observed that these polygons can be used to test if a point p corresponding to the pixel of the rendered image as by counting number of polygons *in front* of p and *behind* p. Denote n_f number of polygons in front of p and n_b number of polygons behind p. Then if $n_f - n_b > 0$ the point is in shadow.

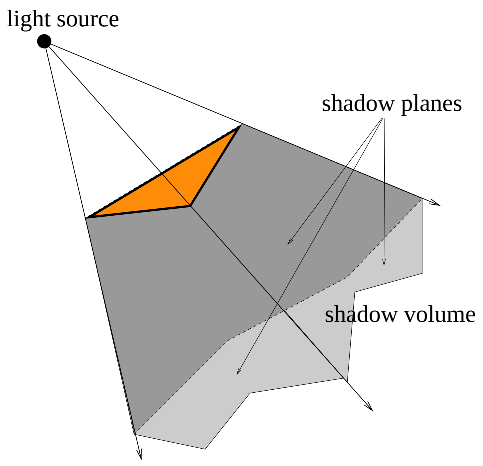


Figure C.1: Shadow volume due to a polygon. The volume is bounded by four planes: the supporting plane of the polygon and three shadow planes.

An efficient real-time implementation of this technique proposed by Heidmann [Heid91] uses OpenGL *stencil buffer* to render the shadow polygons and to count the $n_f - n_b$ for each pixel in the image. Than the color of all pixels with $n_f - n_b > 0$ can be adjusted accordingly.

C.1.4 Shadow volume BSP trees

The *Shadow Volume BSP* (SVBSP) tree proposed by Chin and Feiner [Chin89] is a variant of the BSP tree for the representation of polyhedra [Thib87, Nayl90b]. The SVBSP tree represents a union of shadow volumes cast by convex polygons (occluders). Each internal node of the tree is associated with a *shadow plane* passing through the light source and an edge of the occluder.

The direction of the shadow plane normal is used to determine the half-space in which the occluder and its shadow are located. Each leaf node of the tree corresponds to a semi-infinite pyramid. The leaves are classified as in or out. An in-leaf is associated with a scene polygon that truncates the corresponding pyramid to the shadow frustum. An out-leaf represents an unoccluded pyramid and hence the volume lit by the light source. The complete shadow volume is a union of all shadow frusta corresponding to in-leaves. An example of the SVBSP tree is depicted in Figure C.2.

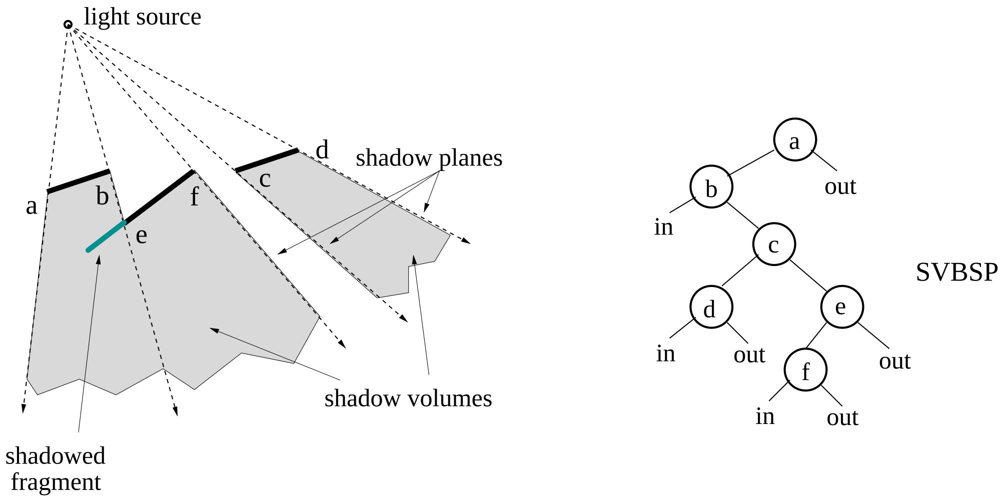


Figure C.2: A 2D example of the SVBSP tree constructed for three polygons.

The SVBSP tree is constructed incrementally by processing the polygons in the front-to-back order with respect to the light source. The contribution of one polygon to the tree is determined as follows:

1. Lit fragments of the polygon are determined.
2. The tree is enlarged by shadow volumes cast by the lit fragments.

The lit fragments of the polygon are determined by *filtering* the polygon down the tree. The filtering is a constrained *depth first traversal* of the tree that starts at the root. For the current node the position of the polygon with respect to node's shadow plane is determined. If the polygon lies completely on the back or the front side of the shadow plane, it is filtered down the back or the front child of the node, respectively. Otherwise, the polygon is split by the shadow plane into two fragments that are filtered down the appropriate children.

Reaching leaf nodes a set of convex polygonal fragments is obtained. These are either lit (out-leaves) or shadowed (in-leaves). In the out-leaves the tree is enlarged by the shadow volumes cast by the lit fragments. For each lit fragment new nodes defined by edges of the fragment are used to replace the corresponding leaf. The lit and shadowed fragments are stored within the original polygon. During rendering the illumination of these fragments can be set accordingly.

The front-to-back ordering of polygons can be achieved by building a BSP tree for the scene polygons and its appropriate traversal [Fuch80]. Alternatively, methods exploiting occlusion dependency graphs can be used [Chen96, Mill96, Jame98, Snyd98]. The SVBSP tree algorithm was designed for static scenes. Chrysanthou and Slater [Chry95] extended the method for dynamic scenes.

C.2 Soft shadows

Rendering soft shadows is significantly more difficult than rendering hard shadows. Soft shadows appear in scenes with areal light sources and/or in scenes with secondary illumination due to diffuse reflections.

A shadow due to an areal light source consists of two parts: *umbra* and *penumbra*. Umbra is a part of the shadow from which the light source is invisible. Penumbra is that part from which the light source is partially visible and partially hidden by some scene objects.

C.2.1 Ray tracing and areal light sources

A straightforward extension of the ray tracing algorithm handles areal light sources by shooting randomly distributed shadow rays towards the light source [Cook84]. The illumination due to the light source is determined by stochastic sampling and Monte Carlo integration.

C.2.2 Soft shadow maps

Heidrich et al. [Heid00] proposed an extension of the shadow map approach for *linear* light sources. They use a discrete shadow map enriched by a *visibility channel* to render soft shadows at interactive speeds. The visibility channel consists of estimates of percentage of the light source visible at the corresponding points. First, the shadow maps are rendered from the endpoints of the linear light source. Then an edge detection algorithm is applied to detect depth discontinuities in the shadow maps. The detected edges are used to form *skin* polygons that are warped into the other shadow map. The visibility channel is updated by merging the visibility estimates with the linearly interpolated estimated visibility corresponding to the skin polygon. Soler and Sillion [Sole98] proposed a shadow map algorithm handling areal light sources. They use convolution to compute visibility estimates in the shadow maps.

C.2.3 Shadow volumes for soft shadows

Heckbert and Herf [Heck97] proposed an algorithm extending the concept shadow volumes to areal light sources. A shadow volume is constructed for each vertex of the light source as with the point light sources. The illumination of a point in shadow is modified according to the number of shadow volumes it is contain in.

C.2.4 Shadow volume BSP tree for soft shadows

An adaptation of the SVBSP tree method for areal light sources was proposed by Chin and Feiner [Chin92]. This approach was further elaborated by Chrysanthou [Chry96] and Chrysanthou and Slater [Chry97] for the use in dynamic scenes.

C.3 Global illumination

Realistic image synthesis aims to solve the *rendering equation* formulated by Kajiya [Kaji86]. The rendering equation expresses the light transport intensity between two surface points x and x' in the scene as:

$$\begin{aligned} I(x,x') &= \boldsymbol{v}(x,x')[e(x,x') + && \text{(C.1)} \\ & \int\limits_{\Omega} \rho(x,x',x'')I(x',x'')dx''] && \text{(C.2)} \end{aligned}$$

where:

- $\boldsymbol{v}(x,x')$ is the *visibility function* that equals 1 is the points x and x' are visible and 0 otherwise,
- $e(x,x')$ is the transfer emittance from x' to x,
- $\rho(x,x',x'')$ is the unoccluded three-point transport reflectance. related to BRDF

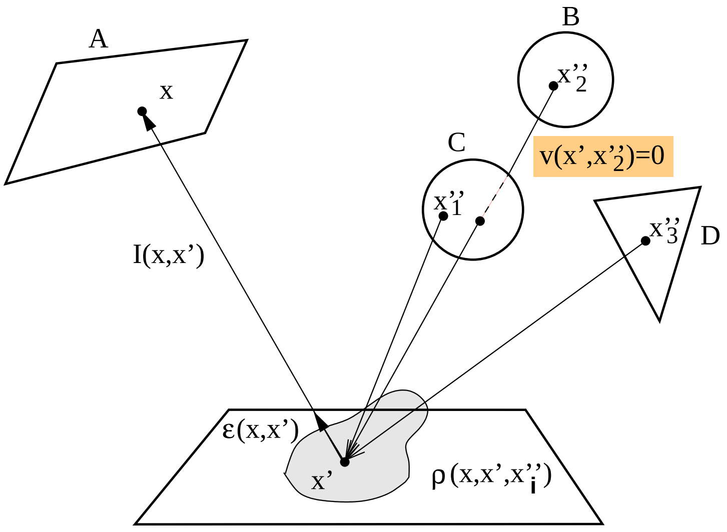


Figure C.3: Geometry involved in the rendering equation. Points x' and x''_2 are occluded and therefore $v(x', x''_2) = 0$.

The terms of the rendering equation are illustrated in Figure C.3. The rendering equation provides context for evaluation of the rendering algorithms according to the quality of the approximation to the equation they provide. The evaluation of visibility function is one of the most demanding tasks in global illumination. The algorithms evaluating the visibility function differ in dependence on the particular rendering algorithm. For example in the classical radiosity algorithm the visibility function is evaluated simultaneously for a whole set of rays that intersect the given patch, whereas in most stochastic methods visibility is evaluated by shooting rays independently.

C.4 Radiosity

The classical *radiosity* method [Gora84] simulates light propagation by solving a system of linear equations describing the light transfers. It handles scenes that consist of perfect diffuse surfaces. The surfaces are subdivided into patches and the *radiosity* B_i of patch A_i is expressed as:

$$B_i = E_i + R_i \sum_{j=1}^{n} B_j F_{ij} \qquad [\frac{W}{m^2}] \tag{C.3}$$

where, E_i is the energy emitted from the patch and F_{ij} is a *form factor* expressing the mutual transfer of energy between patches A_i and A_j. F_{ij} is given as:

$$F_{ij} = \frac{\text{energy leaving patch } A_i \text{ that directly hits } A_j}{\text{energy leaving patch } A_i \text{ in all directions}} \qquad [-] \tag{C.4}$$

This can be expressed as:

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \boldsymbol{v}(dA_i, dA_j) \frac{cos\phi_i cos\phi_j}{\pi r^2} dA_j dA_i \tag{C.5}$$

where $\boldsymbol{v}(dA_i, dA_j)$ is the visibility term that equals 1 if the differential areas dA_i and dA_j are visible and 0 otherwise (see Figure C.4). In the next sections we discuss several methods of computing form factors.

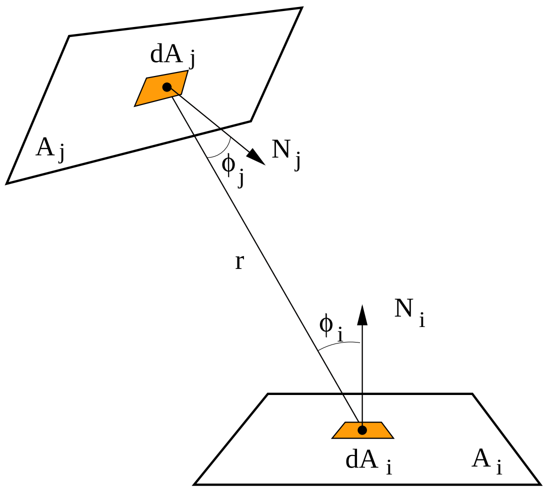


Figure C.4: Geometry for computing a form factor between two polygons.

C.4.1 Ray shooting and form factors

The form factor can be evaluated by shooting random rays and using Monte Carlo integration. This approach was used by several researches [Wall89, Geor90, Shir95, Slus97, Stam98, Stam97].

C.4.2 Hemisphere and hemicube

A simplified form factor evaluation between two patches computes a form-factor of a differential area of P_i with respect to the other patch P_j. The form-factor between the two patches is estimated assuming that the differential form-factor is almost constant across P_i.

The *hemisphere* method [Szir95] projects the visible part of the polygon on a unit hemisphere centered at the middle of the patch. The *hemicube* algorithm proposed by Cohen and Greenberg [Cohe85] uses five item buffers that are erected above the given patch. The scene is rendered into the item buffers using a perspective projection from the center of the patch. The item buffers are scanned and their entries are used to estimate the form factor between the given patch and the patch corresponding to the entry. There are two sources of error in the hemicube algorithm: the finite resolution of the hemicube and the fact that visibility is sampled only at the center of the patch.

C.4.3 BSP trees

Campbell [Camp90] proposed an algorithm computing point-to-polygon form factors using a variant of the Shadow Volume BSP tree. For each vertex of the given patch the algorithm constructs a BSP tree representing a view of the scene from this vertex. The BSP tree is used to efficiently identify fragments of visible patches from the vertex. Then the point-to-polygon form factor is evaluated analytically using a visible fragment of the given patch.

C.4.4 Discontinuity meshing

Following the work of Campbell [Camp90, Camp91], Heckbert [Heck92] and Lischinski et al. [Lisc92] proposed a precise computation of form factors by computing a discontinuity mesh with respect to the light source. A discontinuity mesh is a partition of the scene into patches so that each patch “sees” a topologically equivalent view of the light source. The view of the light source from the patch is called *backprojection*. Boundaries of the resulting mesh correspond to loci of discontinuities in the illumination function.

Discontinuity meshing can be used to analytically calculate form factors with respect to an areal light source: For each patch of the discontinuity mesh that is visible from the light source the form factor can be evaluated using a corresponding backprojection of the light source from the patch [Schr93].

The algorithms of Heckbert [Heck92] and Lischinski et al. [Lisc92] construct a subset of discontinuity mesh by casting linear surfaces corresponding to the vertex-edge visibility events. More elaborated methods capable of creating a complete discontinuity mesh were proposed by Drettakis [Dret94a], Drettakis and Fiume [Dret94b], and Stewart and Ghali [Stew93, Stew94].

C.5 Ray shooting in global illumination

Ray shooting is crucial for many global illumination algorithms that sample light paths by tracing a huge number of rays. The first global illumination algorithm is ray tracing [Whit79]. In its original form ray tracing simulates only specular reflections and point light sources. Many modern methods solve the rendering equation by Monte Carlo integration [Arvo90, Veac94, Kell97, Slus97, Veac97]. All these methods invoke a huge amount of ray shooting queries to sample visibility within the scene.

Ray shooting is also used within many hybrid approaches that combine finite element approach (radiosity) with the stochastic algorithms [Sill89, Kok92, Kok93, Jens95, Tobl97]. In the remainder of this section we review the work on ray shooting acceleration.

C.5.1 Ray shooting acceleration techniques

A naive ray shooting algorithm tests all objects for intersection with a given ray to find the closest visible object along the ray in $O(n)$ time. For complex scenes this linear time complexity is very restrictive since a huge amount of rays is cast to synthesize an image.

Researchers from the computational geometry community aim to establish tight bounds of the time and memory complexities of the proposed algorithms. In order to obtain theoretically provable results the scene description is usually restricted to polygons. De Berg proposed an algorithm [Berg93b] that reaches $O(\log n)$ time complexity with $O(n^{4+\epsilon})$ preprocessing time and storage, where ϵ is an arbitrarily small positive constant, and n is a number of polygons. It was shown that the $O(\log n)$ time complexity is asymptotically optimal [Szir97]. The worst-case-optimal results are important theoretically, but the initial assumptions, the storage complexity and implementation problems usually disallow their use for the real-world data.

An overview of practical acceleration techniques for ray shooting was given by Arvo [Arvo89]. A recent comprehensive survey was presented by Havran [Havr00a]. Most of the proposed methods organize the scene objects in spatial indexing data structures, that allow efficient determination of objects that can possibly be pierced by a given ray. The goal is to minimize the number of ray/object intersection tests. The algorithms use various acceleration data structures such as uniform grids [Fuji86], bounding volume hierarchies [Gold87], octrees [Same89b, Peng87], kD-trees [Fuss88, MacD90, Subr91], irregular grids [Silv97], hierarchies of sorted lists [Four93], or hierarchical grids [Caza95, Caza97a, Klim97]. Some hybrid approaches combine several hierarchical data structures [Subr90, Subr92]. Other methods build on a subdivision of line space or ray space [Arvo87, Simi94, Kwon98]. The main disadvantage of these methods is the high memory complexity of a discrete subdivision of the higher-dimensional (4D or 5D) space.

Another possibility of acceleration is to reduce the number of rays that are traced. Firstly, the coherence of radiance along similar rays can be utilized [Akim91, Loof93, Kok93, Tell96, Havr00b]. Secondly, a visibility preprocessing can avoid ray shooting between invisible regions of the scene [Hain86, Woo90a, Luka98]. Thirdly, some methods trace infinitely many rays enclosed in a beam [Heck84, Zwaa95, Rajk96, Funk98, Tell98].

The *ray bundle tracing* proposed by Szirmay-Kalos [Szir98a, Szir98b] shoots a set of parallel rays through the scene according to randomly sampled direction. This approach allows to exploit a visible surface algorithm for tracing many rays at the same time. When using the z-buffer the algorithm makes use of the dedicated graphics hardware. The disadvantage of the method is that the rays are highly correlated and in general more rays need to be traced to obtain sufficiently precise and unbiased results.

Further acceleration of ray shooting can be achieved by speeding up the ray/object intersection itself. This is usually carried out by object bounding volumes such as simple axis aligned boxes. A different approach that uses duality to speedup the ray/polyhedron intersection was proposed by Kolingerová [Koli97]. Recently, there have been attempts to accelerate ray shooting using SIMD extensions of modern CPUs [Wald02] or a dedicated graphics hardware [Purc02].

Ray shooting can also be accelerated by exploiting temporal coherence of subsequent frames within animation sequences [Glas88, Badt88, Chap90, Jeva92, Bish94, Grol93].

C.5.2 Ray shooting using uniform grid

The *uniform grid* (UG) is one of the first accelerating data structures for ray shooting [Fuji86]. It subdivides the scene into equally sized cells. Within each cell a list of objects that intersect the cell is maintained. Since the UG does not adapt to the distribution of the scene geometry a large number of UG cells can be vacant. On the other hand the ray shooting algorithm for the UG can be implemented very efficiently using a 3D-DDA algorithm [Aman87, Fuji86, Clea88].

Starting from the cell containing the ray origin the UG is traversed along the ray direction. Within each cell objects contained in the corresponding list are tested for intersection with the ray (see Figure C.5). The traversal is terminated when an object intersecting the ray is found or all relevant cells of the UG have been visited. Typically many cells contain no objects, thus the traversal algorithm needs to perform many steps until the intersection is found. To overcome this problem some methods use precomputed zones where no ray/object intersection can occur [Devi89, Cohe94, Semw97].

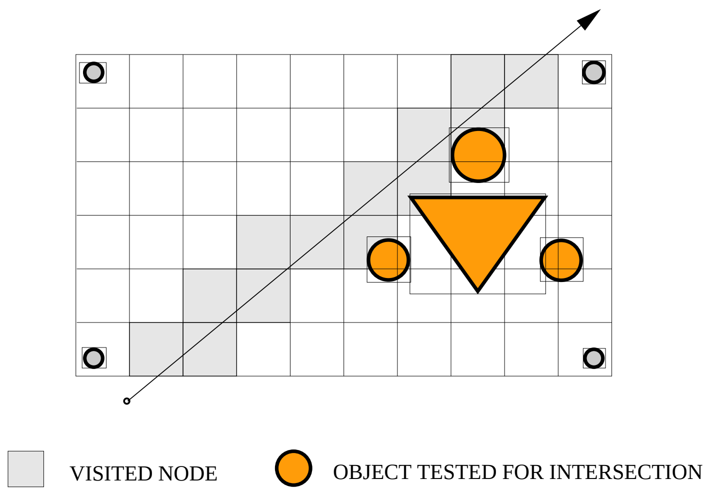


Figure C.5: Ray shooting using the uniform grid.

C.5.3 Ray shooting using kD-tree

kD-trees is a common data structure used to organize multi-dimensional data [Bent75, Same89a]. The fundamental factor influencing the properties of a kD-tree is the positioning of partitioning planes. The construction of kD-trees for ray shooting purposes was first studied by Kaplan [Kapl85]. A more elaborate method of MacDonald and Booth [MacD90] uses a *surface area heuristics* to estimate the cost of the resulting tree. The kD-tree is built using a greedy algorithm that minimizes the expected cost of the tree when choosing a partitioning plane [Havr00a]. Kaplan [Kapl85] introduced the *sequential* traversal algorithm for the kD-tree. A more efficient *recursive* algorithm was proposed by Jansen in [Jans86] and further elaborated by Arvo [Arvo88], Sung and Shirley [Sung92] and Havran et al. [Havr98b]. The recursive algorithm performs a constrained depth first traversal starting from the root of the tree. The traversal order and its constraints are determined by the mutual position of the ray and the partitioning plane corresponding to the currently visited node.

The ability of the kD-trees to adapt to the scene geometry is paid by the overhead of the traversal of the interior nodes of the hierarchy. Additionally even if the location of the origin of the ray is known the recursive traversal proceeds again from the root of the tree. A method overcoming these problems uses neighbor-links (ropes) [MacD90, Havr98a] in the scope of a non-recursive traversal algorithm. Ray shooting using kD-tree is illustrated in Figure C.6.

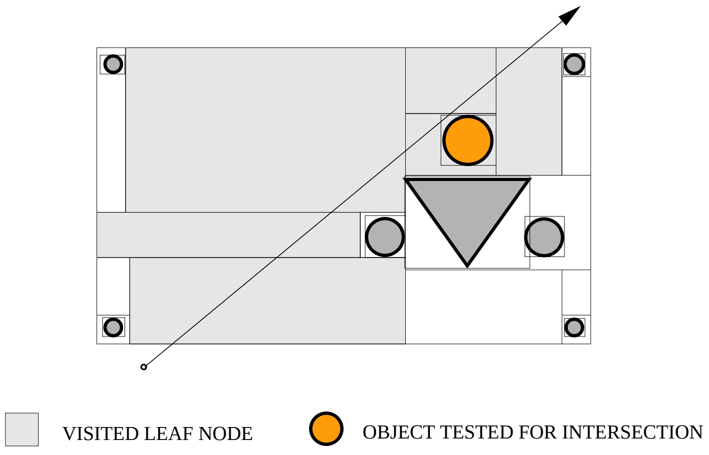


Figure C.6: Ray shooting using the kD–tree.

C.6 Global visibility

The aim of *global visibility* computations is to capture and describe visibility in the whole scene [Dura96]. These methods 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 is still an open problem. Prospectively these techniques provide an elegant method for ray shooting acceleration — the ray shooting problem can be reduced to point location in the line space subdivision.

C.6.1 Visibility complex

Pocchiola and Vegter introduced the visibility complex [Pocc93] that describes global visibility in two-dimensional scenes. The visibility complex has been applied to solve various visibility problems in the plane [Rivi95, Rivi97b, Rivi97a, Orti96]. The visibility complex was generalized to 3D by Durand et al. [Dura96]. Nevertheless, no implementation of the 3D visibility complex is known.

C.6.2 Visibility skeleton

Durand et al. [Dura97] introduced the *visibility skeleton*. Visibility skeleton is a graph describing the skeleton of the 3D visibility complex. The visibility skeleton was implemented and verified experimentally. The results indicate that its worst case complexity $O(n^4 \log n)$ is much better in practice. Recently Duguet and Drettakis [Dugu02] improved the robustness of the method by using robust epsilon-visibility predicates.

Bibliography

[Agar97] P. Agarwal, T. Murali, and J. Vitter. Practical Techniques for Constructing Binary Space Partitions for Orthogonal Rectangles. In *Proceedings of ACM Symposium on Computational Geometry*, pp. 382–384, 1997. Cited on page 31.

[Aire90] J. M. Airey, J. H. Rohlf, and F. P. Brooks, Jr. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In *Proceedings of Symposium on Interactive 3D Graphics*, pp. 41–50, ACM SIGGRAPH, March 1990. Cited on page 2, 24, 39, 84, 100, xvii.

[Akim91] T. Akimoto, K. Mase, and Y. Suenaga. Pixel-Selected Ray Tracing. *IEEE Computer Graphics and Applications*, Vol. 11, No. 4, pp. 14–22, July 1991. Cited on page xxix.

[Alia97] D. G. Aliaga and A. A. Lastra. Architectural Walkthroughs Using Portal Textures. In *Proceedings of IEEE Visualization '97*, pp. 355–362, IEEE, Nov. 1997. Cited on page xxii.

[Alia99] D. G. Aliaga and A. Lastra. Automatic Image Placement to Provide a Guaranteed Frame Rate. In *Computer Graphics (SIGGRAPH '99 Proceedings)*, pp. 307–316, Aug. 1999. Cited on page xxii.

[Aman84] J. Amanatides. Ray Tracing with Cones. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, pp. 129–135, July 1984. Cited on page 27, 28.

[Aman87] J. Amanatides and A. Woo. A Fast Voxel Traversal Algorithm for Ray Tracing. In *Proceedings of Eurographics '87*, pp. 3–10, Aug. 1987. Cited on page xxx.

[Appe68] A. Appel. Some Techniques for Shading Machine Renderings of Solids. In *AFIPS 1968 Spring Joint Computer Conf.*, pp. 37–45, 1968. Cited on page viii.

[Arvo87] J. Arvo and D. Kirk. Fast Ray Tracing by Ray Classification. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, pp. 55–64, July 1987. Cited on page 99, xxix.

[Arvo88] J. Arvo. Linear-time Voxel Walking for Octrees. *Ray Tracing News*, Vol. 1, No. 5, 1988. Cited on page xxxi.

[Arvo89] J. Arvo and D. Kirk. *A survey of ray tracing acceleration techniques*, pp. 201–262. Academic Press, 1989. Cited on page 4, xxix.

[Arvo90] J. Arvo and D. Kirk. Particle Transport and Image Synthesis. In *Computer Graphics (Proceedings of SIGGRAPH'90)*, pp. 63–66, Aug. 1990. Cited on page xxix.

[Asan00] T. Asano, S. K. Ghosh, and T. C. Shermer. Visibility in the Plane. In H. of Computational Geometry, J.-R. Sack, and J. Urrutia, Eds., *Elsevier, 2000*, 2000. Cited on page 69.

[Assa00] U. Assarsson and T. Möller. Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools*, Vol. 5, No. 1, pp. 9–22, 2000. Cited on page 39, 46, 51, x.

[Atal83] M. Atallah. Dynamic Computational Geometry. In *Proceedings of 24th Symposium on Foundations of Computer Science*, pp. 92–99, IEEE Computer Society, Baltimora, 1983. Cited on page xx.

[ATi 02] ATi Co. Graphics hardware specifications. 2002. `http://www.ati.com`. Cited on page ix.

[Avis02] D. Avis. LRS polyhedra enumeration library. 2002. Available at `http://cgm.cs.mcgill.ca/~avis/C/lrs.html`. Cited on page 90, 113, 119.

[Avis96] D. Avis and K. Fukuda. Reverse Search for Enumeration. *Discrete Applied Mathematics*, Vol. 6, pp. 21–46, 1996. Cited on page 90, 113, 119, 132.

[Badt88] S. Badt, Jr. Two Algorithms for Taking Advantage of Temporal Coherence in Ray Tracing. *The Visual Computer*, Vol. 4, No. 3, pp. 123–132, Sep. 1988. Cited on page xxx.

[Baja96] C. L. Bajaj and V. Pascucci. Splitting a Complex of Convex Polytopes in any Dimension. In *Proceedings of 12th Annual ACM Symposium on Computational Geometry*, pp. 88–97, 1996. Cited on page 119.

[Bart98] D. Bartz, M. Meissner, and T. Hüttner. Extending Graphics Hardware for Occlusion Queries in OpenGL. In *Proceedings of the 1998 Workshop on Graphics Hardware, Lisbon, Portugal*, pp. 97–104, 1998. Cited on page 40.

[Bart99] D. Bartz, M. Meißner, and T. Hüttner. OpenGL-assisted occlusion culling for large polygonal models. *Computers and Graphics*, Vol. 23, No. 5, pp. 667–679, Oct. 1999. Cited on page 41.

[Belm01] O. Belmonte, J. Ribelles, I. Remolar, and M. Chover. Searching Triangle Strips Guided by Simplification Criterion. In *Proceedings of WSCG '01*, 2001. Cited on page xx.

[Bent75] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, Vol. 18, No. 9, pp. 509–517, Sep. 1975. Cited on page 35, xxx.

[Berg93a] M. de Berg. Generalized hidden surface removal. In *Proceedings of the 9th Annual Symposium on Computational Geometry (SCG '93)*, pp. 1–10, ACM Press, San Diego, CA, USA, May 1993. Cited on page 39.

[Berg93b] M. de Berg. Ray shooting, Depth Orders and Hidden Surface Removal. In *Lecture Notes in Computer Science*, Springer Verlag, New York, 1993. Cited on page 39, xxix.

[Berg97] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, Heidelberg, New York, 1997. Cited on page 21, 31, 37, v.

[Bern94] M. W. Bern, D. P. Dobkin, D. Eppstein, and R. L. Grossman. Visibility with a Moving Point of View. *Algorithmica*, Vol. 11, No. 4, pp. 360–378, Apr. 1994. Cited on page xx.

[Bish94] G. Bishop, H. Fuchs, L. McMillan, and E. J. Scher Zagier. Frameless Rendering: Double Buffering Considereed Harmful. In A. Glassner, Ed., *Computer Graphics (SIGGRAPH '94 Proceedings)*, pp. 175–176, ACM Press, July 1994. ISBN 0-89791-667-0. Cited on page xxx.

[Bish98] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. Designing a PC Game Engine. *IEEE Computer Graphics and Applications*, Vol. 18, No. 1, pp. 46–53, Jan./Feb. 1998. Cited on page ix.

[Bitt01a] J. Bittner and V. Havran. Exploiting Coherence in Hierarchical Visibility Algorithms. *Journal of Visualization and Computer Animation, John Wiley & Sons*, Vol. 12, pp. 277–286, 2001. Cited on page 5.

[Bitt01b] J. Bittner and V. Havran. Exploiting Temporal and Spatial Coherence in Hierarchical Visibility Algorithms. In *Proceedings of Spring Conference on Computer Graphics (SCCG'01)*, pp. 213–220, IEEE Computer Society, Budmerice, Slovakia, 2001. Cited on page 5.

[Bitt01c] J. Bittner and J. Přikryl. Exact Regional Visibility using Line Space Partitioning. Tech. Rep. TR-186-2-01-06, Institute of Computer Graphics and Algorithms, Vienna University of Technology, March 2001. Available as `ftp://ftp.cg.tuwien.ac.at/pub/TR/01/TR-186-2-01-06Paper.ps.gz`. Cited on page 5, 13.

[Bitt01d] J. Bittner and P. Slavík. Exact Regional Visibility using Line Space Partitioning. Nov. 2001. Submitted to the Computers & Graphics journal. Cited on page 5.

[Bitt01e] J. Bittner, P. Wonka, and M. Wimmer. Visibility Preprocessing for Urban Scenes using Line Space Subdivision. In *Proceedings of Pacific Graphics (PG'01)*, pp. 276–284, IEEE Computer Society, Tokyo, Japan, 2001. Cited on page 5, 23, 82, 100, ix.

[Bitt02a] J. Bittner. Efficient Construction of Visibility Maps using Approximate Occlusion Sweep. In *Proceedings of Spring Conference on Computer Graphics (SCCG'02)*, pp. 163–171, Budmerice, Slovakia, 2002. Cited on page 5.

[Bitt02b] J. Bittner and P. Wonka. Visibility in Computer Graphics. June 2002. Submitted to Journal of Environmental Planning. Cited on page 5.

[Bitt97] J. Bittner. *Global Visibility Computations*. Master's thesis, Department of Computer Science and Engineering, Czech Technical University Prague, January 1997. Also available as `http://www.cgg.cvut.cz/˜bittner/masters.ps.gz`. Cited on page 90, 119, 132.

[Bitt98] J. Bittner, V. Havran, and P. Slavík. Hierarchical Visibility Culling with Occlusion Trees. In *Proceedings of Computer Graphics International '98 (CGI'98)*, pp. 207–219, IEEE, 1998. Cited on page 5, 30, 36, 46, 59, 60.

[Bitt99] J. Bittner. Hierarchical Techniques for Visibility Determination. Tech. Rep. DS-005, Department of Computer Science and Engineering, Czech Technical University in Prague, March 1999. Also available as `http://www.cgg.cvut.cz/˜bittner/publications/minimum.ps.gz`. Cited on page 101.

[Blai98] M. Blais and P. Poulin. Sampling Visibility in Three-Space. In *Proc. of the 1998 Western Computer Graphics Symposium*, pp. 45–52, Apr. 1998. Cited on page 101.

[Bois98] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, 1998. Cited on page 102.

[Brot84] L. S. Brotman and N. I. Badler. Generating Soft Shadows with a Depth Buffer Algorithm. *IEEE Computer Graphics and Applications*, Vol. 4, No. 10, pp. 5–12, Oct. 1984. Cited on page 101.

[Bueh01] C. Buehler, M. Bosse, L. McMillan, S. J. Gortler, and M. F. Cohen. Unstructured Lumigraph Rendering. In *Computer Graphics (SIGGRAPH '01 Proceedings)*, pp. 425–432, 2001. Cited on page 102.

[Cabr87] B. Cabral, N. Max, and R. Springmeyer. Bidirectional Reflection Functions from Surface Bump Maps. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, pp. 273–281, July 1987. Cited on page xix.

[Camp90] A. T. Campbell III and D. S. Fussell. Adaptive Mesh Generation for Global Diffuse Illumination. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, pp. 155–164, Aug. 1990. Cited on page 3, 101, xxviii.

[Camp91] A. T. Campbell, III. *Modeling Global Diffuse Illumination for Image Synthesis*. PhD thesis, CS Dept, University of Texas at Austin, Dec. 1991. Tech. Report TR-91-39. Cited on page 101, xxviii.

[Carl00] I. N. Carlos Andújar, Carlos Saona-Vázquez and P. Brunet. Integrating Occlusion Culling with Levels of Detail through Hardly-Visible Sets. In *Computer Graphics Forum (Proceedings of Eurographics '00)*, pp. 499–506, 2000. Cited on page xx, xxi.

[Carp84] L. Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In H. Christiansen, Ed., *Computer Graphics (SIGGRAPH '84 Proceedings)*, pp. 103–108, July 1984. Cited on page iii.

[Catm75] E. E. Catmull. Computer Display of Curved Surfaces. In *Proceedings of the IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structure*, pp. 11–17, May 1975. Cited on page 1, 16, 39, iii.

[Caza95] F. Cazals, G. Drettakis, and C. Puech. Filtering, Clustering and Hierarchy Construction: A New Solution for Ray-Tracing Complex Scenes. *Computer Graphics Forum*, Vol. 14, No. 3, pp. C/371–382, 1995. Cited on page xxix.

[Caza97a] F. Cazals and C. Puech. Bucket-like space partitioning data-structures with applications to ray-tracing. In *13th ACM Symposium on Computational Geometry*, p. To Appear, Nice, 1997. Cited on page xxix.

[Caza97b] F. Cazals and M. Sbert. Some integral geometry tools to estimate the complexity of 3D scenes. Tech. Rep. RR-3204, The French National Institue for Research in Computer Science and Control (INRIA), July 1997. Cited on page 102.

[Cham96] B. Chamberlain, T. DeRose, D. Lischinski, D. Salesin, and J. Snyder. Fast rendering of complex environments using a spatial hierarchy. In *Proceedings of Graphics Interface '96*, pp. 132–141, May 1996. Cited on page ix.

[Chap90] J. Chapman, T. W. Calvert, and J. Dill. Exploiting Temporal Coherence in Ray Tracing. In *Proceedings of Graphics Interface '90*, pp. 196–204, Canadian Information Processing Society, Toronto, Ontario, May 1990. Cited on page xxx.

[Chaz96] B. Chazelle *et al.* Application Challenges to Computational Geometry: CG Impact Task Force Report. Technical Report TR-521-96, Princeton Univ., Apr. 1996. Cited on page 99.

[Chen96] H. Chen and W. Wang. The Feudal Priority Algorithm on Hidden-Surface Removal. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pp. 55–64, Aug. 1996. held in New Orleans, Louisiana, 04-09 August 1996. Cited on page xxv.

[Chin89] N. Chin and S. Feiner. Near Real-Time Shadow Generation Using BSP Trees. In *Computer Graphics (Proceedings of SIGGRAPH '89)*, pp. 99–106, 1989. Cited on page 3, 30, 42, xxiv.

[Chin92] N. Chin and S. Feiner. Fast object-precision shadow generation for areal light sources using BSP trees. In *Proceddings of Symposium on Interactive 3D Graphics*, pp. 21–30, March 1992. Cited on page 3, 101, xxvi.

[Cho99] F. S. Cho and D. Forsyth. Interactive ray tracing with the visibility complex. *Computers and Graphics*, Vol. 23, No. 5, pp. 703–717, Oct. 1999. Cited on page 4.

[Choi92] H. K. Choi and C. M. Kyung. PYSHA: a shadow-testing acceleration scheme for ray tracing. *Computer-aided design*, Vol. 24, No. 2, Feb. 1992. Cited on page 4.

[Chry92] Y. Chrysanthou and M. Slater. Computing dynamic changes to BSP trees. In *Computer Graphics Forum (EUROGRAPHICS '92 Proceedings)*, pp. 321–332, Sep. 1992. Cited on page 31.

[Chry95] Y. Chrysanthou and M. Slater. Shadow Volume BSP Trees for Computation of Shadows in Dynamic Scenes. In *Proceedings of Symposium on Interactive 3D Graphics*, pp. 45–50, Apr. 1995. Cited on page xxv.

[Chry96] Y. Chrysanthou. *Shadow Computation for 3D Interaction and Animation*. PhD thesis, QMW, Dept of Computer Science, Jan. 1996. Cited on page 101, 113, xxvi.

[Chry97] Y. Chrysanthou and M. Slater. Incremental Updates to Scenes Illuminated by Area Light Sources. In *Proceedings of Eurographics Workshop on Rendering*, pp. 103–114, Springer Verlag, June 1997. Cited on page 3, 101, xxvi.

[Chry98a] Y. Chrysanthou, D. Cohen-Or, and D. Lischinski. Fast Approximate Quantitative Visibility for Complex Scenes. In *Proceedings of Computer Graphics International '98 (CGI'98)*, pp. 23–31, IEEE, NY, Hannover, Germany, June 1998. Cited on page 99, 101.

[Chry98b] 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. Cited on page 100.

[Clar76] J. H. Clark. Hierarchical Geometric Models for Visible Surface Algorithms. *Communications of the ACM*, Vol. 19, No. 10, pp. 547–554, Oct. 1976. Cited on page 39, 51, x, xxi.

[Clea88] J. G. Cleary and G. Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, Vol. 4, No. 2, pp. 65–83, July 1988. Cited on page xxx.

[Cohe02] D. Cohen-Or, Y. Chrysanthou, C. Silva, and F. Durand. A Survey of Visibility for Walkthrough Applications. *To appear in IEEE Transactions on Visualization and Computer Graphics.*, 2002. Cited on page 2.

[Cohe85] M. F. Cohen and D. P. Greenberg. The Hemi-Cube: A Radiosity Solution for Complex Environments. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, pp. 31–40, July 1985. Cited on page 3, xxviii.

[Cohe94] D. Cohen and Z. Sheffer. Proximity clouds - an acceleration technique for 3D grid traversal. *The Visual Computer*, Vol. 11, pp. 27–38, 1994. Cited on page xxx.

[Cohe95] D. Cohen-Or and A. Shaked. Visibility and Dead-Zones in Digital Terrain Maps. *Computer Graphics Forum*, Vol. 14, No. 3, pp. C/171–C/180, Sep. 1995. Cited on page 23, 84, xix.

[Cohe98a] D. Cohen-Or, G. Fibich, D. Halperin, and E. Zadicario. Conservative Visibility and Strong Occlusion for Viewspace Partitioning of Densely Occluded Scenes. In *Computer Graphics Forum (Eurographics '98 Proceedings)*, pp. 243–253, 1998. Cited on page 2, 84, 100, xviii.

[Cohe98b] D. Cohen-Or, A. Solomovic, and D. Levin. Three-dimensional distance field metamorphosis. *ACM Transactions on Graphics*, Vol. 17, No. 2, pp. 116–141, Apr. 1998. Cited on page xxi.

[Cohe98c] D. Cohen-Or and E. Zadicario. Visibility Streaming for Network-based Walkthroughs. In *Proceedings of Graphics Interface '98*, pp. 1–7, June 1998. Cited on page 2, 100.

[Cole89] R. Cole and M. Sharir. Visibility Problems for Polyhedral Terrains. *Journal of Symbolic Computation*, Vol. 7, No. 1, pp. 11–30, Jan. 1989. Cited on page xx.

[Cook84] R. L. Cook, T. Porter, and L. Carpenter. Distributed Ray Tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, pp. 137–45, July 1984. Cited on page 3, xxvi.

[Cook86] R. L. Cook. Stochastic Sampling in Computer Graphics. *ACM Transactions on Graphics*, Vol. 5, No. 1, pp. 51–72, Jan. 1986. Also in Tutorial: Computer Graphics: Image Synthesis, Computer Society Press, Washington, 1988, pp. 283–304. Cited on page 4.

[Coor96a] S. Coorg and S. Teller. A Spatially and Temporally Coherent Object Space Visibility Algorithm. Tech. Rep. TM-546, Department of Computer Graphics, MIT, Feb. 1996. Cited on page 40, 41.

[Coor96b] S. Coorg and S. Teller. Temporally Coherent Conservative Visibility. In *Proceedings of the Twelfth Annual ACM Symposium on Computational Geometry*, Philadelphia, PA, May 1996. Cited on page 40, 46, 49.

[Coor97] S. Coorg and S. Teller. Real-Time Occlusion Culling for Models with Large Occluders. In *Proceedings of the Symposium on Interactive 3D Graphics*, pp. 83–90, ACM Press, Apr. 1997. Cited on page 2, 5, 40, 42, 46, 59, 60, 130, xiv.

[Crow77] F. C. Crow. Shadow Algorithms for Computer Graphics. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, 1977. Cited on page xxiv.

[Daub97] K. Daubert, H. Schirmacher, F. X. Sillion, and G. Drettakis. Hierarchical Lighting Simulation for Outdoor Scenes. In *Proceedings of Eurographics Workshop on Rendering '97*, pp. 229–238, Springer Wein, June 1997. Cited on page 101.

[Deva97] F. Dèvai. On the Computational Requirements of Virtual Reality Systems. In *State of the Art Reports, Eurographics '97*, 1997. Cited on page ix.

[Devi89] O. Devillers. The Macro-regions: an Efficient Space Subdivision Structure for Ray Tracing. In *Computer Graphics Forum (Proceedings of Eurographics '89)*, pp. 27–38, Elsevier / North-Holland, Sep. 1989. Cited on page xxx.

[Dobk97] D. Dobkin and S. Teller. Computer graphics. In J. E. Goodman and J. O'Rourke, Eds., *Handbook of Discrete and Computational Geometry*, Chap. 42, pp. 779–796, CRC Press LLC, 1997. Cited on page 36.

[Down01] L. Downs, T. Möller, and C. H. Séquin. Occlusion Horizons for Driving through Urban Scenes. In *Symposium on Interactive 3D Graphics*, pp. 121–124, ACM SIGGRAPH, 2001. Cited on page 23, xvi.

[Dret94a] G. Drettakis. *Structured Sampling and Reconstruction of Illumination for Image Synthesis*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Jan. 1994. Cited on page 101, xxix.

[Dret94b] G. Drettakis and E. Fiume. A Fast Shadow Algorithm for Area Light Sources Using Backprojection. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pp. 223–230, 1994. Cited on page 3, 99, 101, xxix.

[Dret97] G. Drettakis and F. Sillion. Interactive Update of Global Illumination Using A Line-Space Hierarchy. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pp. 57–64, Aug. 1997. Cited on page 101.

[Dugu02] F. Duguet and G. Drettakis. Robust Epsilon Visibility. In *Computer Graphics (SIGGRAPH '02 Proceedings)*, pp. 567–575, ACM Press/ACM SIGGRAPH, 2002. Cited on page 99, 101, xxxii.

[Dura00] F. Durand, G. Drettakis, J. Thollot, and C. Puech. Conservative Visibility Preprocessing Using Extended Projections. In *Computer Graphics (SIGGRAPH '00 Proceedings)*, pp. 239–248, 2000. Cited on page 2, 19, 69, 83, 84, 94, 95, 100, 133, xviii.

[Dura96] F. Durand, G. Drettakis, and C. Puech. The 3D Visibility Complex: A New Approach to the Problems of Accurate Visibility. In *Proceedings of Eurographics Workshop on Rendering*, pp. 245–256, Eurographics, Springer Wein, June 1996. Cited on page 4, 99, 101, xxxi.

[Dura97] F. Durand, G. Drettakis, and C. Puech. The Visibility Skeleton: A Powerful and Efficient Multi-Purpose Global Visibility Tool. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pp. 89–100, 1997. Cited on page 4, 99, 101, xxxii.

[Dura99] F. Durand. *3D Visibility: Analytical Study and Applications*. PhD thesis, Universite Joseph Fourier, Grenoble, France, July 1999. Cited on page 4, 9, 13, 114.

[Egge92] D. W. Eggert, K. W. Bowyer, C. R. Dyer, H. I. Christensen, and D. B. Goldgof. The Scale Space Aspect Graph. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition '92*, pp. 335–40, IEEE Computer Society, 1992. Cited on page 100.

[Evan96a] F. Evans, S. Skiena, and A. Varshney. Completing Sequential Triangulations is Hard. Tech. Rep., Dept. of Computer Science, State University of New York at Stony Brook, 1996. Cited on page xx.

[Evan96b] F. Evans, S. S. Skiena, and A. Varshney. Optimizing Triangle Strips for Fast Rendering. In *Proceedings of IEEE Visualization '96*, pp. 319–326, 1996. Cited on page xx.

[Flor94] L. D. Floriani and P. Magillo. Visibility Algorithms on Triangulated Digital Terrain Models. In *International Journal of Geographical Information Systems*, pp. 13–41, Taylor & Francis, 1994. Cited on page xix.

[Flor95] L. D. Floriani and P. Magillo. Horizon computation on a hierarchical triangulated terrain model. *The Visual Computer*, Vol. 11, No. 3, pp. 134–149, 1995. Cited on page 23, xix.

[Fole90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Publishing Co., 2nd Ed., 1990. Cited on page 20, 60, iv, viii, xxi.

[Four93] A. Fournier and P. Poulin. A Ray Tracing Accelerator Based on a Hierarchy of 1D Sorted Lists. In *Proceedings of Graphics Interface '93*, pp. 53–61, Canadian Information Processing Society, Toronto, Ontario, May 1993. Cited on page xxix.

[Fran90] W. R. Franklin and M. S. Kankanhalli. Parallel Object-Space Hidden Surface Removal. *Computer Graphics (SIGGRAPH '90 Proceedings)*, Vol. 24, No. 4, pp. 87–94, Aug. 1990. Cited on page 39.

[Fuch80] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On Visible Surface Generation by a Priori Tree Structures. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, pp. 124–133, July 1980. Cited on page 29, 31, 39, 49, iv, xxv.

[Fuji86] A. Fujimoto, T. Tanaka, and K. Iwata. ARTS: Accelerated Ray Tracing System. *IEEE Computer Graphics and Applications*, Vol. 6, No. 4, pp. 16–26, 1986. Cited on page xxix, xxx.

[Fuku02] K. Fukuda. CDD polyhedra enumeration library. 2002. Availabale at `http://www.ifor.math.ethz.ch/~fukuda/cdd_home/cdd.html`. Cited on page 90, 113, 119.

[Fuku96] K. Fukuda and A. Prodon. Double Description Method Revisited. *Lecture Notes in Computer Science*, Vol. 1120, pp. 91–111, 1996. Cited on page 113.

[Funk93] T. A. Funkhouser. *Database and Display Algorithms for Interactive Visualization of Architectural Models*. PhD thesis, CS Division, UC Berkeley, 1993. Cited on page xxi.

[Funk98] T. Funkhouser, I. Carlbom, G. Elko, G. Pingali, M. Sondhi, and J. West. A Beam Tracing Approach to Acoustic Modeling for Interactive Virtual Environments. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pp. 21–32, Addison Wesley, July 1998. Cited on page 102, xxix.

[Fuss88] D. Fussell and K. R. Subramanian. Fast Ray Tracing Using K-D Trees. Technical Report TR-88-07, University of Texas, Austin, Dept. Of Computer Science, March 1988. Cited on page xxix.

[Garr96] W. F. Garrett, H. Fuchs, M. C. Whitton, and A. State. Real-Time Incremental Visualization of Dynamic Ultrasound Volumes Using Parallel BSP Trees. In *Proeedings of IEEE Visualization '96*, Oct. 1996. ISBN 0-89791-864-9. Cited on page 31.

[Geor90] D. W. George, F. X. Sillion, and D. P. Greenberg. Radiosity Redistribution for Dynamic Environments. *IEEE Computer Graphics and Applications*, Vol. 10, No. 4, pp. 26–34, July 1990. Cited on page xxviii.

[Geor95] C. Georges. Obscuration Culling on Parallel Graphics Architectures. Tech. Rep. TR95-017, Department of Computer Science, University of North Carolina, Chapel Hill, 1995. Cited on page 39.

[Ghal96] S. Ghali and A. J. Stewart. Incremental Update of the Visibility Map as Seen by a Moving Viewpoint in Two Dimensions. In *Computer Animation and Simulation '96*, pp. 3–13, Springer-Verlag, 1996. Cited on page 70.

[Gigu90] Z. Gigus and J. Malik. Computing the aspect graph for line drawings of polyhedral objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, No. 2, pp. 113–122, Feb. 1990. Cited on page 100, 106.

[Glas84] A. S. Glassner. Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics and Applications*, Vol. 4, No. 10, pp. 15–22, Oct. 1984. Cited on page viii.

[Glas88] A. S. Glassner. Spacetime ray tracing for animation. *IEEE Computer Graphics and Applications*, Vol. 8, No. 2, pp. 60–70, March 1988. Cited on page xxx.

[Glas95] A. S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, San Francisco, CA, 1995. Cited on page 2.

[Gold87] J. Goldsmith and J. Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, Vol. 7, No. 5, pp. 14–20, May 1987. Cited on page xxix.

[Good97] J. E. Goodman and J. O'Rourke, Eds. *Handbook of Discrete and Computational Geometry*. CRC Press, 1997. Cited on page 70, 90, 107.

[Gora84] C. M. Goral, K. K. Torrance, D. P. Greenberg, and B. Battaile. Modelling the Interaction of Light Between Diffuse Surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, pp. 213–222, July 1984. Cited on page xxvii.

[Gord91] D. Gordon and S. Chen. Front-to-back display of BSP trees. *IEEE Computer Graphics and Applications*, Vol. 11, No. 5, pp. 79–85, Sep. 1991. Cited on page 31, v.

[Gort96] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The Lumigraph. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pp. 43–54, Addison Wesley, Aug. 1996. Cited on page 102.

[Grab01] M. Grabner. Advanced Techniques for Interactive Visualization of Multi-resolution Meshes. In *The Journal of Visualization and Computer Animation*, pp. 241–252, John Wiley & Sons, Ltd., 2001. Cited on page xxi.

[Gran85] C. W. Grant. Integrated Analytic Spatial and Temporal Anti-Aliasing for Polyhedra in 4-Space. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, pp. 79–84, July 1985. Cited on page 102.

[Gran92] C. W. Grant. *Visibility Algorithms in Image Synthesis*. PhD thesis, University of California, Davis, 1992. Cited on page 39, 60, xxiv.

[Gras99] J. Grasset, O. Terraz, J.-M. Hasenfratz, and D. Plemenos. Accurate Scene Display by Using Visibility Maps. In *Spring Conference on Computer Graphics (SCCG '99)*, 1999. Cited on page 59, 60.

[Gree93] N. Greene, M. Kass, and G. Miller. Hierarchical Z-Buffer Visibility. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pp. 231–238, 1993. Cited on page 39, 46, xi, xxii.

[Gree94a] N. Greene and M. Kass. Error-Bounded Antialiased Rendering of Complex Environments. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pp. 59–66, July 1994. Cited on page 40.

[Gree94b] N. Greene. Detecting Intersection of a Rectangular Solid and a Convex Polyhedron. In P. Heckbert, Ed., *Graphics Gems IV*, pp. 74–82, Academic Press, Boston, MA, 1994. Cited on page 45.

[Gree96] N. Greene. Hierarchical Polygon Tiling with Coverage Masks. In H. Rushmeier, Ed., *Computer Graphics (SIGGRAPH '96 Proceedings)*, pp. 65–74, Addison Wesley, Aug. 1996. held in New Orleans, Louisiana, 04-09 August 1996. Cited on page 40, 46, xii.

[Grol93] E. Gröller. Oct-tracing animation sequences. In *Spring Conference on Computer Graphics (SCCG '93)*, pp. 96–101, June 1993. Cited on page xxx.

[Gros98] J. P. Grossman and W. J. Dally. Point Sample Rendering. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '98)*, pp. 181–192, Springer-Verlag Wien New York, 1998. Cited on page xxii.

[Gu97] X. Gu, S. J. Gortier, and M. F. Cohen. Polyhedral Geometry and the Two-Plane Parameterization. In *Proceedings of Eurographics Workshop on Rendering '97*, pp. 1–12, Springer Wein, June 1997. Cited on page 9.

[Hain86] E. A. Haines and D. P. Greenberg. The Light Buffer: A Ray Tracer Shadow Testing Accelerator. *IEEE Computer Graphics and Applications*, Vol. 6, No. 9, pp. 6–16, Sep. 1986. Cited on page 4, xxiii, xxix.

[Hain94] E. A. Haines and J. R. Wallace. Shaft Culling for Efficient Ray-Traced Radiosity. In *Photorealistic Rendering in Computer Graphics (Proceedings of Eurographics Workshop on Rendering '94)*, Springer-Verlag, 1994. Cited on page 101, 115.

[Havr00a] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000. Cited on page 4, 36, 56, xxix, xxxi.

[Havr00b] V. Havran and J. Bittner. LCTS: Ray Shooting using Longest Common Traversal Sequences. In *Computer Graphics Forum (Eurographics '00 Proceedings)*, pp. 59–70, Interlaken, Switzerland, 2000. Cited on page viii, xxix.

[Havr02] V. Havran and J. Bittner. On Improving Kd-Trees for Ray Shooting. In *Journal of WSCG (Proceedings of Winter School on Computer Graphics '02)*, Plzen, Czech Republic, Feb. 2002. Cited on page 41.

[Havr98a] V. Havran, J. Bittner, and J. Žára. Ray Tracing with Rope Trees. In *Proceedings of 13th Spring Conference on Computer Graphics*, pp. 130–139, Budmerice, 1998. Cited on page 50, xxxi.

[Havr98b] V. Havran, T. Kopal, J. Bittner, and J. Žára. Fast Robust BSP Tree Traversal Algorithm for Ray Tracing. *Journal of Graphics Tools*, Vol. 2, No. 4, pp. 15–24, Dec. 1998. Cited on page xxxi.

[Heck84] P. S. Heckbert and P. Hanrahan. Beam Tracing Polygonal Objects. *Computer Graphics (SIGGRAPH'84 Proceedings)*, Vol. 18, No. 3, pp. 119–127, July 1984. Cited on page 27, xxix.

[Heck92] P. S. Heckbert. Discontinuity Meshing for Radiosity. In *Third Eurographics Workshop on Rendering*, pp. 203–216, Bristol, UK, May 1992. Cited on page 3, 59, 101, xxviii, xxix.

[Heck97] P. S. Heckbert and M. Herf. Simulating Soft Shadows with Graphics Hardware. Tech. Rep., CS Dept., Carnegie Mellon U., Jan. 1997. CMU-CS-97-104. Cited on page 3, xxvi.

[Heid00] W. Heidrich, S. Brabec, and H. Seidel. Soft Shadow Maps for Linear Lights. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '00)*, pp. 269–280, Springer-Verlag Wien New York, 2000. Cited on page xxvi.

[Heid91] T. Heidmann. Real Shadows, Real Time. *Iris Universe*, Vol. 18, pp. 28–31, 1991. Silicon Graphics, Inc. Cited on page xxiv.

[Heid99] W. Heidrich, R. Westermann, H.-P. Seidel, and T. Ertl. Applications of Pixel Textures in Visualization and Realistic Image Synthesis. In *ACM Symposium on Interactive 3D Graphics*, ACM/Siggraph, 1999. Cited on page xxiv.

[Hey01] H. Hey, R. F. Tobler, and W. Purgathofer. Real-Time Occlusion Culling with a Lazy Occlusion Grid. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '00)*, pp. 217–222, 2001. Cited on page xix.

[Hink96] A. Hinkenjann and H. Müller. Hierarchical Blocker Trees for Global Visibility Calculation. Research Report 621/1996, University of Dortmund, Aug. 1996. Cited on page 69, 102.

[Hopp96] H. Hoppe. Progressive Meshes. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pp. 99–108, Addison Wesley, Aug. 1996. held in New Orleans, Louisiana, 04-09 August 1996. Cited on page xxi.

[Hopp97] H. Hoppe. View-Dependent Refinement of Progressive Meshes. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pp. 189–198, Addison Wesley, Aug. 1997. Cited on page xxi.

[Hopp98] H. Hoppe. Smooth View-Dependent Level-Of-Detail Control and its Application to Terrain Rendering. In *Proceedings IEEE Visualization'98*, pp. 35–42, IEEE, 1998. Cited on page xxi.

[Huds97] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, and H. Zhang. Accelerated Occlusion Culling Using Shadow Frusta. In *Proceedings of ACM Symposium on Computational Geometry*, pp. 1–10, 1997. Cited on page 2, 5, 40, 42, 46, 59, 60, 130, xiv.

[Huer97] J. Huerta, M. Chover, J. Ribelles, and R. Quiros. Constructing and Rendering of Multiresolution Binary Space Partitioning Trees. In H. P. Santo, Ed., *Edugraphics + Compugraphics Proceedings*, pp. 212–221, GRASP- Graphic Science Promotions & Publications, P.O. Box 4076, Massama, 2745 Queluz, Portugal, Dec. 15, 1997. Cited on page 31.

[Ione98] A. Iones, S. Zhukov, and A. Krupkin. On Optimality of OBBs for Visibility Tests for Frustum Culling, Ray Shooting and Collision Detection. In *Proceedings of Computer Graphics International 1998 (CGI '98)*, pp. 256–263, IEEE Computer Society, June 22–26 1998. Cited on page x.

[Jame98] A. James and A. Day. The Priority Face Determination Tree for Hidden Surface Removal. In *Computer Graphics Forum*, pp. 55–71, march 1998. Cited on page xxv.

[Jans86] F. W. Jansen. Data Structures for Ray Tracing. In L. R. A. Kessener, F. J. Peters, and M. L. P. van Lierop, Eds., *Data Structures for Raster Graphics*, pp. 57–73, Springer-Verlag, New York, 1986. Eurographic seminar. Cited on page xxxi.

[Jens95] H. W. Jensen and N. J. Christensen. Efficiently Rendering Shadows Using the Photon Map. In H. P. Santo, Ed., *Edugraphics + Compugraphics Proceedings*, pp. 285–291, GRASP-Graphic Science Promotions & Publications, P.O. Box 4076, Massama, 2745 Queluz, Portugal, Dec. 12, 1995. Cited on page xxix.

[Jesc02] S. Jeschke, M. Wimmer, and H. Schuman. Layered Environment-Map Impostors for Arbitrary Scenes. In *Proceedings of Graphics Interface '02*, pp. 1–8, May 2002. Calgary, Alberta. Cited on page xxii.

[Jeva92] D. Jevans. Object Space Temporal Coherence for Ray Tracing. In *Proceedings of Graphics Interface '92*, pp. 176–183, Canadian Information Processing Society, Toronto, Ontario, May 1992. Cited on page xxx.

[Jone71] C. B. Jones. A New Approach to the ‘Hidden Line’ Problem. *Computer Journal*, Vol. 14, No. 3, pp. 232–237, Aug. 1971. Cited on page xv.

[Kaji86] J. T. Kajiya. The Rendering Equation. In *Computer Graphics (SIGGRAPH ’86 Proceedings)*, pp. 143–150, Aug. 1986. Cited on page 4, xxvi.

[Kapl85] M. Kaplan. Space-Tracing: A Constant Time Ray-Tracer. In *SIGGRAPH ’85 State of the Art in Image Synthesis seminar notes*, pp. 149–158, Addison Wesley, July 1985. Cited on page 41, xxx, xxxi.

[Kell97] A. Keller. Instant Radiosity. In *Computer Graphics (SIGGRAPH ’97 Proceedings)*, pp. 49–56, Addison Wesley, Aug. 1997. ISBN 0-89791-896-7. Cited on page xxix.

[Klim97] K. S. Klimaszewski and T. W. Sederberg. Faster Ray Tracing Using Adaptive Grids. *IEEE Computer Graphics and Applications*, Vol. 17, No. 1, pp. 42–51, Jan./Feb. 1997. Cited on page xxix.

[Klos01] J. T. Klosowski and C. T. Silva. Efficient Conservative Visibility Culling Using the Prioritized-Layered Projection Algorithm. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 7, No. 4, pp. 365–379, Oct. 2001. Cited on page 2, xix.

[Kok92] A. J. F. Kok and F. W. Jansen. Adaptive Sampling of Area Light Sources in Ray Tracing Including Diffuse Interreflection. *Computer Graphics Forum (Proceedings of Eurographics ’92)*, Vol. 11, No. 3, pp. 289–298, Sep. 1992. Cited on page xxix.

[Kok93] A. J. F. Kok, F. W. Jansen, and C. Woodward. Efficient, Complete Radiosity Ray Tracing Using a Shadow-Coherence Method. *The Visual Computer*, Vol. 10, pp. 19–33, oct 1993. Cited on page xxix.

[Koli97] I. Kolingerová. Convex Polyhedron-Line Intersection Detection using Dual Representation. *The Visual Computer*, Vol. 13, No. 1, pp. 42–49, 1997. Cited on page xxx.

[Kolt00] V. Koltun, Y. Chrysanthou, and D. Cohen-Or. Virtual Occluders: An Efficient Intermediate PVS Representation. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering ’00)*, pp. 59–70, 2000. Cited on page 5.

[Kolt01] V. Koltun, Y. Chrysanthou, and D. Cohen-Or. Hardware-Accelerated From-Region Visibility Using a Dual Ray Space. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering ’01)*, pp. 205–216, 2001. Cited on page 2, 5, 23, 84, 94, 96, 100, 133, xix.

[Kuma96a] S. Kumar and D. Manocha. Hierarcical Visibility Culling for Spline Models. In *Proceedings of Graphics Interface ’96*, pp. 142–150, Canadian Human-Computer Communications Society, May 1996. Cited on page 39, ix.

[Kuma96b] S. Kumar, D. Manocha, W. Garrett, and M. Lin. Hierarchical Back-Face Computation. In *Rendering Techniques (Proccedings of Eurographics Workshop on Rendering ’96)*, pp. 235–244, Springer Wein, June 1996. Cited on page 39, ix.

[Kwon98] B. Kwon, D. S. Kim, K.-Y. Chwa, and S. Y. Shin. Memory-Efficient Ray Classification for Visibility Operations. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 4, No. 3, pp. 193–201, jul–sep 1998. Cited on page xxix.

[Levo96] M. Levoy and P. Hanrahan. Light Field Rendering. In *Computer Graphics (SIGGRAPH ’96 Proceedings)*, pp. 31–42, Addison Wesley, Aug. 1996. held in New Orleans, Louisiana, 04-09 August 1996. Cited on page 102.

[Lisc92] D. Lischinski, F. Tampieri, and D. P. Greenberg. Discontinuity meshing for accurate radiosity. *IEEE Computer Graphics and Applications*, Vol. 12, No. 6, pp. 25–39, Nov. 1992. Cited on page 3, 101, xxviii, xxix.

[Loof93] N. Loofbourrow and S. A. Shafer. Optimizing ray tracing with visual coherence. Tech. Rep. CMU-CS-93-209, Carnegie-Mellon University, Department of Computer Science, 1993. Cited on page xxix.

[Losc97] C. Loscos and G. Drettakis. Interactive High-Quality Soft Shadows in Scenes with Moving Objects. *Computer Graphics Forum*, Vol. 16, No. 3, pp. C219–C230, Sep. 4–8 1997. Cited on page 101.

[Lueb95] D. Luebke and C. Georges. Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets. In *Proceedings of Symposium on Interactive 3D Graphics '95*, pp. 105–106, ACM SIGGRAPH, Apr. 1995. Cited on page 2, 24, 39, xv, xx.

[Luka98] A. Lukaszewski and A. Formella. Fast Penumbra Calculation in Ray Tracing. In *Proceedings of Winter School of Computer Graphics (WSCG'98)*, pp. 238–245, Feb. 1998. Cited on page xxix.

[MacD90] J. D. MacDonald and K. S. Booth. Heuristics for Ray Tracing Using Space Subdivision. *Visual Computer*, Vol. 6, No. 6, pp. 153–65, 1990. Cited on page 36, 41, xxix, xxx, xxxi.

[Mars97] D. Marshall, D. S. Fussell, and A. Campbell III. Multiresolution rendering of complex botanical scenes. In *Proceedings of Graphics Interface '97*, pp. 97–104, May 1997. Cited on page 31.

[Max88] N. L. Max. Horizon mapping: shadows for bump-mapped surfaces. *The Visual Computer*, Vol. 4, No. 2, pp. 109–117, July 1988. Cited on page xix.

[McMi97] L. McMillan. An Image-Based Approach to Three-Dimensional Computer Graphics. Ph.D. Thesis TR97-013, University of North Carolina, Chapel Hill, May 1997. Cited on page xxii.

[Mill96] T. Miller. Hidden-Surfaces: Combining BSP Trees with Graph-Based Algorithms. Tech. Rep. CS-96-15, Department of Computer Graphics, Brown University, Apr. 1996. Cited on page xxv.

[Moll02] T. Möller and E. Haines. *Real-Time Rendering, 2nd edition*. A. K. Peters, 2002. Cited on page 1, 2, 22, 46, ix, xxiii.

[More95] P. Morer, A. M. Garcia-Alonso, and J. Flaquer. Optimization of a Priority List Algorithm for 3-D Rendering of Buildings. *Computer Graphics Forum*, Vol. 14, No. 4, pp. 217–227, Oct. 1995. Cited on page 31.

[Mulm89] K. Mulmuley. An Efficient Algorithm for Hidden Surface Removal. *Computer Graphics (SIGGRAPH '89 Proceedings)*, Vol. 23, No. 3, pp. 379–388, July 1989. Cited on page 39.

[Mura97] T. M. Murali and T. A. Funkhouser. Consistent Solid and Boundary Representations from Arbitrary Polygonal Data. In *Proceedings of Symposium on Interactive 3D Graphics '97*, pp. 155–162, ACM SIGGRAPH, Apr. 1997. Cited on page 31.

[Nagy94] G. Nagy. Terrain visibility. *Computers and Graphics*, Vol. 18, No. 6, pp. 763–773, 1994. Cited on page xix.

[Nayl90a] B. Naylor. Binary Space Partitioning Trees as an Alternative Representation of Polytopes. *Computer–Aided Design*, pp. 250–252, 1990. Cited on page 31.

[Nayl90b] B. Naylor, J. Amanatides, and W. Thibault. Merging BSP Trees Yields Polyhedral Set Operations. *Computer Graphics (SIGGRAPH ’90 Proceedings)*, Vol. 24, No. 4, pp. 115–124, Aug. 1990. Cited on page 30, 31, 99, xxiv.

[Nayl92a] B. F. Naylor. Interactive solid geometry via partitioning trees. In *Proceedings of Graphics Interface ’92*, pp. 11–18, May 1992. Cited on page 31.

[Nayl92b] B. F. Naylor. Partitioning tree image representation and generation from 3D geometric models. In *Proceedings of Graphics Interface ’92*, pp. 201–212, May 1992. Cited on page 27, 28, 30, 31, v.

[Nayl93] B. Naylor. Constructing good partition trees. In *Proceedings of Graphics Interface ’93*, pp. 181–191, Toronto, Ontario, Canada, May 1993. Cited on page 31.

[Nech96] K. Nechvíle and J. Sochor. Form-factor Evaluation with Regional BSP Trees. In *Proceedings of Winter School of Computer Graphics (WSCG ’96)*, pp. 285–293, Feb. 1996. held at University of West Bohemia, Plzen, Czech Republic, 12-16 February 1996. Cited on page 31, 101.

[Nech99] K. Nechvíle and P. Tobola. Local Approach to Dynamic Visibility in a Plane. In *Proceedings of Winter School of Computer Graphics (WSCG’99)*, pp. 202–208, 1999. Cited on page 70.

[Neid93] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley, Reading MA, 1993. Cited on page xx.

[Newe72] M. E. Newell, R. G. Newell, and T. L. Sancha. A Solution to the Hidden Surface Problem. In *Proceedings of the ACM Annual Conference*, pp. 443–450, Boston, Massachusetts, Aug. 1972. Cited on page iv.

[Nire02] S. Nirenstein, E. Blake, and J. Gain. Exact From-Region Visibility Culling. In *Proceedings of Eurographics Workshop on Rendering ’02*, pp. 199–210, 2002. Cited on page 99, 101, 115.

[Nish85] T. Nishita and E. Nakamae. Continuous Tone Representation of 3-D Objects Taking Account of Shadows and Interreflection. *Computer Graphics (SIGGRAPH ’85 Proceedings)*, Vol. 19, No. 3, pp. 23–30, July 1985. Cited on page 101.

[nVID02] nVIDIA Co. Graphics hardware specifications. 2002. `http://www.nvidia.com`. Cited on page ix.

[Orti96] R. Orti, S. Riviere, F. Durand, and C. Puech. Using the Visibility Complex for Radiosity Computation. In *Lecture Notes in Computer Science (Applied Computational Geometry: Towards Geometric Engineering)*, pp. 177–190, Springer-Verlag, Berlin, Germany, May 1996. Cited on page 69, 101, xxxi.

[Pell97] M. Pellegrini. Ray shooting and lines in space. In J. E. Goodman and J. O’Rourke, Eds., *Handbook of Discrete and Computational Geometry*, Chap. 32, pp. 599–614, CRC Press, 1997. Cited on page 9, 99, 102, 107, 114.

[Peng87] Q. Peng, Y. Zhu, and Y. Liang. A Fast Ray Tracing Algorithm Using Space Indexing Techniques. In *Proceedings of Eurographics ’87*, pp. 11–23, North-Holland, Aug. 1987. Cited on page xxix.

[Pfis00] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: Surface Elements as Rendering Primitives. In *Computer Graphics (SIGGRAPH '00 Proceedings)*, pp. 335–342, 2000. Cited on page xxii.

[Plan90] H. Plantinga, C. R. Dyer, and W. B. Seales. Real-Time Hidden-Line Elimination for a Rotating Polyhedral Scene Using the Aspect Representation. In *Proceedings of Graphics Interface '90*, pp. 9–16, May 1990. Cited on page 100.

[Plan93] H. Plantinga. Conservative visibility preprocessing for efficient walkthroughs of 3D scenes. In *Proceedings of Graphics Interface '93*, pp. 166–173, Toronto, Ontario, Canada, May 1993. Cited on page 100.

[Pocc93] M. Pocchiola and G. Vegter. The visibility complex. In *Proceedings of ACM Symposium on Computational Geometry*, pp. 328–337, 1993. Cited on page 4, 69, 71, 101, xxxi.

[Pope98] V. Popescu, A. Lastra, D. Aliaga, and M. de Oliveira Neto. Efficient warping for architectural walkthroughs using layered depth images. In *IEEE Visualization '98 (VIS '98)*, pp. 211–216, IEEE, Washington - Brussels - Tokyo, Oct. 1998. Cited on page xxii.

[Pu98] F.-T. Pu. *Data Structures for Global Illumination and Visibility Queries in 3-Space*. PhD thesis, University of Maryland, College Park, MD, 1998. Cited on page 10, 101, 102, 104, 107, 109, 111.

[Pupp97] E. Puppo and R. Scopigno. Simplification, LOD and Multiresolution, Principles and Applications. In *Eurographics '97 Tutorial*, 1997. Cited on page xxi.

[Purc02] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray Tracing on Programmable Graphics Hardware. In *Computer Graphics (SIGGRAPH '02 Proceedings)*, pp. 703–712, 2002. Cited on page xxx.

[Rajk96] A. Rajkumar, B. Naylor, F. Feisullin, and L. Rogers. Predicting RF coverage in large environments using ray–beam tracing and partitioning tree represented geometry. *Wireless Netwoks*, Vol. 2, pp. 143–154, 1996. Cited on page xxix.

[Rivi95] S. Rivière. Topologically Sweeping the Visibility Complex of Polygonal Scenes. In *Proceedings of ACM Symposium Computational Geometry*, pp. C36–C37, 1995. Cited on page 69, 101, xxxi.

[Rivi97a] S. Rivière. Dynamic visibility in polygonal scenes with the visibility complex. In *Proceedings of ACM Symposium on Computational Geometry '97*, pp. 421–423, ACM Press, New York, June 4–6 1997. Cited on page 69, 101, xxxi.

[Rivi97b] S. Rivière. Walking in the Visibility Complex with Applications to Visibility Polygons and Dynamic Visibility. In *Proceedings of 9th Canadian Conference on Computational Geometry*, pp. 147–152, 1997. Cited on page 69, 101, xxxi.

[Rohl94] J. Rohlf and J. Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real–Time 3D Graphics. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pp. 381–395, July 1994. Cited on page 46, 51, ix.

[Rusi00] S. Rusinkiewicz and M. Levoy. QSplat: A Multiresolution Point Rendering System for Large Meshes. In *Computer Graphics (SIGGRAPH '00 Proceedings)*, pp. 343–352, 2000. Cited on page xxii.

[Sada00] A. Sadagic and M. Slater. Dynamic Polygon Visibility Ordering for Head-Slaved Viewing in Virtual Environments. In *Computer Graphics Forum*, pp. 111–122, Eurographics Association, 2000. Cited on page 100.

[Same89a] H. Samet. *Design and analysis of Spatial Data Structures: Quadtrees, Octrees, and other Hierarchical Methods*. Addison–Wesley, Redding, MA, 1989. Cited on page xxx.

[Same89b] H. Samet. Implementing Ray Tracing with Octrees and Neighbor Finding. *Computers and Graphics*, Vol. 13, No. 4, pp. 445–60, 1989. Cited on page xxix.

[Same90] H. Samet. *Applications of Spatial Data Structures*. Addison–Wesley, Reading, MA, 1990. Cited on page 35.

[Scha00] G. Schaufler, J. Dorsey, X. Decoret, and F. X. Sillion. Conservative Volumetric Visibility with Occluder Fusion. In *Computer Graphics (SIGGRAPH '00 Proceedings)*, pp. 229–238, 2000. Cited on page 2, 19, 69, 83, 84, 95, 100, 133, xviii.

[Scha96] G. Schaufler and W. Sturzlinger. A Three-Dimensional Image Cache for Virtual Reality. *Computer Graphics Forum (Proceedings of Eurographics '96)*, Vol. 15, No. 3, pp. C227–C235, C471–C472, Sep. 1996. Cited on page xxii.

[Scha97] G. Schaufler. Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '97)*, pp. 151–162, Springer Wein, June 1997. Cited on page xxii.

[Schm97] D. Schmalstieg. A Survey of Advanced Interactive 3-D Graphics Techniques. Tech. Rep. TR-186-2-97-05, Institute of Computer Graphics, Vienna University of Technology, 1997. Cited on page ix.

[Schr93] P. Schröder and P. Hanrahan. On the Form Factor Between Two Polygons. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pp. 163–164, 1993. Cited on page 101, xxix.

[Schu69] R. A. Schumacker, R. Brand, M. Gilliland, and W. Sharp. Study for Applying Computer-Generated Images to Visual Simulation. Tech. Rep. AFHRL–TR–69–14, U.S. Air Force Human Resources Laboratory, 1969. Cited on page iv, v.

[Sega92] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast Shadows and Lighting Effects using Texture Mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, Vol. 26, No. 2, pp. 249–252, July 1992. Cited on page xxiv.

[Semw97] S. K. Semwal and H. Kvarnstrom. Directed Safe Zones and the Dual Extend Algorithms for Efficient Grid Tracing during Ray Tracing. In *Proceedings of Graphics Interface '97*, pp. 76–87, May 1997. Cited on page xxx.

[Shad96] J. Shade, D. Lischinski, D. Salesin, T. DeRose, and J. Snyder. Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pp. 75–82, Addison Wesley, Aug. 1996. Cited on page xxii.

[Shad98] J. W. Shade, S. J. Gortler, L. He, and R. Szeliski. Layered Depth Images. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pp. 231–242, Addison Wesley, July 1998. Cited on page xxii.

[Shar92] M. Sharir and M. H. Overmars. A Simple Output-Sensitive Algorithm for Hidden Surface Removal. *ACM Transactions on Graphics*, Vol. 11, No. 1, pp. 1–11, Jan. 1992. Cited on page 39.

[Shim93] I. Shimshoni and J. Ponce. Finite Resolution Aspect Graphs of Polyhedral Objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 19, No. 4, pp. 315–327, 1993. Cited on page 100.

[Shir95] P. Shirley, B. Wade, P. M. Hubbard, D. Zareski, B. Walter, and D. P. Greenberg. Global Illumination via Density Estimation. In *Rendering Techniques '95 (Proceedings of Eurographics Workshop on Rendering '95)*, pp. 219–230, Springer-Verlag, 1995. Cited on page xxviii.

[Sill89] F. Sillion and C. Puech. A General Two-Pass Method Integrating Specular and Diffuse Reflection. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, pp. 335–344, July 1989. Cited on page xxix.

[Silv97] C. T. Silva and J. S. B. Mitchell. The Lazy Sweep Ray Casting Algorithm for Rendering Irregular Grids. *IEEE Transaction on Visualization and Computer Graphics*, Vol. 3, No. 2, pp. 142–157, Apr. 1997. Cited on page xxix.

[Simi94] G. Simiakakis and A. M. Day. Five-dimensional Adaptive Subdivision for Ray Tracing. *Computer Graphics Forum*, Vol. 13, No. 2, pp. 133–140, June 1994. Cited on page 4, 99, 100, xxix.

[Slat92] M. Slater. A Comparison of Three Shadow Volume Algorithms. *The Visual Computer*, Vol. 9, No. 1, pp. 25–38, 1992. Cited on page xxiii.

[Slat97] M. Slater and Y. Chrysanthou. View Volume Culling Using a Probabilistic Caching Scheme. In *Proceedings of ACM Symposium on Virtual Reality Software and Technology (VRST '97)*, pp. 71–78, Lausanne, Switzerland, Sep. 1997. Cited on page 46, x.

[Slus97] P. Slusallek. Photo-Realistic Rendering – Recent Trends and Developments. In *EUROGRAPHICS '97 State-of-the-Art-Report*, Eurographics Association, 1997. Cited on page xxviii, xxix.

[Snyd98] J. Snyder and J. Lengyel. Visibility Sorting and Compositing without Splitting for Image Layer Decomposition. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pp. 219–230, Addison Wesley, July 1998. Cited on page xxv.

[Sojk95] E. Sojka. Aspect Graphs of Three Dimensional Scenes. In *Proceedings of Winter School of Computer Graphics (WSCG '95)*, Feb. 1995. Cited on page 100.

[Sole96] C. Soler and F. Sillion. Accurate Error Bounds for Multi-Resolution Visibility. In *rendering Techniques (Proceedings of Eurographics Workshop on Rendering '96)*, pp. 133–142, Springer Wein, June 1996. Cited on page 101.

[Sole98] C. Soler and F. Sillion. Fast Calculation of Soft Shadow Textures Using Convolution. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pp. 321–332, July 1998. Cited on page 101, xxvi.

[Stam97] M. Stamminger, W. Nitsch, and P. Slusallek. Isotropic Clustering for Hierarchical Radiosity – Implementation and Experiences. In *Proceedings of Winter School of Computer Graphics (WSCG '97)*, 1997. Cited on page xxviii.

[Stam98] M. Stamminger, P. Slusallek, and H. Seidel. Bounded Clustering: Finding Good Bounds on Clustered Light Transport. In *Proceedings of Pacific Graphics '98*, pp. 87–96, IEEE, Oct. 1998. Cited on page xxviii.

[Stew01] A. J. Stewart. Tunneling for Triangle Strips in Continuous Level-of-Detail Meshes. In *Proceedings of Graphics Interface '01*, pp. 91–100, 2001. Cited on page xx.

[Stew93] A. J. Stewart and S. Ghali. An Output Sensitive Algorithm for the Computation of Shadow Boundaries. In *Proceedings of Canadian Conference on Computational Geometry*, pp. 291–296, Aug. 1993. Cited on page 101, xxix.

[Stew94] A. J. Stewart and S. Ghali. Fast Computation of Shadow Boundaries Using Spatial Coherence and Backprojections. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pp. 231–238, 1994. Cited on page 3, 99, 101, xxix.

[Stew96] A. J. Stewart. Fast horizon computation for accurate terrain rendering. Tech. Rep. 349, Department of Computer Science, University of Toronto, June 1996. Cited on page xix.

[Stew97] A. J. Stewart. Hierarchical Visibility in Terrains. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '97)*, pp. 217–228, 1997. Cited on page 23, 84, xx.

[Stew98a] A. J. Stewart and T. Karkanis. Computing the approximate visibility map, with applications to form factors and discontinuity meshing. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '98)*, pp. 57–68, 1998. Cited on page 59, 60, 67.

[Stew98b] A. J. Stewart. Fast Horizon Computation at All Points of a Terrain With Visibility and Shading Applications. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 4, No. 1, pp. 82–93, Jan. 1998. Cited on page xix.

[Stol91] J. Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations*. Academic Press, 1991. Cited on page 9, 10, 71, 85.

[Subr90] K. Subramanian and D. Fussel. Factors Affecting Performance of Ray Tracing Hierarchies. Tech. Rep. CS-TR-90-21, The University of Texas at Austin, July 1990. Cited on page xxix.

[Subr91] K. R. Subramanian and D. S. Fussell. Automatic Termination Criteria for Ray Tracing Hierarchies. In *Proceedings of Graphics Interface '91*, pp. 93–100, June 1991. Cited on page xxix.

[Subr92] K. Subramanian and D. Fussel. A Search Structure based on K-d Trees for Efficient Ray Tracing. Tech. Rep. Tx 78712-1188, The University of Texas at Austin, 1992. Cited on page xxix.

[Subr97] K. R. Subramanian and B. F. Naylor. Converting Discrete Images to Partitioning Trees. *IEEE Transactions on Visualization and Computer Graphics*, Vol. 3, No. 3, pp. 273–288, July 1997. Cited on page 31.

[Suda96] O. Sudarsky and C. Gotsman. Output-Sensitive Visibility Algorithms for Dynamic Scenes with Applications to Virtual Reality. *Computer Graphics Forum*, Vol. 15, No. 3, pp. C249–C258, Sep. 1996. Cited on page 100.

[Sung92] K. Sung and P. Shirley. Ray Tracing with the BSP Tree. In D. Kirk, Ed., *Graphics Gems III*, pp. 271–274, Academic Press, San Diego, 1992. Cited on page xxxi.

[Suth74] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A Characterization of Ten Hidden-Surface Algorithms. *ACM Computing Surveys*, Vol. 6, No. 1, pp. 1–55, March 1974. Cited on page 20.

[Szir95] L. Szirmay-Kalos ed., G. Márton, B. Dobos, T. Horváth, P. Risztics, and E. Kovács. *Theory of Three-Dimensional Computer Graphics*. Vol. 13 of *Technical Sciences: Advances in Electronics,* Akadémiai Kiadó, Budapest, Hungary, Sep. 1995. English revision by Ian A. Stroud. Cited on page xxviii.

[Szir97] L. Szirmay-Kalos and G. Marton. On the Limitations of Worst–case Optimal Ray Shooting Algorithms. In *Winter School of Computer Graphics (WSCG '97),* pp. 562–571, Feb. 1997. Cited on page xxix.

[Szir98a] L. Szirmay-Kalos. Global Ray-bundle Tracing. Technical Report, TR-186-2-98-18, Vienna University of Technology, Vienna, 1998. Cited on page xxx.

[Szir98b] L. Szirmay-Kalos and W. Purgathofer. Global Ray-bundle Tracing with Hardware Acceleration. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '98),* pp. 247–258, Vienna, Austria, June 1998. Cited on page xxx.

[Teic99] M. Teichmann and S. Teller. A Weak Visibility Algorithm with an Application to an Interactive Walkthrough. 1999. Downloaded from the WWW. Cited on page 101.

[Tell91] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. In *Computer Graphics (SIGGRAPH '91 Proceedings),* pp. 61–69, 1991. Cited on page 2, 24, 39, 84, 100, xvi, xvii.

[Tell92a] S. J. Teller. Computing the antipenumbra of an area light source. *Computer Graphics (SIGGRAPH '92 Proceedings),* Vol. 26, No. 2, pp. 139–148, July 1992. Cited on page 13, 90, 91, 100, 101, 109.

[Tell92b] S. J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, CS Division, UC Berkeley, Oct. 1992. Tech. Report UCB/CSD-92-708. Cited on page 24, 89, 100, 102, 104, 106, 107, 109, 110, 114, xvii.

[Tell93a] S. Teller and M. Hohmeyer. Computing the Lines Piercing Four Lines. Technical Report UCB/CSD 93/161, UC Berkeley, Apr. 1993. Cited on page 109.

[Tell93b] S. Teller and P. Hanrahan. Global Visibility Algorithms for Illumination Computations. In *Computer Graphics (SIGGRAPH '93 Proceedings),* pp. 239–246, 1993. Cited on page 101.

[Tell94] S. Teller, C. Fowler, T. Funkhouser, and P. Hanrahan. Partitioning and Ordering Large Radiosity Computations. In *Computer Graphics (SIGGRAPH '94 Proceedings),* pp. 443–450, July 1994. Cited on page 101.

[Tell96] S. Teller, K. Bala, and J. Dorsey. Conservative Radiance Interpolants for Ray Tracing. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '96),* pp. 257–268, Springer Wien, June 1996. Cited on page xxix.

[Tell98] S. Teller and J. Alex. Frustum casting for progressive, interactive rendering. Tech. Rep. MIT LCS TR–740, MIT, January 1998. Cited on page 27, 28, xxix.

[Thib87] W. C. Thibault and B. F. Naylor. Set Operations on Polyhedra Using Binary Space Partitioning Trees. In *Computer Graphics (SIGGRAPH '87 Proceedings),* pp. 153–162, July 1987. Cited on page 30, 31, xxiv.

[Tobl97] R. F. Tobler, A. Wilkie, M. Feda, and W. Purgathofer. A Hierarchical Subdivision Algorithm for Stochastic Radiosity Methods. In *Rendering techniques (Proceedings of Eurographics Workshop on Rendering '97),* pp. 193–204, Springer Wein, June 1997. Cited on page xxix.

[Tobo99] P. Tobola and K. Nechvíle. Linear Size BSP trees for Scenes with Low Directional Density. In *Proceedings of Winter School of Computer Graphics (WSCG'99)*, pp. 297–304, 1999. Cited on page 31.

[Torr90] E. Torres. Optimization of the Binary Space Partition Algorithm (BSP) for the Visualization of Dynamic Scenes. In *Proceedings of Eurographics '90*, pp. 507–518, North-Holland, Sep. 1990. Cited on page 31.

[Veac94] E. Veach and L. Guibas. Bidirectional Estimators for Light Transport. In *Rendering Techniques (Proceedings pf Eurographics Workshop on Rendering '94)*, pp. 147–162, Darmstadt, Germany, June 1994. Cited on page xxix.

[Veac97] E. Veach and L. J. Guibas. Metropolis Light Transport. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pp. 65–76, Addison Wesley, Aug. 1997. Cited on page xxix.

[Vegt90] G. Vegter. The Visibility Diagram: a Data Structure for Visibility Problems and Motion Planning. In *In Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory (SWAT '90)*, pp. 97–110, Springer, 1990. Cited on page 69, 70.

[Wald02] I. Wald, C. Benthin, and P. Slusallek. Interactive Global Illumination Using Fast Ray Tracing. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '02)*, Springer Wien, June 2002. Cited on page xxx.

[Wall89] J. R. Wallace, K. A. Elmquist, and E. A. Haines. A Ray Tracing Algorithm for Progressive Radiosity. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, pp. 315–324, July 1989. Cited on page 3, xxviii.

[Wand01] M. Wand, M. Fischer, I. Peter, F. M. auf der Heide, and W. Straßer. The Randomized z-Buffer Algorithm: Interactive Rendering of Highly Complex Scenes. In *Computer Graphics (SIGGRAPH '01 Proceedings)*, pp. 361–370, 2001. Cited on page xxii.

[Wang98] Y. Wang, H. Bao, and Q. Peng. Accelerated Walkthroughs of Virtual Environments Based on Visibility Preprocessing and Simplification. In *Computer Graphics Forum (Proceedings of Eurographics '98)*, pp. 187–194, 1998. Cited on page 100.

[Warn69] J. Warnock. A Hidden-Surface Algorithm for Computer Generated Half-Tone Pictures. Tech. Rep. TR 4–15, NTIS AD-733 671, University of Utah, Computer Science Department, 1969. Cited on page vi.

[Weil77] K. Weiler and P. Atherton. Hidden Surface Removal Using Polygon Area Sorting. *Computer Graphics (SIGGRAPH '77 Proceedings)*, Vol. 11, No. 2, pp. 214–222, July 1977. Cited on page 16, vii.

[Weis99] E. W. Weisstein. *The CRC Concise Encyclopedia of Mathematics*. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1999. Cited on page 9.

[Welz85] E. Welzl. Constructing the Visibility Graph for n-Line Segments in $O(n^2)$ Time. *Information Processing Letters*, Vol. 20, No. 4, pp. 167–171, May 1985. Cited on page 69.

[West97] R. Westermann and T. Ertl. The VSBUFFER: Visibility Ordering of Unstructured Volume Primitives by Polygon Drawing. In *IEEE Visualization '97*, pp. 35–42, IEEE, Nov. 1997. Cited on page ix.

[Whit79] T. Whitted. An improved illumination model for shaded display. In *Computer Graphics (Special SIGGRAPH '79 Issue)*, pp. 1–14, Aug. 1979. Cited on page 4, xxiii, xxix.

[Wile97] C. Wiley, A. T. Campbell III, S. Szygenda, D. Fussell, and F. Hudson. Multiresolution BSP Trees Applied to Terrain, Transparency, and General Objects. In *Proceedings of Graphics Interface '97*, pp. 88–96, Canadian Information Processing Society, May 1997. ISBN 0-9695338-6-1 ISSN 0713-5424. Cited on page 31.

[Will78] L. Williams. Casting Curved Shadows on Curved Surfaces. *Computer Graphics (SIGGRAPH '78 Proceedings)*, Vol. 12, No. 3, pp. 270–274, Aug. 1978. Cited on page 3, xxiii.

[Wimm01] M. Wimmer, P. Wonka, and F. Sillion. Point-Based Impostors for Real-Time Visualization. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '01)*, pp. 163–176, Springer-Verlag, 2001. Cited on page xxii.

[Wonk00] P. Wonka, M. Wimmer, and D. Schmalstieg. Visibility Preprocessing with Occluder Fusion for Urban Walkthroughs. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '00)*, pp. 71–82, 2000. Cited on page 2, 5, 19, 23, 69, 83, 84, 92, 94, 95, 97, 100, 131, 132, 133, xix.

[Wonk01a] P. Wonka. *Occlusion Culling for Real-Time Rendering of Urban Environments*. PhD thesis, Institute of Computer Graphics, Vienna University of Technology, 2001. Cited on page 23, 131, ix.

[Wonk01b] P. Wonka, M. Wimmer, and F. X. Sillion. Instant Visibility. In *Computer Graphics Forum (Proceedings of Eurographics '01)*, pp. 411–421, Blackwell Publishing, 2001. Cited on page 2, 46, xix.

[Wonk99] P. Wonka and D. Schmalsteig. Occluder Shadows for Fast Walkthroughs of Urban Environments. *Computer Graphics Forum (Proceedings of Eurographics '99)*, Vol. 18, No. 3, pp. 51–60, Sep. 1999. Cited on page 2, xvi.

[Woo90a] A. Woo and J. Amanatides. Voxel Occlusion Testing: A Shadow Determination Accelerator for Ray Tracing. In *Proceedings of Graphics Interface '90*, pp. 213–220, May 1990. Cited on page xxiii, xxix.

[Woo90b] A. Woo, P. Poulin, and A. Fournier. A Survey of Shadow Algorithms. *IEEE Computer Graphics and Applications*, Vol. 10, No. 6, pp. 13–32, Nov. 1990. Cited on page 3, xxiii.

[Yage95] R. Yagel and W. Ray. Visibility Computation for Efficient Walkthrough of Complex Environments. *Presence: Teleoperators and Virtual Environments*, Vol. 5, No. 1, 1995. Cited on page 100, xviii.

[Yama97] F. Yamaguchi and M. Niizeki. Some Basic Geometric Test Conditions in Terms of Plücker Coordinates and Plücker Coefficients. *Visual Computer*, Vol. 13, No. 1, pp. 29–41, 1997. Cited on page 102.

[Zach02] C. Zach. Integration of Geomorphing into Level of detail management for Real-Time Rendering. In *Proceedings of Spring Conference on Computer Graphics (SCCG'02)*, pp. 109–116, Budmerice, Slovakia, 2002. Cited on page xxi.

[Zhan97a] H. Zhang and K. E. Hoff III. Fast Backface Culling Using Normal Masks. In *Proceedings of 1997 Symposium on Interactive 3D Graphics*, pp. 103–106, ACM SIGGRAPH, Apr. 1997. Cited on page 39, ix.

[Zhan97b] H. Zhang, D. Manocha, T. Hudson, and K. E. Hoff III. Visibility Culling Using Hierarchical Occlusion Maps. In *Computer Graphics (Proceedings of SIGGRAPH '97)*, pp. 77–88, 1997. Cited on page 2, 40, 46, xiii.

[Zhan98] H. Zhang. *Effective Occlusion Culling for the Interactive Display of Arbitrary Models*. PhD thesis, Department of Computer Science, UNC-Chapel Hill, 1998. Cited on page xiv.

[Zwaa95] M. van der Zwaan, E. Reinhard, and F. W. Jansen. Pyramid Clipping for Efficient Ray Traversal. In *Rendering Techniques (Proceedings of Eurographics Workshop on Rendering '95)*, Dublin, Ireland, 1995. Cited on page xxix.