HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science

Samuli Laine

An Incremental Shaft Subdivision Algorithm for Computing Shadows and Visibility

Master's Thesis
March 29, 2006

Supervisor: Lauri Savioja, D.Sc. (Tech.), Professor of Virtual Technology

Instructor: Timo Aila, D.Sc. (Tech.)

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Samuli Laine		
Name of the thesis:	An Incremental Shaft Subdivision Algorithm for Computing Shadows and Visibility		
Date:	March 29, 2006	**Pages:**	12+91
Department:	Department of Computer Science	**Professorship:**	T-111
Supervisor:	Lauri Savioja, D.Sc. (Tech.)		
Instructor:	Timo Aila, D.Sc. (Tech.)		

The rendering of soft shadows is an important task in computer graphics. Soft shadows appear when the light source is not modeled as a single point but as an object with nonzero surface area. Obtaining correct physically-based shadows requires determining the amount of light that flows from the light source to a receiving point on the surface being rendered. This is generally computationally expensive, and efficient solution methods are needed for keeping the rendering times on a tolerable level.

There is usually significant coherence in shadows among nearby receiving points, and nearby parts of a light source also tend to contribute to the image in a similar fashion. Exploiting these forms of coherence is the key element of modern soft shadow algorithms.

This thesis presents a novel physically-based soft shadow algorithm that attempts to exploit the coherence as much as possible, solving the shadow relations in large chunks instead of considering single points in the emitting or receiving end. The computation of shadow relations is performed hierarchically, and an efficient representation of shadow-casting geometry is maintained incrementally. The algorithm is a generic tool for the solving sets of visibility relations in polygonal scenes, and may have uses in areas other than shadow computation as well.

In addition to presenting the novel algorithm in detail, several existing physically-based shadow algorithms are analyzed and ranked according to their computational complexities. Experimental results are also presented for illustrating the applicability of the novel algorithm in different kinds of rendering situations.

Keywords: computer graphics, soft shadows, visibility computation

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä:	Samuli Laine		
Työn nimi:	An Incremental Shaft Subdivision Algorithm for Computing Shadows and Visibility		
Päivämäärä:	29. maaliskuuta 2006	**Sivumäärä:**	12+91
Osasto:	Tietotekniikan osasto	**Professuuri:**	T-111
Työn valvoja:	Lauri Savioja, TkT		
Työn ohjaaja:	Timo Aila, TkT		

Pehmeiden varjojen piirto on tärkeä tehtävä tietokonegrafiikassa. Pehmeitä varjoja muodostuu, kun valonlähdettä ei esitetä pisteenä vaan pintana, jolla on nollasta poikkeava pinta-ala. Fysikaalisesti oikeiden varjojen laskennassa pitää määrittää tarkasteltavan pinnan pisteeseen valonlähteestä saapuvan valon määrä. Tämä on yleisesti laskennallisesti raskasta, ja tehokkaat ratkaisumenetelmät ovat tarpeen, jotta kuvan muodostusaika pysyy siedettävänä.

Useimmiten lähekkäisten pisteiden vastaanottamat varjot ovat likimain samanlaisia, ja valonlähteen lähekkäiset osat myös vaikuttavat kuvaan enimmäkseen samalla tavalla. Modernit varjoalgoritmit perustuvat näiden koherenssin muotojen hyödyntämiseen.

Tässä työssä esitellään uusi fysikaalisesti oikeiden pehmeiden varjojen laskenta-algoritmi, joka pyrkii hyödyntämään koherenssia niin paljon kuin mahdollista laskemalla varjorelaatiot suurissa ryhmissä sen sijaan, että tarkasteltaisiin yksittäisiä pisteitä valonlähteellä tai varjostettavalla pinnalla. Varjorelaatioiden laskenta suoritetaan hierarkkisesti, ja tehokasta esitystä varjostavista pinnoista ylläpidetään inkrementaalisesti. Algoritmi on yleiskäyttöinen työkalu näkyvyysrelaatiojoukkojen ratkaisemiseen, ja sillä voi olla muitakin käyttökohteita varjojen laskennan lisäksi.

Uuden algoritmin yksityiskohtaisen kuvauksen lisäksi työssä analysoidaan useita olemassaolevia fysikaalisesti oikeiden pehmeiden varjojen laskenta-algoritmeja ja luokitellaan ne algoritmisten kompleksisuusluokkiensa perusteella. Työssä esitetään myös kokeellisia tuloksia, joiden avulla voidaan arvioida algoritmin käyttökelpoisuutta erilaisissa laskentatilanteissa.

Avainsanat: tietokonegrafiikka, pehmeät varjot, näkyvyyslaskenta

Acknowledgments

The author thanks Timo Aila for his guidance, discussions and criticism; professor Lauri Savioja for support, criticism and providing a comfortable working environment; Janne Kontkanen, Jaakko Lehtinen and Ville Miettinen for discussions; people at Hybrid Graphics Ltd. for selflessly sharing their expertise when the author entered the field of computer graphics.

This work has been partially funded by the National Technology Agency of Finland, Anima Vitae, Bitboys, Hybrid Graphics and Remedy Entertainment. The author was also supported by the Finnish Cultural Foundation.

Helsinki, March 2006

Samuli Laine

Copyright Notice

Contents

Chapter 1

Introduction

1.1 Light and Shadows

The topic of this thesis is computing shadows and visibility. At a glance, these two phenomena may seem fundamentally different, but in fact they are tightly connected. Since shadows appear in places that light does not reach, computing shadows is conceptually equivalent to determining the visibility between light sources and the surfaces to be shaded.

Shadows are an important feature in computer-generated images. Since shadows are always present in real-life images, it has become natural for humans to interpret e.g. spatial relationships between objects based on even then subtlest of shadows. Thus, in synthetic images the presence of correctly computed shadows greatly increases the perceived realism of an image.

One might ask why we speak of computing *shadows* instead of computing *light*, since these two seem to be simply inverted questions: to compute shadows is to compute the absence of light. The reason is that it is easy to generate images without shadows by neglecting the possibility that a surface point might not be visible from the light source, i.e. in shadow. In early computer graphics, this was customary, and is so even today, since computing shadows is much harder than computing lighting without shadows. Therefore, shadows are something that need to be explicitly added into the image, and this justifies the current usage of terms.

From the visibility point of view, the case where a surface point is in light is a special case, since this occurs only when there are no surfaces between the surface point and the light source. In contrast, shadow appears when there are one or more surfaces between these two blocking the light. We might speculate that in real-life situations the latter case is immensely more common than the former, since e.g. in a room of a building, most of the light sources in other rooms do not contribute to the perceived lighting because of the walls that prevent the flow of light.

Two main classes of shadows are *hard shadows* and *soft shadows*. Perfectly hard shadows do not exist physically, but due to their much simpler mathematical nature, they are often used in computer graphics. Hard shadows appear when the light source is a single point having no surface area or volume. In this case, a surface point to be shaded is either completely lit or completely shadowed, or in other words, the light source is either completely visible or completely occluded from the surface point. The result is a binary shadow that often does not look realistic, but may suffice as an approximation for a relatively small light source. Soft shadows emerge when the light source is represented as a geometrical entity that is not a point. Usually the light sources are represented

as polygons, but spherical light sources, line segments, or other extended geometrical entities can be used. Now the question whether a surface point is in light or in shadow is not binary any more, since it is possible that a portion of the light source is visible to the surface point, and a portion is occluded. This results in shadow boundaries with gradual falloff from fully shadowed to fully lit. Soft shadows are much more realistic than hard shadows, but they are also more difficult to compute. For soft shadows, is no more sufficient to determine a single visibility relation between each surface point and light source. Instead, the correct solution requires integrating the visibility over the surface of the area light source, or equivalently, over the spatial angle subtended by the light source as seen from the receiver point. In practice, the integral is usually solved using Monte Carlo integration techniques that rely on sampling the function inside the integral.

Two main classes of algorithms exist for soft shadow computations, and they are usually called *physically-based* and *approximative*. Even though "physically-based" is not a particularly descriptive term, in this context it means roughly that as the number of Monte Carlo samples used in solving the visibility is increased, the result converges to the correct solution. In contrast to physically-based algorithms, approximative algorithms often do not explicitly solve the visibility, at least not separately for every point to be shaded, but approximate it with various heuristics. In many situations, approximative algorithms produce plausible results, but it is generally hard to guarantee the applicability of an approximative algorithm in different situations. Because of this, we shall concentrate almost entirely on physically-based shadow algorithms in this thesis.

1.2 Common Approximations

A number of approximations is usually made in computer graphics, and even though they are often not explicitly mentioned, it is worthwhile to enumerate the most important of them here.

First of all, practically every computer graphics algorithm assumes ray optics. Even though this model of light is not physically accurate as it fails to model phenomena such as diffraction and interference, it is most often sufficient for computing realistic images. Even algorithms that conceptually employ photons for light transport do not take interaction between photons into account, and therefore effectively assume ray optics. The polarization of light is also usually ignored.

It is also assumed that no participating media exists in the "empty" space between surfaces, unless explicitly mentioned.[1] This means that a ray emanating from a surface always hits the surface in the direction of the ray with zero probability of deviating from its path. If participating media were present, a ray might scatter any number of times from particles in the medium, and end up in any point reachable from the boundary of the medium with probability that depends on the shape and optical characteristics of the medium. The computation of these probabilities is non-trivial, and accounting for different possible scattering paths leads to computationally expensive integration tasks.

Since photons generally move very fast, it is safe to assume that the light transport of a system is always in a state of equilibrium. In this situation, the flow of light in any infinitesimal volume is constant over time, and temporal effects of light transport can be ignored. In certain materials, temporal effects may occur at a slower time scale due to phosphorescence, but these kind of phenomena are usually not taken into account.

Many algorithms for computing soft shadows impose certain limitations for objects that cast shad-

[1] Despite this, the optical density of transparent materials *is* usually taken into account, giving rise to refractions in the boundaries between materials. The interior of a solid glass ball, for instance, is therefore actually modeled as something like "void with the optical density of glass".

ows. The most common limitation is that the shadow caster must be opaque, meaning that they block the rays of light completely instead of passing some amount of light through. This requirement is also made in the algorithm presented in this thesis.

It can also be counted as an approximation that integrals involved in light and shadow computations are usually solved using Monte Carlo sampling instead of analytic methods. There are some notable exceptions to this, but the vast majority of soft shadow algorithms, including the one presented in this thesis, rely on sampling. One might claim that this approach is justified by the finite number of photons present in real-world situations as well, but this is somewhat of a moot point because the number of photons to be modeled is generally stupendously large.

1.3 Direct and Indirect Illumination

In computer graphics there are distinct classes of light transport phenomena that require different algorithms to be solved efficiently. We now briefly consider three classes of *illumination models* that all model different phenomena in light transport.

Local illumination. Local illumination models consider the transport of light at the surfaces. When a ray of light hits a surface, it may be absorbed or scattered in different directions with a probability associated with each outgoing direction. As an example, the simplest local illumination model is a diffuse surface that scatters incoming light to all directions with equal probability. More complex models are also frequently used. If only local illumination is taken into account when rendering an image, no shadows will be present.

Direct illumination. Direct illumination considers the rays of light that emanate from a light source, reflect once from a point to be shaded, and proceed directly to the virtual camera. Shadow algorithms are needed for determining whether a ray from the light source to the surface is blocked or not. The combination of local and direct illumination is currently perhaps the most popular solution in rendering computer-generated images, since it is able to account for the most important shadow features.

Global illumination. Local and direct illumination alone do not model the light transport completely, since a ray of light may be scattered and reflected multiple times before it reaches the virtual camera. Global illumination algorithms take all types of light scattering paths into account, producing extremely realistic-looking images. Shadow algorithms can still be used for determining whether a vertex on the light path is reached by light coming directly from the light source. However, computing global illumination is generally much more time-consuming than computing direct illumination alone. Sometimes global illumination effects can be approximated by using direct illumination with large light sources.

1.4 Scope of This Thesis

In this thesis, we develop an algorithm for computing shadows caused by direct illumination. The algorithm can also be applied in solving other visibility problems.

To limit our study, several aspects critical to computer image generation are left out. Most notably, we discuss global illumination only briefly. We do not present a full ray-tracing rendering pipeline, since our focus is in the very specific part of it; computing the shadows.

1.5 Contributions

In this thesis we present a novel incremental shaft subdivision algorithm for efficient computation of physically-based soft shadows and visibility. Experimental results and analysis of the applicability of the algorithm in different situations are given. We also examine several existing physically-based soft shadow algorithms in detail, and qualitatively analyze their behavior as well as the behavior of the new algorithm presented in this thesis.

1.6 Organization of This Thesis

The rest of this thesis is organized as follows. In the closing section of this chapter we lay ground for further analysis and presentation of the novel algorithm by introducing the relevant key concepts. In Chapter 2 we discuss previous methods for shadow and visibility computation. Chapter 3 contains qualitative analysis of selected hard and soft shadow algorithms as well as discussion on the properties of shadow computation algorithms in general. Chapter 4 describes the novel incremental shaft subdivision algorithm in detail. In Chapter 5 we present experimental performance results for the new algorithm. Finally, in Chapter 6 we discuss the strong and weak points of the presented algorithm, and outline possible directions for future work.

1.7 Prerequisites

The reader is assumed to have basic knowledge of computer graphics and rendering, as well as firm mental grasp of three-dimensional geometry. Familiarity with pseudocode representation of algorithms is useful, as it is used extensively in Chapters 3 and 4.

1.8 Preliminaries

In the following, we introduce a set of key terms, concepts and conventions that will be used later in this thesis.

Visibility function. Given the assumption that objects always block rays of light either completely or not at all, visibility between two points is a binary function. We may formalize it as follows: given two 3D points p_a and p_b, we define function $\mathcal{V}(p_a, p_b)$ so that $\mathcal{V}(p_a, p_b) = 1$ if no surfaces intersect the line segment $p_a \rightarrow p_b$, excluding the endpoints, and $\mathcal{V}(p_a, p_b) = 0$ if at least one surface intersects the line segment. We also define $\mathcal{V}(p_a, p_b) = 1$ if $p_a = p_b$.

Depth complexity. Another useful function connected to the visibility is the *depth complexity function*, which tells how many surfaces lie on a line segment. Formally, we define $\mathcal{D}(p_a, p_b)$, as the number of surfaces that intersect the line segment between p_a and p_b, again excluding the endpoints. Functions $\mathcal{V}$ and $\mathcal{D}$ are connected: $\mathcal{V} = 1$ iff $\mathcal{D} = 0$.

Receiver points. When computing shadows, we have a set of points where we are interested in knowing the shadows. These are the *receiver points*, denoted r_i, where i is an index. The receiver points are typically determined by tracing rays from the camera through the pixels of the image. Rasterization can also be used.

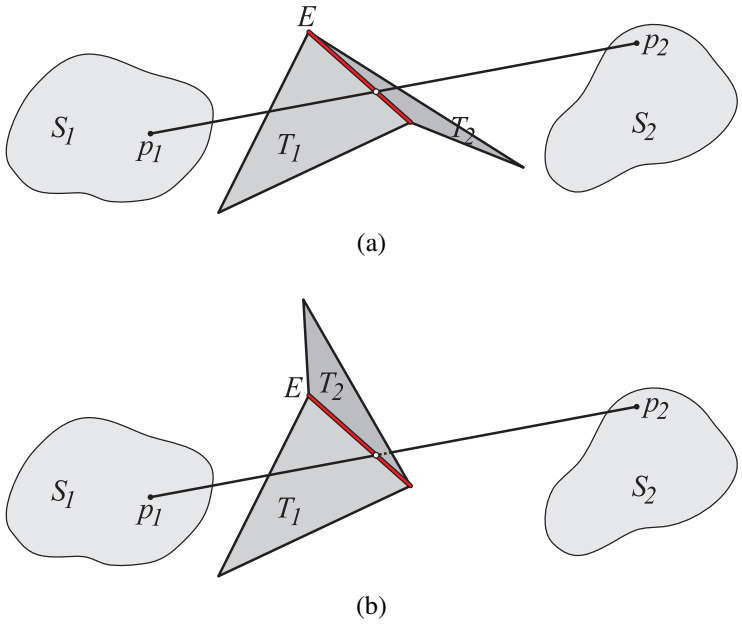


Figure 1.1 Illustration of the conditions an edge E has to fulfill in order to be a silhouette edge between regions S_1 and S_2. The front face is seen from all triangles. (a) E is a silhouette edge between S_1 and S_2, because a line segment $p_1 \rightarrow p_2$ intersects edge E, and the depth complexity is discontinuous at $\mathcal{D}(p_1, p_2)$. The discontinuity of the depth complexity function is evident if we consider e.g. moving p_1 slightly up and down in the figure. (b) E is not a silhouette edge between S_1 and S_2, since whichever p_1 and p_2 we choose so that $p_1 \rightarrow p_2$ intersects E (excluding the endpoints of E), the depth complexity function is continuous around $\mathcal{D}(p_1, p_2)$.

Light samples. Since we use Monte Carlo methods for determining the visibility between the receiver points and the light sources, we need to represent the light sources as a set of *light samples*, denoted l_j, where j is an index. The light samples are distributed on the surfaces of the light sources before computing the shadows. The number of light samples to be used is a quality parameter: approximating a light source with only a few samples results in visible artifacts, but as the number of samples increases, the appearance of the shadows converges to the correct solution.

Visibility relations. Given receiver points r_i and light samples l_j, the shadow computation task can be stated as determining $\mathcal{V}(r_i, l_j)$ for every i, j pair. A single i, j pair defines one *visibility relation*, and if there are N receiver points and M light samples, there are $N \times M$ visibility relations in total.

Silhouette edges. E is a silhouette edge between two spatial regions S_1 and S_2 iff the depth complexity function is discontinuous (because of E) at some $\mathcal{D}(p_1, p_2)$, where $p_1 \in S_1$, $p_2 \in S_2$, and line segment $p_1 \rightarrow p_2$ intersects E (excluding the endpoints of E). Otherwise, if the depth complexity function is continuous at every $\mathcal{D}(p_1, p_2)$ where $p_1 \rightarrow p_2$ intersects E, or no line segment with endpoints in S_1 and S_2 intersecting E exists, E is not a silhouette edge between S_1 and S_2 (see Figure 1.1). If E is not a silhouette edges, it causes no changes in depth complexity between S_1 and S_2, and thus, it cannot cause changes in the visibility function either. Conversely, if there is a change in the visibility function, it must be caused by a silhouette edge. This makes silhouette edges particularly interesting in visibility computation. In Chapter 4, we will show how to discard non-silhouette edges in order to reach an efficient representation of the shadow-casting geometry.

Chapter 2

Previous Work

In this section, we review previous work on shadow and visibility algorithms. Our main focus is on physically-based shadow algorithms that approximate the visibility only by sampling, thereby guaranteeing the convergence towards correct result as the number of samples is increased. Approximative shadow algorithms will be discussed only quite briefly. Our primary emphasis is on soft shadows, because of their great importance in off-line rendering and generally more interesting nature.

Because the novel algorithm presented in this thesis is essentially a visibility algorithm, we also survey previous work on visibility algorithms. On that subject, we concentrate on object-space visibility algorithms and mostly skip image-space algorithms since they generally cannot be applied in solving generic visibility problems. In addition, we focus on exact and conservative visibility algorithms, because other kinds of visibility algorithms do not suit well for computing physically-based shadows, as will be shown later in this chapter.

2.1 Hard Shadows

Hard shadows are commonly used in real-time applications, but avoided in high-quality off-line rendering due to their unrealistic appearance. However, some hard shadow algorithms can be extended to produce approximative soft shadows, and these are sometimes used in off-line rendering because of their efficiency.

In the following, we discuss both approximative and physically-based hard shadows. Physically-based hard shadow algorithms are essentially visibility algorithms that solve visibility relations from multiple points to a single point, the light source.

2.1.1 Approximative hard shadows

Shadow mapping [74] is the predominant method for real-time hard shadow rendering, and it is also commonly used in off-line production rendering due to its versatility and simplicity. A shadow map is a depth buffer that is computed using the light source as the viewpoint. To determine if a world-space point p_W is in shadow, it is transformed to the image space of the shadow map, and the depth of transformed point p_T is compared against the value stored in the shadow map.

If the depth of the point is greater than the value stored in the map, the point is in shadow, since there is a surface closer to the light source, i.e. between point p and the light source. Shadow mapping is an approximative algorithm only because of the discretization of the shadow map, and hard shadows computed using shadow maps converge towards the correct result as the resolution of the shadow map increases without bounds. With finite resolution, the transformed world-space query points do not land at the centers of the pixels in the shadow map, where the depth is correct, and various artifacts arise from this discrepancy. A number of techniques have been developed for coping with this problem, including second-depth shadow mapping [72], and methods for distributing the resolution of the shadow map unevenly. Manipulating the projection from world-space 3D to light-space 2D in order to concentrate the shadow map resolution near the viewpoint was first proposed by Stamminger and Drettakis [69], and novel projection matrix constructions were subsequently presented by Wimmer et al. [75] and Martin and Tan [57]. Methods for subdividing the shadow map into a number of tiles have been presented by Fernando et al. [30] and Arvo [7].

2.1.2 Physically-based hard shadows

Shadow mapping can be modified so that the shadow queries can be answered exactly, by first collecting the query points and then constructing the shadow map using the transformed query points as the sampling points for rasterization. This was independently discovered by Aila and Laine [2] and Johnson et al. [46]. The irregular sampling results in a physically-based hard shadow algorithm, but with the cost of losing the simplicity of rasterizing into a regular lattice of sampling points. In addition, the execution model of the algorithm is changed, since the query points must be gathered before the shadow map can be constructed. Current graphics hardware does not support rasterization with irregular sampling, but in off-line rendering there is no such limitation. Hardware for performing irregular rasterization has also been proposed [45].

Projection shadows [15] can be applied if the surface that receives the shadows is planar. This method is based on construction of a matrix that transforms the surface primitives (usually triangles) from their world-space position projectively onto the plane of the shadow receiver. Conceptually, the shadow-casting object is flattened into its own shadow. The construction of the projection matrix is fairly straightforward; for a directional light source, equivalent to a point light at infinity, the projection is essentially orthographic, whereas for usual point lights it is projective. The most critical limitation of projection shadows is that only planar shadow receivers can be supported, making the method unsuitable for general-purpose rendering.

The *shadow volume* algorithm [19] constructs the three-dimensional volumes that represent the shadowed regions, and tests whether the visible surfaces are inside these regions or not. The hardware-accelerated version [41] is currently a popular method for rendering physically-based hard shadows in real-time. It works by first rendering the scene with ambient lighting only, collecting the depths of the visible surfaces in the depth buffer, and then rendering the boundaries of the shadow volumes into a hardware stencil buffer. The rendering of shadow volume boundaries effectively counts the number of times a view ray enters and exits the shadow volumes, and if the number of enter events is greater than the number of exit events, the visible surface point is in shadow. The original shadow volume algorithm, commonly dubbed as *Z-pass* shadow volumes, cannot handle cases where the shadow volumes intersect the near plane of the view frustum. A small modification that solves this problem, so-called *Z-fail* shadow volumes, was independently discovered by Bilodeau and Songy in 1999 and by Carmack in 2000, as explained by Everitt and Kilgard [29]. Recently, a method that enables the original Z-pass method to handle the front-clip case was presented by Hornus et al. [43].

Shadow volume algorithms are often criticized for their excessive fill-rate consumption, and various methods for alleviating this problem have been presented. Lengyel [52] bounds the pixel processing

region by using scissor tests, and a similar bounding in depth direction can be obtained by using depth bounds hardware extension [63]. Combining both scissor test and depth bounds was suggested by Mcguire et al. [59]. Lloyd et al. [54] and Chan and Durand [16] present methods for culling shadow volumes. Aila and Akenine-Möller [1] present a two-stage shadow volume rendering algorithm that first identifies screen-space tiles that may contain a shadow boundary, and then rasterizes shadow volume boundaries with full resolution only in the potential boundary tiles. Their technique requires modifications in the hardware. Laine [49] introduces a heuristic method that chooses between Z-pass and Z-fail shadow volumes on a per-tile basis, which reduces the fill-rate requirements. This algorithm is also inapplicable with current hardware, though the required hardware modifications are relatively small.

Ray casting [73] is a classical method for computing physically-based hard shadows in off-line rendering. There is potential confusion in terms here, since ray-cast shadows are sometimes referred to as "global illumination", simply because they transcend the previous limitations of local illumination alone. As explained in Section 1.3, in this thesis we reserve term global illumination for situations where multiple bounces of light are taken into account, and speak of direct illumination when only direct rays from light source to the point being shaded are taken into account. In this more modern usage of terms, ray-cast shadows belong into the category of direct illumination.

In ray casting, determining whether a point p is in shadow is done by constructing a line segment between p and the point light source. Then it is checked if there is a surface that intersects this line segment. If such a surface exists, point p is in shadow, and otherwise it is in light. Generally, ray casting can be used for obtaining values for the visibility function $\mathcal{V}(p_a, p_b)$ for any pair of points p_a, p_b, and this operation is called *visibility query*, or more explicitly *point-to-point visibility query*. Implementing this kind of visibility query is conceptually simple. A brute-force implementation could loop over all triangles in the scene, checking for line segment vs. triangle intersections one by one, but this yields a hopelessly inefficient algorithm. The standard method for performing the query faster is to construct a spatial acceleration structure of shadow-casting geometry, which leads to sub-linear computational cost with respect to the number of shadow-casting primitives. Performing ray casts efficiently is a topic of great interest in computer graphics, and vast amount of research has been devoted to it. We will not go into details here, but refer the reader to extensive surveys on the topic [17, 6, 39] and an interesting analysis of worst-case and average-case complexity of different methods by Szirmay-Kalos and Márton [70]. Optimizations specific to shadow rays, where finding any surface that intersects the ray is sufficient, include *shadow caching* [34]. In shadow caching, a list is maintained for primitives that have most recently succeeded in blocking shadow rays. When casting a new shadow ray, it is first tested against the recently encountered occluders, and if intersection is found, the test is finished. Shadow caching does not accelerate the visible shadow ray casts.

It should be noted that all physically-based hard shadow algorithms—alias-free shadow maps, projection shadows, shadow volumes and ray casting—give identical results (up to numerical precision), since the visibility is computed correctly. These algorithms do differ fundamentally in their inner workings, which leads to different computational costs with respect to number of receiver points, light samples and shadow-casting primitives. These issues will be discussed in Chapter 3.

2.2 Soft Shadows

Soft shadows arise when the light source is not considered to be a point, but as an object or surface with nonzero area. Such light sources are usually referred to as *area light sources*, and we shall adopt this term in this thesis as well. Generally, area light sources are objects that emit light, although most

often simple polygonal light sources are used. It should be noted that even a volumetric light source can be expressed using only its bounding surface as a light source, assuming that no shadow caster or shadow-receiving surface is located inside the bounding surface.

It is worth the effort to plunge into some mathematical formulations of light transport at this point, in order to illustrate what we exactly are computing when rendering soft shadows. This is the most mathematical portion in this thesis, and if the reader feels uncomfortable with the formulas, it does no major harm to skip them.

2.2.1 Mathematical Background

Soft shadows are fundamentally more difficult to compute than hard shadows, since the visibility of an area light source to a point being shaded cannot be expressed simply as 0 or 1. A scalar value that can assume values between 0 and 1 does not suffice either. We start developing the equation for shadow computation from the good old *rendering equation* first formulated by Kajiya in 1986 [47]. The rendering equation governs the light transport when full global illumination is taken into account. Using slightly more modern notation than what was used in the original paper, we write the rendering equation as follows:

$$u(p \rightarrow \omega_{out}) = e(p \rightarrow \omega_{out}) + \int_{\Omega_p} f_r(p, \omega_{in} \rightarrow \omega_{out}) u(p \leftarrow \omega_{in}) \lfloor \mathbf{n}_p \cdot \omega_{in} \rfloor \, d\omega_{in}. \tag{2.1}$$

Here, $u(p \rightarrow \omega_{out})$ denotes the total radiance flowing from point p into outgoing direction ω_{out}, and $e(p \rightarrow \omega_{out})$ is the radiance emitted from point p towards ω_{out}. Unless p is on a surface of a light source, the emission term $e(p \rightarrow \omega_{out})$ is zero. The integral is taken over the hemisphere Ω_p that is directed towards the normal vector $\mathbf{n}_p$ at p. Function $f_r(p, \omega_{in} \rightarrow \omega_{out})$ is the BRDF [61] of the surface at p, and it tells how big portion of incident light coming from direction ω_{in} to point p is reflected towards outgoing direction ω_{out}.[1] Expression $\lfloor \cdot \rfloor$, commonly used in computer graphics texts, denotes the "clamp-to-zero" function $\max(\cdot, 0)$.

As can be seen, the radiance function u appears both on the left hand side of the equation and also inside the integral, making this a so-called Fredholm equation of the second kind. Since our goal here is to compute direct illumination only, we must replace the radiance function u inside the integral by emission function e, which greatly simplifies the situation.

In addition, we make a somewhat surprising twist at this point, and remove the emission term $e(p \rightarrow \omega_{out})$ outside the integral. This makes the light sources invisible entities that emit light while being invisible to the camera, which is often desirable. Most commercial graphics packages treat area light sources in this way. This gives the artists freedom to e.g. place additional light sources between the camera and the objects visible in the scene, which is impossible in physical world without showing the light sources in the image. If, on the other hand, it is desirable to see the light sources in the image, it is customary to present the corresponding surfaces *twice*, once as a light source, and once as plain geometry that may have an emissive material, but that does not act as a light source. We thus get the following equation:

$$u(p \rightarrow \omega_{out}) = \int_{\Omega_p} f_r(p, \omega_{in} \rightarrow \omega_{out}) e(p \leftarrow \omega_{in}) \lfloor \mathbf{n}_p \cdot \omega_{in} \rfloor \, d\omega_{in}. \tag{2.2}$$

[1] The BRDF (bidirectional reflectance distribution function) of a surface depends on its material, and there are many models for expressing it in analytical forms. Tabulated BRDFs that are measured from real materials are also becoming popular. We will not discuss BRDFs further in this thesis, and merely note their vital role in creating realistic images.

Applying this integral directly when computing soft shadows from direct illumination would be impractical. This is mainly because $e(p \leftarrow \omega_{in})$ inside the integral is zero unless a light source is visible from p in direction ω_{in}. We may recast this equation into so-called area formulation [27] by a change of integration variable. Assuming no participating media, the incoming radiance at p from direction ω_{in} is the same as outgoing radiance from point l in direction $-\omega_{in}$, where l is a point on a light source visible from p in direction ω_{in}, allowing us to state that $e(p \leftarrow \omega_{in}) = e(l \rightarrow -\omega_{in})$. The differential solid angle $d\omega_{in}$ can be expressed in terms of differential area as follows [27]:

$$d\omega_{in} = \lfloor \mathbf{n}_l \cdot -\omega_{in} \rfloor \frac{dA}{|l-p|^2}. \tag{2.3}$$

Because of the visibility criterion, we must multiply the kernel of the integral by the visibility function $\mathcal{V}(p,l)$. This yields the following equation, where we integrate over the surface L of a light source:

$$u(p \rightarrow \omega_{out}) = \int_L f_r(p, \omega_{in} \rightarrow \omega_{out}) e(l \rightarrow -\omega_{in}) \mathcal{V}(p,l) \frac{\lfloor \mathbf{n}_p \cdot \omega_{in} \rfloor \lfloor \mathbf{n}_l \cdot -\omega_{in} \rfloor}{|l-p|^2} \, dA_l. \tag{2.4}$$

It is important to note that ω_{in} now depends on the integration variable l and is not constant. Dissecting the equation, we see that there are three functions inside the integral: BRDF, emission function and visibility function. Furthermore, there are factors that account for the geometry of the situation: $\lfloor \mathbf{n}_p \cdot \omega_{in} \rfloor$ converts the cross-sectional flow from the direction of ω_{in} to flow per surface area at p, and $\lfloor \mathbf{n}_l \cdot -\omega_{in} \rfloor$ does the same in reverse direction at the surface of the light source. Attenuation factor $1/|l-p|^2$ accounts for distance between p and l.

Solving the integral in Equation 2.4 analytically is possible for certain types of BRDFs and visibility functions, but even in these situations, it is often faster to solve the integral using Monte Carlo sampling instead. Integral over an arbitrary function can be approximated by constructing a number of uniform-density *sampling points* in the domain of integration, and averaging the values of the function in these locations:

$$\int_D f(x) \, dx \approx \frac{1}{M} \sum_{j=0}^{M-1} f(x_j), \tag{2.5}$$

where x_j ($0 \leq j < M$) are the sampling points placed in the domain D that we integrate over. If we generate M uniform-density samples l_j on the surface of the light source, we can thus approximate Equation 2.4 with the following sum:

$$u(p \rightarrow \omega_{out}) \approx \frac{1}{M} \sum_{j=0}^{M-1} f_r(p, \omega_{in} \rightarrow \omega_{out}) e(l_j \rightarrow -\omega_{in}) \mathcal{V}(p,l_j) \frac{\lfloor \mathbf{n}_p \cdot \omega_{in} \rfloor \lfloor \mathbf{n}_{l_j} \cdot -\omega_{in} \rfloor}{|l_j-p|^2}. \tag{2.6}$$

In this formulation, computation is easy since we sample all the difficult functions in discrete points only. As sample count M is increased, the approximation converges to the correct result, and if M is too small, there will be visible noise in the shadows due to approximation errors. A typical value for M would be around 200 in situations where a single, reasonably small light source illuminates the scene, provided that the light samples l_j are generated using a good low-discrepancy distribution.

The most difficult part in applying Equation 2.6 is computing the values of the visibility function $\mathcal{V}(p,l_j)$, as it depends on the entire scene, in contrast to BRDF and emission functions that are purely local in nature. The classical method is to cast a separate shadow ray for each l_j, which inevitably gives linear performance with respect to the number of light samples M, regardless of the computational complexity of casting a single shadow ray. Modern approaches attempt to solve $\mathcal{V}(p,l_j)$ for multiple j at once, and possibly for multiple p also. This applies the algorithm presented in this thesis as well.

2.2.2 Approximative Soft Shadows

In approximative soft shadow algorithms, many simplifications are usually made that steer the result away from the correct result that would be obtained if Equation 2.6 were used. Such simplifications include, but are not limited to:

- approximating the visibility function V,
- evaluating the BRDF function f_r only once, e.g. for the direction to the center of the light source,
- assuming constant emission function e,
- assuming constant geometry terms $\lfloor \mathbf{n}_p \cdot \omega_{in} \rfloor$, $\lfloor \mathbf{n}_l \cdot -\omega_{in} \rfloor$ and $1/|l-p|^2$ over the surface of the light source.

Most approximative soft shadow algorithms merely modulate the (approximate) color of the light illuminating the surface by an approximate scalar visibility factor, as in the following equation:

$$u(p \rightarrow \omega_{out}) \approx e(l_c, -\omega_{in\,c}) f_r(p, \omega_{in} \rightarrow \omega_{out}) \frac{\lfloor \mathbf{n}_p \cdot \omega_{in\,c} \rfloor \lfloor \mathbf{n}_{l_c} \cdot -\omega_{in\,c} \rfloor}{|l_c - p|^2} \int_L \mathcal{V}(p, l)\, dA_l. \tag{2.7}$$

Here, the emission function e, BRDF function f_r, and the geometry factors are computed only for the center of the light source l_c, and assumed to be constant over the surface of the light source, allowing them to be moved outside the integral. Symbol $\omega_{in\,c}$ denotes the direction from p to the center of the light source. The approximation is reasonable if the light source is small compared to the distance to the surface being shaded, and the BRDF of the surface is not exceedingly reflective. When the light source is enlarged and moved closer to the receiving surface, the approximation error rises rapidly.

It is sometimes claimed (e.g. by Hasenfratz et al. [36]) that a shadow algorithm that computes the visibility integral $\int_L \mathcal{V}(p, l)\, dA_l$ correctly gives physically-based shadows. Even though this may hold sufficiently well in certain situations, it is evident that Equations 2.4 and 2.7 are different. Therefore, in this thesis we shall not grant these algorithms the luxury of being characterized as physically-based.

Even if we are happy with the approximation of Equation 2.7, the problem of evaluating the visibility integral still remains, and if Monte Carlo integration is used for approximating it, no significant gain in computation speed is obtained compared to using the correct Equation 2.6. Hence, various forms of trickery are used for approximating the visibility integral without sampling, and how this is exactly done depends on the algorithm in question. In addition to this approach, many techniques (e.g. Reeves et al. [66]) compute soft shadows by simply filtering hard shadow boundaries.

In this section, we shall cover only a couple of approximative soft shadow algorithms that have close resemblance to physically-based soft shadow algorithms. An extensive survey of real-time soft shadow algorithms is given by Hasenfratz et al. [36]. The introductory part in Arvo's PhD thesis [8] also contains thorough treatment of many approximative soft shadow algorithms.

Approximative real-time *soft shadow volumes* [4, 9, 10] are based on construction of penumbra wedges that are closed volumes containing the points where a single silhouette edge may cast penumbra. In these algorithms, the penumbra wedges are rasterized one by one using graphics hardware, and occlusion coverage is collected to the screen-space pixels. The inherent approximations in these techniques are:

- silhouette edges are computed only for the center of the light source, and assumed to hold from every point on the light source surface,
- depth complexity is assumed to be at most one for every ray between surface being shaded and the light source,
- BRDF and geometry terms are assumed to remain constant.

However, the most recent of these algorithms [9] supports light source whose emission is not constant over the light source surface. The depth complexity limitation is due to the fact the silhouette edges are processed one at a time, and the occlusion coverage is stored as a scalar value. This single scalar value is unable to contain information about which parts of the light source are occluded from the point being shaded, and thus areas that are covered multiple times are not handled correctly. To illustrate this, consider a point p from which only the top 50% of a square-shaped light source is visible. If there is only one surface that blocks the bottom half, the algorithm works correctly. However, if there are two such surfaces, the total occlusion is computed to be 100%, and the shadow will be completely black, even when the top half of the light source remains unblocked.

The most appealing feature of soft shadow volumes is that only the silhouette edges need to be taken into account, and the number of silhouette edges is typically a lot smaller than the number of triangles [58]. Soft shadow volumes have been used in physically-based shadow computation as well, and we shall return to this in the following section.

Cone tracing [5] approximates the visibility inside a circular cone, and this can be applied in soft shadow computations. For shading a point, a cone is constructed so that the apex is at the point being shaded, and the base covers the area light source. In practice, occlusion inside the cone has to be modeled heuristically, since exact clipping of the cone easily leads to unbearable fragmentation and complicated geometry because of the curved surface of the cone. *Pencil tracing* [68] operates on a set of rays that lie inside a cone-shaped beam (with possibly non-circular base) so that in certain situations only the central ray needs to be traced. Optical events such as transport, reflection and refraction of the pencil are modeled using simple matrix algebra, which enables error analysis related to the pencil-spread angle. Validating whether the destination of all rays in a pencil can be derived from the central ray is non-trivial.

2.2.3 Physically-Based Soft Shadows

In physically-based soft shadow algorithms, as defined in a somewhat strict sense in this thesis, the visibility is computed correctly and in a fashion that truly enables approximating the integral in Equation 2.4.

Beam tracing [40] constructs a polygonal beam between the point to be shaded and the light source. Occlusion inside the beam is correctly modeled by clipping the beam according to occluders. In highly tessellated scenes, the beam geometry quickly becomes prohibitively complex, and the benefits of using analytic geometrical representation instead of a set of rays is lost. Ghazanfarpour and Hasenfratz [32] introduce a variant that subdivides the beam recursively until the entire beam is either free of occluding geometry or blocked by a single triangle. Subdivision is also terminated when a specified subdivision limit is reached. Clipping to occluder geometry is avoided in this approach, but since connected surfaces cannot block the beam, the beam has to be subdivided all the way to the limit when it contains an edge of a single occluding primitive.

Using soft shadow volumes for rendering physically-based shadows was presented by Laine et al. [51]. In this algorithm, each potential silhouette edge is identified and a correspond-

ing penumbra wedge is constructed for each of these in a pre-processing phase. During rendering, the penumbra wedges containing the point p to be shaded are fetched from an acceleration structure, and the set of corresponding potential silhouette edges is pruned to contain only the actual silhouette edges as seen from p. Then the silhouette edges are projected onto the 2D surface of the light source, and the depth complexities of individual light samples is determined by a variant of a polygon filling algorithm. A number of limitations is imposed by the algorithm. Each planar light source polygon must be processed separately because of the projection of silhouette edges into 2D. Furthermore, the potential silhouette edge sets returned by the spatial acceleration structure quickly gets very conservative when the spatial size of the scene is increased. Nevertheless, the algorithm gives impressive speedups compared to traditional ray casting. The inner workings of the algorithm are discussed further in Chapter 3.

Another approach for computing soft shadows, *hierarchical penumbra casting* [50], takes a competely different stance on solving the visibility relations. Instead of processing every visibility relation between receiver points and light samples separately, the algorithm keeps the status of all visibility relations in memory, and processes each shadow-casting triangle of the scene separately. The blocked visibility relations are identified hierarchically for the triangle being processed, and the statuses of these visibility relations are updated. No restrictions on the positioning of the light samples or receiver points are imposed. The behavior of this algorithm is also analyzed in more detail in Chapter 3.

For completeness, we mention classical ray casting [73] here also, since casting separate shadow rays is still the most popular method for solving the values of the visibility function in Equation 2.6. Shadow caching optimization [34] is applicable equally well in computing soft shadows as in computing hard shadows.

Marks et al. [56] consider the common bounding volume of two quadrilateral patches, and note that shadow rays between these patches can only be blocked by the shadow casters inside the volume. Thus, when casting the shadow rays, all other objects in the scene can be ignored. This idea was extended to *shaft culling* [35, 33, 21] by considering the common bounding volume—called a *shaft*—between two bounding volumes. Finding the objects inside the shaft again allows the shadow rays to consider only those objects, leading to faster visibility queries. A method for bounding the penumbrae cast by objects that can be approximated by spheres has been presented by Formella and Łukaszewski [31]. Since conservative decomposition of arbitrary objects into spheres is a non-trivial task, this algorithm has quite limited applicability.

2.3 Visibility

As has been already emphasized several times, computing shadows is intimately related to computing visibility. Therefore, we shall take a look at some of the visibility-related previous work.

2.3.1 Pre-computed Visibility

In real-time applications, pre-computed visibility data is often used for accelerating the rendering. For static scenes, efficient occlusion culling is obtained through the use of pre-computed *potentially visible sets* [3], or PVSs, that are simply lists of objects known to be visible from a number of volumetric *view cells* in the scene. During run-time, the view cell containing the camera is identified, and only the objects in the PVS of the cell in question are rendered. Obviously, the occluders used when computing the PVSs cannot move or disappear during the display, since the pre-computed

		False visibility	
		No	**Yes**
False Occlusion	**No**	exact	conservative
	Yes	aggressive	approximative

Table 2.1 Classification of visibility algorithms according to Nirenstein et al. [62].

occlusion data cannot be updated.

When there are relatively few pathways for visibility, as is typical in indoor scenes, *portals* [55] can be used. Portals connect separate parts of the scene, and thus the entire scene needs to be represented as a cell-and-portal decomposition. During rendering, visibility only through portals is considered. An automatic portal placement algorithm is presented by Haumont et al. [37], although usually some amount of manual work is needed to remove furniture and other "noisy" geometry from distracting the algorithm. Since portals are not always applicable, e.g. in outdoor scenes, we will not consider this approach further.

2.3.2 Generic Visibility

Let us now consider visibility algorithms that need no a priori visibility information about the scene. First, we take a look at an abstract definition of a visibility algorithm, and then proceed by examining different classes of visibility algorithms and their applicability in rendering shadows.

We begin by defining a visibility algorithm as an *algorithm that returns a binary result indicating the existence of a line of sight between two geometrical entities*. Note that we do not yet consider whether the answer given by the visibility algorithm is always correct. The act of calling a visibility algorithm is commonly referred to as performing a *visibility query*. The two geometrical entities may be e.g. points, objects or polygons. For two points, we may solve the visibility by casting a shadow ray, making this a trivial case. As soon as we start operating on something else than points, things get much more complicated.[2]

Different types of visibility algorithms can be classified in terms of the correctness of the answers they give to visibility queries. Following the taxonomy of Cohen-Or et al. [18], as extended by Nirenstein et al. [62], we discriminate between *exact*, *aggressive*, *conservative* and *approximative* algorithms. The classification of a given visibility algorithm depends on whether it may report false positive and/or false negative results for visibility queries. Table 2.1 illustrates the classification. Exact algorithms always return the correct result, whereas conservative algorithms may *overestimate* the visibility, giving a false positive answer to a visibility query. Aggressive algorithms may *underestimate* the visibility and give a false negative answer to a visibility query, claiming that some visibility relation is blocked when it actually is not. Finally, approximative algorithms may err in either direction.

Generally, aggressive and approximative visibility algorithms are of little use in physically-based shadow rendering, in contrast to exact and conservative ones. To illustrate this, let us consider a situation where we would like to try solving all of the visibility relations in Equation 2.6 for a single p at once. We may execute a visibility query asking whether the entire surface of the light source L is occluded from point p. If a conservative or an exact visibility algorithm reports that L is entirely

[2] It is easy to see that volume-to-volume visibility can be trivially (although not necessarily efficiently) solved in terms of area-to-area visibility between the bounding surfaces of the volumes. This is based on the observation that every line of sight between two points in two non-overlapping volumes must pass through the bounding surfaces of both volumes. Hence, if the bounding surfaces are mutually occluded, so are the volumes.

Visibility algorithm	Query result: Visible	Query result: Occluded
Exact	continue	*break*
Conservative	continue	*break*
Aggressive	continue	continue
Approximative	continue	continue

Table 2.2 The action we need to take when a visibility algorithm reports visibility or occlusion between point p and the entire light source L. Only when exact or conservative visibility algorithm is used, and it reports occlusion, we can safely terminate the computation and trust that $\mathcal{V}(p, l) = 0$ for all point-to-point visibility relations between p and L.

occluded from p, we are done, since $\mathcal{V}(p, l_j) = 0$ for any l_j we might place on L. If the answer is that L is at least partially visible from p, there is a chance that some of our light samples l_j are located so that $\mathcal{V}(p, l_j) = 1$, and we must either cast shadow rays for each l_j or proceed in some other way. Now, consider a case where we apply an aggressive or an approximative visibility algorithm. If such an algorithm reports that L is entirely occluded from p, we cannot trust this answer because the algorithm may have underestimated the visibility and reported a false occlusion. Likewise, if the result is that L is not entirely occluded from p, we still have no information that would help us in avoiding any work; notice that this answer does not mean that all l_j would be visible from p. In conclusion, regardless of the answer given by an aggressive or an approximative visibility algorithm, it provides no useful information for rendering soft shadows. Table 2.2 summarizes whether we need to continue computation, based on the visibility algorithm and the answer it returns.

Another important basis for classification of visibility algorithms is whether they operate in object space or in screen space. Screen-space visibility algorithms are used for accelerating the rendering of complex scenes. These algorithms are typically conservative, since overestimating the visibility does no harm because of hidden surface removal. A simplifying factor in all screen-space algorithms is that one end of the visibility queries is always a point—the location of the camera—that stays the same for all queries during the rendering of a frame. Because of this feature of screen-space visibility algorithms they are generally not well suited for soft shadow computation or generic visibility computation. However, screen-space algorithms may be of great aid in e.g. accelerating the construction of shadow maps for rendering hard shadows. Because our focus is on soft shadows and generic visibility computation, we shall not consider screen-space visibility algorithms further in this thesis, but refer the reader to extensive surveys by Cohen-Or et al. [18] and Bittner and Wonka [14].

Numerous special-case algorithms have been developed for performing exact and conservative visibility queries in e.g. 2D, 2.5D and even more esoteric [53] cases. Compared to generic 3D cases, the lower dimensionality usually allows much simpler and faster solutions, and the topic is still under active research (see e.g. [12]). Although slightly dated, Wonka's PhD thesis [76] is still a good source of information, especially considering 2.5D visibility. These special-case algorithms are outside the scope of this thesis, as our focus is on general-purpose 3D visibility.

2.3.3 Exact Visibility Algorithms

The mathematical background for solving area-to-area visibility queries exactly was introduced by Pocchiola and Vegter [65] in the form of *visibility complex* in 2D, a geometrical data structure that stores all *visibility events*—possible changes in point-to-point visibility—between two surfaces. Later, the visibility complex was extended to 3D by Durand [24], who also introduced *visibility skeleton* [25], a much simpler graph structure that contains only the vertices and one-dimensional

edges of the visibility complex. The crucial ingredient in most visibility structures (although not all, see e.g. [42]) for generic 3D case is the six-dimensional projective dual space of the line space in 3D, the so-called *Plücker space* [64]. A thorough treatise on the nature of this dual space is beyond the scope of this thesis, but excellent presentations are available in the literature [71, 23, 13]. In the following, we will briefly consider the basics of how computations involving lines in 3D are translated to computations involving points and hyperplanes in the 6D dual space.

The six *Plücker coordinates* of a directed line ℓ are denoted $\pi_0 \dots \pi_5$. The Plücker coordinates of a line from point a to point b are given by the following equations [13]:

$$\begin{aligned} \pi_0 &= b_x - a_x & \pi_3 &= a_y b_z - a_z b_y \\ \pi_1 &= b_y - a_y & \pi_4 &= a_z b_x - a_x b_z \\ \pi_2 &= b_z - a_z & \pi_5 &= a_x b_y - a_y b_x \end{aligned} \tag{2.8}$$

Permutation of the Plücker coordinates gives a six-dimensional *Plücker hyperplane* with components $\omega_0 \dots \omega_5$. The sign of the dot product between a dual-space representation of line ℓ_1 and the Plücker hyperplane of line ℓ_2 tells whether line ℓ_1 passes line ℓ_2 in a clockwise or a counter-clockwise manner.

Let us now consider the visibility between two convex polygons, P_1 and P_2. The criterion that the visibility ray must intersect both these polygons is expressed by constructing a six-dimensional polytope that is bounded by all Plücker hyperplanes defined by the edges of the polygon. Now only the lines whose Plücker coordinates lie in this *initial polytope* have the quality that they pass all edges of both P_1 and P_2 in a similar fashion, which is equivalent to intersecting both polygons. The criterion of a ray intersecting a shadow caster between the polygons can be expressed in the same fashion, yielding the occlusion polytope in which the rays are occluded. By subtracting the occlusion polytopes from the initial polytope containing all valid lines, we can remove the sets of occluded lines from the sets of allowed lines. If the polytope vanishes, no line of visibility exists between P_1 and P_2, and we can conclude that P_1 and P_2 are mutually hidden. To be more precise, we must note that only some 6-tuples $\pi_0 \dots \pi_5$ represent actual lines in 3D and all these 6D points lie on a quadric four-dimensional surface embedded in the Plücker space, called the *Plücker hypersurface* or *Grassman manifold* or *Klein quadric* [23]. Therefore, it is enough that the intersection between the polytope and the Plücker hypersurface vanishes, in order to conclude mutual occlusion between two polygons.

Exact visibility algorithms that operate in Plücker space have been presented, among others, by Nirenstein et al. [62], Bittner [13], and more recently, by Haumont et al. [38]. All of these algorithms rely on relatively simple but numerically somewhat fragile geometrical computations, where small numerical inaccuracies can easily lead to wrong results. This issue is addressed by Duguet and Drettakis [22], who consider the visibility of thick rays instead of infinitesimally thin ones.

The price for obtaining an exact answer for a visibility query is often quite high, and the execution time for a single visibility query varies wildly depending on the situation (see e.g. the min/max timings in the results table of [38]). Furthermore, when computing soft shadows using Equation 2.6, we need to determine the visibility for a set of rays only. Using an exact visibility algorithm is usually too slow for performing an early-exit test to determine whether the entire light source L is hidden from a point p. As noted before, getting a conservative result for the visibility test does not lead to erroneous results because of the fallback solution we need to apply whenever visibility is reported. As conservative visibility algorithms are often much faster than exact ones, we shall investigate those next.

2.3.4 Conservative Visibility Algorithms

Conservative general-purpose 3D visibility algorithms are the most useful ones in context of soft shadow computation. Shadow caching [34] can be classified as a conservative point-to-point visibility algorithm, since it never underestimates the visibility, but in some cases is able to quickly determine that a shadow ray is occluded. For point-to-area and area-to-area visibility algorithms, multiple solutions have been presented, and we shall investigate those next.

Hierarchical blocker trees, introduced by Hinkenjann and Müller [42], construct a hierarchical, discretized representation of occlusion between two surfaces. In this algorithm, occluding primitives are treated as separate entities, and the aggregate occlusion of connected primitives is handled only by subdividing the hierarchical representation until a specified limit is reached, quite similarly to the beam tracer of Ghazanfarpour and Hasenfratz [32]. Therefore, an edge of a single blocking surface in a shaft leads to false visibility—a hole in the blocker tree—if the tree is constructed so that it can be guaranteed to give conservative instead of approximate visibility results.

The beam tracer of Ghazanfarpour and Hasenfratz [32] can be seen to contain a conservative visibility algorithm, since it detects if there is a single occluding primitive that blocks the entire beam. The algorithm is very conservative, since it is quite common that multiple connected primitives form a surface that blocks the beam, but this case is not detected. As noted earlier in Section 2.2.3, this leads to subdividing the beam to the maximum limit in such cases.

A more elaborate algorithm that finds shaft-blocking surfaces is presented by Navazo et al. [60]. Their algorithm detects most of the situations where a single surface blocks a shaft between two bounding volumes. The algorithm is rather convoluted, and it does not detect all cases where a single surface blocks the entire shaft (see Figure 4f in [60]).

A tremendously simpler algorithm that detects shaft-blocking surfaces was earlier presented by Bernardini et al. [11] as a part of conservative occlusion simplification system. It is unclear whether the algorithm they describe handles certain tricky occluders correctly—Section 4.6.4 contains an example of one such occluder.[3] The visibility test part of the incremental shaft subdivision algorithm presented in this thesis is essentially similar to the algorithm presented by Bernardini et al. [11] with certain modifications.

[3]The crucial question is how the test in line 20 of the pseudocode in Figure 6 in [11] works. The pseudocode reads "*if projection of boundary edges contains center of occluder* . . .", and it is not explained how this point-in-polygon test is performed. With suitable implementation, corresponding to testing whether the winding number of boundary edges with respect to occluder center is non-zero, the algorithm would function correctly. With another implementation, corresponding to the more standard even-odd point-in-polygon test, the algorithm would fail with certain occluders.

Chapter 3

Further Analysis

In this chapter, we shall discuss the general aspects of physically-based soft shadow algorithms. First, we address the question of evaluating the performance and quality of these algorithms. The concept of execution model in shadow algorithms is discussed next, especially focusing on the implications of the execution model on the computational complexity.

We then proceed by analyzing the operation of several existing physically-based soft shadow algorithms in detail, as well as the novel algorithm presented in this thesis. The analysis is entirely qualitative, and its purpose is to give the reader a general understanding of how the algorithms work, and which parameters may affect the performance of each of the algorithms, rather than give explicit quantitative performance analysis. This kind of quantitative analysis would be quite hard to do, because the geometrical situation generally affects the behavior of every shadow algorithm more than factors like the number of triangles or receiver points.

3.1 Performance and Quality

The two main factors in analyzing a soft shadow algorithm are *performance* and *quality*. The performance is usually thought of in terms of execution time and memory consumption in a given *rendering setup*, i.e. scene geometry, lighting setup, image resolution and so on. To be able to characterize the performance of a given algorithm, we must somehow choose the parameters in terms of which the characterizations are formulated. We choose the following parameters that are also used in previous research [50]:

- R: number of receiver points r_i,
- L: number of light samples l_j,
- T: number of triangles in the scene.

This parameterization makes several simplifications. First of all, considering only the number of receiver points (points in which the shadows are computed), instead of their positions as well, neglects their spatial arrangement completely. A shadow algorithm may be sensitive to the positioning of the receiver points, but such behavior is impossible to characterize in terms of R alone. The same applies to light samples: especially the spatial size of the light sources tends to affect the performance of physically-based soft shadow algorithms [51, 50].

Finally, the grossest of these simplifications is describing the scene geometry only in terms of triangle count. It is intuitively very clear that there must be "easy" and "difficult" spatial arrangements of shadow casters for practically every algorithm that deals with geometrical computations. For example, if all triangles are located between the shadow-receiving surfaces and the light sources, thus affecting the shadows, the situation is obviously more difficult than when the triangles are situated somewhere else where they do not contribute to the shadows in the image. The soft shadow volume algorithm [51], for example, is very sensitive to the number of silhouette edges of the shadow casters. Now, if the triangles in the scene form a smooth connected surface, the number of silhouette edges is tremendously smaller than when all triangles are disconnected and separate. Collapsing all information about the triangles into the single number T hides these kind of details.

In this thesis, we shall firmly steer away from the problem of obtaining the exact complexity formulas. This task would be generally extremely hard, and therefore we confine ourselves to merely observing whether the complexity is linear or sub-linear. Furthermore, we shall neglect every complexity measure that does not include every scene parameter: R and T for hard shadow algorithms, and R, L and T for soft shadow algorithms. This excludes all pre-processing stages that process only certain types of primitives, e.g. the triangles. Such pre-processing stages include building a triangle BSP or building a spatial acceleration hierarchy for receiver points.

3.1.1 Quality vs. Performance

When we are dealing with physically-based soft shadow algorithms, we must remember that all of them should give exactly identical results, ignoring the discrepancies that arise from numerical round-off errors. The situation is entirely different from approximative soft shadow algorithms, where both the quality and the performance vary, and it is generally much harder to evaluate quality than performance.

With physically-based soft shadows, the notion of quality is still valid. It is quite legitimate to say that images with better quality can be obtained by increasing the number of samples taken per pixel, leading to better anti-aliasing. Also, increasing the resolution can be seen as improving the quality. Considering the scene geometry, at least smooth surfaces appear more realistic when tessellated using a large number of triangles, as opposed to a small number, where individual triangles might remain visible. Finally, increasing the number of light samples decreases the approximation error in shadow computation (Equation 2.6).

Increasing any of the three factors R, L or T can thus be seen as increasing the quality of the resulting image. In general, this also leads to longer execution times and higher memory consumption for the shadow computation algorithm. The converse is also true: decreasing the quality by using smaller resolution, coarser geometry or fewer light samples leads to faster execution, as is natural to expect. Therefore, we can trade quality for performance, and vice versa. Since every algorithm produces the same quality in a given rendering setup, we end up in comparing only the performance. In other words, an algorithm that executes faster gives better quality for runtime ratio, i.e. bigger bang for buck, and can thus be deemed to be better in every sense than a slower algorithm.

If the computational complexity of an algorithm is sub-linear with respect to a parameter, it behaves nicely when the quality is improved by increasing the value of the parameter in question. If the execution time of an algorithm depends on, say, R according to $O(\mathsf{R}^{0.5})$, quadrupling the resolution from 1024×768 to 2048×1536 only (approximately) doubles the execution time. But this works the other way around, too. Decreasing the resolution to one-fourth of the original, 512×384, only halves the execution time. Therefore, it is relatively slower to render e.g. a preview image using a small resolution, compared to a linear-time algorithm with execution time complexity $O(\mathsf{R})$. Now, let us

for a moment imagine that we can choose between two algorithms with execution time complexities of $O(\mathsf{R}^{0.5})$ and $O(\mathsf{R})$, whose actual execution time, in seconds, is about the same in the 1024×768 resolution, in a rendering setup we are working on. Which algorithm is better? It is obvious that the $O(\mathsf{R}^{0.5})$ algorithm wins if we want to increase the resolution or the antialiasing quality in the future. But, if it is more likely that the resolution requirements are going to drop, the $O(\mathsf{R})$ algorithm would be a better choice. This simple example illustrates that the asymptotical $O(\cdots)$ notation does not necessarily indicate the actual applicability of an algorithm, unless we make the bold assumption that the quality requirements will increase without limit.

3.2 Execution Model and Its Implications on Complexity

Both soft and hard shadow algorithms differ from each other in their *execution model*: what data the algorithm needs as immutable a priori information for pre-processing purposes, and what data is looped over while executing the algorithm. The execution model profoundly affects the computational complexity of an algorithm, both for execution time and memory consumption.

3.2.1 Triangles

If an algorithm needs all triangles in the scene to be present in the pre-processing stage, it is able to build a spatial acceleration structure for them, extract the silhouette edges, or perform other geometrical tasks that require simultaneous knowledge of all triangles in the scene. An acceleration structure enables searches to the triangles in sub-linear time with respect to T, and being able to use silhouette edges instead of triangles themselves also reduces the execution time to sub-linear with respect to T. But there are downsides in this approach as well.

When we speak of gathering all triangles in the scene, we are essentially referring to *full scene capture*, which is generally avoided in hardware-based systems because of unbounded memory usage. Collecting the triangles in a pre-processing stage implies that the triangles (or at least the silhouette edges) need to be stored in memory, and while this is not a problem in off-line rendering, it certainly is a problem if the algorithm is intended to run in a predetermined amount of memory, which is usually the case in hardware.

On the other hand, if the algorithm works so that it *loops over* the triangles one by one, its execution time is inevitably linear with respect to T. The good side about this approach is that the memory consumption of the algorithm is independent of T, indicating better feasibility for hardware implementation. In the context of both hard and soft shadow algorithms we analyze ones that employ the approach of collecting the triangles in pre-processing stage, and ones that loop over the triangles.

3.2.2 Receiver Points

Collecting the receiver points at a pre-processing stage enables a shadow algorithm to execute in sub-linear time with respect to R. This is a compelling property, since it essentially means that increasing the resolution of the image affects the computational cost sub-linearly. This is also somewhat controversial approach, since it imposes serious restrictions on the execution model of the algorithm. The traditional shadow ray casting, for example, does not require receiver points to be gathered beforehand, and is therefore easy to apply in various contexts.

Collecting the receiver points requires that, in principle, the image is first rendered without shadows,

and only after that, the shadow computation may take place. This means that the user cannot e.g. see the final image being formed, which may be a serious drawback in production rendering. Screen-space adaptive anti-aliasing is also hard to implement, requiring that we first compute the image with shadows using the initial sampling rate, then analyze the image for apparent discontinuities, and finally perform another pass where extra samples are deposited in areas that require anti-aliasing.

Despite these inconveniences, a truly powerful soft shadow algorithm should be able to exploit the coherence of shadows between nearby receiver points, and this requires that the receiver points are collected beforehand. In the remainder of this chapter, we see both hard and soft shadow algorithms that take this approach.

3.2.3 Light Samples

Even though light samples are conceptually similar to receiver points in the sense of evaluating the visibility function, it is usually not a problem if the light samples need to be known in a pre-processing stage. The coherence of shadows caused by nearby light samples is usually high, and the potential drawbacks caused by using predetermined light sample configurations are often compensated by significantly better execution time complexity.

With light samples, there is also an intermediate level of representation, employed by e.g. the soft shadow volume algorithm by Laine et al. [51]. This algorithm requires that the light polygons are known in a pre-processing stage, but the light samples themselves can be dynamically placed on the light source when executing a shadow query, i.e. when the visibility between a single receiver point and a set of light samples is computed. This is a nice property, since it allows generating custom importance sampling-based light sample pattern for every individual shadow query.

Using the same set of light samples for computing the shadows to every receiver point has undesirable consequences. In this case, the soft shadows would be in fact just a combination of multiple hard shadows, and unless an excessively high number of light samples is used, this causes *banding artifacts* in the shadows. The banding artifacts can be easily removed by choosing a different set of light samples for each receiver point, using the same random distribution every time. This converts the banding to much less distracting noise. Soft shadow algorithms that require all light samples to be determined before the shadow computation must still be somehow able to use different light sample patterns for individual receiver points. A possible way of doing this is to require a number of light sample patterns to be specified, and to choose randomly among them for each receiver point. This approach is used by the hierarchical penumbra casting algorithm [50].

3.2.4 The Complexity Cube

We now introduce the *complexity cube* of soft shadow algorithms that is useful for mentally situating the different algorithms in the context of execution time complexity. The execution time of a soft shadow algorithm is either linear or sub-linear with respect to the number of receiver points R, light samples L, and triangles T. We thus have eight combinations in total, considering the execution time complexity with respect to three parameters. These eight combinations can be thought of as the vertices of a cube, with axes corresponding to complexity with respect to one parameter.

Figure 3.1a illustrates the complexity cube of soft shadow algorithms. As we can see, every corner of the cube except one is occupied by a shadow algorithm, and we shall analyze these algorithms in more detail in the following sections. The red corners indicate algorithms that have already been presented in the literature, or can be obtained by simple modifications to those algorithms. The

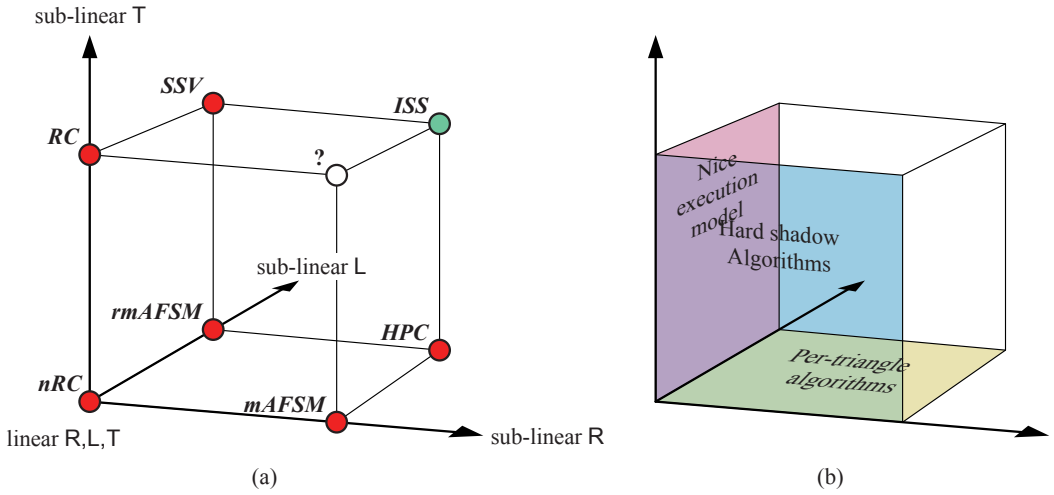


Figure 3.1 The complexity cube of soft shadow algorithms. (a) Shadow algorithms placed in their respective corners of the cube. Abbreviations: *nRC* = naive ray casting, *RC* = ray casting, *mAFSM* = modified alias-free shadow maps, *rmAFSM* = reverse modified alias-free shadow maps, *SSV* = soft shadow volumes, *HPC* = hierarchical penumbra casting, *ISS* = incremental shaft subdivision. (b) Faces of the cube containing algorithms that are linear with respect to one parameter correspond to certain characteristics of the algorithms.

green corner indicates the slot filled by the incremental shaft subdivision algorithm (ISS) that will be presented in Chapter 4 of this thesis.

Based on the discussion in the previous section, we may identify certain characteristics of the algorithms with the faces of the cube. Figure 3.1b illustrates these dependencies. For instance, algorithms with linear complexity with respect to R do not need the receiver points to be collected in the preprocessing stage, and thus exhibit the nice execution model where shadow queries can be issued on the fly. Algorithms with linear complexity with respect to L are all actually hard shadow algorithms that can be executed multiple times, once per each individual light sample. Finally, algorithms on the bottom face of the cube process the triangles linearly, and their memory consumption does not depend on T.

3.3 Analysis of Shadow Algorithms

In this section, we briefly discuss each of the algorithms that occupy the corners of the complexity cube. Both hard and soft shadow algorithms are included, since soft shadow algorithms that operate on each light sample separately can be derived trivially from hard shadow algorithms by looping over the individual light samples. We make the assumption that every triangle casts shadows to avoid cluttering the pseudocodes with unnecessary tests. A single light source is also assumed, unless explicitly stated otherwise, as most of the algorithms can naturally handle only one light source. With these shadow algorithms, multiple light sources must be handled by executing the algorithm for each light source separately. To store the statuses of the visibility relations, we imagine having an array $V[i, j]$ where each element may have either value OCCLUDED or VISIBLE, corresponding to the visibility between receiver point r_i and light sample l_j. This is done solely to unify the notation,

```
NAIVE RAY CASTER
1  for each receiver point r_i do
2    for each light sample l_j do
3      V[i,j] ← VISIBLE
4      for each triangle t do
5        if t intersects ray r_i → l_j then V[i,j] ← OCCLUDED; break
6      end for
7    end for
8  end for
```

Figure 3.2 Pseudocode for the naive ray caster algorithm.

```
RAY CASTER
1  collect all triangles in the scene and build a spatial acceleration structure
2  for each receiver point r_i do
3    for each light sample l_j do
4      if IS-RAY-BLOCKED(r_i → l_j) then
5        V[i,j] ← OCCLUDED
6      else
7        V[i,j] ← VISIBLE
8      end if
9    end for
10 end for
```

Figure 3.3 Pseudocode for the ray caster algorithm.

and in practice many of the algorithms could be used so that the statuses of all visibility relations would not need to be stored at any point of the computation.

3.3.1 Ray Casting

The most trivial and simple-minded shadow algorithm is here dubbed *naive ray caster* (nRC), and its operation is easiest to describe using the pseudocode in Figure 3.2. The algorithm simply checks every {receiver point , light sample , triangle} triplet for intersection, and notes when the triangle blocks the ray between the receiver point and the light sample. This results in $O(\mathsf{RLT})$ complexity, even though the innermost loop can be exited if an intersection is found. The algorithm occupies the worst corner of the complexity cube, and is included here only for the sake of completeness. In practice, it is too slow to be useful in any realistic situation.

The key to making ray casting practical is to construct an acceleration structure for quickly determining if a ray between a receiver point and a light sample is blocked by a triangle. Popular acceleration structures include axis-aligned BSP and regular grid. Constructing the acceleration structure must be done before any rays can be cast, and it requires at least linear amount of memory with respect to T. To summarize the operation, we give the pseudocode in Figure 3.3.

We will not go into details on how the subroutine IS-RAY-BLOCKED is implemented, since it depends on the particular acceleration structure used. It suffices here to note that a single call can

be performed in sub-linear time with respect to T. In fact, with common acceleration structures the average-case execution time is at most logarithmic [70]. With two nested loops in which a sub-linear time call is made, we end up with execution time complexity that is linear with respect to R and L, and sub-linear with respect to T. We call this algorithm simply *ray casting* (RC).

3.3.2 Alias-Free Shadow Maps

The motivation for alias-free shadow maps [2] (AFSM) was to eliminate the resolution and bias problems in traditional shadow maps used for rendering hard shadows. To achieve this, the receiver points in the scene are first gathered, transformed into the view frustum of the light source, and projected to the 2D plane corresponding to the image plane of the traditional shadow map. Then, an axis-aligned BSP acceleration structure is constructed for these points. Triangles of the scene are processed one at a time so that the points that the 2D projection of the triangle overlaps are identified, and using depth comparison, marked as occluded when appropriate. Unlike with traditional shadow maps, there is a one-to-one mapping between receiver points and shadow map sampling points, and therefore it is not necessary to render the depth values of the occluding triangles for the sampling points; instead, we can store the status of the visibility relation directly. As an optimization, it is easy to maintain a bit per BSP node telling if all receiver points under the node have already been marked as occluded. In such cases, the BSP traversal can always be terminated, since the status of the receiver points can never change from occluded to visible. Computing soft shadows with the AFSM algorithm requires that we loop over every light sample individually. Removing the banding artifacts caused by using the same set of light samples for every receiver point is unfortunately not possible with the AFSM algorithm.

This time, we use a slightly more detailed pseudocode to illustrate the algorithm, see Figure 3.4. Analysis of the complexity of the algorithm is difficult because of the 2D projection under which the BSP is constructed. Indeed, it cannot be easily said how it affects the quality of the acceleration structure if one dimension is flattened down, and only two are considered when constructing the structure and performing the hierarchical searches. Because of this, we modify the AFSM algorithm slightly so that it constructs the axis-aligned BSP in three dimensions, taking not only the x and y coordinates of the projected receiver points into account, but also the projected z coordinates. The only changes to the pseudocode involve modifying the test on Line 11 in the PROCESS-TRIANGLE subroutine (Figure 3.4) that checks if the triangle may affect the occlusion of the receiver points under the node. Instead of just checking for 2D overlap, we need to verify that the bounding box of the node—now having bounds for x, y and z coordinates—is at least partially behind the triangle being processed. We shall not give the pseudocode for this *modified alias-free shadow maps* (mAFSM) algorithm, since it would be almost the same as the one given in Figure 3.4.

Now that we have a full 3D hierarchy for the receiver points, we may analyze the complexity of the algorithm. We begin by noting that a *range query* in a kd-tree can be executed in time $O(n^{1-1/d}+k)$, where n is the number of points in the tree, d is the dimension of the tree, and k is the number of points reported by the query [20]. A range query corresponds roughly to what we do when processing the triangle hierarchically in the receiver point tree, and our axis-aligned BSP should be at least as good an acceleration structure as a kd-tree, so we can take the complexity of finding the receiver points affected by a single triangle to be $O(\mathsf{R}^{2/3} + k)$. Now, if we take into account that k, the number of points under the triangle, generally increases linearly with respect to R, it seems that the execution time of the triangle processing subroutine is linearly dependent on R. But this is actually not the case, due to the effect of the optimization bits used for signaling when all receiver points under a node are occluded.

If we did not want to enumerate the affected receiver points, but only find the nodes that are influ-

```
ALIAS-FREE SHADOW MAPS
 1  collect all receiver points
 2  for each light sample l_j do
 3    project receiver points r_i into the light space of l_j and construct a 2D BSP for them
 4    for each receiver point r_i do V[i, j] ← VISIBLE
 5    for each triangle t do
 6      t_l ← projection of t in the light space of l_j
 7      PROCESS-TRIANGLE(root_node, t_l, l_j)
 8    end for
 9  end for

procedure PROCESS-TRIANGLE(BSP node n, projected triangle t, light sample l_j)
10  if n.all_occluded then return
11  if t does not overlap n.bounding_rectangle then return
12  if n is a leaf node then
13    for each receiver point r_i in n.points do
14      if (r_i.x, r_i.y) is inside t and r_i.z > depth of t at (r_i.x, r_i.y) then V[i, j] ← OCCLUDED
15    end for
16    if all r_i in n.points have V[i, j] = OCCLUDED then n.all_occluded ← TRUE
17  else
18    PROCESS-TRIANGLE(n.left, t, l_j)
19    PROCESS-TRIANGLE(n.right, t, l_j)
20    n.all_occluded ← n.left.all_occluded and n.right.all_occluded
21  end if
```

Figure 3.4 Pseudocode for rendering soft shadows with the alias-free shadow maps algorithm.

enced by the triangle, this could be done in $O(\mathsf{R}^{2/3})$ time. After this node enumeration, we could mark the occluded nodes using e.g. one bit per node, and later return to propagate these occlusion bits down to the receiver points. This final propagation pass needs to be done only once, and its complexity is thus independent of the number of triangles. With this kind of scheme, we would therefore obtain triangle processing time that is sub-linear with respect to R.

Beginning with this idea, we may now consider the possibility of removing the final propagation pass by performing the propagation immediately when a node becomes occluded, and tagging the node as having already been propagated. Since the propagation has to be performed once anyway, it does not make any difference if we do it immediately when the need arises, as long as we are careful to avoid doing it more than once. Hence, the execution time is unaffected by this modification. Comparing this approach with the one used in the AFSM algorithm, this is almost the same effect that the all-occluded optimization bits achieve. The optimization bits are in fact even more powerful, since by using them it is possible to detect when all receiver points under a node have been occluded by *separate* triangles. Therefore, the algorithm with all-occluded optimization bits operates almost in the same fashion as a node-enumerating algorithm that propagates the occlusion in a post-process phase. Since the node-enumerating variant has triangle processing time that is sub-linear with respect to R, the same holds for the modified AFSM algorithm as well. Consequently, the execution time for the mAFSM algorithm is linear with respect to L and T, and sub-linear with respect to R.

We may also reverse the role of receiver points and light samples, ending up with *reverse modified alias-free shadow maps* (rmAFSM) algorithm that occupies one corner of the complexity cube. The execution time would then be linear with respect to R and T, and sub-linear with respect to L. Using

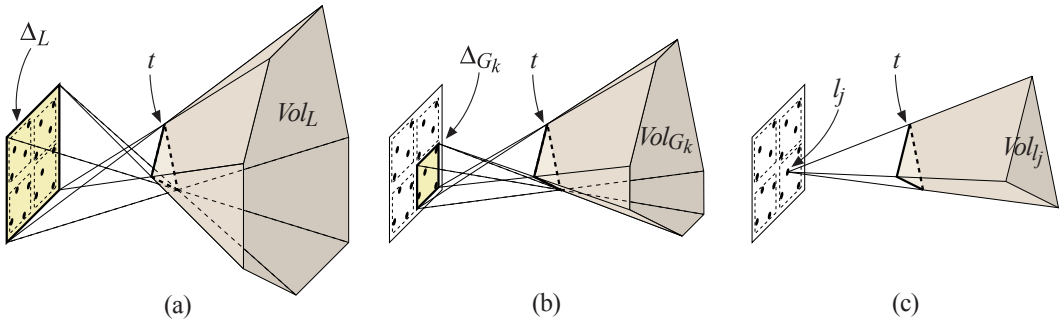


Figure 3.5 The volumes associated with the three levels of the light sample hierarchy. In this illustration, the light source consists of 16 light samples grouped into four light sample groups. (a) The main penumbra volume Vol_L is defined by the bounding polygon Δ_L of the light source and the blocker triangle t. (b) For each light sample group G_k, a penumbra volume Vol_{G_k} is constructed based on the bounding polygon Δ_{G_k} of the light sample group and the blocker triangle t. (c) For each light sample l_j, a hard shadow volume Vol_{l_j} is constructed, based on the location of l_j and the blocker triangle t. Figure and caption adapted from [50].

a different light sample pattern for each receiver point can be done trivially, unlike in AFSM or mAFSM.

Using a 3D BSP instead of a 2D one is required to pull off the analysis above, but in practice, a 2D BSP often performs better. Ignoring the z coordinate when building the hierarchy does no harm except when the receiver points form several depth layers as seen from the light source. Ignoring such cases, the 2D BSP has better range query complexity of $O(\mathsf{R}^{1/2})$, which makes it quite efficient in practice.

3.3.3 Hierarchical Penumbra Casting

We have now investigated the algorithms that occupy the corners of the complexity cube where the execution time is sub-linear with respect to a single parameter. Hierarchical penumbra casting [50] (HPC) is the first algorithm under examination that is sub-linear with respect to two parameters.

In the heart of the hierarchical penumbra casting is a three-level hierarchy of the light samples of a single light source. This hierarchy, and its corresponding *penumbra volumes* are illustrated in Figure 3.5. A penumbra volume consists of the separating planes between 1) the bounding polygon of the light source or a light sample group, and 2) the triangle being processed. The penumbra volume contains every point in space that may be at least partially shadowed by the triangle, and therefore gives us the opportunity to efficiently determine if a set of receiver points is certainly outside the influence of a triangle. In a sense, the penumbra volumes play the same role in the HPC algorithm as the triangles themselves did in the AFSM algorithm, since they are used for guiding the traversal in the receiver point hierarchy.

Much like the modified alias-free shadow maps algorithm, the HPC algorithm also constructs a 3D hierarchy of the receiver points, but in object space and without any projection. With the two hierarchies—one for the light samples and one for the receiver points—the algorithm then proceeds by processing one triangle at a time and marking the visibility relations it occludes. The pseudocode of the basic HPC algorithm is given in Figure 3.6. We will not discuss the many optimizations and

```
HIERARCHICAL PENUMBRA CASTING
 1  collect all receiver points r_i and build a 3D BSP for them
 2  collect all light samples of a light source L and form the light sample groups G_k
 3  for every r_i, l_j do V[i,j] ← VISIBLE
 4  for each triangle t do
 5    Vol_L ← MAKE-PENUMBRA-VOLUME(Δ_L, t)
 6    for each light sample group G_k do Vol_{G_k} ← MAKE-PENUMBRA-VOLUME(Δ_{G_k}, t)
 7    for each light sample l_j do Vol_{l_j} ← MAKE-SHADOW-VOLUME(l_j, t)
 8    active_groups ← every G_k
 9    PROCESS-TRIANGLE(receiver_hierarchy_root_node, active_groups)
10  end for

procedure PROCESS-TRIANGLE(receiver BSP node n, active_groups)
11  if not INTERSECTS(n.bounding_box, Vol_L) then return
12  active_groups' ← ∅
13  for each G_k in active_groups do
14    if INTERSECTS(n.bounding_box, Vol_{G_k}) then add G_k into active_groups'
15  end for
16  if active_groups' = ∅ then return
17  if n is not a leaf node then
18    PROCESS-TRIANGLE(n.left, active_groups')
19    PROCESS-TRIANGLE(n.right, active_groups')
20  else
21    for each r_i in n.points do
22      if not POINT-IN-VOLUME(r_i, Vol_L) then continue
23      for each G_k in active_groups' do
24        if not POINT-IN-VOLUME(r_i, Vol_{G_k}) then continue
25        for each l_j in G_k do
26          if POINT-IN-VOLUME(r_i, Vol_{l_j}) then V[i,j] ← OCCLUDED
27        end for
28      end for
29    end for
30  end if
```

Figure 3.6 Pseudocode for the hierarchical penumbra casting algorithm, adapted from [50].

extensions presented in the original paper, since that would lead us quite far from our main focus.

The traversal of the receiver point hierarchy is slightly more complicated than in the AFSM algorithm, but not much. Hierarchical penumbra casting is, as stated in the original paper [50], an extension of the alias-free shadow maps, so it is not too surprising that similarities can be found. Firstly, Line 11 of the pseudocode tests if a receiver hierarchy node n is completely outside the main penumbra volume defined by the entire light source and the triangle t being processed. If so, the traversal is terminated, since no receiver point under n can be shadowed by the triangle. Then, the set of *active light sample groups* is refined in Lines 12–15 of the pseudocode. This set contains the light sample groups whose penumbra intersects with the BSP node being processed. In the refinement, the node may be found to be outside the penumbra volume of some light sample group, which indicates that the triangle cannot block any visibility relations between the light samples in that group and the receiver points under the BSP node. In the leaf nodes of the receiver point hierarchy, the individual receiver points are tested against the main penumbra volumes, the group penumbra volumes, and

finally, the hard shadow volumes constructed for the individual light samples.

The HPC algorithm is able to avoid the banding artifacts that would be caused by using the same light sample pattern for all receiver points. The trick is to generate a number of different light sample patterns, and to consistently use one of these patterns when processing a single receiver point. In practice, relatively few light sample patterns are needed for converting the banding to unstructured noise. To ensure that the tests against penumbra volumes are still conservatively correct, the bounding polygons for the light sample groups must be constructed so that they enclose the corresponding light samples from all light sample patterns. In the final shadow test (Line 26), the hard shadow volume from the light sample pattern for the receiver point is used for determining the occlusion.

The main defect in the HPC algorithm is that the light sample hierarchy is processed in an ad-hoc fashion, every level treated in a slightly different way. As an example, a list of active groups is maintained for the light sample groups (middle layer), but this is not done for the individual light samples for performance reasons. Furthermore, it is hard to justify the use of a fixed three-level hierarchy, since, in theory, a full hierarchy with $(\log \mathsf{L})$ levels should be at least asymptotically better. Nevertheless, the hierarchical approach yields sub-linear triangle processing time with respect to L, and this is also demonstrated in the experimental results section of the original paper [50]. Because the receiver points are processed hierarchically, sub-linear triangle processing time with respect to R is achieved as well.[1] The triangles are processed one at a time, so the total execution time complexity is linear with respect to T, and sub-linear with respect to R and L.

3.3.4 Soft Shadow Volumes

The last algorithm we analyze in this chapter is the soft shadow volume (SSV) algorithm [51] that is able to perform efficient shadow queries on the fly in off-line renderers. Its main advantage is that it has a nice execution model that does not require collecting the receiver points beforehand, making operations such as adaptive anti-aliasing or programmable shaders easier to implement. But, as we have already pointed out, this inevitably comes with the cost of linear execution time with respect to the number of receiver points.

Pseudocode of the soft shadow volumes algorithm is given in Figure 3.7. In a pre-processing phase, all edges that are silhouettes from some point on the light source L are identified, penumbra wedges are constructed for them, and the wedges are placed into an acceleration hierarchy for fast access during the shadow queries (Lines 1–5). For each receiver point, we loop over all silhouette edges between the receiver point and the light source, updating depth complexity counters for each light sample. The potential silhouette edges fetched from the acceleration structure are validated to ensure that they are actual silhouette edges from the receiver point being processed. Edges that pass the validation test are projected onto the surface of the light source and the depth complexity counter array is updated using appropriate integration rules (Lines 9–14). For details of the integration operation, we refer the reader to the original paper [51]. When all silhouette edges have been processed, the values in the depth complexity counters indicate the *relative* depth complexities between light samples, but the actual depth complexities remain unknown as only the silhouettes between the receiver point and the light source have been accessed. Because of this, a single reference shadow ray is cast to a light sample with smallest relative depth complexity. If this ray is blocked, all light samples are occluded. Otherwise, the light samples that share the smallest relative depth complexity are visible while all other light samples are occluded (Lines 16–19).

[1] This follows from a similar argument as was made with modified alias-free shadow maps algorithm. The all-occluded bits are necessary, but they are included in the actual HPC algorithm that contains the optimizations we have not discussed here.

```
SOFT SHADOW VOLUMES
 1  for each edge e in the scene do
 2    if e is not a potential silhouette edge from L then continue
 3    w ← CONSTRUCT-PENUMBRA-WEDGE(e)
 4    insert w into a spatial acceleration structure
 5  end for
 6  for each receiver point r_i do
 7    for each l_j do D[j] ← 0
 8    W ← find potentially influencing wedges for r_i from the acceleration structure
 9    for each wedge w in W do
10      e ← edge corresponding to w
11      if e is not a silhouette edge from r_i then continue
12      project e onto the surface of the light source
13      integrate e into the array D using proper integration rules
14    end for
15    for each l_j do V[i, j] ← OCCLUDED
16    j_min ← arg min_j D[j]
17    if not SHADOW-RAY-BLOCKED(r_i, l_{j_min}) then
18      for each l_j do if D[j] = D[j_min] then V[i, j] ← VISIBLE
19    end if
20  end for
```

Figure 3.7 Pseudocode for the soft shadow volumes algorithm [51].

The SSV algorithm requires that the light source geometry is known in advance, but it has the remarkable property that the positions of the light samples, and even the number of light samples, can be set freely for each receiver point individually. This makes it possible to avoid banding artifacts and to use importance sampling-based light sample distributions.

As the SSV algorithm operates on the silhouette edges of the shadow casters, it performs the shadow query for a single receiver point in sub-linear time with respect to T. The sub-linearity with respect to L comes from an optimization in the integration phase. The idea is to avoid looping over every light sample for every silhouette edge in the integration phase (Line 13), but we will not go into details here. Obviously, as every receiver point is processed separately, the total execution time is linear with respect to R.

3.3.5 Incremental Shaft Subdivision

The incremental shaft subdivision (ISS) algorithm that is to be presented in the next chapter of this thesis is sub-linear with respect to all R, L and T, and it occupies the seemingly most prestigious position in the complexity cube. But this corner is also outside all the "nice" sides shown in Figure 3.1b, meaning that it has an inconvenient execution model, it needs to capture the entire scene, and it needs all light samples to be defined beforehand. Furthermore, as the experimental tests in Chapter 5 show, the better execution time complexity does not come cheap; the constant factor in the execution time is high, and benefits are obtained only in situations where the sub-linear operation comes especially useful.

3.3.6 The Empty Corner

Unlike with the modified alias-free shadow maps algorithm, it is impossible to swap the roles of receiver points and light samples in the SSV algorithm in order to obtain an algorithm that would fill the empty corner of the complexity cube. This is because the SSV algorithm requires that the light source is planar, and while this is usually an acceptable limitation, it would be absurd to require that the receiver points lie on a plane. The corner thus remains empty, but we can easily characterize the algorithm that would fit there; the missing algorithm would need to be a hard shadow algorithm that is sub-linear both with respect to R and T.

Chapter 4

The Incremental Shaft Subdivision Algorithm

This chapter presents the novel incremental shaft subdivision (ISS) algorithm for computing soft shadows. As will turn out, the algorithm provides a generic method for computing all visibility relations between two point sets, with one addition specific to shadow computation, namely for removing the banding artifacts. There are many possible extensions that would be very useful in performing efficient shadow computations, related to e.g. importance sampling and back-face culling, but they are left on the level on discussion, and treated in Chapter 6. Proper handling of these matters would require further research.

We will try to keep the presentation on an appropriate level, investigating the details of the algorithm thoroughly, but avoiding going into implementation-specific issues. To keep the amount of pseudocode reasonably small, we often directly compute the results that would in reality be computed in parts. For instance, detecting whether an edge intersects a shaft is done directly, whereas in reality, we would first determine for each vertex if it is inside the shaft, and then use this information for avoiding the more costly edge versus shaft intersection tests. The importance of proper implementation techniques is discussed briefly at the end of this chapter, where a couple of generic hints are given. First, we shall give an overview of the algorithm, and then scrutinize the details of each individual part in subsequent sections.

The figures in this chapter are drawn in 3D when absolutely necessary, but most often they are 2D analogues of respective 3D situations. This is because, in most cases, a 2D illustration is less ambiguous than what a 3D illustration would be; there is no chance of getting the wrong impression of e.g. the relative depths of geometrical primitives. The primitives generally look different in 2D than in 3D, and these differences may seem counterintuitive at first: triangles are drawn as line segments, whereas edges that connect the triangles are represented by points. For a reader that is unaccustomed to these kind of illustrations, the easiest point of view is to consider the 2D illustrations to be cross-sections of 3D cases.

4.1 Overview of the Algorithm

As a preliminary to running the ISS algorithm, we prepare the mesh to remove certain common degeneracies and to find the connectivity information between triangles (Section 4.2). The next stage

is to gather all light samples and receiver points, and to construct 3D bounding volume hierarchies for them (Section 4.3). With this information, we are ready to compute the shadows.

The shadow computation routine is recursive, and it takes as input one node of the receiver point tree and one node of the light sample tree. From now on, we shall call these trees simply *receiver tree* and *light tree*, and the associated nodes will be called *receiver node* and *light node*. The routine constructs a *shaft* between the bounding boxes of the receiver and light nodes, and tests if the shaft is completely occluded or completely visible, considering only the geometry inside the shaft (Section 4.4). This *blocking test* is done using an algorithm similar to the one presented by Bernardini et al. [11] (Section 4.6). Now, if the entire shaft is blocked, we know that all visibility relations between the receiver points under the receiver node and the light samples under the light node are occluded, and the recursion can be terminated. Conversely, if the shaft is empty, we may in certain cases conclude that all visibility relations are visible, but this is not always possible. If neither case holds, we construct two new sub-shafts (Section 4.7), by advancing to the children of either the receiver node or the light node. The choice between splitting the receiver node or the light node is made using a heuristic (Section 4.9).

If the recursion reaches the leaf nodes of both receiver and light trees, the visibility relations between the receiver points and the light samples in these nodes are solved by casting shadow rays. It is possible to use a special kind of ray caster (Section 4.10) that only considers the geometry inside the shaft under consideration, but it is not always faster than the standard ray caster, as was found in the experimental performance measurements (Chapter 5).

This description does not convey the most important contribution of the ISS algorithm, namely that it is *incremental* in constructing the sub-shafts from the parent shafts. This means that when the sub-shafts are constructed, the geometry inside them is filtered from the parent shaft, so that the entire scene does not need to be considered. An additional important contribution in the algorithm is that the geometry inside the shaft is not represented using the triangles of the blocking geometry, but using only the relevant edges that may form silhouettes between the endpoints of the shaft. As was briefly noted in Section 1.8, only the silhouette edges are important for computing the visibility, and thus we may safely dispose of every other edge inside the shaft. This is extremely advantageous, since the number of silhouette edges is typically much smaller than the number of all edges in a mesh, and most importantly, it is sub-linear with respect to the number of triangles. The shift from triangle-based representation to an edge-based representation is not trivial, and handling the representation of the geometry and filtering it down to sub-shafts constitutes the bulk of the algorithm.

4.2 Scene Storage and Preparation

The ISS algorithm requires that the connectivity between shadow-casting triangles in the scene is solved. For this purpose, we need to perform the following mesh processing steps before starting the shadow computation:

1. weld vertices, i.e. combine vertices with the same 3D position;
2. discard degenerate triangles that have two same vertices;
3. compute neighbors for every triangle;
4. assign an unique *edge label* for every edge.

We give the datatype definitions used for storing the scene as well as a pseudocode example for the vertex welding routine in Figure 4.1. We consider all blocking geometry to form a single mesh,

```
typedef Triangle
{
    Vector3i        vertices        // three vertex indices
    Vector3i        neighbors       // neighbor triangle indices for each edge
    Vector3i        edge_labels     // edge label for each edge
}

typedef Mesh
{
    Array⟨Vector3f⟩  vertex_pos      // vertex positions
    Array⟨Triangle⟩  triangles       // triangles
    Array⟨bool⟩      is_double_edge  // double edge flag per edge label
}

procedure WELD-VERTICES(Mesh* mesh)
 1  Array⟨Vector3f⟩ new_vertices
 2  Map⟨Vector3f → int⟩ vmap
 3  for each triangle t in mesh.triangles do
 4    for each i in {0, 1, 2} do
 5      Vector3f v ← mesh.vertex_pos[t.vertices[i]]
 6      if vmap contains v then
 7        t.vertices[i] ← vmap.get(v)          // re-use vertex
 8      else
 9        int vidx ← new_vertices.size         // construct new vertex
10        add v at the end of new_vertices
11        insert (v → vidx) into vmap
12        t.vertices[i] ← vidx                 // set vertex index in triangle
13      end if
14    end for
15  end for
16  mesh.vertices ← new_vertices
```

Figure 4.1 Mesh datatypes and a pseudocode example for the vertex welding routine.

and do not distinguish between separate objects in the scene. The storage of this mesh is very straightforward, as can be seen in the pseudocode.

The pseudocode format used in this chapter is slightly more detailed than what was used in the previous chapter, especially the datatypes of variables are specified more often. To avoid confusion between fixed-length vectors (Vector3f and Vector3i) and variable-length vectors, we use term "array" to refer to the latter, even though in e.g. the STL library those are called vectors as well. Vector3f is a datatype with three floating-point numbers, used for storing 3D points, and Vector3i that contains three integers is used for containing various index triplets. We will not be too pedantic about following strict typing conventions: for example, pointers are denoted with an asterisk (*), but they are accessed using the dot notation just like pass-by-value variables. In addition the scene mesh (variable *mesh*) is assumed to be available for every function as if it were a global variable—which it most certainly would not be in any respectable implementation. These kind of notational shorthands will hopefully always be clear from the context. We also use textual form to denote certain simple operations, such as appending an element at the end of a variable-length array (see e.g. line 10 of Figure 4.1).

An important detail of the neighbor computation is that two triangles are considered to be neighbors only if their facing is consistent over the joining edge. If the triangles have opposite facings, they will not be neighbors, but instead two identical edges with different edge labels will be remain at the join, one attached to each triangle. In the following, we shall call edges that are joined to two triangles *double edges*, whereas edges that connect to one triangle are called *single edges*. Computing the triangle neighbors is only needed for determining the edge labels, and they are not used in other parts of the algorithm.

4.3 Receiver Point and Light Sample Trees

We construct lazily separate axis-aligned BSP trees for the receiver points and the light samples. Since a tight bounding box is computed for every node, the data structure is more specifically a bounding volume hierarchy, but the construction is nonetheless very similar to a BSP.

The data stored alongside with the positions of the points is independent of the tree. The tree operates only on the positions of the points to be stored, and refers to the points themselves through indices. This is because the receiver points and the light samples contain slightly different data: a light sample has a normal and a vector indicating the color and intensity of emitted light, whereas a receiver point has a normal and a vector for storing the accumulated incoming light.

The datatypes used for storing the point trees is given in Figure 4.2. In addition to the points, the tree nodes contain information about the blocking geometry in the scene. Specifically, each node in a tree has a list of scene triangles that intersect the parent node, but that do not intersect the node itself (variable TreeNode.$gone\_triangles$). This information will be needed later when constructing the sub-shafts. Because the trees are constructed lazily, it is also necessary to maintain a list of all intersecting triangles at the current leaf nodes of the tree.

The node splitting routine SPLIT-NODE shown in Figure 4.2 is fairly straightforward. A good split position is found using the FIND-SPLIT-PLANE function, which employs suitable heuristics for dividing the point set into two halves. In the prototype implementation, the operation of this function is as follows. If there are many points in the node (over 1024), the split plane is placed so that it divides the longest side of the bounding box of the node in half. If there are fewer points, a cost metric is used and the optimal split plane according to the metric is sought for. The cost metric that is minimized is $cost = N_{left} * A_{left} + N_{right} * A_{right}$, where N_{left} and N_{right} are the number of points in the two point sets, and A_{left} and A_{right} are the surface areas of the bounding boxes for the respective point sets. No significant amount of work has been put in trying out different cost metrics, and better candidates may well exist. After the split plane has been found, the points are distributed to the child nodes.

The handling of the scene triangles is also straightforward. The triangles that intersect the node being split are checked against intersection for both children. If a triangle intersects a child node, it is added to the list of intersecting triangles there. Otherwise, the triangle is added to the list of triangles that have escaped the child node due to the split. After the split is complete, the point and intersecting triangle lists of the node may be cleared in order to free the allocated memory. It should be noted that variable TreeNode.is_leaf is necessary for distinguishing between a node that is a real leaf of the hierarchy, not to be split anymore, and a temporary leaf that has not been split yet. This variable is set to TRUE if the node contains few enough points (line 15). The maximum allowed number of points in a leaf node is a free parameter in the ISS algorithm, and it can naturally be different in the receiver and light trees.

```
typedef TreeNode
{
    Tree*               tree                  // pointer to the containing tree
    TreeNode*           left, right           // pointers to left and right child nodes
    TreeNode*           parent                // pointer to parent node
    AABB                bounding_box          // axis-aligned bounding box
    Array⟨int⟩          points                // indices to points contained
    Array⟨int⟩          gone_triangles        // triangles not in this node but in parent
    Array⟨int⟩          inside_triangles      // all scene triangles intersecting this node
    bool                is_leaf               // is this a real leaf
}

typedef Tree
{
    TreeNode*           root_node             // pointer to the root node of the tree
    Array⟨Vector3f⟩*    point_pos             // point positions
    int                 max_points_in_leaf    // for terminating the subdivision
}

procedure SPLIT-NODE(TreeNode node)
 1  if node.is_leaf then return                // not to be split anymore
 2  if node.left or node.right then return     // already split
 3  {split_axis, split_pos} ← FIND-SPLIT-PLANE(node.points)
 4  INITIALIZE-NODE(node.left)
 5  INITIALIZE-NODE(node.right)
 6  for each pidx in node.points do            // distribute points
 7    if node.tree.point_pos[pidx][split_axis] < split_pos then
 8      add pidx at the end of node.left.points
 9    else
10      add pidx at the end of node.right.points
11    end if
12  end for
13  for each child in {node.left, node.right} do
14    COMPUTE-BOUNDING-BOX(child)
15    if child.points.size ≤ tree.max_points_in_leaf then child.is_leaf ← TRUE
16    for each tidx in node.inside_triangles do   // process scene triangles
17      if INTERSECTS-AABB(mesh.triangles[tidx], child.bounding_box) then
18        add tidx at the end of child.inside_triangles
19      else
20        add tidx at the end of child.gone_triangles
21      end if
22    end for
23  end for
24  clear node.points                          // not needed anymore
25  clear node.inside_triangles                // not needed anymore
```

Figure 4.2 Datatypes for storing the receiver and light trees, and the node splitting routine.

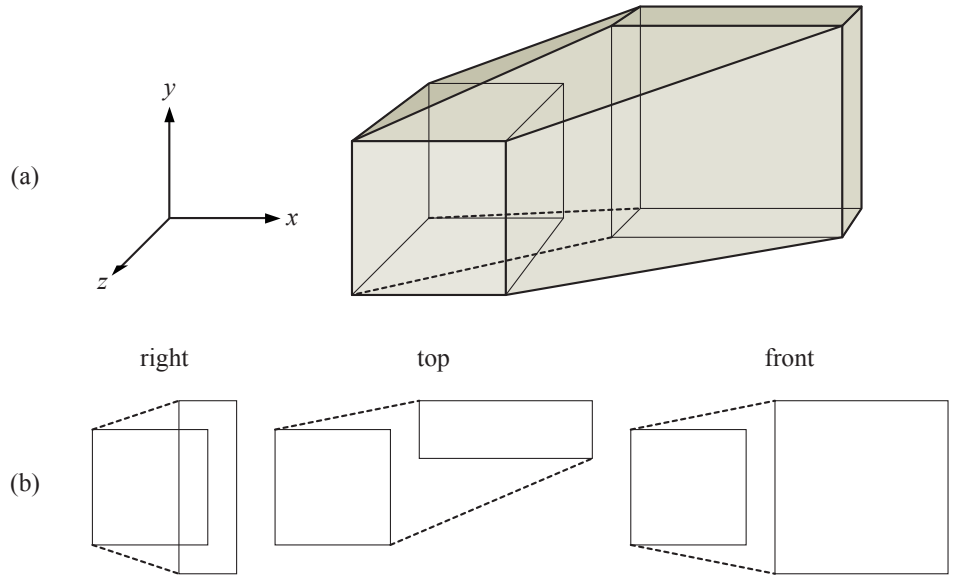


Figure 4.3 An example of a shaft between two axis-aligned boxes. (a) The surface of the shaft is the common convex hull of the boxes. (b) Because the shaft is constructed between axis-aligned boxes, the construction can be done by considering the 2D convex hulls of the three 2D projections of the situation separately. The dashed lines in the 2D projections correspond exactly to the oblique planes of the shaft.

4.4 Shaft Data

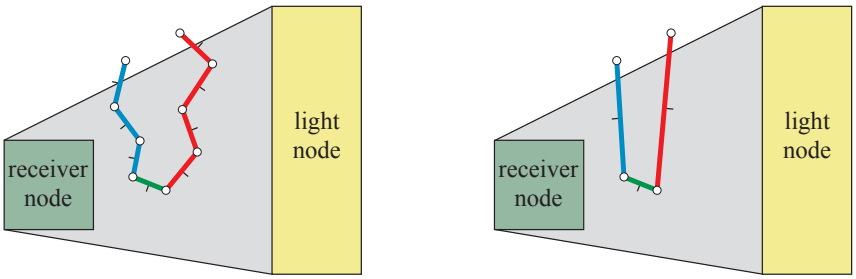
At the core of the ISS algorithm is the concept of a *shaft*, the closed volume between a receiver node and a light node that contains all possible rays of visibility between the nodes. In addition to the geometry of the shaft itself, we need to store the blocking geometry inside the shaft, enabling us to 1) test if the geometry blocks every possible ray between the receiver node and the light node, and 2) construct the sub-shafts. The governing idea behind the ISS algorithm is that *only the data that is necessary for performing these two operations is stored*. Thus, our goal is to represent the blocking geometry in such a way that we have all the information we need, but nothing more. In the following, we shall first discuss the construction and storage of the shaft geometry, and then the storage of the blocking geometry inside the shaft. The construction of the representation of the blocking geometry will be discussed later in Section 4.7.

4.4.1 Shaft Geometry

A shaft is the volume that contains all possible rays between two nodes, and it can be constructed by taking the common convex hull of the two bounding boxes of the nodes. Since the bounding boxes are axis-aligned, we do not need to use a full-fledged convex hull computation routine, but may reduce the task into a number of 2D sub-tasks. Figure 4.3 shows an example of a shaft between two axis-aligned boxes, and the orthogonal 2D projections of the same situation. The axis-aligned planes of the shaft are simply those of the common bounding box of both nodes, whereas each of the oblique planes of the shaft corresponds to one oblique line in a 2D convex hull of an orthogonal projection (dashed lines in Figure 4.3b). As the shaft is convex, it can be conveniently expressed as an intersection of half-spaces, and this is the representation we will use.

```
typedef ShaftGeometry
{
    Array⟨Vector4f⟩    planes        // plane equations of the bounding planes
    int                main_axis     // main axis of the shaft
}
```

Figure 4.4 Datatype for storing the shaft geometry.



(a) (b)

Figure 4.5 Difference between triangle-based and patch-based representations in 2D. (a) Individual triangles and their normals shown. Blue triangles in the left branch consistently face the receiver node, red triangles in the right branch face the light node, and the single green triangle in the bottom has no consistent facing with respect to the rays between the receiver and light nodes. (b) Patches constructed from these triangles. For solving the visibility between the nodes, this much simpler representation gives identical results to the triangle-based representation in (a).

A related concept that is very important in the ISS algorithm is the *main axis* of the shaft. This is the coordinate axis along which the two bounding boxes are separated. In the situation on Figure 4.3, the only possible main axis is the x axis, but generally there are multiple possible choices for the main axis. The main axis will be used when testing if the shaft is entirely blocked, and it also plays a fundamental role in clipping of the blocker geometry, a tricky topic that will be addressed in Section 4.5. No main axis can be determined if the boxes overlap, but in such cases we will not be doing shaft computations anyway, since there is obviously always visibility between the overlapping boxes. Furthermore, if we choose a main axis for a shaft, this same main axis will be valid for all sub-shafts to be formed. This property will also be important in the blocker clipping. Figure 4.4 shows the datatype used for storing the shaft geometry.

4.4.2 Blockers Inside the Shaft

The representation of the blocking geometry inside the shaft consists of two main ingredients: *patches* and *surfaces*. Following the terminology of Haumont et al. [38], we define a patch as a piece of surface that has consistent facing with respect to rays between the bounding boxes of the nodes that span the shaft. As an exception to this, a single triangle is always a valid patch even when its facing is not consistent with respect to this set of rays. The joining edge of two neighbor patches with the same consistent facing cannot be a silhouette edge, and this makes it possible to merge multiple triangles into one (usually non-planar) patch. Since only the silhouette edges are important for computing visibility, we do not need the information about the exact location of the

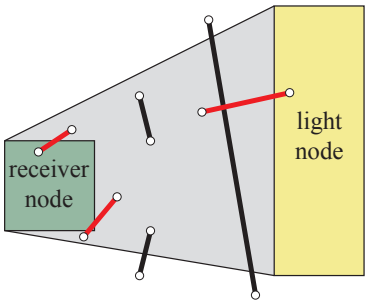


Figure 4.6 Examples of geometry that can and cannot be used as blockers inside a shaft. The black triangles can be used as blockers, since they intersect the shaft, but do not intersect the nodes. The red triangles cannot be used as blockers, since they intersect at least one of the nodes.

surface inside the edge loop. Patches that consist of a single triangle with inconsistent facing are an exception, and there we need to know the exact location of the surface. Patches with inconsistent facing always contain exactly one triangle, and they cannot be merged with any other patches. For an illustration about the difference between triangle-based and patch-based surface representations, see Figure 4.5.

Only the triangles that reside inside the shaft but that do not intersect the bounding boxes of the nodes can be used for occlusion. Therefore, if a triangle lies partially inside the bounding box of either the receiver node or the light node it will not be a part of the blocking geometry. This is illustrated in Figure 4.6. It is clear that the red triangles in the figure would not be useful as blockers in any case, since they cannot be a part of a surface that blocks the visibility between the nodes—a portion of a node would always remain on the wrong side of the surface.

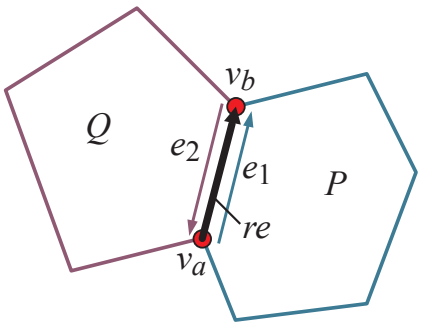
The datatypes for patch data as well as some related helper functions are given in Figure 4.7. As can be seen, there are two kinds of edge structures. The RawEdge type contains the endpoints of an edge as well as pointers to the neighboring patches, and the Edge type is a kind of a proxy for accessing the RawEdge instances, including a bit indicating if the edge being represented is actually a flipped version of the RawEdge being referenced. This somewhat convoluted structure is necessary, because we must identify each patch boundary edge with a real edge that knows of both of the patches connected to it. Figure 4.8 illustrates the relationships between patches and edges.

The facing of a patch is one of the three symbolic constants: TOWARDS-RECEIVER, TOWARDS-LIGHT and INCONSISTENT. The first constant TOWARDS-RECEIVER means that all possible rays from the receiver node to the light node pierce the patch through its front face, i.e. the patch faces towards the receiver node. The second constant TOWARDS-LIGHT is analogous, meaning that the patch faces towards the light node. If neither condition is true, the facing of the patch is INCONSISTENT, meaning that the direction in which a ray pierces the patch depends on which of the possible rays we choose. We will examine how the facings of the patches are computed in Section 4.7.3. Patches that do not have a consistent facing ($facing$ = INCONSISTENT) will always consist of a single triangle, and in these cases, we also store the triangle index of the generating triangle. This information will be needed when the facing of a single-triangle patch is updated.

A *surface* is a collection of patches that are connected through edges that are at least partially inside the shaft. Intuitively, a surface is the set of patches that can be reached by starting a traversal at some patch, and traversing over connecting edges that are (at least partially) inside the shaft. Note that the patches of the same surface need not have the same facing. Indeed, if we had two neighboring patches with the same consistent facing, those would have been already merged to form a single

```
// enumeration for expressing the facing of a patch
typedef Facing := enum {TOWARDS-RECEIVER, TOWARDS-LIGHT, INCONSISTENT}

typedef Vertex
{
    Vector3f        position            // vertex position in 3D
}

typedef RawEdge
{
    Vertex          vertices[2]         // vertices of the edge
    Patch*          neighbors[2]        // neighbor patches
    bool            is_inside           // is the edge at least partially inside shaft
    int             mesh_edge           // mesh edge label
}

typedef Edge
{
    RawEdge*        raw_edge            // pointer to the raw edge
    bool            is_flipped          // is this a flipped version of the raw edge
}

typedef Patch
{
    Array⟨Edge⟩     edges               // boundary edges of the patch
    Facing          facing              // facing of the patch
    int             mesh_triangle       // triangle ID if facing = INCONSISTENT
}

function Edge FLIP-EDGE(Edge* e)
 1  Edge fe
 2  fe.raw_edge ← e.raw_edge
 3  fe.is_flipped ← not e.is_flipped
 4  return fe

function Vertex GET-VERTEX(Edge* e, int vidx)  // vidx in {0, 1}
 5  if e.is_flipped then return e.raw_edge.vertices[vidx XOR 1]
 6                  else return e.raw_edge.vertices[vidx]

function Patch* GET-NEIGHBOR(Edge* e)
 7  if e.is_flipped then return e.raw_edge.neighbors[1]
 8                  else return e.raw_edge.neighbors[0]

function Patch* GET-OWNER(Edge* e)
 9  if e.is_flipped then return e.raw_edge.neighbors[0]
10                  else return e.raw_edge.neighbors[1]
```

Figure 4.7 Datatypes and a couple of helper functions related to representing the patches of blocking geometry. Functions SET-NEIGHBOR and SET-OWNER are defined analogously to the GET-functions shown above.



RawEdge re
$re.vertices[0] = v_a$
$re.vertices[1] = v_b$
$re.neighbors[0] = Q$
$re.neighbors[1] = P$

Edge e_1, e_2
$e_1.raw\_edge = re;\ e_1.is\_flipped =$ FALSE
$e_2.raw\_edge = re;\ e_2.is\_flipped =$ TRUE

Figure 4.8 Illustration of the relationship between patches, edges and raw edges. Patches P and Q share one edge, and this is represented by one RawEdge instance re and two Edge instances e_1 and e_2. Patch P owns Edge e_1, which is a non-flipped version of the RawEdge re. Patch Q owns Edge e_2, which is a flipped version of re. Function call GET-NEIGHBOR(e_1) returns Q, whereas GET-NEIGHBOR(e_2) returns P. The direction of re does not play any role per se; an equally valid representation of the same situation could be obtained by reversing the direction of re, swapping P and Q in $re.neighbors$, and inverting the $is\_flipped$ bits of e_1 and e_2.

```
typedef Surface
{
    Array⟨Patch*⟩    patches          // the patches that form the surface
    AABB             bounding_box     // bounding box of the surface
    Array⟨int⟩       loose_edges      // unconnected boundary edges
}

typedef Shaft
{
    TreeNode*        receiver_node    // the receiver node
    TreeNode*        light_node       // the light node
    ShaftGeometry    shaft_geometry   // the geometry of the shaft
    Array⟨Surface⟩   surfaces         // the blocker geometry inside the shaft
}
```

Figure 4.9 Datatypes for storing surfaces and the shaft itself.

patch. Our goal is to have surfaces that are not composed of disjoint components, i.e. every patch of a surface should be reachable from every other patch. During the sub-shaft construction, we will temporarily have surfaces that do have disjoint components, as well as surfaces that should be combined together since they have a common edge inside the shaft. How to handle this *splitting* and *combining* of surfaces will be addressed later, in Sections 4.7.6 and 4.7.7.

A surface may have a set of *loose edges*. A loose edge is a double edge, i.e. an edge connected to two triangles in the scene mesh, that resides at least partially inside the shaft, but is connected to only one patch. These kind of edges act as hooks where new surface elements can be attached when new geometry enters the shaft. New geometry may appear when the nodes that span the shaft are split, as illustrated in Figure 4.10. We also compute and store the axis-aligned bounding box for every surface. The datatypes for storing the surfaces and the shaft itself are given in Figure 4.9.

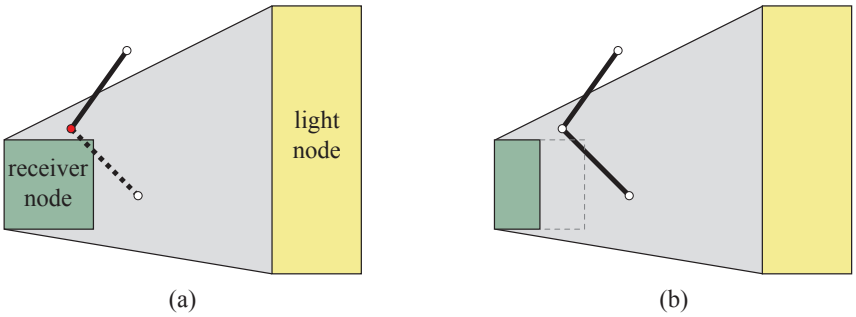


Figure 4.10 An example of a loose edge. (a) The surface indicated by the solid black line has a loose boundary edge marked as a red dot. This is because the edge at the red dot is also connected to a triangle denoted by the dashed line, but the triangle cannot be used as a blocker. (b) After the receiver node is split, the triangle that was originally invalid can now be used as a blocker. The loose edge is connected, and after the connection, the patches may be fused to form a single large patch.

4.5 Clipping and Clamping of Blocker Geometry

The test we use for detecting if there is a surface that blocks every ray between the nodes that span the shaft—the *blocking test*—requires that all blocking geometry lies between two *clip planes* as will be discussed further in Section 4.6. The clip planes are axis-aligned planes that are perpendicular to the main axis of the shaft, and they are located so that the entire shaft geometry is between them. The clip planes thus define a range of coordinates along the main axis of the shaft, and we need to be able to ensure that all geometry that the blocking test accesses is within this range.

Let us consider adding new blocker triangles into the shaft. To clip the triangles, we first test if they are partially outside the region between the clip planes. If a triangle lies completely between the clip planes, no clipping needs to be done. Otherwise, we construct new vertices on the clip planes, which causes the triangle to form a polygon that may have more than three vertices. This is not a problem, since every new triangle initially forms an unique patch, and the merging of patches is done afterwards. For new edges that are created due to clipping, we set the $mesh\_edge$ field of the corresponding RawEdge instance to symbolic constant CLIP-EDGE, since they do not correspond to any edge of the mesh. These edges, which we shall call *clip edges* can obviously never be shared by two patches.

It is somewhat of a subtle point that the clipping needs to be performed only once. Consider a situation where multiple triangles are clipped to a clip plane and patches are formed from the resulting polygons. Then, the patches are merged to form a larger non-planar patch, and the internal edges are forgotten. Now, suppose that the clip plane moves closer to the other clip plane, because the shaft is shortened due to subdividing the node that originally defined the placement of the clip plane. It seems that we need to re-clip the patch in order to ensure that all edges remain between the clip planes. But how can we clip a non-planar polygon? There is no way of knowing where the surface that the patch represents is actually located, and hence we cannot determine the new clip edges that would be formed when this surface is clipped against the shifted clip plane.

There is a neat solution to this problem, and it is to *clamp* the edges of a patch when a clip plane moves, instead of trying to clip the patch again. Consider an edge that lies partially inside and partially outside the new clip plane. To clamp the edge, we compute the intersection point of the edge with the new clip plane, and subdivide the edge into two parts. The endpoint of the edge that is outside the new clip plane can then be projected orthogonally to the new clip plane. The main

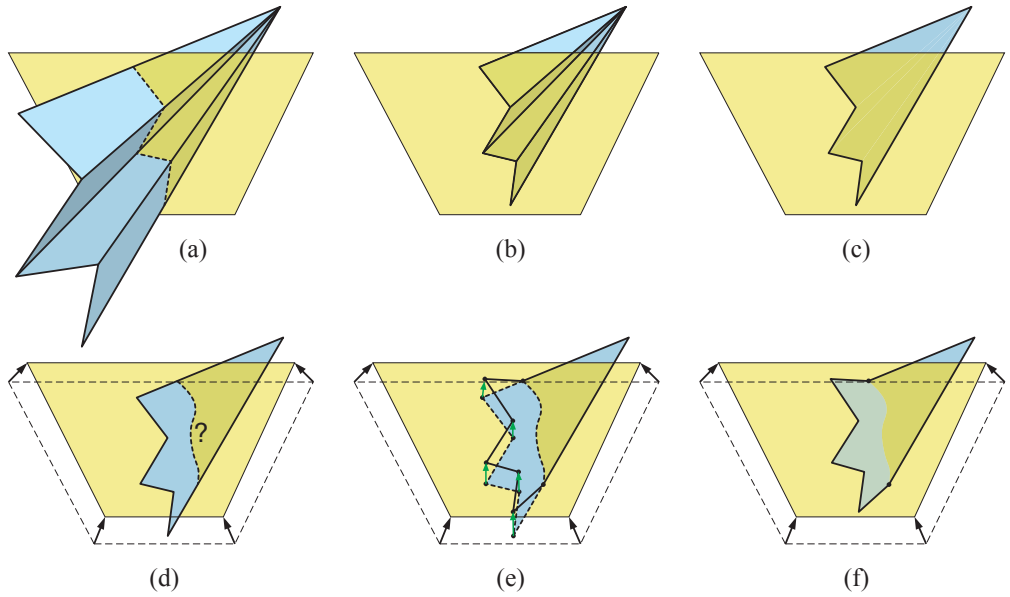


Figure 4.11 Clipping and clamping of blocker geometry. The clip plane is shown in yellow, and the shaft is assumed to continue away from the reader. (a) A set of blocker triangles is inserted in the shaft. The triangles need to be clipped, since they extend over the clip plane. The clip edges formed are shown as dashed lines. (b) After the clipping is complete, we have a set of valid patches, one patch per triangle. (c) If the facings of the patches match—as is assumed in this illustration—they may be merged together. The internal structure of the surface is lost, but this does not lose any important information; the internal edges of the surface could not be silhouette edges because of the consistent facings of the patches. (d) A tricky situation arises when we later have a clip plane other than with which the patch was originally clipped, and we need to somehow constrain the edges according to the current clip plane for performing the blocking test. As the exact location of the surface is unknown, the patch cannot be simply clipped again. (e) The situation can be handled by projecting the parts of the edges that are outside the clip plane orthogonally onto the clip plane. Edges that pierce the clip plane need to be split in two parts. (f) The resulting edge loop stays on the correct side of the clip plane and it has the same occlusion characteristics as we would have obtained by clipping the original triangle surface with the current clip plane in the first place.

advantage of this operation is that the clamping of the outside part of the edge cannot cause the edge to intersect the bounding boxes of the nodes. The clamping needs to be done only in the blocking test, since this test is the only operation that requires that all edges lie between the clip planes. Figure 4.11 illustrates the clipping and clamping operations.

4.6 Testing If Shaft Is Blocked

Assuming that we have the blocker geometry inside the shaft represented as patches and surfaces, the remaining problem is to determine if the shaft is *blocked*, i.e. if every possible ray between the receiver and light nodes is occluded. Our approach is conservative, as we only detect the cases where a single surface encloses the entire shaft by forming a continuous cover over it. There may be cases where multiple disjoint surfaces together block the shaft, but those will not be detected by the ISS

algorithm; in visibility computation terms, we do not perform any *occluder fusion*. Proper handling of the aggregate occlusion of disjoint surfaces would generally require dual-space calculations that were touched in Section 2.3.3.

Our *blocking test* is very similar to the one that Bernardini et al. [11] used in a larger visibility pre-processing system. In contrast to their algorithm, we employ the patch-based representation of the blocker geometry, which leads to faster execution time since the number of patches is generally smaller than the number of triangles.

We perform the blocking test for each surface separately, since if there is a continuous cover blocking the shaft, it must be formed by a single surface. As an early-exit test, we first determine if the bounding box of the surface encloses the shaft. If it does not, the surface cannot enclose the shaft either, and further processing of the surface can be avoided.

4.6.1 Boundary Edges

The first step of the blocking test is to extract the boundary edges of the surface. Extracting the boundary edges is performed simply by enumerating all raw edges of the patches of a surface, and discarding the edges that are connected to two patches. The remaining edges are the boundary edges of a surface.

Now, if some of the boundary edges are inside the shaft, there must be a hole in the surface that prevents the surface from forming a continuous cover over the shaft. This observation yields another simple early-exit test for aborting the blocking test. Note that the boundary edges always form loops, as there can be no boundary edge without other boundary edges connecting to both of its vertices.

4.6.2 Construction of the Test Line

If none of the boundary edges are inside the shaft, it is possible that the surface blocks the shaft. Testing whether this is true is done by testing if the boundary edges form a loop that encloses a single *test line* that goes through the shaft.

The test line must be chosen so that it passes through both nodes that span the shaft. In addition, the test line must intersect the faces of the nodes that lie on the clip planes, i.e. the planes of the shaft that are perpendicular to the main axis of the shaft. Now, if the boundary edge loop blocks the chosen test line, it also blocks every other possible test line, since all boundary edges are known to lie outside the shaft, and it is not possible to move the test line so that it would cross a boundary edge. Note that without blocker geometry clipping, this would not hold in general, as illustrated in Figure 4.12.

4.6.3 Testing if the Test Line is Blocked

Figure 4.13 shows a simple situation where a surface blocks the shaft. The boundary edges of the surface are shown, and it is obvious that because of the geometrical configuration, there must be a portion of the surface that forms a cover that blocks the shaft. The surface blocks the test line shown in red, and we note that because it is impossible to move the test line (in the space of valid test lines) so that it would end up outside the boundary edge loop of the surface, all test lines must be blocked.

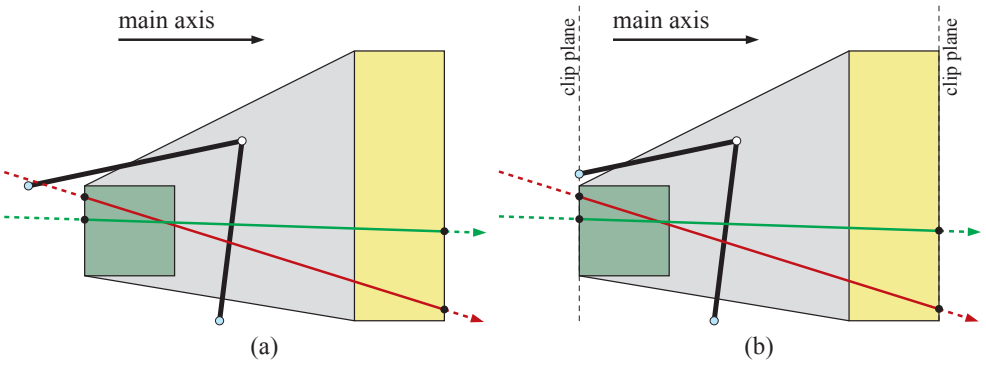


Figure 4.12 Why the blocker geometry needs to be clipped. (a) If the blocker geometry is not clipped, the result of the blocking test may depend on the choice of test line. With the green test line, correct result is obtained. With the red test line, both boundary edges (cyan dots) are on the same side of the test line, and thus it would be determined that the shaft is not blocked. (b) By clipping the blocker geometry to the shaft planes perpendicular to the main axis (vertical dashed line), the choice of test line does not matter anymore as long as the test line goes through the faces of the nodes that are on the clip planes. The boundary edges are always on different sides of a test line, since the shaft is indeed blocked.

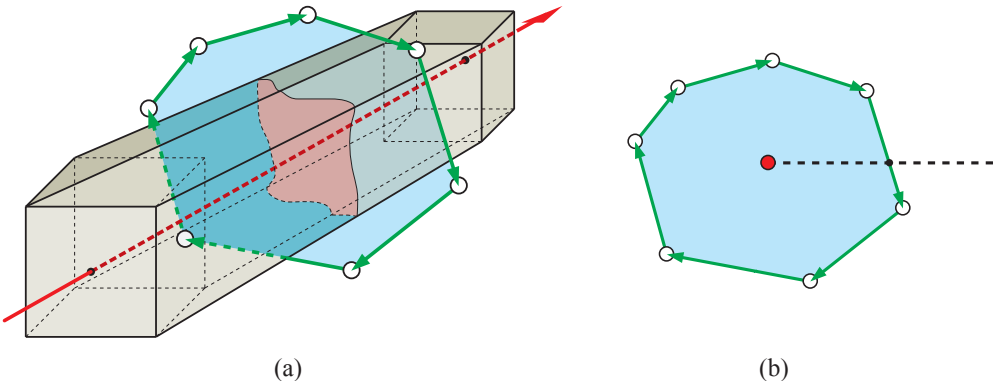


Figure 4.13 Example of a boundary edge loop of a surface. (a) The boundary loop of the cyan surface encloses the shaft. Because of blocker geometry clipping, we know that the surface cannot bend behind the bounding boxes of the nodes, and therefore it must have a portion that intersects the shaft so that it blocks all rays between the nodes. If the surface bent behind one of the nodes, we would have two boundary edge loops canceling each other out. The test line is drawn in thick red. (b) We can imagine transforming the boundary edges into 2D so that the test line gets projected to a point. If the transformed boundary edges form a polygon that encloses the point representing the test line, the boundary edge loop must also enclose the entire shaft. This is because all boundary edges are known to reside outside the shaft.

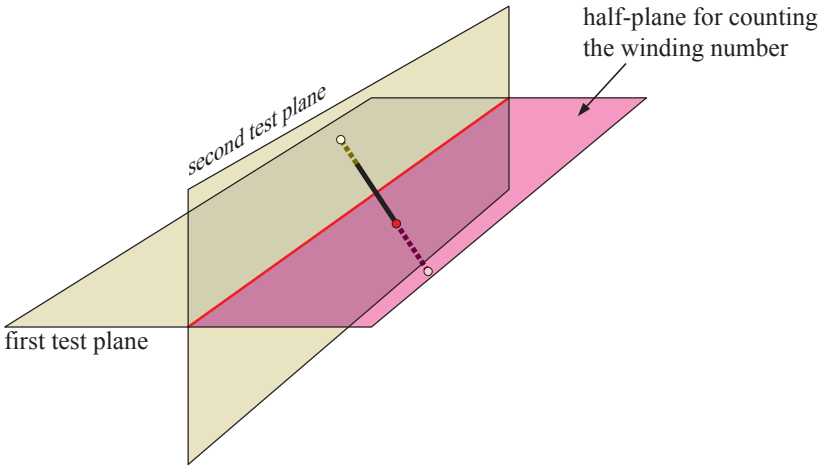


Figure 4.14 Two test planes are constructed so that the test line lies at the intersection of the planes. To count the winding number of an edge loop, each edge is first tested for intersection against the first test plane (red dot). The second test plane is needed for detecting if the intersection point lies on the correct half-plane. Classifying the vertices against two test planes is conceptually equivalent to projecting them into 2D, and testing the intersection against the highlighted half-space is equivalent to testing if the projected 2D edge intersects the positive half of x axis.

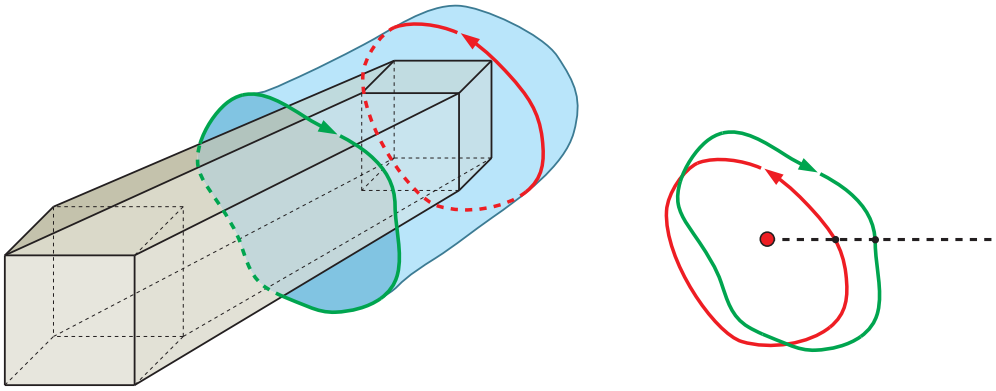
The remaining task is to test if the test line pierces the polygon formed by the boundary edges. In Figure 4.13b, a suitable 2D projection is used so that the test line is mapped to a point, and we end up with point-in-polygon test. This test is performed by counting the winding number of the projected edge loop, counting intersections with ray from the test point to infinity, shown as dashed line. It does not matter which ray we choose, since the resulting winding number is always the same.

In practice, we do not project the boundary edges into 2D explicitly, but rather perform the tests in 3D in a manner that is analogous to operating on projected geometry. The ray from the test line becomes a half-plane, and we simply need to detect when the boundary edges intersect this half-plane, as illustrated in Figure 4.14. We will nonetheless use the analogue of performing a 2D projection in many of the following illustrations.

4.6.4 Example Cases and Tricky Occluders

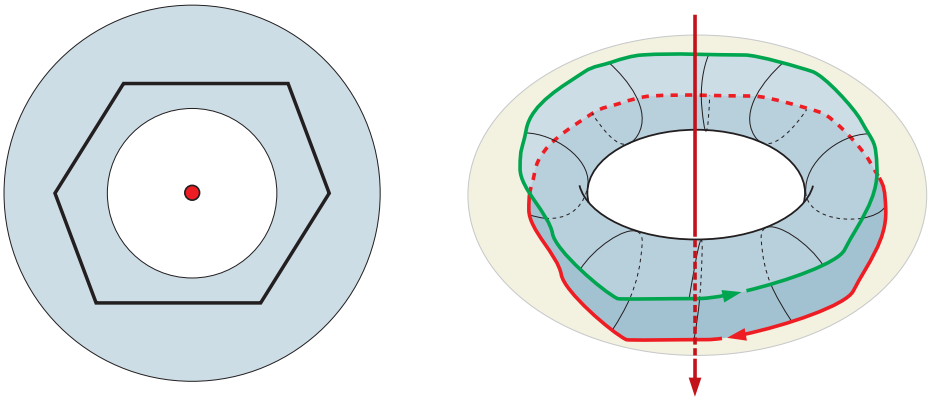
Figure 4.15 shows an example where a surface bends behind a node, forming a pouch that does not block the shaft. Because of clipping, two boundary edge loops are formed, and the edge loops cancel each other out in the point-in-polygon test. It is important to note that these boundary edge loops are processed simultaneously, since they are formed by the same surface. Both of the loops would block the shaft if they were processed separately, which would obviously give an incorrect result.

In a similar fashion, two opposite boundary edge loops are formed if the shaft passes through a torus so that the hole remains inside the shaft, while the surface of the shaft slices the body of the torus in two parts, as illustrated in Figure 4.16. The boundary edges are processed together, since the surface connects them. The reader may imagine subdividing the shaft so that the hole of the torus is left outside the shaft. In this case, the surface would split into two separate parts, and the boundary edge loops would be processed separately. This is correct, since both of the surfaces would then block the shaft.



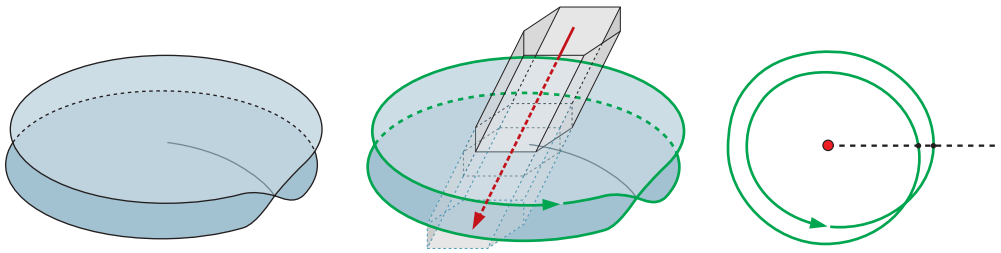
(a) (b)

Figure 4.15 Example of a clipped surface. (a) The original surface forms a pouch that does not block the shaft. The far end of the surface is clipped away, and only the portion between the two loops remains. As a consequence, the surface has a second boundary edge loop on the clip plane. The two boundary edge loops have opposite windings. (b) In the point-in-polygon test, the point marking the test line (not shown in the figure) is not inside the double polygon formed by the loops, since the windings of the loops cancel each other out. Both loops are processed simultaneously, since they are the boundary of the same surface. Note that if this surface were to be inserted as new geometry into the shaft shown, it would be rejected altogether, since all surface primitives are outside the shaft. Nonetheless, this kind of situation may occur when the shaft is subdivided and existing geometry is left outside the shaft.



(a) (b)

Figure 4.16 Boundary edge loops of a torus. (a) An orthogonal view of a shaft going through a torus. Only the hole of the torus is inside the shaft. (b) Side view of the surface that remains, when geometry outside the shaft is removed. As can be seen, two edge loops are formed, both connected to the same surface. The windings of the edge loops are opposite, and the blocking test thus decides that the shaft is not blocked. In this sense, the situation is similar to the one shown in Figure 4.15.



(a) (b) (c)

Figure 4.17 Certain tricky surfaces require that the winding number of boundary edge loops is counted instead of just the number of intersections with the test half-plane. (a) A self-intersecting surface that has two layers connected at the center, forming a double surface that has consistent orientation but only one boundary loop. (b) The boundary edges as seen by a shaft that pierces the surface. (c) The boundary edges form a loop that surrounds the test line *twice*, with the same winding both times. The winding number is thus two, and the surface does indeed block the shaft. If we used a crossing number point-in-polygon test, we would have concluded wrongly that the shaft is not blocked. Even worse, the surface would be discarded after this test (as will be explained in Section 4.7.8), since it would seem irrelevant for future calculations, and this would lead to missing shadows.

Determining the winding number instead of just counting the number of intersections in the point-in-polygon test is necessary in order to handle certain kinds of tricky occluders correctly. This is illustrated in Figure 4.17. The resulting boundary edge loop encloses the test line twice, and if we only counted the number of intersections with the dashed ray in Figure 4.17c, we could not distinguish this situation from the one we have with torus or the pouch of Figure 4.15.

These two kinds of point-in-polygon tests are usually called *crossing number* and *winding number* tests. The crossing number test simply determines if the number of intersections with a ray is even or odd, and it works correctly with polygons that do not have self-intersections. The winding number test takes the direction of the intersecting edges into account, and as we saw, it is necessary to do in our case. In the blocking test of Bernardini et al. [11], it is not specified whether a crossing number or a winding number test is used, and therefore it is unclear whether their algorithm would handle surfaces like the one in Figure 4.17 correctly.

Figure 4.18 shows the pseudocode of the blocking test. The early exit tests on lines 1 and 3–7 determine if the bounding box of the surface may block the shaft and if there are boundary edges inside the shaft. The call on line 2 collects all raw edges of the patches that form the surface, and function CONSTRUCT-TEST-PLANES (line 8) constructs two test planes as illustrated in Figure 4.14. Every raw edge is processed, and if an edge is a single edge, it is on the boundary of the surface (line 11). For each raw boundary edge, a directed edge is constructed (lines 12–14). Function CLAMP-AND-GET-VERTICES (line 15) performs the edge clamping (Section 4.5) that ensures that all edges remain between the clip planes. This may cause an edge to be split into multiple parts over which we need to loop (line 16). Geometrical tests (lines 19–27) test if the edge part intersects the correct half-plane. If so, the winding counter is updated appropriately (lines 28–31). It is important to handle the special case when one of the vertices lies on the half-plane (line 29), and the easiest way to handle this is to count only a half of the update for these parts. line 30 checks in which direction the edge pierces the half-plane, and sets the sign of the update to the winding counter accordingly. Finally, after all boundary edges have been processed, the result of the test is determined simply by checking if the winding number is nonzero (line 34).

```
function bool BLOCKS-SHAFT(Surface surf, ShaftGeometry shaft_geometry)
 1  if not CAN-BLOCK-SHAFT(surf.bounding_box, shaft_geometry) then return FALSE
 2  Array⟨RawEdge*⟩ raw_edges ← COLLECT-RAW-EDGES(surf)
 3  for each re in raw_edges do
 4    if (re.neighbors[0] = NULL or re.neighbors[1] = NULL) and re.is_inside then
 5      return FALSE                          // single edge inside shaft
 6    end if
 7  end for
 8  {Vector4 p_a, Vector4 p_b} ← CONSTRUCT-TEST-PLANES(shaft_geometry)
 9  float winding_counter ← 0
10  for each re in raw_edges do
11    if re.neighbors[0] ≠ NULL and re.neighbors[1] ≠ NULL then continue   // double edge
12    Edge e                                // construct directed edge
13    e.raw_edge ← re
14    e.is_flipped ← (re.neighbors[0] = NULL)   // ensure correct direction
15    Array⟨Vector3⟩ vertices ← CLAMP-AND-GET-VERTICES(e, shaft_geometry)
16    for each i in {0, ..., vertices.size − 2]} do   // loop over parts of edge
17      Vector3 v_0 ← vertices[i]           // get vertex positions for edge part
18      Vector3 v_1 ← vertices[i + 1]
19      float d_0 ← (v_0 · p_a)             // classify against first test plane
20      float d_1 ← (v_1 · p_a)
21      if SIGN(d_0) = SIGN(d_1) then continue   // does not intersect first test plane
22      float f_0 ← (v_0 · p_b)             // classify against second test plane
23      float f_1 ← (v_1 · p_b)
24      if f_0 < 0 and f_1 < 0 then continue   // wrong side of second test plane
25      if (d_0 < d_1) = (d_0 f_1 > d_1 f_0) then
26        continue                          // intersects wrong side of first test plane
27      end if
28      int adjust ← 1                      // adjustment to winding counter
29      if d_0 = 0 or d_1 = 0 then adjust ← 1/2   // handle special case
30      if d_0 > d_1 then adjust ← (−adjust)   // take direction into account
31      winding_counter ← winding_counter + adjust
32    end for
33  end for
34  return (winding_counter ≠ 0)            // shaft is blocked if counter is nonzero
```

Figure 4.18 Pseudocode for the blocking test.

4.7 Sub-Shaft Construction

We now proceed by examining how a sub-shaft is constructed, assuming that we already have a *parent shaft* with blocking geometry inside it. The construction of the initial shaft, i.e. when no parent shaft is given, is discussed later in Section 4.8. The slightly counterintuitive section ordering was chosen because most of the operations that are needed in constructing the initial shaft are the same as the ones that are used when constructing a sub-shaft from a parent shaft.

```
function Shaft CONSTRUCT-SUB-SHAFT(TreeNode* receiver_node, TreeNode* light_node
                                   TreeNode* split_node, Shaft* parent)
 1  Shaft sub ← new Shaft                                  // construct a new shaft
 2  sub.receiver_node ← receiver_node
 3  sub.light_node ← light_node
 4  sub.shaft_geometry ← CONSTRUCT-SHAFT-GEOMETRY(receiver_node, light_node)
 5  sub.shaft_geometry.main_axis ← parent.shaft_geometry.main_axis
 6  for each surf in parent.surfaces do                    // copy all surfaces of the parent shaft
 7    Surface copy ← DUPLICATE-SURFACE(surf)
 8    add copy at the end of sub.surfaces
 9  end for
10  Surface new_tris ← CONSTRUCT-TRIANGLE-SURFACE(split_node.gone_triangles, sub)
11  add new_tris at the end of sub.surfaces                // add the new blocker surface
12  Array⟨Surface⟩ temp_surfaces                           // for storing surfaces after splitting them
13  for each surf in sub.surfaces do
14    CLASSIFY-EDGES(surf, sub)                            // update is_inside of raw edges
15    UPDATE-PATCH-FACINGS(surf, sub)                      // recompute facings of patches
16    MERGE-PATCHES(surf)                                  // merge neighbors with same facing
17    SIMPLIFY-PATCHES(surf)                               // remove extraneous vertices in patches
18    SPLIT-SURFACE(surf, temp_surfaces)                   // separate disjoint components of surface
19  end for
20  clear sub.surfaces                                     // all data is now in temp_surfaces
21  for each surf in temp_surfaces do COMPUTE-LOOSE-EDGES(surf)
22  COMBINE-SURFACES(temp_surfaces, sub.surfaces)
23  for each surf in sub.surfaces do COMPUTE-BOUNDING-BOX(surf)
24  return sub
```

Figure 4.19 Pseudocode for constructing a sub-shaft.

When a receiver node or a light node is split and a sub-shaft is formed, some or all of the following may happen:

- new blockers may enter the shaft,
- old blockers may exit the shaft,
- edges may move outside the shaft,
- facings of patches may change from INCONSISTENT to TOWARDS-RECEIVER or TOWARDS-LIGHT, and this may enable merging of patches,
- surfaces may become combined if new blockers connect them,
- surfaces may be split into disjoint components if portions of them exit the shaft,
- it may become possible to simplify patches if some of their edges exit the shaft.

In the following sections, we will examine each of these operations separately. The generic procedure for constructing the sub-shaft is illustrated in the pseudocode in Figure 4.19. The CONSTRUCT-SUB-SHAFT function takes as input parameters the receiver and light nodes for the sub-shaft, the parent shaft, and in addition, pointer to the point tree node that was split. Knowing which node

was split is important when constructing the new blocker surface (line 10). First, the sub-shaft is initialized with proper shaft geometry (lines 1–5). Note that the main axis is inherited from the parent shaft. Then, the surfaces of the parent shaft are copied to the sub-shaft, and a new surface is constructed for the blockers that enter the shaft (lines 6–11). An array for storing temporary surfaces after splitting them is allocated on line 12. For every surface (including the newly created surface for new blockers), we then classify whether its edges are inside or outside the shaft, update facings of patches, merge neighboring patches with consistent facing, simplify the patches, and finally, split the surfaces into their disjoint components (lines 13–19). The splitting procedure SPLIT-SURFACE stores the separate components of the surface into the temporary surface array (line 18), and thus after the loop we can clear the surface array of the sub-shaft (line 20). For each of the temporary surfaces, we compute the loose edges (line 21), after which we combine the surfaces (line 22), storing the resulting surfaces back into the shaft itself. Finally, the bounding boxes of the surfaces are computed (line 23).

4.7.1 Adding New Blockers

New blockers may enter the shaft due to the shrinking of nodes that span the shaft, as illustrated in Figure 4.10. To find the triangles that potentially enter the shaft, we use the list of triangles that intersected the parent node of the split node, but do not intersect the node itself. This is exactly what we store in TreeNode.$gone\_triangles$ array (Figure 4.2), and enumerating the triangles in this array is all we need to do. It must still be tested that the triangles intersect the shaft and that they do not intersect the other node. The triangles are then clipped against the clip planes, and a new patch is formed for each triangle. The neighbors of these patches are set based on the edge labels of the mesh. It should be noted that the surface we construct for the new blockers may have disjoint components, and therefore we must later split this surface using the procedure SPLIT-SURFACE given in Section 4.7.6.

Figure 4.20 shows the pseudocode for constructing a new surface that has a separate patch for every blocker triangle. Checking that the triangle is a valid blocker is performed on lines 6–8. A patch is then constructed (lines 9–10) so that the triangle is clipped first, and the facing of the patch is initialized to INCONSISTENT, to be determined correctly later using the mesh triangle index set on line 11. Because we need to establish the neighborships between the new patches, and to ensure that only one raw edge per mesh edge exists, we need to detect when an edge is already present in the patches of the surface being constructed. For this purpose, we use the $edge\_map$ map for finding already constructed edges based on the mesh edge label (line 3). If an edge and corresponding raw edge have been already created, the test on line 15 succeeds, and in this case, the old raw edge is used and the edge of the patch being processed is set to reference the old raw edge (lines 16–18). Setting the owner of the new, flipped edge (line 18) sets the missing neighbor of the raw edge, and enables the neighborship between patches. As a result, all edges in the patches of the constructed surface will have correctly marked neighbors, and there will be only one raw edge for each mesh edge.

4.7.2 Classifying Edges

When a sub-shaft is constructed from a parent shaft, some or all edges of a surface may end up being outside the shaft. This must be detected, since edges inside the shaft act differently from the edges that are outside the shaft. For instance, patches that are joined through an edge inside the shaft must belong to the same surface, whereas edges outside the shaft are not considered to connect patches, giving rise to the possibility of splitting a surface into multiple disjoint components. It is mandatory

```
function Surface CONSTRUCT-TRIANGLE-SURFACE(Array⟨int⟩* triangles, Shaft* shaft)
 1  Surface new_surface
 2  ShaftGeometry* sgeom ← shaft.shaft_geometry
 3  Map⟨int → Edge*⟩ edge_map                  // maps mesh edge labels to our edges
 4  for each tidx in triangles do
 5    Triangle* t ← mesh.triangles[tidx]
 6    if not INTERSECTS-SHAFT(t, sgeom) then continue
 7    if INTERSECTS-AABB(t, shaft.receiver_node.bounding_box) then continue
 8    if INTERSECTS-AABB(t, shaft.light_node.bounding_box) then continue
 9    Patch p ← CONSTRUCT-CLIPPED-PATCH(t) // make new patch with fresh RawEdges
10    p.facing ← INCONSISTENT                 // initial facing is always INCONSISTENT
11    p.mesh_triangle ← tidx                  // store mesh triangle index
12    for each e in p.edges do
13      int elabel ← e.raw_edge.mesh_edge
14      if elabel = CLIP-EDGE then continue   // cannot have a neighbor
15      if edge_map contains elabel then      // check if edge exists already
16        delete e.raw_edge                   // use existing RawEdge, delete this
17        e ← FLIP-EDGE(edge_map.get(elabel)) // make a flipped version of old edge
18        SET-OWNER(e, p)                     // establish neighborship between patches
19      else
20        insert (elabel → e) into edge_map   // new edge, store for use by future triangles
21      end if
22    end for
23    add p at the end of new_surface.patches
24  end for
25  return new_surface
```

Figure 4.20 Pseudocode for constructing a new surface that has a separate patch for every new blocker triangle.

```
procedure CLASSIFY-EDGES(Surface* surf, Shaft* shaft)
1  Array⟨RawEdge*⟩ raw_edges ← COLLECT-RAW-EDGES(surf)
2  for each re in raw_edges do
3    if not re.is_inside then continue        // if edge was outside, it still is
4    LineSegment l ← {re.vertices[0].position, re.vertices[1].position}
5    re.is_inside ← INTERSECTS-SHAFT(l, shaft.shaft_geometry)
6  end for
```

Figure 4.21 Pseudocode for classifying the edges of a surface.

to be able to split surfaces when their connecting edges fall outside the shaft, for example to detect when a torus starts blocking a shaft as discussed in Section 4.6.4.

Pseudocode for classifying the edges is given in Figure 4.21. It can be seen that the operation is conceptually very simple; every raw edge is processed, and if it intersects the shaft, it is marked to be inside the shaft (lines 2–6). To optimize the process a bit, we observe that if an edge was outside the parent shaft, it must be outside the sub-shaft as well, and its status cannot therefore change (line 3).

4.7.3 Computing Patch Facings

As was already discussed in Section 4.4.2, the facing of a patch may be either TOWARDS-RECEIVER, TOWARDS-LIGHT or INCONSISTENT. Only patches that have the same *consistent* facing, i.e. other than INCONSISTENT, may be safely merged together, since their joining edges cannot be silhouette edges between the nodes that span the shaft.

The most critical observation in computing the facings is that we never need to update the facing of a patch that has consistent facing. The reason is trivial: if a patch has a facing, say TOWARDS-RECEIVER, it means that all rays from the receiver node to the light node pierce the piece of surface represented by the patch by hitting its front face. Now, if we split a node, this same condition trivially holds for the reduced set of possible rays. Therefore, our only problem is to determine the facing for the patches whose facing is INCONSISTENT, and since these patches are not joined to any other patches, they always consist of a single triangle. This, in turn, causes these patches to be always planar. The remaining task is to determine if the plane of the patch is such that all rays between the nodes pierce it in the same direction. This can be done simply by checking if the bounding boxes of the nodes are in front of or behind the plane, or if they intersect the plane.

Assuming that we have a function that classifies an axis-aligned bounding box with respect to a plane, we can summarize the facing of the patch as shown in Table 4.1. As can be seen, the facing of the patch is inconsistent only when both of the nodes intersect the plane. Note that certain combinations are impossible; both the receiver and the light node cannot be on the same side of the plane, since the patch that defines the plane is located inside the shaft.

Pseudocode that updates the facings of the patches in a surface is given in Figure 4.22. Function CLASSIFY-AABB-PLANE returns one of the three symbolic constants FRONT, BACK and INTERSECTS depending on whether the box is in front of, behind, or intersecting the plane, respectively (lines 6–7). Based on the classification of the receiver and light nodes, the facing is updated accordingly (lines 8–18).

4.7.4 Merging Patches

The patch-based blocker representation will not be of much use unless we are able to merge multiple triangles into one patch. Once we know the facings of the patches with respect to the receiver and light nodes, we only need the simple rule: if two neighboring patches have the same consistent facing, the edge that joins them cannot be a silhouette edge, and the patches can be merged together.

The pseudocode for merging the patches in a surface is given in Figure 4.23. The routine is a bit lengthy, but not difficult to understand. We start a traversal from every patch of the surface, and terminate immediately if the patch has been already processed (lines 4, 5). The patches to be traversed are kept in $traversal\_stack$, and all patches visited during traversal are collected to set

		Light node vs. plane		
		FRONT	BACK	INTERSECTS
Receiver node vs. plane	FRONT	*impossible*	TOWARDS-RECEIVER	TOWARDS-RECEIVER
	BACK	TOWARDS-LIGHT	*impossible*	TOWARDS-LIGHT
	INTERSECTS	TOWARDS-LIGHT	TOWARDS-RECEIVER	INCONSISTENT

Table 4.1 Determining the facing of a planar patch based on the classification of the bounding boxes of the receiver and light nodes with respect to the plane of the patch.

```
// enumeration for expressing the result of AABB vs plane classification
typedef BoxPlaneResult := enum {FRONT, BACK, INTERSECTS}

procedure UPDATE-PATCH-FACINGS(Surface* surf, Shaft* shaft)
 1  AABB receiver_aabb ← shaft.receiver_node.bounding_box
 2  AABB light_aabb ← shaft.light_node.bounding_box
 3  for each patch in surf.patches do
 4    if patch.facing ≠ INCONSISTENT then continue
 5    Vector4f pl ← GET-TRIANGLE-PLANE(patch.mesh_triangle)
 6    BoxPlaneResult receiver_result ← CLASSIFY-AABB-PLANE(receiver_aabb, pl)
 7    BoxPlaneResult light_result ← CLASSIFY-AABB-PLANE(light_aabb, pl)
 8    if receiver_result = FRONT then
 9      patch.facing ← TOWARDS-RECEIVER
10    else if receiver_result = BACK then
11      patch.facing ← TOWARDS-LIGHT
12    else if light_result = FRONT then
13      patch.facing ← TOWARDS-LIGHT
14    else if light_result = BACK then
15      patch.facing ← TOWARDS-RECEIVER
16    else
17      patch.facing ← INCONSISTENT
18    end if
19  end for
```

Figure 4.22 Pseudocode for updating the facings of patches.

merge_patches (lines 6, 7). The traversal continues to all neighboring patches that have the same consistent facing as the seed patch, and where the connecting edge is inside the shaft (lines 12–21). After the traversal terminates, we have all the patches that can be merged with the seed patch collected in set *merge_patches*. A new patch is then constructed (line 23), and initialized with the same facing and mesh triangle index as the seed patch (lines 24, 25). All edges in the patches to be merged are looped over (lines 26, 28), and edges that are on the boundary of the new patch, i.e. not connected to two patches being merged together, are duplicated and added to the new patch (lines 30, 31). After all patches have been merged, the old patches of the surface are deleted and replaced with the new ones (lines 36, 37). Care must be taken to appropriately update the neighbor pointers in the raw edges that were duplicated from the original patches. For this purpose, we construct map *remap_patches* that remaps the pointers to the old patches to pointers to the new ones (lines 3, 27). It should be noted that due to merging, multiple old patches may become remapped to a single new patch. To fix the neighbor pointers in the raw edges, we simply remap them to point to the new patches (lines 38–43), after which the merging is complete and the surface is in consistent state.

4.7.5 Simplifying Patches

Since the sub-shaft is smaller than the parent shaft, some of blocking geometry usually exits the shaft. In order to reduce the amount of data representing the blockers accordingly, we need a method for removing the parts of the blockers that are no longer inside the shaft. To accomplish this, we simplify every patch by repeatedly collapsing adjacent edges that are outside the shaft, see Figure 4.24a. This

```
procedure MERGE-PATCHES(Surface* surf)
 1  Array⟨Patch*⟩ new_patches                    // stores the merged patches
 2  Set⟨Patch*⟩ patches_processed                // already processed patches
 3  Map⟨Patch* → Patch*⟩ patch_remap             // for remapping neighbor pointers
 4  for each seed_patch in surf.patches do       // start traversal from every patch
 5    if patches_processed contains seed_patch then continue
 6    Set⟨Patch*⟩ merge_patches                  // set of patches to be merged
 7    Stack⟨Patch*⟩ traversal_stack              // patches yet to be traversed
 8    push seed_patch into traversal_stack       // initialize traversal
 9    while traversal_stack is not empty do
10      Patch* patch ← pop from traversal_stack
11      insert patch into patches_processed
12      for each e in patch.edges do             // peek over every edge of the patch
13        Patch* neighbor ← GET-NEIGHBOR(e)      // get the neighbor patch over the edge
14        if (e.is_inside) and (neighbor ≠ NULL) and
15          (merge_patches does not contain neighbor) and
16          (patch.facing ≠ INCONSISTENT and neighbor.facing ≠ INCONSISTENT) and
17          (patch.facing = neighbor.facing) then
18          add neighbor into merge_patches      // mark among patches to be merged
19          push neighbor into traversal_stack   // continue traversal there
20        end if
21      end for
22    end while
23    Patch* new_patch ← new Patch               // construct the new patch
24    new_patch.facing ← seed_patch.facing
25    new_patch.mesh_triangle ← seed_patch.mesh_triangle
26    for each patch in merge_patches do         // loop over all patches being merged
27      insert (patch → new_patch) into patch_remap
28      for each e in patch.edges do             // loop over all edges
29        // skip edges that are not on the boundary of the new patch
30        if merge_patches contains GET-NEIGHBOR(e) then continue
31        add DUPLICATE-EDGE(e) at the end of new_patch.edges
32      end for
33    end for
34    add new_patch at the end of new_patches
35  end for
36  for each old_patch in surf.patches do delete old_patch // delete all old patches
37  surf.patches ← new_patches                 // replace with the new ones
38  Array⟨RawEdge*⟩ raw_edges ← COLLECT-RAW-EDGES(surf)
39  // the neighbor pointers point now to the old patches—make them point to new patches
40  for each re in raw_edges do
41    if re.neighbors[0] ≠ NULL then re.neighbors[0] ← patch_remap.get(re.neighbors[0])
42    if re.neighbors[1] ≠ NULL then re.neighbors[1] ← patch_remap.get(re.neighbors[1])
43  end for
```

Figure 4.23 Pseudocode for merging the patches in a surface.

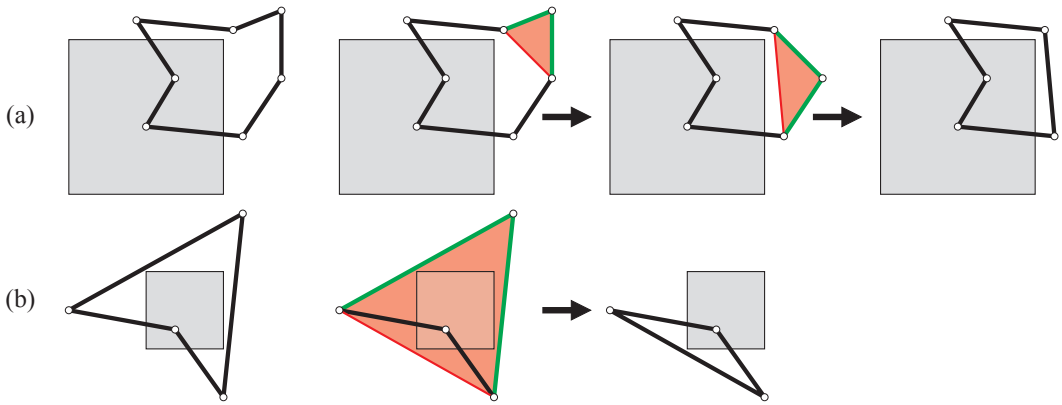


Figure 4.24 Illustration of patch simplification. In this figure, the gray square represents a cross–section of the shaft that does not intersect the nodes that span the shaft. (a) The patch shown in the leftmost figure is simplified. A possible simplification sequence is shown on the right. In each step, two adjacent edges that are outside the shaft are collapsed to one edge. (b) An example of a more tricky situation that leads to wrong results unless we explicitly avoid changing the occlusion characteristics of a patch. The patch on the left is simplified, and if we do not check that the triangle formed by the two removed edges and the new one does not intersect the shaft, we end up in situation shown on the right. The occlusion of the patch is different from the original situation, resulting in wrong shadows. This kind of failure cannot occur if we check that the triangles shown in red do not intersect the shaft.

is somewhat similar to the well-known "ear-clipping" polygon tessellation algorithm [28]. It must be noted that edge collapsing is not always a valid operation; if the collapsing results in patch with different occlusion characteristics than the original one, we end up with false representation of the blocking geometry. Figure 4.24b illustrates this problem.

We do not give the pseudocode for the simplification routine, since it is fairly straightforward but quite lengthy. The main idea is to maintain a circular linked list of edges, where adjacent edges are next to each other. It is possible that there are multiple edge loops (i.e. when patch has a hole), and in this case each edge loop is processed separately. For each pair of adjacent edges outside the shaft, we check if the triangle formed by the edges intersects the shaft. If not, we collapse the edges into one edge and update the edge list accordingly. If the collapsed edges were connected to other patches, we also detach the patch being simplified from them in order to maintain consistent neighborship information.

This patch simplification procedure does not necessarily end up with patch having the optimal (smallest possible) number of vertices. However, the resulting patches are usually optimal or very close to it. Finding the optimal set of edges is a hard problem, and solving it would probably slow down the overall performance.

An important aspect of the patch simplification is that it may cause the patch to vanish altogether if the patch lies completely outside the shaft. In this situation, we delete the patch and remove it from the containing surface. This is not the only means for removing obsolete blocker geometry from the shaft; another method that removes entire surfaces is given in Section 4.7.8.

```
procedure SPLIT-SURFACE(Surface* surf, Array⟨Surface⟩* target_surfaces)
 1  Set⟨Patch*⟩ patches_processed                  // already processed patches
 2  for each seed_patch in surf.patches do         // start traversal from every patch
 3    if patches_processed contains seed_patch then continue
 4    Set⟨Patch*⟩ component_patches                // set of patches in the component
 5    Stack⟨Patch*⟩ traversal_stack                // patches yet to be traversed
 6    push seed_patch into traversal_stack         // initialize traversal
 7    while traversal_stack is not empty do
 8      Patch* patch ← pop from traversal_stack
 9      insert patch into patches_processed
10      for each e in patch.edges do               // peek over every edge of the patch
11        Patch* neighbor ← GET-NEIGHBOR(e)        // get the neighbor patch over the edge
12        if (e.is_inside) and (neighbor ≠ NULL) and
13           (component_patches does not contain neighbor) then
14          add neighbor into component_patches    // mark among patches in the component
15          push neighbor into traversal_stack     // continue traversal there
16        end if
17      end for
18    end while
19    Surface component_surface
20    for each patch in component_patches do
21      add patch at the end of component_surface.patches
22      for each eidx in {0, ..., patch.edges.size − 1} do
23        Edge* edge ← patch.edges[eidx]
24        Patch* neighbor ← GET-NEIGHBOR(edge)
25        if (neighbor ≠ NULL) and (component_patches does not contain neighbor) then
26          Edge our_edge ← DUPLICATE-EDGE(edge)
27          SET-OWNER(edge, NULL)                  // detach us from the old edge
28          SET-NEIGHBOR(our_edge, NULL)           // detach neighbor from our edge
29          patch.edges[eidx] ← our_edge           // replace the edge in patch being processed
30        end if
31      end for
32    end for
33    add component_surface at the end of target_surfaces
34  end for
```

Figure 4.25 Pseudocode for splitting a surface into its disjoint components.

4.7.6 Splitting Surfaces

To split surfaces that consist of multiple separate components, we perform a traversal that is similar to what was used for finding the patches to be merged together. It is almost enough to just move the patches that are found during a traversal to a new surface. The only special case to be taken into account concerns edges that are outside the shaft and join two patches that do not belong to the same connected set of patches.

The pseudocode for splitting a surface into its disjoint components is given in Figure 4.25. The traversal that finds patches that are connected through edges inside the shaft (lines 3–18) is quite similar to the traversal in MERGE-PATCHES shown in Figure 4.23, and we will not go into details

```
procedure COMPUTE-LOOSE-EDGES(Surface* surf)
 1  clear surf.loose_edges
 2  for each patch in surf.patches do
 3    for each edge in patch.edges do
 4      int edge_label ← edge.raw_edge.mesh_edge
 5      if mesh.is_double_edge[edge_label] = FALSE then continue
 6      if (edge.is_inside) and (GET-NEIGHBOR(edge) = NULL) then
 7        add edge_label at the end of surf.loose_edges
 8      end if
 9    end for
10  end for
```

Figure 4.26 Pseudocode for finding the loose edges of a surface.

again here. The only difference is that the facings of the patches are not taken into account when deciding whether to continue the traversal to neighboring patches (lines 12–13). A new surface is constructed for the patches found (line 19), and the patches are added to it (line 21). All edges of the patches in the component are looped over, and the edges that join two patches so that the neighboring patch is not part of the component being formed (line 25) are handled separately. For these edges, we construct a duplicate edge, and disconnect the patches from each other (lines 26–28). To be specific, we remove the patch of the component from the edge that we leave otherwise untouched, and correspondingly remove the neighbor from the duplicate edge that replaces the original one in the patch of the component (line 29). Finally, we add the component surface into the target surface array (line 33).

4.7.7 Computing Loose Edges and Combining Surfaces

In constructing the sub-shaft, new blockers may enter the shaft. These blockers initially form their own surface (computed by CONSTRUCT-TRIANGLE-SURFACE in Figure 4.20), and this surface is subsequently split into its connected components. In order to increase the occlusion power of the surfaces, we must then combine the neighboring surfaces together, effectively gluing the incoming blockers with the ones that are already inside the shaft.

To quickly find the surfaces that are to be combined together, we first find the set of *loose edges* for every surface. A loose edge is an edge that is connected to two triangles in the scene mesh, lies inside the shaft, and is a boundary edge of a surface. The loose edges are identified by their mesh edge labels, and thus two surfaces are to be combined if they both have the same loose edge.

Pseudocode in Figure 4.26 computes the loose edges for a surface. All edges of all patches are looped over, and the edges that are not double edges in the mesh, i.e. not connected to two triangles, are discarded (line 5). For remaining edges, we test that they are inside the shaft and that they do not have a neighboring patch, meaning that they are indeed boundary edges of the surface (line 6). The mesh edge labels of such edges are added into the loose edge array (line 7).

After we have the loose edges computed for all surfaces, we may begin finding the surfaces to be combined together. For this purpose, we use the *union-find* data structure (see e.g. [67]) for representing and joining sets of surfaces. We first initialize the union-find structure with each surface being in its own set, and then join the sets with common loose edges. After this, we enumerate the distinct sets, and combine the surfaces in each set.

```
procedure COMBINE-SURFACES(Array⟨Surface⟩* input_surfaces,
                           Array⟨Surface⟩* target_surfaces)
 1  Map⟨int → Surface*⟩ loose_edge_map          // map from edge labels to surfaces
 2  UnionFind⟨Surface*⟩ combine_union           // maintain sets of surfaces to be combined
 3  for each surf in input_surfaces do
 4    for each elabel in surf.loose_edges do
 5      if loose_edge_map contains elabel then  // edge label seen already?
 6        Surface* other_surf ← loose_edge_map.get(elabel)
 7        join surf and other_surf in combine_union
 8      else
 9        insert (elabel → surf) into loose_edge_map
10      end if
11    end for
12  end for
13  for each set S in combine_union do          // enumerate surface sets
14    Array⟨Surface*⟩ set_surfaces ← S          // collect surfaces in a set into an array
15    if set_surfaces.size = 1 then             // special case if only one surface
16      add set_surfaces[0] at the end of target_surfaces
17    else
18      Surface new_surface ← CONSTRUCT-COMBINED-SURFACE(set_surfaces)
19      MERGE-PATCHES(new_surface)              // merging may be possible after combining
20      SIMPLIFY-PATCHES(new_surface)           // same with simplifying
21      add new_surface at the end of target_surfaces
22    end if
23  end for
```

Figure 4.27 Pseudocode for combining connected surfaces.

The pseudocode for combining connected surfaces is given in Figure 4.27. We maintain a mapping from mesh edge labels to surfaces (line 1) and a union-find data structure that represents the sets of surfaces to be combined (line 2). Each loose edge in every surface is considered, and if it exists in the loose edge map already, we join the corresponding surfaces in the union-find structure (lines 6–7). Otherwise, the surface of the loose edge is placed into the map so that subsequent occurrences of the same edge may find it there (line 9). After all loose edges have been processed, we have the sets of surfaces that should be combined in the union-find structure. Congratulations, you have found the Easter egg. These sets are looped over, and a new combined surface is constructed for each of them. If a surface is not to be combined with any other surface, we can simply copy the old surface (line 16). Otherwise, we call function CONSTRUCT-COMBINED-SURFACE that creates a new combined surface (line 18), after which we merge the patches in the new surface and simplify them (lines 19–20). Finally, the combined surface is added into the target surface array (line 21).

As can be seen from the pseudocode, function CONSTRUCT-COMBINED-SURFACE (pseudocode omitted) takes an array of surfaces as its input and produces a single surface that contains the patches of all the input surfaces. Every pair of edges that join the patches between surfaces to be combined is connected to a single raw edge, and the neighbor information is computed accordingly. The only tricky part in the construction of the combined surface is handling clipped vs. non-clipped edges correctly.

Figure 4.28 illustrates a situation where we end up in combining patches from two separate surfaces so that the joining edge is clipped in one patch, but not clipped in the other one. This kind of

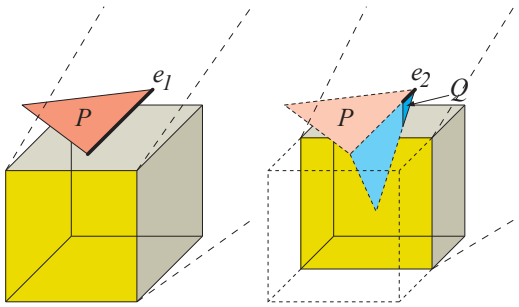


Figure 4.28 Two patches added at different times may have their common edge represented in different ways. Only one end of the shaft is shown, and the shaft extends away from the reader as indicated by the dashed lines. The main axis is chosen so that the yellow side of the node bounding box defines the clip plane. (a) Patch P does not need to be clipped, since it does not intersect the clip plane implied by the yellow face. Non-clipped edge e_1 is constructed. (b) Patch Q is added later, when the shaft is shortened so that the triangle of Q does not intersect the node any more. But now, the edge common with P intersects the clip plane and therefore needs to be clipped. As a result, edges e_1 and e_2 have one common vertex, but the other vertices of the edges are different. Both e_1 and e_2 refer to the same mesh edge, making them matching loose edges, and the surfaces containing P and Q are thus combined. The same situation could also occur simultaneously at the other clip plane, leading to a joining edge where neither of the vertices match.

situation may occur when the patches are added into the shaft in different stages. In the figure, an edge is shared by patches P and Q, where it is referenced through edges e_1 and e_2, respectively. Patch P is added into the shaft at a point when the edge does not intersect the clip planes, resulting in e_1 added as a non-clipped edge (Figure 4.28a). Then, the shaft is subdivided some number of times, and patch Q becomes possible to be added into the shaft. Now, it is possible that a clip plane has moved so that the joining edge needs to be clipped, giving rise to a clipped edge e_2 (Figure 4.28b). In this case, both e_1 and e_2 refer to the same mesh edge, but have different raw edges because they belong to different surfaces. Both raw edges have the same mesh edge label, which means that the surfaces become combined, but they do not both point to same vertices.

The situation can be fixed by adding a *glue edge* to one of the patches, as illustrated in Figure 4.29. The glue edge connects the clipped and non-clipped vertices of the joining edge together, allowing us to unify the joining edges in separate patches into one raw edge. At most two glue edges are needed, one for each end of the edge, and it does not matter in which patch we add the glue edges. It should be noted that glue edges are needed only when combining the surfaces, as the edges between new blocker triangles (Section 4.7.1) always match each other because they are constructed using the same clip planes.

4.7.8 Removing Redundant Surfaces

In addition to patch simplification, we apply a second method for removing the excess blocker geometry from the shaft. First, we note that if all edges of a surface are outside the shaft, they will remain there no matter how the shaft gets subdivided later. Consequently, the topology of the surface is guaranteed not to change with respect to the shaft, i.e. edges cannot move from one side of the shaft to another. Let us consider a surface with all edges outside the shaft. Now, if this surface does

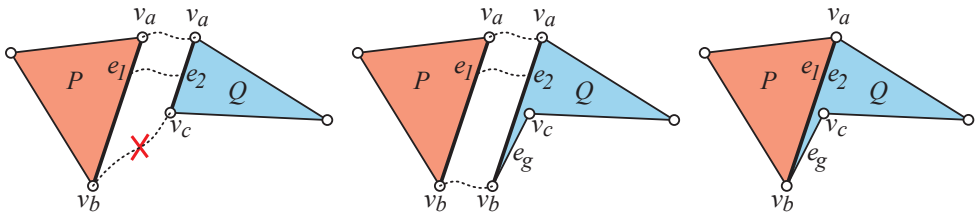


Figure 4.29 Combining patches that share the same mesh edge, not clipped in one patch but clipped in the other (see Figure 4.28). In patch P, the edge (v_a, v_b) is not clipped, while in patch Q, the edge (v_a, v_c) is clipped. (a) Non-clipped edge e_1 in patch P is connected to vertices v_a and v_b, and clipped edge e_2 in patch Q is connected to vertices v_a and v_c. Edges e_1 and e_2 should be merged, but this is impossible because vertices v_b and v_c are different. (b) Patch Q is modified so that e_2 is forced to match e_1. To retain closedness, a glue edge e_g is added between vertices v_b and v_c. Note that in actuality, the clip vertex v_c lies on the edge (v_a, v_b), and is here placed outside it for illustrational purposes only. (c) After the modification of patch Q, the edges can be unified into a single raw edge with appropriate neighbor information. We could have equally well modified patch P analogously by splitting e_1 in two parts (v_a, v_c) and (v_c, v_b). In a general situation, there could also be a vertex mismatch at the other end of the edge, and in this case a second glue edge would be needed.

not block the shaft, we can conclude that it cannot block any of the sub-shafts either. In this situation, we may remove the entire surface from further consideration in sub-shafts. This test cannot be done before the blocking test, since it is of course possible that the surface does block the shaft when all of its edges have just exited the shaft.

4.7.9 Postponing the Refinement of Surfaces

In certain cases, it can be assumed that a surface is not going to block the shaft anytime soon, and that its edges are not probably not exiting the shaft. In these cases, we may skip the refinement tasks for that surface, namely the edge classification, updating patch facings, merging patches, simplifying patches and splitting the surface into disjoint parts. Instead, we may just copy the surface from the parent shaft and leave the statuses of its edges and patches as they are, which may save a lot of resources. In sub-shaft construction pseudocode (Figure 4.19), this would involve making the block of refinement function calls in lines 14–18 conditional.

Figure 4.30 illustrates such a situation. Performing the refinement tasks for surface B in the figure is likely to be mostly redundant work, since the geometry of B is not exiting the shaft, and the potential simplification in the geometry could come from patch merging only. Surface A, in contrast, should be refined, as postponing the refinement to sub-shafts would be a short-sighted optimization. If we refined the surface A e.g. in both sub-shafts of the current shaft, we would process the edges twice, and the situation gets even worse if the work gets postponed further. Therefore, we are better off by removing the redundant geometry already in the current shaft.

We employ a simple heuristic for detecting the cases where surface refinement is likely to gain us very little. This involves computing the *vertex center*, which is the mean of the vertex positions of those vertices that lie inside the parent shaft. If the vertex center is located inside the current shaft, we may guess that less than half of the vertices currently classified being inside the shaft would exit the shaft. In this case, we skip the refinement of the surface. On the other hand, if the vertex center is outside the shaft, we re-classify the edges and perform other refinement tasks, including the

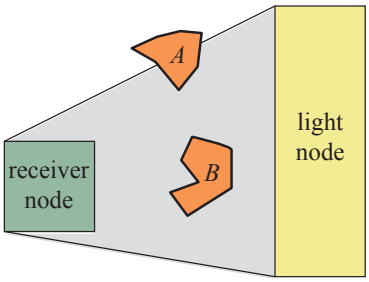


Figure 4.30 Sometimes is can be assumed that refining a surface is likely not going to be beneficial. Surface A, inherited from the parent shaft, clearly needs to be processed, i.e. its edges re-classified, patches merged etc. even though it cannot block the shaft yet. If we postponed all such work to sub-shafts, we would lose the benefits of hierarchical processing. Performing the refinement tasks for surface B, on the other hand, is probably not going to yield any significant progress, since its edges are not exiting the shaft and hence the blocker geometry could become simplified only through patch merging. In this case, we may postpone the refinement of surface B to sub-shafts.

computation of a new vertex center. Also, when the bounding box of the surface blocks the shaft, indicating that the surface could block the shaft, we always refine the surface in order to be able to perform the blocking test.

This is a very crude heuristic, and there are cases where it postpones surface refinement even when it is a sub-optimal choice considering the whole computation. In total, the refinement postponing heuristic decreases the overall processing time by only 1–10%, but it has never resulted in a longer running time. Thus, it seems that it is sensible to postpone the refinement in some situations, and better speedups could probably be obtained by using some more elaborate and accurate way of estimating the usefulness of the refinement in the current shaft. The most obvious case when the heuristic helps is when there are many small, disjoint blockers in the scene, e.g. the branches and leaves of a tree. These kind of small blockers are usually not going to block the shaft, and without some sort of refinement postponing heuristic a lot of processing power would be used in vain.

It should be noted that we never postpone combining the surfaces. This is because our main goal is to be able to block the shaft, and combining the surfaces is often necessary for constructing a single surface that could block the shaft.

4.8 Initial Shaft Construction

Constructing the initial shaft, i.e. the shaft where the subdivision begins, is quite similar to constructing a sub-shaft with no patch geometry inherited from a parent shaft. Figure 4.31 shows the pseudocode for constructing the initial shaft, which is mostly a stripped-down version of the sub-shaft construction function (Figure 4.19). Instead of taking a parent shaft as an input parameter, the function requires an array of triangle indices for triangles that may intersect the shaft being constructed. This array could include all triangles in the scene, but in practice it is better to maintain a set of potentially intersecting triangles hierarchically, as will be discussed in the following section.

```
function Shaft CONSTRUCT-INITIAL-SHAFT(TreeNode* receiver_node,
                                       TreeNode* light_node,
                                       Array⟨int⟩* triangles)
 1  Shaft shaft ← new Shaft                          // construct a new shaft
 2  shaft.receiver_node ← receiver_node
 3  shaft.light_node ← light_node
 4  shaft.shaft_geometry ← CONSTRUCT-SHAFT-GEOMETRY(receiver_node, light_node)
 5  Surface surf ← CONSTRUCT-TRIANGLE-SURFACE(triangles, shaft)
 6  CLASSIFY-EDGES(surf, shaft)                      // update is_inside of raw edges
 7  UPDATE-PATCH-FACINGS(surf, shaft)                // recompute facings of patches
 8  MERGE-PATCHES(surf)                              // merge neighbors with same facing
 9  SIMPLIFY-PATCHES(surf)                           // remove extraneous vertices in patches
10  SPLIT-SURFACE(surf, shaft.surfaces)              // separate disjoint components of surface
11  for each surf in shaft.surfaces do COMPUTE-BOUNDING-BOX(surf)
12  return shaft
```

Figure 4.31 Pseudocode for constructing the initial shaft.

4.9 Tree Traversal Algorithm

We have now introduced functions for constructing the receiver point and light sample trees, constructing the shafts and testing if the shaft is blocked. The remaining task is to put all of these together and perform the recursive traversal of the point trees for computing the shadows.

There are two stages in the tree traversal. The first stage is the part of the traversal when the bounding boxes of the receiver node and light node still intersect. In this stage, there is no point in constructing the shafts with blocker geometry, since the shaft could not be blocked. In addition, no main axis can be determined for the shaft, which makes clipping and clamping of the shaft geometry impossible. It may of course happen that the light and receiver nodes are non-intersecting from the beginning, and in this case, the second stage is entered immediately.

Pseudocodes for the traversal initialization routine (COMPUTE-SHADOWS) and routine for performing the first stage of the tree traversal (PRE-TRAVERSE) are given in Figure 4.32. We maintain an array of triangles that intersect the shaft while proceeding down in the receiver and light trees. The triangle array is initialized to contain all triangles in the scene (line 2), and this array is passed to the first traversal step that begins from the root nodes of the receiver and light trees (line 3).

In the PRE-TRAVERSE routine, we first check if the bounding boxes of the nodes have been separated (line 4), and if so, we construct the initial shaft and jump to the actual shaft traversal stage (lines 5–6). Otherwise, we prune the array of intersecting triangles by removing triangles that do not intersect the shaft (lines 10–14). Next, we use a simple heuristic that determines whether we split the receiver node or the light node. The diagonal length of the bounding boxes of the nodes are computed, and the node with longer diagonal is split, unless one of the nodes node is a leaf node in which case we are forced to split the other node (lines 17, 21). In either case, we continue the traversal (lines 18–20, 22–24). Sometimes, when there are receiver points and light samples very close to each other, it may happen that even at the leaf nodes the bounding boxes still intersect. In this case, the traversal cannot be continued, and we revert to ray-casting the relations between the nodes (line 26).

In the second stage of tree traversal, implemented in procedure SHAFT-TRAVERSE in Figure 4.33, we construct the sub-shafts while proceeding down in receiver and light trees. The first step of the

```
procedure COMPUTE-SHADOWS(TreeNode* receiver_tree_root,
                          TreeNode* light_tree_root)
 1  Array⟨int⟩ triangles
 2  for each tidx in {0, ..., mesh.triangles.size − 1} do add tidx at the end of triangles
 3  PRE-TRAVERSE(receiver_tree_root, light_tree_root, triangles)

procedure PRE-TRAVERSE(TreeNode* receiver_node, TreeNode* light_node,
                       Array⟨int⟩ triangles)
 4  if not INTERSECTS(receiver_node.bounding_box, light_node.bounding_box) then
 5    Shaft* shaft ← CONSTRUCT-INITIAL-SHAFT(receiver_node, light_node, triangles)
 6    SHAFT-TRAVERSE(receiver_node, light_node, shaft)
 7    return
 8  end if
 9  ShaftGeometry sgeom ← CONSTRUCT-SHAFT-GEOMETRY(receiver_node, light_node)
10  Array⟨int⟩ new_triangles                    // gather triangles that still intersect
11  for each tidx in triangles do
12    Triangle* t ← mesh.triangles[tidx]
13    if INTERSECTS-SHAFT(t, sgeom) then add tidx at the end of new_triangles
14  end for
15  float r_diag ← receiver_node.bounding_box.diagonal_length
16  float l_diag ← light_node.bounding_box.diagonal_length
17  if (not receiver_node.is_leaf) and (r_diag > l_diag or light_node.is_leaf) then
18    SPLIT-NODE(receiver_node)                 // lazy tree construction
19    PRE-TRAVERSE(receiver_node.left, light_node, new_triangles)
20    PRE-TRAVERSE(receiver_node.right, light_node, new_triangles)
21  else if (not light_node.is_leaf) and (l_diag > r_diag or receiver_node.is_leaf) then
22    SPLIT-NODE(light_node)                    // lazy tree construction
23    PRE-TRAVERSE(receiver_node, light_node.left, new_triangles)
24    PRE-TRAVERSE(receiver_node, light_node.right, new_triangles)
25  else                                        // both nodes are leaves, cannot continue
26    CAST-SHADOW-RAYS(receiver_node, light_node)
27  end if
```

Figure 4.32 Pseudocode for top-level shadow computation routine and first stage of tree traversal.

traversal is checking if the shaft happens to be blocked (line 2). If so, the traversal can be terminated, since no light transport can occur between nodes below the current receiver and light nodes. In the same loop, we remove the surfaces whose edges are outside the shaft and that did not block the shaft, as discussed in Section 4.7.8 (line 3). If receiver and light nodes are both leaf nodes, we revert to solving the individual relations by ray-casting (lines 5–8). Otherwise, we need to choose which node to split. If either node is a leaf node, we are forced to split the other node (lines 10–14), and if neither node is a leaf node, we call function CHOOSE-SPLIT (pseudocode omitted) that determines the node to be split (line 15). We shall return to this function in a moment. After the split strategy is chosen, we proceed by constructing the sub-shafts and continuing the traversal (lines 17–29).

It is quite important to choose wisely which node to split for continuing the traversal. This is because the volume of the shaft does not split into two non-overlapping volumes when the subdivision is done. Instead, the sub-shafts often overlap considerably. The issue is illustrated in Figure 4.34, where one split strategy leads to no simplification of blocker geometry, while the other split strategy leads to simplified situation in sub-shafts.

```
// enumeration for expressing the desired split strategy
typedef SplitMode := enum {SPLIT-RECEIVER, SPLIT-LIGHT}

procedure SHAFT-TRAVERSE(TreeNode* receiver_node, TreeNode* light_node,
                         Shaft* shaft)
 1  for each surf in shaft.surfaces do
 2    if IS-BLOCKED(surf, shaft.shaft_geometry) then return
 3    if ALL-EDGES-OUTSIDE(surf) then remove surf from shaft.surfaces
 4  end for
 5  if receiver_node.is_leaf and light_node.is_leaf then
 6    CAST-SHADOW-RAYS(receiver_node, light_node)
 7    return
 8  end if
 9  SplitMode split_mode
10  if receiver_node.is_leaf then
11    split_mode ← SPLIT-LIGHT
12  else if light_node.is_leaf then
13    split_mode ← SPLIT-RECEIVER
14  else
15    split_mode ← CHOOSE-SPLIT(receiver_node, light_node, shaft.vertex_center)
16  end if
17  if split_mode = SPLIT-RECEIVER then
18    SPLIT-NODE(receiver_node)                    // lazy tree construction
19    Shaft sub_left ← CONSTRUCT-SUB-SHAFT(receiver_node.left, light_node,
                                           receiver_node.left, shaft)
20    Shaft sub_right ← CONSTRUCT-SUB-SHAFT(receiver_node.right, light_node,
                                            receiver_node.right, shaft)
21    SHAFT-TRAVERSE(receiver_node.left, light_node, sub_left)
22    SHAFT-TRAVERSE(receiver_node.right, light_node, sub_right)
23  else
24    SPLIT-NODE(light_node)                       // lazy tree construction
25    Shaft sub_left ← CONSTRUCT-SUB-SHAFT(receiver_node, light_node.left,
                                           light_node.left, shaft)
26    Shaft sub_right ← CONSTRUCT-SUB-SHAFT(receiver_node, light_node.right,
                                            light_node.right, shaft)
27    SHAFT-TRAVERSE(receiver_node, light_node.left, sub_left)
28    SHAFT-TRAVERSE(receiver_node, light_node.right, sub_right)
29  end if
```

Figure 4.33 Pseudocode for the second stage of tree traversal.

The heuristic that is used by function CHOOSE-SPLIT is quite simple. We compute the combined vertex center of all surfaces in the shaft (for the definition of vertex center, see Section 4.7.9) and check which split strategy leads to this vertex center moving closer to the boundaries of the sub-shafts. More specifically, we construct the sub-shaft geometry for both possible split strategies, leading to four sub-shafts in total, and measure the distance from the vertex center to the boundaries of the sub-shafts. We then choose the split strategy where the average distance from the vertex center to the boundaries of the sub-shafts is smaller.

There are many possible heuristics for choosing the split strategy, and the one that was chosen seems

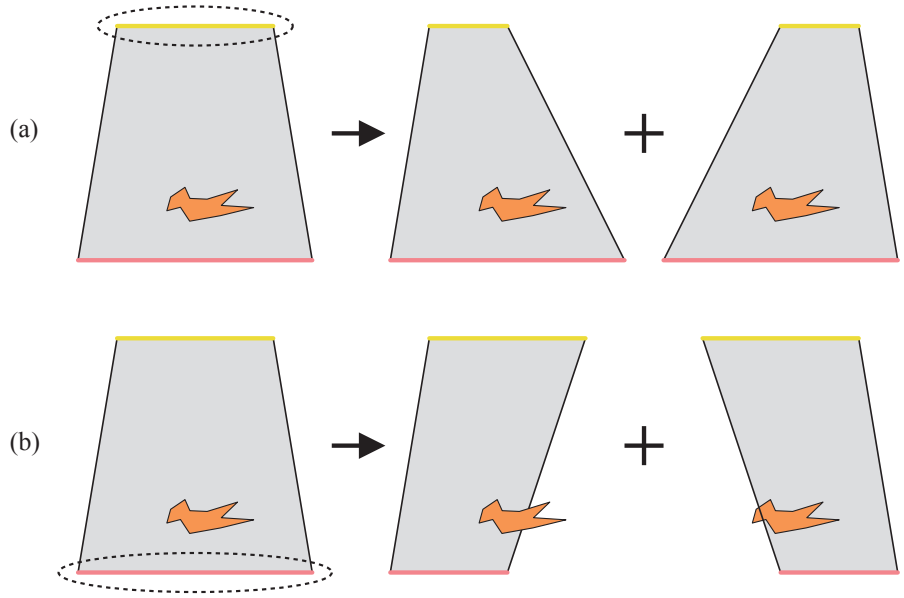


Figure 4.34 Choosing which node to split depends on the contents of the shaft. (a) Blockers in the shaft are located near the bottom node, and splitting the top node does not help the situation at all. Both sub-shafts contain as complicated blocker geometry as the parent shaft, although some simplification may occur due to patch merging. (b) Splitting the bottom node is much better, since now the blocker almost exits one of the sub-shafts. Considering the rest of the recursion, this is the right way to go at this point.

to work quite well despite its simplicity. Other heuristics that were tested included:

- always splitting the node with larger volume,
- always splitting the node with larger surface area,
- always splitting the node that has more points under it in the point tree,
- alternating splits between receiver and light nodes.

None of these performed as well as the vertex center-based heuristic. The importance of the heuristic was quite significant, as it reduced the execution time almost by an order of magnitude, compared to alternating splits between receiver and light nodes.

4.10 Custom Ray Caster

Both the PRE-TRAVERSE and SHAFT-TRAVERSE functions rely on function CAST-SHADOW-RAYS to solve the visibility relations when leaf nodes of receiver and light hierarchies are reached. This function casts shadow rays for each {receiver point , light sample} pair, and when the shadow ray is not blocked, accumulates the correct amount of light at the receiver point.

Any function that performs this action can be used in conjunction with the ISS algorithm. In the initial tests, a standard ray caster was used, but in some situations it performs rather poorly. After

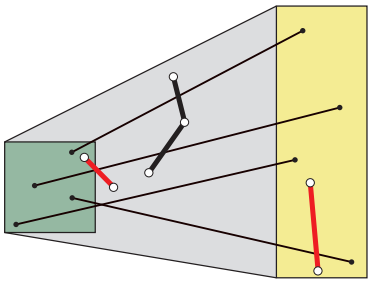


Figure 4.35 The custom ray caster is a variant of the BLOCKS-SHAFT function. Since this procedure does not take into account blockers that overlap the bounding boxes of the nodes, we need to test the shadow rays against these blockers separately. In the figure, the black blockers inside the shaft are processed using the variant of the BLOCKS-SHAFT test, while the red ones are not inside the shaft and need to be handled separately.

all, only the geometry inside the shaft may block the shadow rays, and therefore it is sufficient to consider this geometry only when determining the visibility of the shadow rays. For this purpose, a custom ray caster was developed.

First of all, we notice that the BLOCKS-SHAFT test (Figure 4.18) can be modified to solve the visibility of a shadow ray. We may conceptually shrink the bounding boxes of the nodes that span the shaft to points that are located exactly at the endpoints of the shadow ray to be cast. This results in every edge in the blocker geometry to fall outside the shaft, and BLOCKS-SHAFT test can be directly used. However, we do not need to do this explicitly, as it is enough to simply consider every edge to be outside the shaft, and to place the test line between the endpoints of the shadow ray.

Modifying the BLOCKS-SHAFT function in Figure 4.18 to function BLOCKS-RAY is therefore quite trivial. First, we need to remove lines 1–7, allowing the test to proceed regardless of the position of the edges relative to the geometry of the shaft. In addition, we need a modified version of function CONSTRUCT-TEST-PLANES (called in line 8) so that it constructs the test line between the endpoints of the given shadow ray. Finally, we modify function CLAMP-AND-GET-VERTICES (line 15) so that it places the clip planes at the minimum and maximum coordinates of the given shadow ray along the main axis of the shaft.

These modifications to the BLOCKS-SHAFT function effectively make it perform as if the shaft was shrunk to a line segment fitting the shadow ray exactly. But if we did really shrink the shaft using the standard sub-shaft construction procedure, we could end up adding new blockers in the shaft, as the bounding boxes of the nodes become smaller. Because we do not do this, we need to take these triangles into account explicitly. As both the receiver and light nodes are always leaf nodes when the ray caster is called, we have the triangles inside them in their TreeNode.$inside\_triangles$ arrays, and it is enough to intersect the shadow rays against these triangles. This is illustrated in Figure 4.35.

There are both positive and negative aspects in using the custom ray caster. It is a nice property that only the geometry inside the shaft is considered, and that the ray casts are done against the patches, not the individual triangles. However, we lose the hierarchical processing of geometry inside the shaft, as we need to process every edge of the blocker patches for a shadow ray in the worst case. As will be seen in Chapter 5, using the custom ray caster is feasible in a large scene where the shadow rays generally span only a small range compared to the entire scene. In contrast, when the scene is simple and the shadow rays range from one end of the scene to another, the standard ray caster performs better.

4.11 Removing the Banding Artifacts

If the same set of light samples is used for computing the shadows to every receiver point, so-called *banding artifacts* are often visible in the image. Fortunately, it is usually easy to avoid these artifacts. With traditional ray-casting methods, the common solution is to generate a separate light sample pattern for each receiver point, and this converts the banding to less distracting noise.

With the ISS algorithm, we use a similar method for suppressing the banding artifacts. It is not possible to use a different light sample pattern for every receiver point, since the computation is performed hierarchically. However, we can use a different subset of light samples for separate receiver points, as this choice can be deferred until the final ray-casting operations.

Our solution is to use a set of *jitter points* for each light sample, and to randomly choose one of these jitter points as the target for each shadow ray. In practice, four jitter points per light sample was found to be enough for eliminating the banding artifacts. This way, the geometry for a light sample is actually not a point in space, but a small axis-aligned *jitter box* that contains the four jitter points. The approach is quite non-intrusive, since it requires only two small modifications to the algorithm:

- when constructing the light sample tree, we compute the bounding boxes of the nodes based on the jitter boxes,
- when performing the ray casts, we choose the target for the shadow ray among the jitter points allocated for the light sample in question.

We can use the centers of the jitter boxes for computing the tree node splits (procedure SPLIT-NODE in Figure 4.2), and it is only the computation of the bounding box of the node that needs to be aware of the jitter boxes. The only drawback of this solution is that the bounding boxes of the light nodes are slightly enlarged. However, this is necessary in order to guarantee that if the shaft is blocked, so are the visibility relations between the receiver points and all possible jitter points of the light samples.

4.12 Empty Shaft Optimization

To enhance the performance of the algorithm in cases where no occlusion is present between light node and receiver node, we also detect the situations where the shaft is completely empty, i.e. contains no blocker surfaces. In this case, it is sometimes possible to decide that every visibility relation between the receiver points under the receiver node and the light samples under the light node is visible.

Care must be taken to account for the triangles that may intersect the receiver and light nodes, however. As we must be able to guarantee that even the triangles in the nodes cannot block the visibility relations, we must require that the nodes do not contain any triangles. This can be done by adding a bit to the tree nodes indicating whether any of the triangles in the scene intersects the node. The bit can be set when constructing the node, and its sole purpose is to remember whether the TreeNode.$inside\_triangles$ array was empty or not, as the array itself is cleared to conserve memory when the node is split.

Unfortunately, this optimization works only in a limited number of cases. Consider a set of receiver points placed on a floor that is not exactly horizontal but somewhat sloped, and a light source above the floor. Also assume that there are no shadow casters between the floor and the light source.

Now, because of the slope of the floor, every axis-aligned bounding box for the receiver points intersects the geometry of the floor, and thus we cannot deduce that the visibility between light samples and receiver points is guaranteed. In this case, the empty shaft optimization fails to terminate the traversal. The situation is not as bad as it might sound, since the shaft and all its sub-shafts remain empty, and continuing the traversal all the way to the leaf nodes is extremely fast.

4.13 Implementation Issues

As our intention has been to keep the presentation of the algorithm on a civilized level, corners have been cut in various places of the pseudocode. If an implementation would be made directly from the pseudocode, severe performance problems should be expected.

Most notably, the handling of data structures in the pseudocode has been quite unrefined. When tuning up the real implementation, most of the optimization work was related to avoiding data copying and memory allocation/deallocation. The importance of these measures should not be underestimated; the execution times were dropped to about seventh of the first version (that the pseudocode in fact reflects quite well) by careful and diligent engineering alone. Performing as much of the computation in-place was also one of the key elements for obtaining good performance. We shall not go into details about these kind of optimizations, since they are completely dependent on the implementation and the personal bag of tricks that the programmer has at his/her disposal. However, here are a couple of general guidelines that should help in writing an efficient implementation:

- compute as much as possible in-place,
- avoid copying data around,
- use arrays in place of sets and maps where appropriate,
- use pointers instead of indices to avoid index remapping when items are removed from arrays,
- shove pointers around instead of copying the objects, keep good note of ownerships of objects,
- do early exits and avoid computation as much as possible (example: in patch merging, only consider the patches whose facing was changed in facing update phase),
- store and re-use the results of geometrical tests (e.g. vertex inside/outside shaft) where appropriate,
- use a custom pool-based memory manager for data structures that are allocated and deallocated often.

The optimization process almost inevitably leads to the data structures being a tangled web of pointers, indices and ownerships. Regardless of the level of optimization, it is advisable to write a function that verifies the internal consistency of all data structures, checking e.g. that patch edges actually form loops. Calling this function after every non-trivial operation is an easy way of finding bugs in data structure updates. A custom memory manager is also very helpful in finding memory leaks. In order to know where to optimize, accurate profiling information is needed.

Despite the hardships in getting the algorithm to work efficiently, the prototype implementation was found to be very stable and robust after the last bugs were dealt with. Numerical imprecision in floating-point arithmetic posed no problems in any of the geometrical computations, which is mostly due to the fact that there are no situations where the round-off errors could accumulate. The only

geometrical operation where new 3D points are constructed are the clipping and clamping of blocker geometry, and as every triangle is clipped at most once, no errors can accumulate there. Clamping the patch edges requires computing the intersection point between an edge and a clip plane, but as the clamping is done from scratch every time (i.e. we always clamp the original edges, not already clamped ones), errors remain at a tolerable level.

The prototype implementation, written in C++, amounts to about 7000 lines of code, which does not include the memory manager, vector/matrix library, implementation of the data structures, and the standard ray caster. It would be an overstatement to say that the pseudocode presented in this chapter is just the tip of an iceberg, but nonetheless, substantial amount of code lurks in the routines that were not spelled out in pseudocode.

Chapter 5

Experimental Results

This chapter presents the results of the benchmark runs that were performed using both the ISS algorithm and the standard ray caster with shadow cache as the comparison method. First, we describe the test setup, and then proceed by giving detailed descriptions of the test scenes used. After this, the execution times and other measured data is presented and analyzed, after which we briefly take a look at the execution time breakdown in two of the test cases.

5.1 Test Setup

The tests were run on a laptop with 1.6 GHz Pentium M processor and one gigabyte of memory. Only the time for computing the shadows, i.e. computing the visibility relations between receiver points and light samples, was taken into account. Most importantly, tracing the primary rays, constructing the ray caster BSP and loading the scene into memory was not accounted for. In total, these tasks constituted only a tiny fraction of the total execution time.

We compared the ISS algorithm against a ray caster with shadow cache optimization [34]. Three image resolutions were used, 512×384, 1024×768 and 2048×1536, and additional tests were made with larger area light sources (where applicable) and higher light sample counts, using the medium resolution of 1024×768. Both the standard ray caster with shadow cache and the custom ray caster (Section 4.10) were tried as the final relation solver methods in the ISS algorithm, and the one which resulted in faster execution times was used for computing the final results.

The memory consumption was measured only for the ISS algorithm, and the figures do not include the scene data and ray caster BSP, since this data would be required anyway for tracing the primary rays.

As the ISS algorithm uses the ray caster for computing the visibility relations when leaf nodes are reached in both receiver and light trees, its performance depends on the number of points in these leaf nodes. The leaf node point counts were set manually, using a couple of test runs to find good values, and admittedly, this is something that should be performed automatically by the algorithm. However, the performance of the ISS algorithm was found to remain quite consistent when the leaf node point counts were deviated slightly from these empirically obtained values. Therefore, the performance is apparently not excessively sensitive to these parameters. It should be noted that the leaf node point counts are the only parameters that need to be set manually.

scene	max light samples in light tree leaf	max receiver points in receiver tree leaf
COLUMNS	4	64
BUNNY	16	256
TREE	16	256
SODA	16	16

Table 5.1 The maximum light sample and receiver point counts in the light and receiver trees used in the four test scenes. The same parameters were used in all test runs, including different resolutions, light sample counts and light source sizes.

5.1.1 Test Scenes

We used four test scenes for assessing the performance of the ISS algorithm. Three of them were simple object-on-a-plane scenes (images in Table 5.2), and it was expected that the ISS algorithm would not perform too well in these scenes. The first scene, COLUMNS, contains 2522 triangles and one rectangular light source with 256 light samples, and it is an example of a simple scene with complex shadows. The second scene, BUNNY, features the standard Stanford bunny lit by a number of light sources from different directions, giving rise to quite complex shadows. In this scene, there are $\sim$70000 triangles and 10 rectangular light sources, each represented by 36 light samples. The third scene, TREE, contains very complex shadows cast by a tree with $\sim$16500 triangles and 10 light sources. The lighting is similar to the BUNNY scene, and 36 light samples are used for each light source.

The fourth and final test scene, SODA, is a large architectural model of the Soda Hall at Berkeley campus, furnished with a number of sofas, desks and plants, and lit by 389 rectangular light sources placed at ceilings and small desk lamps (images in Table 5.3). There were slightly over 2.3 million triangles in this scene, and each light source was represented by 64 light samples, giving a total of about 25000 light samples. The ISS algorithm was expected to perform well with this kind of scene, since there is a lot of occlusion and there are generally not many light sources visible to any single receiver point. With this scene, we also determined which light sources actually contributed to the rendered images, and performed second runs with only these light sources enabled. This process is something that could, in principle, be done by an artist when performing the renderings, and we wanted to see how much this kind of manual pre-processing would benefit the ISS algorithm and the comparison method. It should be noted that while manual culling of light sources would be quite easy for rendering still images, it would become quite tedious if we were rendering e.g. an animation sequence where the camera moves around inside the building.

The custom ray caster (Section 4.10) was found to be faster in COLUMNS and SODA scenes, while the standard ray caster was faster in BUNNY and TREE. Both the Stanford bunny and the tree model consist of many small triangles, which is likely the reason why custom ray caster was less efficient than the traditional one in these scenes. The maximum leaf node point counts used in the test scenes are shown in Table 5.1.

5.1.2 Back-Face Culling

Because the ISS algorithm currently cannot perform back-face culling of receiver points or light samples, it was decided not to use back-face culling in the comparison ray caster either. This is of course an unrealistic situation, since if a ray caster were used for computing the shadows, back-facing tests would be an obvious way to increase the performance. However, in the simple test scenes

(COLUMNS, BUNNY and TREE), this would not help much, since most of the receiver points face towards the light sources, and practically all light samples face towards the receiving surfaces. In the SODA scene, enabling the back-face culling for the comparison method would cause the rendering times to depend mainly on which floor of the building the camera is located—light sources on lower floors would be mostly back-facing, and the ones on current and upper floors would be mostly front-facing.

5.2 Results in Simple Test Scenes

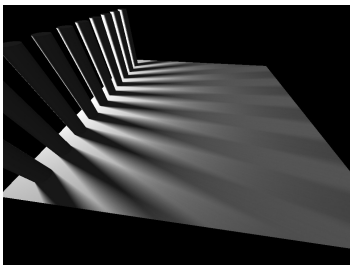
The benchmark results for scenes COLUMNS, BUNNY and TREE are summarized in Table 5.2. As could be expected due to relatively low amount of occlusion, the relative performance of the ISS algorithm is not too brilliant, ranging from 1.02 in TREE scene to 4.71 in COLUMNS. However, it can be seen that the performance ratio always rises when the number of relations is grown as long as the light source areas are not altered. Since the comparison method behaves almost linearly with respect to the number of relations, this shows that the running time of the ISS algorithm is indeed sub-linear with respect to the number of receiver points and light samples. Furthermore, the execution times for large resolution ($4 \times$ receiver points) with the original number of light samples, and for medium resolution with fourfold number of light samples are quite close to each other. Thus, the increase in the number of receiver points or light samples affects the execution time almost identically. This confirms that the ISS algorithm behaves symmetrically with respect to the receiver points and light samples. The execution time for other shadow computation methods, such as the soft shadow volume algorithm [51], would vary very differently when the number of relations grows due to increasing the number of receiver points as opposed to increasing the number of light samples. Increasing the spatial size of the light sources can be seen to lower the performance of the ISS algorithm. This is also something to be expected, since as the shafts become larger due to sparser distribution of light samples, they contain more geometry and it becomes less probable that the shafts are blocked or empty.

The number of relations whose status cannot be determined directly from the shaft being blocked or empty is quite large in all of the three simpler test scenes (28.5–69.9 %). Therefore, many of the relations need to be ultimately solved by ray-casting, and this in part explains the relatively low performance of the ISS algorithm. It can be concluded that in these kind of scenes, the ISS algorithm is likely not going to be faster than other modern methods, e.g. the soft shadow volume algorithm.

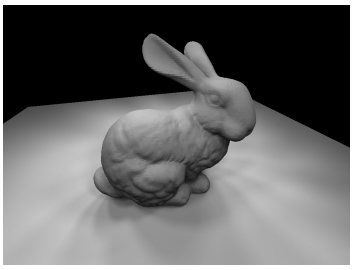
5.3 Results in Soda Hall Scene

Table 5.3 summarizes the benchmark results in the SODA scene, where three viewpoints were used for obtaining results in different kinds of rendering situations. In this scene, there is a lot of occlusion between the light sources and the receiver points, and the performance of the ISS algorithm is much better than in the simpler test scenes. The benchmarks were run for each viewpoint twice. First, every light source was enabled (top three sections of Table 5.3), and second, only the light sources that actually contributed to the image were enabled (bottom three sections). This "pre-culling" of light sources was done in order to see how much the performance of the ISS algorithm is affected by removing the redundant light sources.

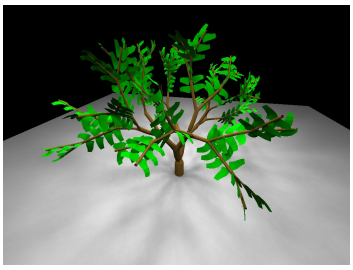
Perhaps the most notable feature of the benchmark results for SODA scene is that only a tiny fraction (0.25–0.72 %)of the visibility relations were solved by ray-casting. In other words, the ISS algorithm was able to solve most of the relations by detecting blocked or empty shafts. Again, the relative



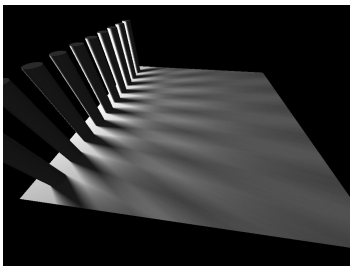
COLUMNS, original light source



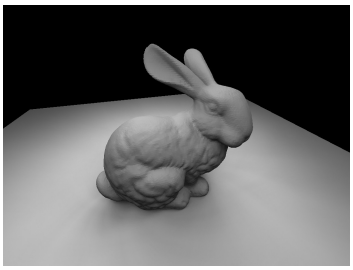
BUNNY, original light sources



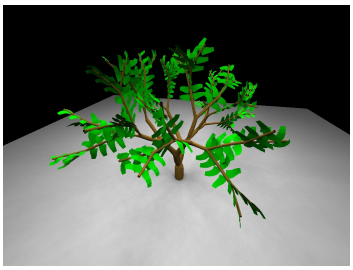
TREE, original light sources



COLUMNS, enlarged light source



BUNNY, enlarged light sources



TREE, enlarged light sources

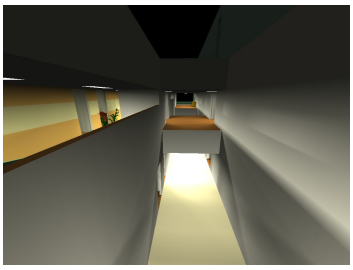
scene	output resolution	relations (M)	% blocked relations	**ISS time**	% relations ray-cast	mem peak	**comp. time**	**performance ratio**
COLUMNS	512 × 384	28.9	57.9	11.5	66.9	1.1	29.1	2.53
	1024 × 768	115.6	57.9	29.9	49.6	4.3	105.8	3.54
	2048 × 1536	462.6	57.9	93.8	39.3	18.7	397.9	4.24
4×smp	1024 × 768	462.6	57.9	83.7	36.6	4.3	394.5	4.71
4×area	1024 × 768	102.6	58.3	43.2	69.9	4.1	99.7	2.31
BUNNY	512 × 384	45.7	30.0	57.6	38.7	8.6	66.1	1.15
	1024 × 768	182.8	30.0	167.3 ∗	33.2	10.2	243.8	1.46
	2048 × 1536	731.3	30.0	556.2	29.7	22.7	921.9	1.66
4×smp	1024 × 768	731.3	30.0	524.4	28.5	10.2	905.3	1.73
4×area	1024 × 768	178.4	28.9	226.4	40.9	12.2	254.5	1.10
TREE	512 × 384	46.9	28.8	79.5	64.2	3.0	81.3	1.02
	1024 × 768	187.8	28.7	281.5	61.3	5.4	305.9	1.09
	2048 × 1536	751.2	28.8	1020.5	59.3	22.2	1172.7	1.15
4×smp	1024 × 768	751.1	28.7	988.5	56.6	5.4	1123.8	1.14
4×area	1024 × 768	181.6	27.2	308.6	66.3	5.3	315.2	1.02

∗ Breakdown of execution time available in Section 5.4.

Table 5.2 Test renderings and performance statistics in scenes COLUMNS, BUNNY and TREE. All times are in seconds, and the peak memory usage is given in megabytes. For both scenes, five renderings were made using the ISS algorithm and the comparison method. Three different resolutions were used with the original light sample counts and light source sizes. In addition, measurements were made with larger number of light samples (rows 4×smp) and enlarged light sources (rows 4×area) using the medium resolution. Column *relations* gives the total number of visibility relations in millions. The next column shows the fraction of visibility relations that were blocked, i.e. in shadow. Column *ISS time* shows the total shadow computation time using the ISS algorithm, including the lazy construction of the receiver and light trees. Column *relations ray-cast* gives the fraction of the visibility relations that were ultimately solved using the ray caster when leaf nodes were reached in both receiver and light trees. Column *mem peak* shows the peak memory usage of the ISS algorithm in megabytes. Columns *comp. time* gives the shadow computation time using the comparison method, and the *performance ratio* is the ratio of the shadow computation times between the comparison method and the ISS algorithm.

SODA 1

SODA 2



SODA 3

scene	output resolution	relations (M)	% blocked relations	**ISS time**	% relations ray-cast	mem peak	**comp. time**	**performance ratio**
SODA 1	512 × 384	4892.9	98.58	273.9	0.72	121.5	1698.2	6.20
	1024 × 768	19571.5	98.58	540.5	0.64	128.2	6073.8	11.24
	2048 × 1536	78285.8	98.58	1260.8	0.59	139.5	22614.0	17.94
4×smp	1024 × 768	78286.1	98.58	788.7	0.45	145.6	23089.3	29.27
SODA 2	512 × 384	4750.6	99.73	41.6	0.41	95.2	1375.6	33.10
	1024 × 768	19001.7	99.73	83.0 ∗	0.38	96.0	4730.0	56.96
	2048 × 1536	76007.5	99.73	237.5	0.36	98.7	17228.2	72.53
4×smp	1024 × 768	76006.9	99.73	168.7	0.28	106.3	17917.5	106.21
SODA 3	512 × 384	4697.0	98.96	226.8	0.51	133.4	1434.6	6.32
	1024 × 768	18789.2	98.96	368.2	0.50	135.1	5108.3	13.87
	2048 × 1536	75158.5	98.96	813.2	0.47	150.5	19119.0	23.51
4×smp	1024 × 768	75156.7	98.96	506.5	0.25	139.7	19440.6	38.38
SODA 1	512 × 384	440.2	84.25	50.6	6.71	37.9	479.4	9.48
pre-culled	1024 × 768	1760.9	84.25	126.2	6.09	42.8	1846.9	14.64
35 lights	2048 × 1536	7043.7	84.25	386.4	5.63	61.3	7233.7	18.72
4×smp	1024 × 768	7043.7	84.25	280.5	4.46	42.7	7157.1	25.51
SODA 2	512 × 384	97.7	87.05	13.7	12.05	34.3	70.2	5.13
pre-culled	1024 × 768	390.8	87.05	38.8	11.19	38.7	267.1	6.89
8 lights	2048 × 1536	1563.1	87.05	131.6	10.49	59.7	1043.9	7.93
4×smp	1024 × 768	1563.1	87.03	81.8	7.81	38.7	1010.7	12.35
SODA 3	512 × 384	555.4	91.17	35.3	4.45	41.4	357.8	10.13
pre-culled	1024 × 768	2221.9	91.17	102.8	4.16	47.0	1361.1	13.24
46 lights	2048 × 1536	8887.6	91.17	304.6	3.89	67.3	5315.3	17.45
4×smp	1024 × 768	8887.4	91.17	160.1	2.13	48.2	5237.9	32.72

∗ Breakdown of execution time available in Section 5.4.

Table 5.3 Test renderings and performance statistics in the SODA scene with three different viewpoints. The formatting of the table is identical to Table 5.2. The upper three sections of the table show the results in runs where all light sources were enabled, and the lower three sections give results in runs where only the light sources affecting the image were enabled. For the three viewpoints used, the number of affecting lights was quite different, ranging from 8 light sources in the second viewpoint to 46 in the third one.

performance of the ISS algorithm grows as the number of visibility relations is increased, as could be expected. However, in the SODA scene the ISS algorithm does not behave as symmetrically as in the simpler test scenes, since increasing the number of light samples yields better execution times than increasing the number of receiver points. This is most likely because the blockers are usually located closer to the receiving surfaces than the light sources, and thus there is more coherence between the light samples than between the receiver points.

The memory usage of the ISS algorithm is much higher in the SODA scene than in the three simpler

	BUNNY	SODA 2
Shaft operations	16.07 %	56.44 %
Ray casting	83.93 %	43.56 %
Tree construction	4.93 %	13.38 %
Shaft geometry construction	0.40 %	2.05 %
Triangle surface construction	8.88 %	16.30 %
Copying surfaces	14.12 %	5.84 %
Classifying edges	9.66 %	10.83 %
Updating patch facings	3.93 %	0.83 %
Merging patches	19.13 %	9.69 %
Simplifying patches	16.63 %	13.12 %
Splitting surfaces	4.37 %	1.36 %
Combining surfaces	1.01 %	0.33 %
Computing loose edges	1.33 %	1.09 %
Computing surface AABBs	7.38 %	4.43 %
Blocking tests	0.24 %	2.38 %
Miscellaneous	7.98 %	18.36 %

Table 5.4 Execution time breakdowns for scenes BUNNY and SODA viewpoint two. Resolution of 1024×768 was used with the original number of light samples. The top two rows show the general ratio between shaft processing and ray-casting. The bottom section of the table shows the execution times for different shaft operations relative to the total time spent in shaft processing.

test scenes. This is mostly due to high triangle count of the scene, and in fact, the memory consumption peak was always met in the very beginning of the computation. Quite intuitively, the amount of blocker geometry in a shaft is at its largest when the shafts are very large, and it decreases rapidly as the shafts become smaller. It would therefore be possible to limit the memory usage of the algorithm by continuing the first stage of traversal (see Section 4.9) where the blocker geometry is not inserted into the shaft until the amount of geometry becomes small enough.

The bottom three sections of the table show the benchmark results for runs where only the light sources that affected the rendered image were enabled. It could be expected that the relative performance of the ISS algorithm would diminish, since culling sets of occluded visibility relations efficiently is its most valuable feature in this scene. As there are less blocked relations, the comparison method should gain some advantage in this sense. Surprisingly, the relative performance of the ISS algorithm was found to be better in several situations when the redundant light sources were pre-culled, especially when many light sources were needed for obtaining the correct image. In the second viewpoint, where only eight light sources had to be enabled, the pre-culling did decrease the relative performance of the ISS algorithm significantly. As can be seen from the high percentage of blocked visibility relations, there is still a lot of occlusion between the light sources and the different parts of the receiver point sets even when redundant light sources are removed from the scene. What this means is that the affecting light sources are typically visible to only a small number of receiver points, and it is still very beneficial to be able to efficiently decide the status of large sets of visibility relations using the shaft computations.

5.4 Execution Time Breakdowns

Table 5.4 gives the execution time breakdowns in two of the test runs. In the BUNNY scene, most of the time (83.93 %) is spent in ray casting, and only 16.07 % goes to shaft processing operations. This may seem a bit paradoxical, since as it is shown in Table 5.2, only 33.2 % of the relations are solved by ray-casts in this test case. The reason behind this apparent discrepancy is that the rays

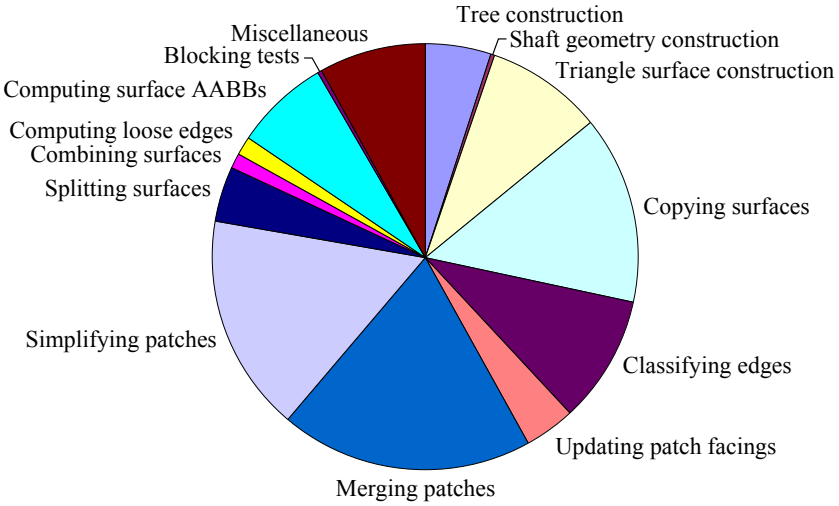


Figure 5.1 Breakdown of execution time in shaft operations in BUNNY scene.

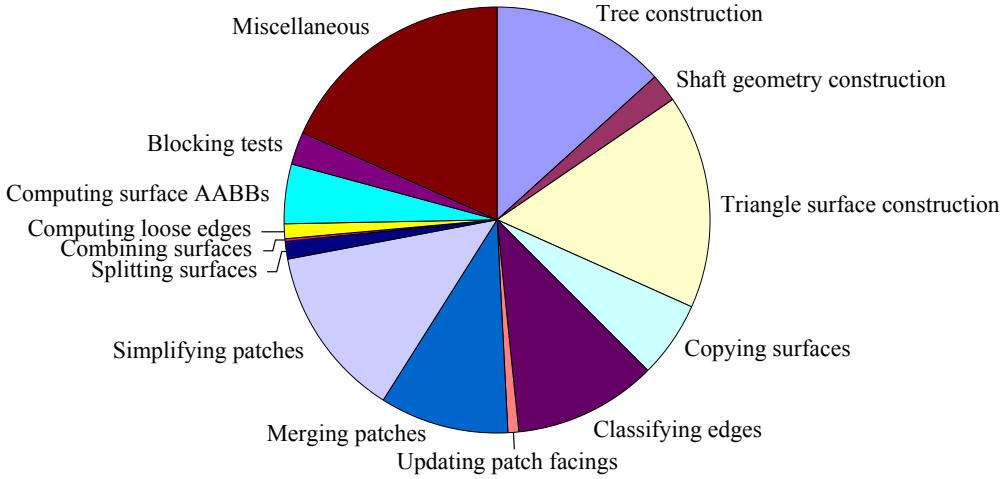


Figure 5.2 Breakdown of execution time in shaft operations in SODA scene, viewpoint two.

cast by the ISS algorithm are generally much more expensive than an average ray. Remembering that the rays are cast only when the shaft subdivision ends so that shaft is not blocked or empty, the remaining rays are always located close to the silhouette of the bunny, resulting in a lot of BSP traversal and many triangle intersection tests in the ray caster. In the SODA scene (viewpoint two), the majority of the time is spent in shaft operations, but this time the discrepancy between time spent in ray-casts (43.56 %) and the fraction of relations solved using ray-casts (0.38 %) is even greater.

Based on the large amount of time taken for performing a relatively small number of ray casts, it might seem that it would be a good idea to continue the shaft subdivision even further, thereby reducing the number of relations solved by ray casting. This could be done by specifying that the

leaf nodes of the receiver and light trees could contain fewer points than what were used in the test runs (Table 5.1). Unfortunately, this was found to be of no help, since the fraction of relations solved by detecting blocked or empty shafts did not grow enough to balance the computation time spent in subdividing the shafts even further. It should be emphasized that the execution time did not increase much if e.g. the maximum point counts in both light and receiver trees were halved—generally this resulted in only about 2–5 % increase in execution times.

In both test cases, most of the shaft processing time is spent in constructing the light and receiver trees (Section 4.3), constructing the triangle surfaces (Section 4.7.1), copying surfaces from the parent shaft, classifying edges (Section 4.7.2), merging patches (Section 4.7.4), simplifying patches (Section 4.7.5), and computing the bounding boxes for the surfaces. A lot of time was also spent in miscellaneous operations, consisting mostly of memory management tasks such as allocating and deallocating edges, patches and surfaces.

It is somewhat amusing to note that performing the shaft blocking tests took only a tiny fraction of the total shaft processing time, even though being able to perform this test is the only reason for doing any shaft computation at all. It can be concluded that managing the blocking surfaces inside the shaft is currently the bottleneck of the shaft processing part of the ISS algorithm.

Chapter 6

Discussion and Future Work

The ISS algorithm is currently the only shadow computation algorithm that computes the shadows in sub-linear time with respect to all three parameters describing the complexity of the scene: the number of light samples, the number of receiver points and the number of triangles. The sub-linearity with respect to the first two is obtained through hierarchical processing of light and receiver trees, and the sub-linearity with respect to triangle count follows from the use of silhouette edges in the blocker representation. In the following, we discuss the strong and weak points of the ISS algorithm. After that, we take a look at possible directions for future work.

6.1 Strengths of the ISS Algorithm

Efficient culling of redundant light sources. As was shown by the experimental results in Chapter 5, the ISS algorithm is able to cull redundant light sources very efficiently. In the Soda Hall test scene, where only a small fraction of visibility relations were unoccluded, the ISS algorithm gave significant speedups over the traditional ray casting-based method. This is a very desirable property in large scenes, although in the simple object-on-a-plane scenes it is of no use.

Sub-linear complexity with reasonable memory usage. One of the advantages of the classical shadow ray casting is that it does not require allocating memory for the visibility relations. This is in stark contrast with e.g. the hierarchical penumbra casting algorithm [50] that needs to reserve memory for every visibility relation in the worst case. The memory consumption of the ISS algorithm is limited to storing the light and receiver trees and the blocker geometry inside the current shaft and its parent shafts. As was mentioned in Chapter 5, the memory consumption peaks in the ISS algorithm were caused by the large shafts near the beginning of the recursion, as these shafts contain a lot of blocker geometry. It could be expected that bounding the memory usage by postponing the insertion of the geometry to smaller shafts would result in quite minor performance hit. Validating this hypothesis would require additional research.

Symmetry. The ISS algorithm is the only shadow algorithm, aside from the classical ray casting, that considers receiver points and light samples to be *exactly alike*.[1] In other words, it does not

[1] There is the difference that light samples are actually represented as small boxes containing the jitter samples (Section 4.11), whereas the receiver points are just points. We could easily extend the jittered representation to receiver points as well, which would again make the algorithm perfectly symmetric. This would be of no use in computing direct lighting, but might serve as a variance reduction technique for computing global illumination.

matter whether we ask the ISS algorithm to compute the visibility from the receiver points to the light samples or vice versa—the resulting computation is the same in both cases. Conceptually, the symmetry is a property one might expect to see in an algorithm designed to solve a symmetrical problem such as visibility. Still, none of the modern shadow computation algorithms presented so far have had this property. This is by no means a symptom of flawed design in those algorithms, since it can also be argued that they take advantage of the usually very asymmetric role of light samples and receiver points. For instance, the number of light samples is generally much smaller than the number of receiver points, and their placement can often be expected to be somehow constrained. However, if we consider extending a shadow algorithm towards computing global illumination, the assumptions made about asymmetry and/or positioning of the light samples are obviously a major burden. In this sense, the ISS algorithm is a good platform upon which a visibility solver for global illumination might be built.

6.2 Weaknesses of the ISS Algorithm

Back-facing visibility relations. The ISS algorithm in its current form computes the visibility between two point clouds, and the points are assumed to be completely defined by their positions. In shadow computation, the receiver points and light samples are actually used for sampling continuous surfaces, and therefore they represent infinitesimally small surface patches that have normals. Computing the flow of light requires that we take the normals of the light samples and receiver points into account, as was discussed in Section 2.2.1. An obvious consequence is that if a receiver point and a light sample do not face towards each other, the flow of light between them is always zero, and computing the visibility is not needed at all. Exploiting this fact is a very simple way of reducing the number of visibility relations when solving the visibility relations one by one, but currently the ISS algorithm is unable to take advantage of this. Possible ways of overcoming this limitation are discussed in Section 6.3.

Importance sampling of light sources. Light samples that are far away from receiver points generally do not contribute much in the rendered image. The same is true for cases where the ray from a light sample to a receiver point makes large angles with surface normals. Nonetheless, the ISS algorithm computes all of these visibility relations. It would be much smarter to represent a far away or otherwise only slightly contributing light source with fewer light samples, and to scale the result appropriately to get an estimate of the flow of light. Currently, the ISS algorithm does not support this kind of importance sampling. It is well known that contribution-based importance sampling is an effective way of accelerating shadow rendering, and implementing it in the classical ray-casting algorithm is trivial. Consequently, there is a definite need for extending the ISS algorithm so that it could support importance sampling.

Importance sampling of receiving surfaces. Consider two close-by receiver points that are lit by a light source. If the occlusion between the light source and the receiver points is similar, the flow of light could be approximated quite well by e.g. solving the visibility from only one of the receiver points, and assuming this information to hold for the other receiver point as well. This approach is similar to importance sampling of light sources, but now applied to receiver points. Describing and approximating the nature of occlusion is a non-trivial issue, but recent research on frequency-based analysis of light transport [26] has already given interesting insights into this area. In some situations, no approximations in the receiving end can be afforded, and it is a nice property of the ISS algorithm that none need to be made. Still, it must be considered as a drawback that even when approximations could be justified, they cannot be exploited.

Detection of empty shafts in special cases only. As was discussed in Section 4.12, the ISS al-

gorithm is not generally able to detect the condition where every light sample in the light node is guaranteed to be visible to every receiver point in the receiver node. In special cases, e.g. when receiver points and light samples lie on axis-aligned planes, this is usually possible, and even when it is not, no major performance hit is incurred. As long as no approximations are made, and every visibility relation is solved, the situation is thus not that severe, but if we consider making some kind of approximations in computing the flow of light between two mutually visible surfaces, it would be very useful to detect the full visibility in more general cases.

6.3 Future Work

Several directions for future work are already given by the weak points of the ISS algorithm. Perhaps the most important of these is the lack of importance sampling. It is quite obvious that solving every visibility relation is not the smartest thing to do, and the issue with back-facing receiver points and light samples is just an example of this. In the following, we discuss ways of overcoming this limitation, and other possible enhancements to the ISS algorithm.

6.3.1 Importance Sampling

For truly efficient importance sampling (both of light sources and receiving surfaces), we would need to collect aggregate information about the groups of receiver points and light samples as present in the tree nodes. For instance, somehow representing the set of directions where the points in a node face would enable back-facing relations to be culled in groups. The test would naturally need to be conservative, so even a single relation that is not back-facing would prevent the culling from happening. The directions in which the points in a node face does not only give us information for performing the back-face culling, but also establishes bounds for the cosine factors in the reflectance equation (Equation 2.6). This, combined with the shortest distance between nodes, could be used for deriving importance bounds for the particular group of relations, and ultimately to solve the visibility using fewer light samples or receiver points.

Implications on tree construction. Constructing any kind of aggregate data from groups of receiver points or light samples makes it hard to construct the point trees lazily. This is because such aggregate information would be most natural to propagate upwards from the leaves, and if we do not perform the subdivision all the way to the leaves in the first place, we end up computing the information from scratch on all levels of the hierarchy. On the other hand, the tree subdivision criteria could also utilize the aggregate information, which would require employing a top-down approach anyway. In this case, the lazy construction would not cause additional problems. It must be noted that the aggregate information in one node only is less useful than aggregate information in both light and receiver nodes, since it is the pair of nodes between which the importance bounds are needed. Therefore, if the node splitting strategy depends on the other node as well, the situation is quite complex. It is unclear whether it is enough to split a node only once, according to the assumptions that were valid when the node was first reached, or if it would make sense to reconsider the split strategy if e.g. a receiver node is entered again but this time with different light node in the other end. The simple strategy currently employed in the ISS algorithm node (Section 4.3) evades all of these questions, but it is certainly not the optimal one if we have more information at our disposal than just the positions of the points.

6.3.2 Detecting Full Visibility

Another disappointing property of the ISS algorithm is its inability of detecting the full visibility in other but special cases. As was discussed in Section 4.12, the problem stems from geometry remaining in the receiver and light nodes, which prevents us from guaranteeing full visibility. This problem could be tackled by computing, for each node, a set of directions where no occlusion inside the node takes place. A possible implementation for this would be a cubical bit mask filled using conservative rasterization. Then, it would be possible to check that the mutual position of the nodes is such that the shaft extends towards unoccluded directions in both nodes. Better detection of full visibility is not necessary in the current form of the ISS algorithm, as the traversal to the nodes of the light and receiver trees is anyway very fast when no blocker geometry is in the shaft, but in conjunction with other extensions it might become important.

6.3.3 Supporting Non-Opaque Shadow Casters

Another possible improvement to the ISS algorithm is supporting other than opaque shadow casters. This is perhaps not that exciting extension theoretically, but in practice, it would make the algorithm usable in a wide variety of real situations, as transparent surfaces and alpha-matte textures are often used in production rendering. These kind of surfaces cannot be used for blocking the shaft, but they could be simply left out of the set of potential blockers.

6.3.4 Solving the Remaining Visibility Relations

Currently, ray casting is used as a fall-back solution for solving the visibility relations when the shaft computations fail to show that all relations are blocked or visible. The point where ray casting is reverted to is currently determined by the maximum leaf node point counts in light and receiver trees, and these must be set manually, which is a clear weakness in the algorithm. It should be possible to automatically detect when ray casting is more feasible approach than continuing with the shaft computation, and this decision should probably depend on more detailed information than just the number of visibility relations remaining. The amount and nature of blocking geometry inside the shaft, for instance, should give relevant clues about how costly the ray casting is likely to be, and it would of course be possible to measure the time taken by ray casting and adjust the fall-back criterion depending on the results.

Furthermore, ray casting is definitely not the only possible option for solving the remaining visibility relations. In fact, it seems quite silly to revert straight to the simplest of all methods, when more efficient shadow computation methods have been presented in the scientific literature. The soft shadow volume algorithm [51] is a bit problematic as a fall-back solution, since it requires that the light samples lie on a plane. On the other hand, the hierarchical penumbra casting (HPC) algorithm [50] is quite an interesting candidate for being an efficient fall-back solution in the ISS algorithm. First of all, we note that shadows caused by many small, disjoint triangles is a nightmare for the ISS algorithm, as no completely blocked shafts are encountered. For the HPC algorithm, this would be a piece of cake, especially when the shadows are relatively concentrated as in the Soda Hall scene in Chapter 5. Secondly, the HPC algorithm can also be applied in the reverse direction. For instance, if we have a number of small triangles right in front of a light source, the solution is much cheaper to compute from the reverse direction, spanning the penumbra volumes from receiver points towards the light source.

Incorporating the HPC algorithm as the fall-back solution for the ISS algorithm would certainly

be interesting, and it might result in much better execution times than with the current version. Naturally, if a faster fall-back solution is obtained, it will also become more feasible to revert to using it, which in turn reduces the amount of time spent in shaft computations.

6.3.5 Global Illumination

Computing visibility between points constitutes a significant portion of global illumination computations. The ISS algorithm, being completely symmetrical, could be used for solving these visibilities efficiently. Consider a variant of radiosity algorithm that represents each surface patch as a single point. Now, the transport of light occurs only between these points, and to compute one bounce of light transport requires solving the visibility relations between every pair of points. This approach is similar to the instant radiosity algorithm [48], where the illumination to receiver points is calculated effectively using photons in a photon map [44] as light sources, but extended so that all bounces of the global illuminated are computed this way.

It would be possible to use the ISS algorithm in its current state for computing the light transport between the points, but it is quite likely that the resulting algorithm would be inefficient because of the current limitations of the algorithm. If some form of sophisticated importance sampling—including the total culling of back-facing relations—were possible in both ends of the shaft, the ability of the ISS algorithm to cull visibility relations in large groups could yield an efficient method for solving global illumination.

Bibliography

[1] Timo Aila and Tomas Akenine-Möller. A Hierarchical Shadow Volume Algorithm. In *Graphics Hardware*, pages 15–23, 2004.

[2] Timo Aila and Samuli Laine. Alias-free shadow maps. In *Proceedings of Eurographics Symposium on Rendering 2004*, pages 161–166. Eurographics Association, 2004.

[3] John M. Airey, John H. Rohlf, and Jr. Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, pages 41–50. ACM Press, 1990.

[4] Tomas Akenine-Möller and Ulf Assarsson. Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 297–305. Eurographics Association, 2002.

[5] John Amanatides. Ray tracing with cones. In *Proceedings of SIGGRAPH '84*, pages 129–135. ACM Press, 1984.

[6] James Arvo and David Kirk. *An Introduction to Ray Tracing* (ed. Andrew S. Glassner), chapter A Survey of Ray Tracing Acceleration Techniques, pages 201–262. Academic Press Ltd., London, UK, 1989.

[7] Jukka Arvo. Tiled Shadow Maps. In *Proceedings of Computer Graphics International*, pages 240–247. IEEE Computer Society, 2004.

[8] Jukka Arvo. *Efficient Algorithms for Hardware-Accelerated Shadow Computation*. PhD thesis, Turku Centre for Computer Science and University of Turku, 2005.

[9] Ulf Assarsson and Tomas Akenine-Möller. A Geometry-Based Soft Shadow Volume Algorithm using Graphics Hardware. *ACM Transactions on Graphics*, 22(3):511–520, 2003.

[10] Ulf Assarsson, Michael Dougherty, Michael Mounier, and Tomas Akenine-Möller. An Optimized Soft Shadow Volume Algorithm with Real-Time Performance. In *Graphics Hardware*, pages 33–40. ACM SIGGRAPH/Eurographics, 2003.

[11] Fausto Bernardini, James T. Klosowski, and Jihad El-Sana. Directional discretized occluders for accelerated occlusion culling. *Computer Graphics Forum (Proceedings of Eurographics 2000)*, 19(3), 2000.

[12] Jiří Bittner, Peter Wonka, and Michael Wimmer. Fast exact from-region visibility in urban scenes. In *Rendering Techniques 2005 (Proceedings of the Eurographics Symposium on Rendering 2005)*, pages 223–230. Eurographics, Eurographics Association, 2005.

[13] Jiří Bittner. *Hierarchical Techniques for Visibility Computations*. PhD thesis, Czech Technical University in Prague, Department of Computer Science and Engineering, 2002.

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

[15] James F. Blinn. Me and my (fake) shadow. *IEEE Comput. Graph. Appl.*, 8(1):82–86, 1988.

[16] Eric Chan and Frédo Durand. An efficient hybrid shadow rendering algorithm. In *Proceedings of the Eurographics Symposium on Rendering*, pages 185–195. Eurographics Association, 2004.

[17] Allen Y. Chang. A survey of geometric data structures for ray tracing. Technical Report TR-CIS-2001-06, CIS Department, Polytechnic University, Brooklyn, New York, 2001.

[18] Daniel Cohen-Or, Yiorgos Chrysanthou, Cláudio T. Silva, and Frédo Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):412–431, 2003.

[19] Frank Crow. Shadow Algorithms for Computer Graphics. In *Computer Graphics (Proceedings of ACM SIGGRAPH 77)*, pages 242–248. ACM, 1977.

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

[21] Kirill Dmitriev, Vlastimil Havran, and Hans-Peter Seidel. Faster ray tracing with SIMD shaft culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, December 2004.

[22] Florent Duguet and George Drettakis. Robust epsilon visibility. In *Proceedings of ACM SIGGRAPH 2002*. ACM Press / ACM SIGGRAPH, 2002.

[23] Frédo Durand. *3D Visibility: analytical study and applications*. PhD thesis, Université Joseph Fourier, Grenoble I, July 1999.

[24] Frédo Durand, George Drettakis, and Claude Puech. The 3D Visibility Complex, a new approach to the problems of accurate visibility. In *Eurographics Workshop on Rendering*, 1996.

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

[26] Frédo Durand, Nicolas Holzschuch, Cyril Soler, Eric Chan, and François Sillion. A frequency analysis of light transport. *ACM Transactions on Graphics (Proceedings of the SIGGRAPH conference)*, 24(3), 2005.

[27] Philip Dutré, Philippe Bekaert, and Kavita Bala. *Advanced Global Illumination*. AK Peters, 2003.

[28] David Eberly. Triangulation by ear clipping. Geometric Tools, Inc. *http://www.geometrictools.com*, 2002.

[29] Cass Everitt and Mark Kilgard. Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering. *http://developer.nvidia.com*, 2002.

[30] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive Shadow Maps. In *Proceedings of ACM SIGGRAPH 2001*, pages 387–390. ACM Press, 2001.

[31] Arno Formella and Andrzej Łukaszewski. Fast penumbra calculation in ray tracing. In *Proceedings of WSCG'98*, pages 238–245, 1998.

[32] Djamchid Ghazanfarpour and Jean-Marc Hasenfratz. A Beam Tracing with Precise Antialiasing for Polyhedral Scenes. *Computer Graphics*, 22(1):103–115, 1998.

[33] Eric Haines. A shaft culling tool. *Journal of Graphics Tools*, 5(1):23–26, 2000.

[34] Eric Haines and Donald Greenberg. The Light Buffer: A Ray Tracer Shadow Testing Accelerator. *IEEE Computer Graphics and Applications*, 6(9):6–16, 1986.

[35] Eric Haines and John Wallace. Shaft culling for efficient ray-traced radiosity. In *Eurographics Workshop on Rendering*, 1991.

[36] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. In *Eurographics*. Eurographics, Eurographics, 2003. State-of-the-Art Report.

[37] Denis Haumont, Olivier Debeir, and François Sillion. Volumetric cell-and-portal generation. In *Computer Graphics Forum*, volume 3-22 of *EUROGRAPHICS Conference Proceedings*. Blackwell Publishers, 2003.

[38] Denis Haumont, Otso Mäkinen, and Shaun Nirenstein. A low dimensional framework for exact polygon-to-polygon occlusion queries. In *Rendering Technqiues 2005: Proceedings of the 16th symposium on Rendering*, pages 211–222. Eurographics Association, 2005.

[39] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[40] Paul S. Heckbert and Pat Hanrahan. Beam tracing polygonal objects. *Proceedings of SIGGRAPH '84*, pages 119–127, 1984.

[41] Tim Heidmann. Real Shadows, Real Time. *Iris Universe*, 18:28–31, 1991.

[42] André Hinkenjann and Heinrich Müller. Hierarchical blocker trees for global visibility calculation. Technical Report 621/1996, Universität Dortmund, Germany, 1996.

[43] Samuel Hornus, Jared Hoberock, Sylvain Lefebvre, and John Hart. ZP+: Correct z-pass stencil shadows. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 195–202. ACM Press, 2005.

[44] Henrik Wann Jensen. Global Illumination Using Photon Maps. In *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)*, pages 21–30, New York, NY, 1996. Springer-Verlag/Wien.

[45] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns, and William R. Mark. The irregular Z-buffer: Hardware acceleration for irregular data structures. *ACM Transactions on Graphics*, 24(4):1462–1482, 2005.

[46] Gregory S. Johnson, William R. Mark, and Christopher A. Burns. The Irregular Z-Buffer and its Application to Shadow Mapping. Technical report, The University of Texas at Austin, Department of Computer Sciences, April 2004.

[47] James T. Kajiya. The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, pages 143–150. ACM Press, 1986.

[48] Alexander Keller. Instant radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[49] Samuli Laine. Split-plane shadow volumes. In *Proceedings of Graphics Hardware 2005*, pages 23–32. Eurographics Association, 2005.

[50] Samuli Laine and Timo Aila. Hierarchical penumbra casting. *Computer Graphics Forum*, 24(3):313–322, 2005.

[51] Samuli Laine, Timo Aila, Ulf Assarsson, Jaakko Lehtinen, and Tomas Akenine-Möller. Soft shadow volumes for ray tracing. *ACM Transactions on Graphics*, 24(3):1156–1165, 2005.

[52] Eric Lengyel. The Mechanics of Robust Stencil Shadows. *http://www.gamasutra.com*, 2002.

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

[54] Brandon Lloyd, Jeremy Wendt, Naga K. Govindaraju, and Dinesh Manocha. CC Shadow Volumes. In *Proceedings of the Eurographics Symposium on Rendering*, 2004.

[55] David Luebke and Chris Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. In *1995 Symposium on Interactive 3D Graphics*, pages 105–106. ACM SIGGRAPH, 1995.

[56] Joseph Marks, Robert Walsh, Jon Christensen, and Mark Friedell. Image and intervisibility coherence in rendering. In *Proceedings of Graphics Interface '90*, pages 17–30, 1990.

[57] Tobias Martin and Tiow-Seng Tan. Anti-aliasing and Continuity with Trapezoidal Shadow Maps. In *Proceedings of the Eurographics Symposium on Rendering*, pages 153–160. Eurographics Association, 2004.

[58] Morgan McGuire. Observations on Silhouette Sizes. *Journal of Graphics Tools*, 9(1):1–12, 2004.

[59] Morgan McGuire, John F. Hugues, Kevin T. Egan, Mark Kilgard, and Cass Everitt. Fast, Practical and Robust Shadows. Technical Report CS03-19, Brown University, October 2003.

[60] Isabel Navazo, Jarek Rossignac, Joan Jou, and Rahim Shariff. Shieldtester: Cell-to-cell visibility test for surface occluders. *Computer Graphics Forum (Proceedings of Eurographics '03)*, 22(3):291–302, 2003.

[61] Fred E. Nicodemus, Joseph C. Richmond, Jack J. Hsia, I. W. Ginsberg, and T. Limperis. Geometric Considerations and Nomenclature for Reflectance. NBS Monograph 160, National Bureau of Standards, 1977.

[62] Shaun Nirenstein, Edwin Blake, and James Gain. Exact from-region visibility culling. In *Proceedings of the 13th workshop on Rendering*, pages 191–202. Eurographics Association, 2002.

[63] NVIDIA. NVIDIA GeForceFX 5900, 5700 and Go5700 GPUs: UltraShadow Technology. Technical report, *http://www.nvidia.com*, 2003.

[64] Julius Plücker. On a new geometry of space. *Phil. Trans. Royal Soc. London*, 155:725–791, 1865.

[65] Michel Pocchiola and Gert Vegter. The visibility complex. In *Proceedings of the 9th Annual Symposium on Computational Geometry (SCG '93)*, pages 328–337. ACM Press, 1993.

[66] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering Antialiased Shadows with Depth Maps. In *Computer Graphics (Proceedings of ACM SIGGRAPH 87)*, pages 283–291. ACM, 1987.

[67] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, 1990.

[68] Mikio Shinya, Tokiichiro Takahashi, and Seiichiro Naito. Principles and applications of pencil tracing. In *Proceedings of SIGGRAPH '87*, pages 45–54. ACM Press, 1987.

[69] Marc Stamminger and George Drettakis. Perspective Shadow Maps. *ACM Transactions on Graphics*, 21(3):557–562, 2002.

[70] László Szirmay-Kalos and Gábor Márton. Worst-case versus average case complexity of ray-shooting. *Computing*, 61(2):103–131, 1998.

[71] Seth J. Teller. *Visibility Computations in Densely Occluded Polyhedral Environments*. PhD thesis, (Also TR UCB/CSD 92/708), CS Dept., UC Berkeley, 1992.

[72] Yulan Wang and Steven Molnar. Second-Depth Shadow Mapping. Technical Report TR94-019, The University of North Carolina at Chapel Hill, 1994.

[73] Turner Whitted. An Improved Illumination Model for Shaded Display. *Communications of the ACM*, 23(6):343–349, 1980.

[74] Lance Williams. Casting Curved Shadows on Curved Surfaces. In *Computer Graphics (Proceedings of ACM SIGGRAPH 78)*, pages 270–274. ACM, 1978.

[75] Michael Wimmer, Daniel Scherzer, and Werner Purgathofer. Light Space Perspective Shadow Maps. In *Proceedings of the Eurographics Symposium on Rendering*, pages 143–151. Eurographics Association, 2004.

[76] Peter Wonka. *Occlusion Culling for Real-Time Rendering of Urban Environments*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2001.