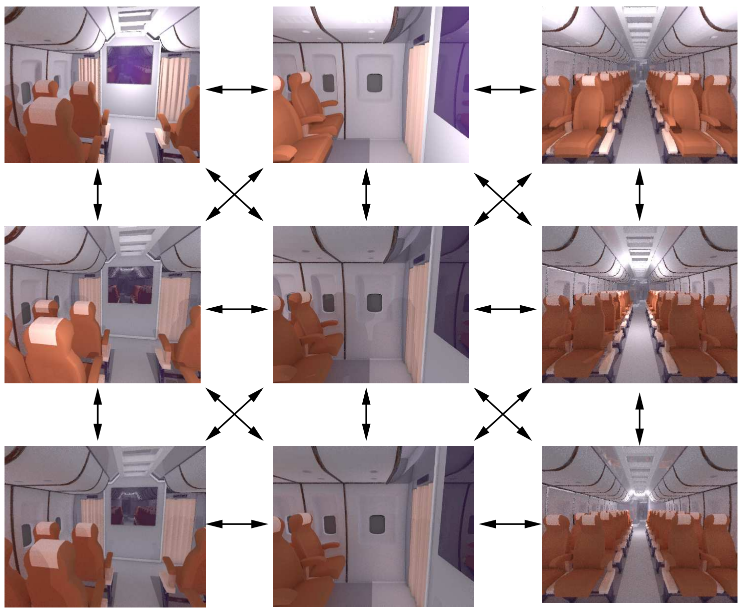
Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics

Fast Photorealistic Techniques to Simulate Global Illumination in Videogames and Virtual Environments

Àlex Méndez Feliu
Ph. D. dissertation

Advisor: **Mateu Sbert Casasayas**



Girona
April, 2007

Preface

This document is the Ph. D. dissertation entitled *Fast Photorealistic Techniques to Simulate Global Illumination in Videogames and Virtual Environments*, presented by Àlex Méndez i Feliu to the **Universitat Politècnica de Catalunya** to obtain the doctorate degree.

Abstract

To compute global illumination solutions for rendering virtual scenes, physically accurate methods based on radiosity or ray-tracing are usually employed. These methods, though powerful and capable of generating images with high realism, are very costly. In this thesis, some techniques to simulate and/or accelerate the computation of global illumination are studied. The obscurances technique is based on the supposition that the more occluded is a point in the scene, the darker it will appear. It is computed by analyzing the geometric environment of the point and gives a value for the indirect illumination for the point that is, though not physically accurate, visually realistic. This technique is enhanced and improved in real-time environments as videogames. It is also applied to ray-tracing frameworks to generate realistic images. In this last context, sequences of frames for animation of lights and cameras are dramatically accelerated by reusing information between frames.

Just a Tale

It was the end of 2001 and I was in Mexico.

One Sunday, few weeks after tragic September 11th and exactly the day U. S. A. attacked Afghanistan, my ex-teacher and friend Isaac Rudomín and I joined for breakfast. We had not seen in months and many things had happened during that time.

2001 was not really a good year for me, at least professionally. I arrived to Mexico in January with the only objective to be close to my girlfriend, but I never imagined how hard it is to be a foreigner in a strange land, even when they speak your own language. I did not find a job in six months and when I found it, it was in substitution for a pregnant secretary. I answered phone calls and wrote faxes to customers and suppliers: an awful job for a shy programmer with the dream to work in computer graphics.

Meanwhile my friend Isaac had been traveling through Europe, gone to conferences and met people. Among the people he met, there was Roel Martínez and Mateu Sbert from the Universitat de Girona.

That Sunday, in a beautiful *all-you-can-eat* restaurant in the middle of the Naucalpan Gardens, he told me about Girona Graphics Group and the things they were doing. Then I devised the idea of coming back to my country to do what I always wanted to do and it looked fascinating. I contacted Mateu and a few months later I came to Girona.

There was only one problem to solve, but it had an easy solution: soon my girlfriend had to become my wife, but that is another story...

Funding

My work in this dissertation has been possible thanks to grant FPI number BES-2002-1570 from the Ministerio de Ciencia y Tecnología from the Spanish Government.

This research has also been supported by projects TIC2001-2416-C03 and TIN2004-07451-C03-01 from the Spanish Government, GameTools Project from the VIth Framework Program from the European Union, 2001/SGR/00296 and ACI2002-52 projects from the Catalan Government, Spanish-Austrian joint-action HU2000-0011 and Hungarian-Spanish Joint Action HH2004-0001.

Acknowledgements

Thanks to all the people that in any way has been involved with this work.

To my advisor Mateu Sbert, for his brain is a source of ideas that feeds all our research, for his patience and for his support.
To Isaac Rudomín, for increasing my passion for this field and to Roel Martínez, for being the link to Mexico from Girona, and viceversa.
To László Szirmay-Kalos, for his kindness, his knowledge and his support during my stay in Budapest and all the chances we have had to meet in the past and hopefully in the future.
To László Neumann, for his colors and his ideas. Thanks to Coloroid Ltd. and to professor Antal Nemcsics for his kind permission to use the Color Harmony Designer software.
To my other co-authors, Jordi Catà, Nicolau Sunyer and Sergi Funtané, aquesta tesi us deu més coses del que sembla.
The kitchen, aircraft and stairs model is courtesy of LightWork Design Ltd.
Margarita García and Astrid Southon helped with programming obscurances within RenderPark.
Thanks to Janne Kontkanen, Simon Premože and Per Christensen for their kind permission to use their images.
Als meus companys. Al Carles i a l'Albert, per tots aquests cafès. A la Marité i al Narcís, per omplir-nos els caps de setmana. A la *quitxalla*, l'Isma, el Fran i la Tere, molts ànims, que teniu molt de futur. Al Gus, per cuidar-nos a tots.
Als meus amics, al Roberto, al Joan i, especialment, al Paco, pel que hem compartit i el que ens queda per fer junts.
A la meva família. Als meus pares, a les meves germanes, a les iaies i a la tieta, que sovint no han acabat d'entendre per què he seguit aquest camí a la vida, però sempre m'han fet suport.
A la petita Ariadna, la meva neboda, la nova estrella que il.lumina la família, i primera de les que han de venir.
A l'Estela, la meva llum i les meves ombres, la meva vida. El meu món real.

Àlex Méndez i Feliu
Girona, 24 d'Abril del 2007

Contents

List of Figures

Chapter 1

Introduction

This dissertation has been done in the context of computer graphics, in particular in the field of image synthesis. Its aim is to research new techniques to accelerate the computation of global illumination, leading to photorealistic results in shorter computation time. The results of this dissertation are of application to videogames and virtual reality, as well as to production rendering for movies and animations.

1.1 Motivation

Computing global illumination is a very costly process, as the interaction of light within all surfaces of the scene has to be taken into account several times to achieve an accurate computation of direct and indirect illumination.

Radiosity [32, 82] and some ray-casting techniques as path-tracing [43] and bidirectional path-tracing [89] can be used for the computation of global illumination, and in the last twenty years a lot of research has been done to improve and accelerate these techniques.

Nowadays, global illumination for virtual environments is used by two kinds of applications: on the one hand those that require real-time rendering, as videogames and virtual reality applications, and on the other hand those that use high quality realistic rendering, as movie production or architecture.

The real-time applications can pre-compute radiosity and store it in a special kind of texture called lightmap. The illuminated scenes are shown in scene walkthroughs, and some direct illumination (or object shadows) can be computed in real-time for every frame, but physically accurate global illumination can not be recomputed at real-time frame rates, because if any object or light source moves, it requires recomputing all illumination of the scene from scratch.

In applications that require high quality realistic rendering, computing a single image can take hours in a personal computer. This is much more problematic if we need to compute an animation, even if it is a few seconds long, at a rate of 24 frames per second the number of images to compute increases dramatically. In production rendering in the industry, very fast and specialized parallel hardware is used. The rooms where these powerful machines are placed are called rendering farms, and they work 24 hours a day to compute a few seconds of an animation.

For all these reasons, doing research on the development of techniques that accelerate the process of computing global illumination can lead to save much time and money in the production of movies and to achieve high realism in videogames and the visualization of virtual environments, that require real-time frame rates.

1.2 Objectives

The main objective is to research and find techniques that simulate or accelerate global illumination and that give visually plausible and realistic results.

In particular, on the one hand we improve a technique that simulates global illumination, the obscurances, in different ways, so we obtain more realistic results with little added computation time. We add color bleeding, study different functions for the computation of the obscurances, study the generation of samples over an hemisphere and propose some solutions for the special cases that present problems.

On the other hand, obscurances and other global illumination techniques are used in animations. For moving objects in videogame environments, obscurances around the objects are recomputed. Moving lightsources combined with obscurances accelerate the computation of a series of ray-traced frames in a movie. For a moving camera in ray-tracing environments, hits in the scene are reused, and, finally, this last technique can be combined with light animation, too.

1.3 Scope of the problem

We will first focus on the obscurances, first introduced in [96] and [40], a technique that simulates global illumination and creates realistic images with much less computational effort. We will also point out some techniques to accelerate the computation of true global illumination when computing a series of frames in an animation.

The obscurances are computed to simulate the indirect illumination of a scene. The direct lighting is computed apart and in an independent way. The decoupling of direct and indirect lighting is a big advantage, and we will take profit from this. We can easily add color bleeding effects without adding computation time (introduced in [60]). Another advantage is that to compute the obscurances we only need to analyze a limited environment around the point.

For diffuse virtual scenes, the radiosity can be precomputed and we can navigate the scene with a realistic appearance. But when a small object moves in a dynamic real-time virtual environment, as a videogame, the recomputation of the global illumination of the scene is prohibitive. Thanks to the limited reach of the obscurance computation, we can recompute the obscurances only for the limited environment of the moving object for every frame and still have real-time frame rates (also in [60]).

Obscurances can also be used to compute high quality images, or sequences of images for an animation, in a ray-tracing-like environment (discussed in [62] and in [56]). This allows us to deal with non-diffuse materials and to research the use of a commonly diffuse technique as obscurances in general environments (in [59]). For static cameras, using light animation only affects to direct lighting, and if we use obscurances for the indirect lighting, thanks to the decoupling of direct and indirect illumination, the computation of a series of frames for the animation is very fast (algorithm introduced also in [56]). The next step is to add camera animation, reusing the obscurances results between frames (presented in [58]).

Using this last technique of reusing the illumination of the hit points between frames for a true global illumination technique as path-tracing, we study how we can reuse this information in an unbiased way (in [64]).

Besides, in [57] a study of different sampling techniques for the hemisphere is made, and in [61], obscurances are computed with the depth-peeling technique and using GPU.

1.4 Overview

In this section we overview the structure of this dissertation and summarize its contents. Previous work is divided in two chapters. The first one, chapter two, presents a general overview of the scientific background of global illumination. The second one, chapter three, presents the basic concepts of the obscurances and surveys similar techniques, as ambient occlusion, and their improvements. The core of this dissertation comprises chapters four to seven. Finally, the conclusions and future work are presented.

1.4.1 Chapter 2: Background

In this chapter, a general overview of the scientific background of this dissertation is presented. The chapter is conceived to introduce some concepts needed to understand and situate the context of this work. Starting with some mathematical and physical concepts, we get into the concepts and algorithms to compute global illumination, finishing with a short overview of the previous work in the reuse of paths for global illumination.

We start by reviewing some concepts and techniques of the Monte Carlo probabilistic methods for solving complex equations. Then we move to explain the light as a physical concept in section 2.2 and its computational models (section 2.3). Next some classic algorithms to compute the illumination of a scene are reviewed, as radiosity (section 2.4), ray-tracing (section 2.5), path-tracing (section 2.6) and others (section 2.7). Finally the previous work on reusing paths is reviewed.

1.4.2 Chapter 3: From obscurances to ambient occlusion: A survey

This chapter is conceived as a survey and compiles the basic concepts of obscurances and related techniques as ambient occlusion, their improvements and additions.

1.4.3 Chapter 4: Improving obscurances

In this chapter we look more deeply into the obscurances concept by adding some improvements, studying different options for actual implementations of the technique and introducing some problems that might occur if we use obscurances.

In section 4.1 the effect of color transfer between surfaces is added to the original obscurances equations. In section 4.2, some different functions ρ for the obscurances computation are analyzed and in section and 4.3 we study the effect of different values for the maximum distance parameter d_{max}. Next in section 4.4 a problem of the obscurances and one possible solution is introduced. Next in section 4.5 we study different methods to sample the hemisphere for the obscurances and study their relative efficiencies. Finally, in section 4.6, a different algorithm to account for average intensity and average reflectivity of the scene is introduced.

1.4.4 Chapter 5: Obscurances in diffuse environments: videogames

The idea of obscurances was first thought to be used in videogame environments, by precomputing the obscurance values and saving them in obscurance maps, using these maps in real-time while navigating the scene. As obscurances are computed locally, we show in section 5.1 that they can be recomputed in real-time in the environments of a moving object and we present an algorithm to do that.

The advantage of obscurances with respect to radiosity in this context is that they are much faster to compute, and can even be accelerated by using GPU techniques. In section 5.2 we use the depth peeling technique to compute the obscurances for objects and scenes, using programmable shaders of modern GPU cards.

1.4.5 Chapter 6: Obscurances in non-diffuse environments

There are some environments and materials in which the perceived illumination depends on the point of view, this is, the relative position of the eye and the object with respect to the light. We will use ray-tracing techniques (see [31]) to generate images with obscurances in which the illumination of the scene depends on the point of view (i. e., the scene contains objects with non-diffuse materials) and to include specular and translucent effects.

In this chapter we first envisage how the obscurances can be plugged in ray-tracing-like algorithms (section 6.1) and in another section (6.2) we study how we can modify the obscurances concept to deal with non-diffuse materials.

1.4.6 Chapter 7: Animations: reuse of information between frames

In this chapter we study how we can save computation time when computing a series of frames in an animation and some of the elements of the scene move, as the light sources or the camera.

First we study a technique to reuse information between neighbor frames in camera animation. The technique presented here is general and useful for several techniques that compute radiance of a hit point, including path tracing and obscurances. Using examples of path tracing, an unbiased solution is presented. Next, in section 7.2, we will apply the same technique to obscurances, but simplified, as obscurances are always computed for diffuse materials. A technique to reuse indirect lighting (computed with obscurances) between frames with still camera and moving light sources is presented in section 7.3. Finally (section 7.4) both camera and light sources animation are combined in a single algorithm, leading to a new concept that we will call *frame array*; a multi-dimensional array of frames that can be navigated to form movies that present different combinations of animations of light and camera.

1.4.7 Chapter 8: Conclusions and future work

In the last chapter we present the final conclusions and the main contributions of this dissertation are summarized. We list the publications that support this dissertation, as well. In addition, we envisage the lines for future work.

1.5 Summary

In this chapter we have introduced this dissertation. First we have presented the motivation of this research and next we have presented our main objectives. Then we have introduced a more concrete explanation of the scope of our problems and how the possible solutions are devised. Finally the structure of this document is presented.

Chapter 2

Background

This chapter describes briefly the basic science behind the core of this thesis and some of the previous work in global illumination. We start by reviewing some concepts and techniques of the Monte Carlo probabilistic methods for solving complex equations. Then we move to explain the light as a physical concept in section 2.2 and its computational models (section 2.3). Next some classic algorithms to compute the illumination of a scene are reviewed, as radiosity (section 2.4), ray-tracing (section 2.5), path-tracing (section 2.6) and others (section 2.7). Finally the previous work on reusing paths is reviewed.

2.1 Monte Carlo techniques

The Monte Carlo methods are probabilistic methods used to find an approximate solution to integral equations that have difficult analytical solution. These equations may have a very complex domain or a high number of degrees of freedom, and this makes them hard to solve in classic analytical way. These methods are useful for computations in the matters of physics of light. In this section we will summarize the theoretical basis of Monte Carlo and the related methods, as Quasi-Monte Carlo [67] and Multiple Importance Sampling [89].

2.1.1 How does it work?

We have a function $g(x)$ to be integrated over a given domain D. The Monte Carlo method allows to integrate the function $g(x)$ over its domain D by generating a sequence of independent samples on D according to a probability density function (*pdf*) $f(x)$. The value of the integral can be seen as the expected value of the random variable $\frac{g(x)}{f(x)}$ with pdf $f(x)$ (2.1), and this can be estimated by sampling the variable on D using $f(x)$ as pdf, obtaining the unbiased estimator (2.2):

$$I = \int_D g(x)dx = \int_D \frac{g(x)}{f(x)} f(x)dx = E_f \left[\frac{g(x)}{f(x)} \right] \tag{2.1}$$

$$I \approx \langle I \rangle = \frac{1}{N} \sum_{\imath=1}^{N} \frac{g(x)}{f(x)} \tag{2.2}$$

The samples on D according to the density function $f(x)$ are usually obtained from the inverse of the distribution function $F(x)$. This procedure is known as *inversion method* [67], and consists in computing the sequence of samples x_k from $F^{-1}(\xi_k)$. $< \xi_k >$ is a sequence of realizations of independent random variables with uniform distribution in $[0,1)^d$, d being the dimension of the integration domain. In practice, such a sequence $< \xi_k >$ can be obtained from the $[0,1)$ values provided by the computer random generator.

2.1.2 Error in Monte Carlo integration

Monte Carlo methods are probabilistic, based on sampling values from random variables. The value of the integral is seen as an expected value, and variance must be considered. Let us consider we are integrating a square integrable function, that is, a function that belongs to L^2 [67]. Then the error in the Monte Carlo estimation (or convergence rate) is proportional to $\sqrt{N}$, where N is the number of samples taken. As an example, this means that the number of samples has to be multiplied by 100 to reduce the error by one order of magnitude. The variance for the estimator (2.2), that is, the expected value of the quadratic error for an unbiased estimator, is given by

$$V(\langle I \rangle) = \frac{1}{N}\left(\int_D \frac{g(x)^2}{f(x)}dx - I^2\right) \tag{2.3}$$

2.1.3 Importance Sampling

We can see in equation (2.3) that the variance depends on the probability density function $f(x)$ used in the Monte Carlo sampling. It can be shown that the minimum variance is obtained taking $f(x) = \frac{|g(x)|}{I}$ [44]. Since the value of the integral I is unknown, density functions that mimic the integrand have to be used. These functions are called *importance functions*. The sampling according to these importance functions is called *importance sampling*. In other words, importance sampling consists of sampling more points in the regions where $|g(x)|$ is greater. This technique is widely used in Monte Carlo methods.

2.1.4 Multiple importance sampling

Multiple importance sampling, introduced by Veach [89], is a Monte Carlo variance reduction technique consisting of generating the samples according to n different pdf's, and properly combining these samples in order to obtain low variance estimators.

The multi-sample model allows one to optimally combine n estimators via weighting functions, obtaining a new unbiased estimator. Suppose we are integrating a function $f(x)$ on a given domain (assuming that f can be evaluated for any point on the domain). We can use n sampling techniques, represented by the corresponding probability density functions $p_1, ..., p_n$. Let $N_1, ..., N_n$ be the number of samples taken from each density function, and let $N = \sum_{i=1}^{n} N_i$ be the total number of samples. The multi-sample estimator is defined (see Veach [89]) as

$$F = \sum_{i=1}^{n} \frac{1}{N_i} \sum_{j=1}^{N_i} w_i(X_i^j) \frac{f(X_i^j)}{p_i(X_i^j)}, \tag{2.4}$$

where $w_i(X_i^j)$ is the weight of the j-th sample drawn from p_i. Note that the estimator F is in fact a weighted sum of the Monte Carlo estimators $\frac{f(X_i^j)}{p_i(X_i^j)}$ obtained using each sampling technique i.

The weighting functions $w_i(x)$ can be chosen in different ways, but they have to fulfill two conditions for F to be unbiased:

1. if $f(x) \neq 0$, $\sum_{i=1}^{n} w_i(x)$ must be equal to 1;
2. if $p_i(x) = 0$, $w_i(x)$ must be 0.

These conditions imply that, at any point at which $f(x) \neq 0$, at least one of the $p_i(x)$ must be positive (i.e., at least one sampling technique must be able to generate samples there). Thus, it is not necessary for every p_i to sample the whole domain, namely, some of the p_i can be specialized sampling techniques that concentrate on specific regions of the domain, and are 0 for the rest of it. However, note that each of these pdf's has to have sum 1 for the whole domain.

A good choice for the weighting functions, that fulfills the conditions above, is the balance heuristic. This strategy consists of taking the weight of each estimator proportional to the corresponding pdf times the number of samples for this technique:

$$w_i(x) = \frac{N_i p_i(x)}{\sum_{k=1}^{n} N_k p_k(x)}. \tag{2.5}$$

2.1.5 Quasi-Monte Carlo

A quasi-Monte Carlo method can be seen as a deterministic version of a Monte Carlo method, in the sense that the random samples in the Monte Carlo method are replaced by well-chosen deterministic samples specially designed to be as much evenly distributed on the domain as possible. Thus, quasi-Monte Carlo integration follows the equation (2.2) but replacing Monte Carlo random samples by deterministic samples uniformly distributed on the domain.

We can construct sequences of points uniformly distributed on the domain using both Monte Carlo and quasi-Monte Carlo generation. In this last case we call them *quasi-Monte Carlo sequences* (see [67]). Quasi-Monte Carlo generation designs the sequences of points to be as evenly distributed as possible (or, in simple words, trying to fill empty spaces).

The regularity (even distribution) of the samples on the integration domain happens to be more relevant for integration than true randomness. This is the reason for quasi-Monte Carlo integration to perform better than Monte Carlo integration.

2.2 Physics of the light

Light is a form of energy that the eye receives from the environment and is interpreted by the brain, allowing us to get information about our environment.

The physical properties of the light can be interpreted in two forms: waves or photons. Many of the observable properties of light can be understood if it is regarded as a wave which travels at a finite speed c ($3 \times 10^8 m/s$). But not all properties of light can be explained with Maxwell's electrodynamic theory and for some properties, the light can be understood as a big amount of traveling tiny photons. Some light is emitted from the light sources, as the sun or a bulb, and it travels at a high velocity. Different objects, depending on their material properties, can absorb, reflect or refract the electromagnetic waves (or photons) through which the light is propagated. Absorbed light usually is transformed to another kind of energy as heat.

Light interacts with all near-by materials and some of the originally emitted photons can finally arrive at our eye. The eye receives the signals and transmits them to the brain, that interprets them and gives us information about the colors (that depend on the lightwave longitude), shapes, textures, even the depth (thanks to binocular vision) of the objects of our environment.

The visible light is a part of the spectrum of the electromagnetic radiation. Electromagnetic radiation is classified into types according to the frequency of the wave: these types include, in order of increasing frequency, radio waves (low frequency), microwaves, terahertz radiation, infrared radiation, visible light, ultraviolet radiation, X-rays and gamma rays (high frequency).

In optics, **radiometry** is the field that studies the measurement of electromagnetic radiation in optical spectrum. This part of the spectrum includes infrared, visible and ultraviolet light, while **photometry** is the science that studies the measurement of visible light in units that are applicable to the human eye perception. The difference between both fields is mainly the units of measurement.

Radiant flux Φ is the ratio of **radiant energy** Q per unit of time. **Irradiance** E is the radiant flux coming from any direction per unit of area over a surface. **Radiant intensity** I is the change of power with respect to a solid angle. We can imagine a ray arriving or leaving a point of a surface, the **radiance** L is the radiant power that the ray contains.

Table 2.1 shows a list of these physical magnitudes, their symbols, their equations and the units of measurement for radiometry and photometry.

RADIOMETRY			
Concept	**Symbol**	**Equation**	**Unit**
Radiant energy	Q, Q_e		J (Joule)
Radiant flux	Φ, Φ_e	$\Phi = dQ/dt$	W (Watt)
Irradiance	E, E_e	$E = d\Phi_{in}/dA$	Wm^{-2}
Radiant exitance	M, M_e	$M = d\Phi_{out}/dA$	Wm^{-2}
Radiant intensity	I, I_e	$I = d\Phi/d\omega$	Wsr^{-1}
Radiance	L, L_e	$L = d^2\Phi/d\omega(dA\cos\theta)$	$Wm^{-2}sr^{-1}$
PHOTOMETRY			
Concept	**Symbol**	**Equation**	**Unit**
Luminous energy	Q_v		lms
Luminous flux	Φ_v	$\Phi_v = dQ_v/dt$	lm (lumen)
Illuminance	E_v	$E_v = d\Phi_{v_{in}}/dA$	lmm^{-2}
Luminous emittance	M_v	$M_v = d\Phi_{v_{out}}/dA$	lmm^{-2}
Luminous intensity	I_v	$I_v = d\Phi_v/d\omega$	cd (candela)
Luminance	L_v	$L_v = d^2\Phi_v/d\omega(dA\cos\theta)$	cdm^{-2}

Table 2.1: *Names, symbols, equations and magnitudes of radiometry and photometry*

2.3 Programming the light: local and global illumination

2.3.1 Local illumination and global illumination

To compute the illumination of a virtual environment, we need to find mathematical models that describe the physical interaction of light with the objects and materials. These models have to describe the lighting effects in such a way that the computer can simulate with realism and detail the illumination of the scene. Some examples of lighting effects that increase realism in the scene are: diffuse and specular reflections, shines, refractions, color bleeding, caustics, shadows and shades, participating media, etc.

The computational techniques to simulate illumination can be divided into two kinds: local and global illumination. In **local illumination** only the light that arrives to every surface directly from the light sources is taken into account. This is called **direct lighting**. When using local illumination, the indirect lighting, if used, is normally introduced as a constant ambient value by the user. On the other hand, in **global illumination**, besides direct lighting, the indirect lighting coming from all interreflections between the objects of the scene is taken in to account. For this reason, global illumination is very costly to compute. Different global illumination algorithms are described in [27].

2.3.2 The ray

Light energy travels following a straight line, so the simplest computational model to work with light is the ray, a straight line that has its origin in a point and transports an amount of light energy until it intersects some other point of the scene. Thus, the ray has to have also the capability of intersecting the objects of the scene, this has to be computationally fast and efficient, as millions of rays have to be traced to compute a single image.

2.3.3 The color

The spectrum is simplified into three components that give us an amount of red, green and blue intensities, respectively. Combining these tree values we can obtain many of the colors that the eye can see.

2.3.4 BRDFs and BTDFs

Some of the most important mathematical models to use are those that implement the form in which the light is reflected and/or transmitted in the different surfaces. For each material we can introduce a **bidirectional reflectance distribution function** (BRDF) that models how the light energy reflected gets distributed for each incident angle, and a **bidirectional transmittance distribution function** (BTDF), that works the same but for transmitted light in transparent and translucent objects. The opaque objects can have BRDF's that go from perfectly specular to perfectly diffuse. The perfectly diffuse objects, called also lambertian, are those that distribute part of the incident light energy over a surface point among the hemisphere over this point with probability proportional to the cosinus of the angle to the normal.

We can consider the light as a bundle of rays with direction and intensity. If one of these rays arrives to a perfectly specular surface, it bounces out with a certain angle that depends on the incident angle and an equal or lesser intensity, depending on the **reflection factor** (**absorption**) of the material.

Most of the materials are not ideally specular or diffuse, but semi-speculars. For these kind of materials, the rays bounce off the surface in all directions, but with a higher probability they take a direction close to a certain angle that depends on the incident angle.

For transparent materials, the simplest BRDFs just change the ray angle for every different material the ray traverses, depending on a **refraction index** of every medium. We perceive two effects. First, considering the rays arriving to the eye, we can see through the transparent object the objects that are behind it, but deformed or displaced. Considering light rays, they change their trajectory depending on the shape of the object and produce an effect called **caustics**.

For translucent materials, rays get into a surface and scatter inside the object until they get absorbed or get out through another point of the same or other surface of the object.

Besides BRDF's and BRDFs, there are other functions that take into account other degrees of freedom of the transport of light energy. We can consider non-homogeneous material or all the different wavelengths of the spectrum of a ray, instead of RGB discretization. We can even consider different colors for entering and exitant rays (**fluorescent** materials), or time depending functions (**phosphorescent** materials: those that are capable of keeping light energy and emit it for a while, even when they have stopped receiving light).

2.3.5 The rendering equation

The rendering equation [43] describes light transport in a closed vacuum environment:

$$L(x,w) = L_e(x,w) + \int_S \rho(x,w,w')L(x',w')G(x,x')dA' \tag{2.6}$$

where:

- x and x' are two surface points,
- w and w' are the outgoing directions at x and x' respectively,
- dA' is a differential area at point x',
- $L(x,w)$ is the total exiting radiance (reflected + emitted) at point x in the direction w,
- $L_e(x,w)$ is the emitted radiance at point x in the direction w,
- $\rho(x,w,w')$ is the BRDF (see section 2.3.4),
- $G(x,x')$ is a geometrical term, equal to $\frac{V(x,x')\cos\theta\cos\theta'}{|x-x'|^2}$, where θ and θ' are the angles between the directions w, w' and the respective normals at points x, x', and $V(x,x')$ is a visibility function, equal to 1 if x and x' are mutually visible and 0 otherwise,
- S is the set of surface points.

The rendering equation describes the exchange of energy between all surfaces and the final result is the distribution of light at every point of the environment.

2.4 Radiosity

In essence, radiosity is about simulating the balance of radiant power between the surfaces of the scene. The name of the radiosity family of techniques comes from its objective, that is computing the radiance for each point of the scene, as the light intensity per solid angle emitted at every point.

Proposed by Goral et al. in 1984 [32], the idea was to apply the heat transfer methods to image synthesis. These methods can be applied to light interreflection between lambertian surfaces. In this case, the radiosity at each point is equal in all directions of the hemisphere over the point, so it is independent of the point of view. In the original Goral paper, all patches[1] were visible to all others. Occlusions were introduced later.

The main advantage is that radiosity can be computed previous to the scene visualization, saving its values in textures or in discrete patches, and do a walk-trough over the scene without the need of recomputing all or part of the illumination for each frame.

2.4.1 Radiosity equation

Radiosity, or more commonly in optics, *radiant exitance* is the power per unit area leaving a surface. We recall here the definition of *radiance*, as the power per unit solid angle per unit projected source area:

$$L(A,\theta,\phi)dA\cos\theta d\omega = d^2\phi \tag{2.7}$$

Integrating the previous equation over the hemisphere, we get the total radiant power from dA:

$$d\phi = \int_\Omega d^2\phi = dA\int L(A,\theta,\phi)\cos\theta d\omega \tag{2.8}$$

This expression gives us radiosity per area unit, that for a point x is:

$$B(x) = \frac{d\phi}{dA} = \int L(A,\theta,\phi)\cos\theta d\omega \tag{2.9}$$

2.4.2 Form factors

Now we want to solve a problem: in a diffuse and closed environment, knowing the *exitance*, as the energy leaving from a *patch*, what portion of the flux will arrive to any other patch? Fig. 2.1 shows this problem.

The relative position and orientation of both patches $A_\imath$ and $A_\jmath$ are arbitrary. Patch $A_\imath$ is an emitter that sends a certain quantity of the flux $\Phi_\imath$, while $A_\jmath$ receives a portion of the emitted flux ($\Phi_{\imath\jmath}$). The relation $\Phi_{\imath\jmath}/\Phi_\imath$ is called **form factor** from $A_\imath$ to $A_\jmath$ and will be represented as $F_{A_\imath - A_\jmath}$ or $F_{\imath\jmath}$. Total emitted flux by $A_\imath$ is $\Phi_\imath = M_\imath A_\imath$, where $M_\imath$ is the emitted energy and $A_\imath$, its area. Unfortunately, computing $F_{\imath\jmath}$ analytically can be extremely difficult.

Consider two differentials of area $dA_\imath$ and $dA_\jmath$ as it is shown in fig. 2.2, where $dA_\imath$ is the emitter. The fraction of flux emitted by $dA_\imath$ and received by $dA_\jmath$ is the differential of form factor from $dA_\imath$ to $dA_\jmath$ and is represented as $dF_{dA_\imath - dA_\jmath}$.

[1] *Patch* is defined as each one of the small parts of the scene into which the surfaces are divided to account for their radiosity.

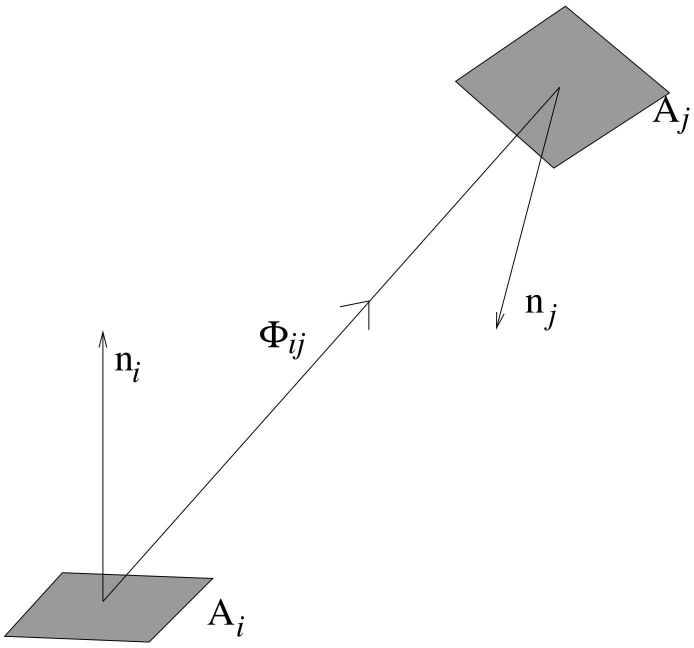


Figure 2.1: Patch A_j receives the flux Φ_{ij} from A_i.

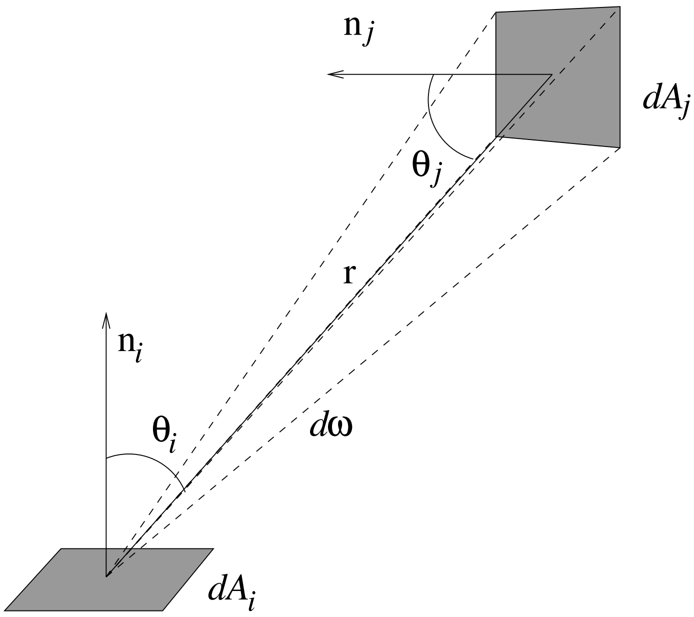


Figure 2.2: Geometry of the form factor between two differentials of surface patches.

Computing form factors is a pure geometrical function, so no emittances or reflectances of patches are involved. The form factor between two patches depends on the distance and inclination that they respectively have, assuming visibility between them:

$$dF_{dA_\imath - dA_\jmath} = \frac{\cos\theta_\imath \cos\theta_\jmath}{\pi r^2} dA_\jmath \tag{2.10}$$

where $\theta_\imath$ is the angle between normal vector at $dA_\imath$ and the vector from $dA_\imath$ to $dA_\jmath$, $\theta_\jmath$ is the angle between normal vector at $dA_\jmath$ and the vector from $dA_\jmath$ to $dA_\imath$, and r is the distance between the two areas.

Integrating (2.10), we get the form factor of a differential of area $dA_\imath$ to a finite area:

$$dF_{dA_\imath - A_\jmath} = \int_{A_\jmath} \frac{\cos\theta_\imath \cos\theta_\jmath}{\pi r^2} dA_\jmath \tag{2.11}$$

Form factor from $A_\imath$ to $A_\jmath$ is obtained integrating (2.11) for every point of $A_\imath$:

$$F_{\imath\jmath} = \frac{1}{A_\imath} \int_{A_\imath} \int_{A_\jmath} \frac{\cos\theta_\imath \cos\theta_\jmath}{\pi r^2} dA_\jmath dA_\imath \tag{2.12}$$

Visibility has to be taken into account, though: two patches may not see each other. In this way we include in equation (2.12) the operator $V_{\imath\jmath}$, that has value 1 if there is visibility and 0 otherwise:

$$F_{\imath\jmath} = \frac{1}{A_\imath} \int_{A_\imath} \int_{A_\jmath} \frac{\cos\theta_\imath \cos\theta_\jmath}{\pi r^2} V_{\imath\jmath} dA_\jmath dA_\imath \tag{2.13}$$

The form factor as it is explained here does not take into consideration the media between the patches: there is no absorption, refraction or scattering of light. In other words, we have non-participative media.

To sum up, the form factor is a non-dimensional constant that represents the fraction of the light emitted from a patch and received by another one. It provides information about relative position and orientation between surfaces and their visibility.

The form factors have these properties:

- Energy conservation: $\sum F_{\imath\jmath} = 1$, all emitted energy arrives to some place, no energy is lost in the way,
- $F_{\imath\imath} = 0$, the form factor of a patch with himself is always zero for flat or convex surfaces,
- reciprocity: $A_\imath F_{\imath\jmath} = A_\jmath F_{\jmath\imath}$,
- additivity: $F_{\imath(\jmath \cup k)} = F_{\imath\jmath} + F_{\imath k}$ where $\imath$, $\jmath$, k are different patches.

2.4.3 Solving the radiosity matrix

If patches $p_\imath$ and $p_\jmath$ are lambertian surfaces, form factor $F_{\imath\jmath}$ points out the fraction of flux emitted by $p_\imath$ and received by $p_\jmath$. Reciprocally, factor $F_{\jmath\imath}$ indicates proportion of flux emitted by $p_\jmath$ and received by $p_\imath$. In any case, we must remind that form factors themselves are not considered flux, only proportions.

Flux leaving $p_\jmath$ is $\Phi_\jmath = B_\jmath A_\jmath$. The fraction of flux received by patch $p_\imath$ is $B_\jmath A_\jmath F_{\jmath\imath}$. From this one, the flux immediately reflected by $p_\imath$ is $\rho_\imath B_\jmath A_\jmath F_{\jmath\imath}$, where $\rho_\imath$ is the reflectance of $p_\imath$. In this way:

$$B_{\imath\jmath} = \rho_\imath B_\jmath A_\jmath F_{\jmath\imath} / A_\imath \tag{2.14}$$

where we define $B_{\imath\jmath}$ as the exitance of $p_\imath$ because of the flux received from $p_\jmath$. Using the reciprocal relation, we can write the later expression as:

$$B_{\imath\jmath} = \rho_\imath B_\jmath F_{\imath\jmath} \tag{2.15}$$

To compute final exitance $B_\imath$ from patch $p_\imath$, we have to take into consideration flux received by $p_\imath$ from all other patches $p_\jmath$. In this way,

$$B_\imath = E_\imath + \rho_\imath \sum_{\jmath=1}^{n} B_\jmath F_{\imath\jmath} \tag{2.16}$$

where $E_\imath$ is the initial exitance (or **emittance**) of patch $p_\imath$ and it is the radiosity emitted by patch $p_\imath$ by itself. If $E_\imath > 0$, patch $p_\imath$ is (or is part of) a light source.

When we isolate term $E_\imath$ we get:

$$E_\imath = B_\imath - \rho_\imath \sum_{\jmath=1}^{n} B_\jmath F_{\imath\jmath} \tag{2.17}$$

We can express this equation for all patches p_1 to p_n as a set of n linear equations, that can be presented in matrix form:

$$\begin{bmatrix} E_1 \\ E_2 \\ \cdots \\ E_n \end{bmatrix} = \begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \cdots & -\rho_2 F_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \cdots & 1-\rho_n F_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \cdots \\ B_n \end{bmatrix} \tag{2.18}$$

and can also be expressed as $E = (I - T)B$ where I is the $(n \times n)$-identity matrix, B is the final $(n \times 1)$-vector of exitances, E is the $(n \times 1)$-vector of emittances and T is a $(n \times n)$-matrix, where its $\imath\jmath$ elements are $\rho_\imath F_{\imath\jmath}$.

In conclusion, to solve radiosity we need to solve a set of linear equations that contain all surface reflectances, the form factors and the exitances of the different patches. We also need the emittances of the patches, as they will be the known initial values that will allow us to solve the equation. The hard part of all this work is to previously determine the form factors for every pair of patches.

These equations are solved for every needed wavelength (usually, red, green and blue). The system can be solved in many different ways, like direct inverting the matrix (Gauss), Gauss-Seidel method, etc. Refer to [23] for details of these methods.

2.4.4 Cornell Box

To test the realism of radiosity algorithms, in Cornell University a box with intensely painted inner walls was constructed. It included a cube and an cuboid (rectangular parallelepiped). All this composition was illuminated by a squared lamp from the ceiling. Fig. 2.3 shows one of the simulations of radiosity algorithm. We will use this model to show the results of some of our algorithms in the core of this thesis.

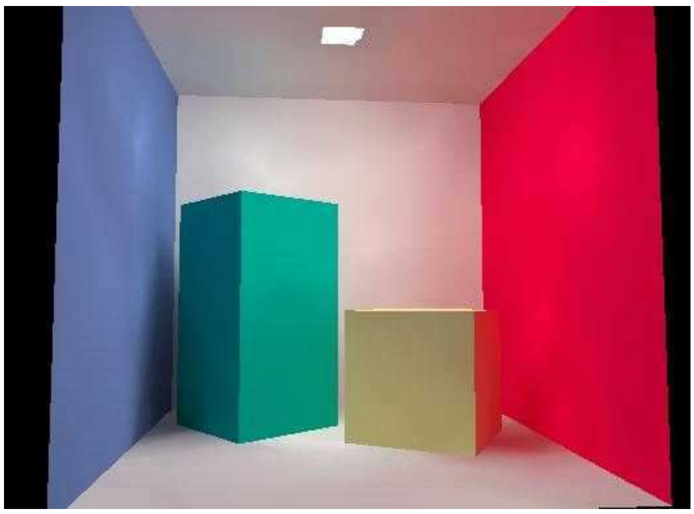


Figure 2.3: Cornell box.

2.5 Ray-tracing

The aim of the ray-tracing family of techniques [31] for rendering is to compute a color for each pixel in an image or in a set of images. The three main elements in ray-tracing are the camera (and the eye, among its parameters), the grid of pixels in the image and the scene. From the eye, one or more rays are cast through each pixel that are intersected with the scene; this is how we find the hit points. Knowing a hit point we can compute its illumination following any suitable technique. By sending rays from the hit point to the light sources and computing visibility, we take care of direct illumination. The objects that are not directly illuminated are visualized by means of an ambient light intensity.

The main advantage of ray-tracing with respect to radiosity is its capability of computing the illumination of perfectly reflective and refractive materials, as the change of directions of the light ray is well known physically and easy to compute. Due to the dependence on the eye position, radiosity techniques do not suit. On the other hand, real-time algorithms to compute and visualize ray-traced images are still far from being realized nowadays, though some attempts have been made. Some interactive ray-tracing with global illumination effects has been achieved in a cluster of parallel personal computers thanks to the increase in computing speed, optimized acceleration structures and extensive use of cache [91].

2.5.1 The concept

The couple of linked words *ray-tracing* may refer to different things. The most common meaning in computer graphics is the idea of finding efficiently intersections between a ray and the objects of a scene. This simple concept can be used in many different algorithms, and ray-tracing may also refer to the family of algorithms that are based on casting rays from the eye through the pixels of an image to find the colors of the picture. There are also many algorithms, including global illumination and ambient occlusion ones, that use ray tracing for computing images, but they have very little in common with the classical ray-tracing algorithm for rendering.

The main idea behind any kind of ray-tracing algorithm is to efficiently find intersections of a ray with a scene, defined as a set of geometric primitives. The ray $R(t) = O + tD$ is described by its origin O and direction D, and may additionally have a parameter t_{max} that specifies a *maximum distance* up to which the ray is looking for intersecting objects.

At this level we have three different problems to solve: finding the *closest* intersection, finding *any* intersection and finding *all* the intersections. To find the closest intersection is the most fundamental operation in a ray tracer. It requires to find the closest geometric primitive, the intersecting point and the distance to it. Additionally, other parameters may be required for a later shading of the point, such as local surface properties or surface normal. Finding any intersection is the problem commonly known as *visibility*. This is a simpler problem than finding the *closest* intersection. Finding *all* intersections is a less common problem used on some lighting algorithms such as in [75].

The scene consists in a set of geometric primitives. These primitives can be any kind of object capable of being intersected by a ray, as polygons, triangles, cones, spheres or parametric surfaces. In [31], an excellent overview of different ray-primitive intersections can be found.

2.5.2 Algorithm: recursive ray-tracing

The idea of using ray-shoting for computing images was introduced by Arthur Appel [11] as an alternative method for solving the *hidden surface* problem for rendering solid objects.

The rays are generated from *the eye* (the focal point) of the virtual camera through each pixel, traced into the scene, and the closest object is determined (see fig. 2.4). The color of this ray is then determined based on the properties of this object. Additionally, visibility rays to the light sources may be shot to account for direct lighting.

In [95] the recursive form of the ray-tracing algorithm was introduced. It casts secondary rays to account for indirect effects like reflection or refraction, by following the path of the reflected or refracted ray (using well-known physical equations) and recursively compute direct and ambient lighting.

Cook [24] extended the range of ray-tracing effects to include more realistic effects like glossy reflections, smooth shadows from area light sources, motion blur and depth of field. This was achieved by modeling all these effects with a probability distribution, which allowed for computing them via stochastic sampling. Originally this algorithm was named *distributed* ray-tracing, but due to the later use of parallelization techniques for ray-tracing and the common use of the word *distributed* for parallel algorithms running in different CPU's and different memory spaces, Cook's enhancement of ray-tracing changed its name

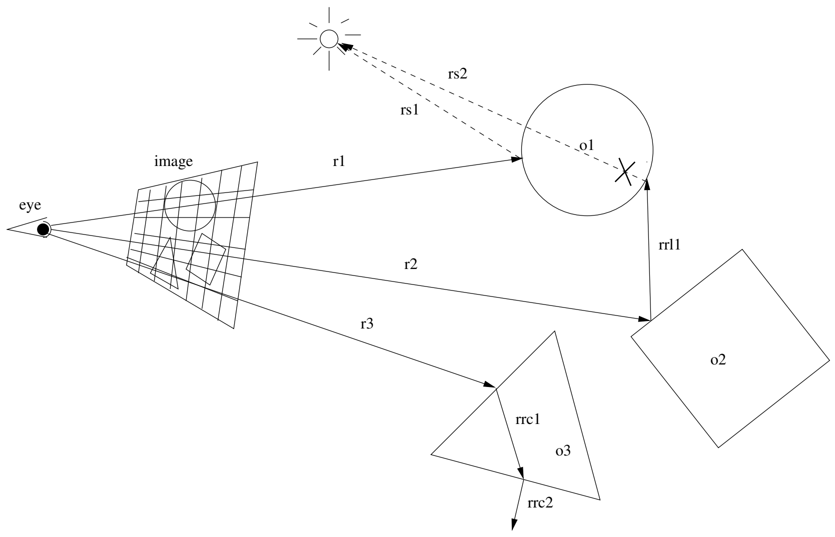


Figure 2.4: Schema of the ray-tracing algorithm. The rays are cast from the eye through the pixels to intersect the scene. In this example, ray **r1** hits a diffuse object **o1** and its direct illumination is computed shooting a shadow ray **rs1**. Ray **r2** hits a specular object **o2** that reflects it (**rrl1**) to the diffuse object o1. A shadow ray **rs2** is also cast at this point but there is no visibility and the point remains in shadow, computing only its ambient light. The third ray **r3** hits a transparent object **o3** that refracts the ray (**rrc1** and **rrc2**), and continues its path recursively.

to *distribution ray-tracing.*

2.5.3 Acceleration techniques

Computing a single image with ray tracing usually involves millions of rays and their respective intersections with the scene, specially since the introduction of Cook's distribution ray tracing. That is why lots of research has been done aiming to reduce this high cost and/or accelerate ray-tracing computation.

One possible approach is to reduce the number of primary rays to cast in the image plane. This can be done using some kind of adaptive sampling, i. e., sending more rays for those pixels that present high variation of sampled values and less rays for those pixels that have always the same or similar values. A closer description of this method appears in [31]. Other methods to reduce primary rays from the eye include *vertex tracing* [88] and *render cache* [93]. The number of secondary rays can also be reduced by taking advantage of shadow coherence in the case of direct illumination rays or other techniques for reflected and refracted rays. These techniques are briefly surveyed in [91].

Also surveyed in [91], the most interesting techniques to accelerate ray-tracing are the ones based in accelerating the intersections themselves, by reducing the number of intersections of the ray with the scene, or by using some kind of acceleration structure for the scene. For complex objects it is usually better to try to intersect first the ray with a more simple structure as its bounding volume. Another option is to use some spatial subdivision as grids, octrees, BSPs, etc., to first try to intersect the ray with the closest environment. This last technique will be very useful for obscurances, as we only need to look for intersections in a limited environment.

2.6 Path-tracing

The path-tracing algorithm [43] is based in the same principles that ray-tracing, but aimed to obtain a global illumination image. A ray is cast from the eye to the scene through a pixel, and direct lighting computed the same way as ray-tracing, but instead of computing local illumination for indirect lighting using an ambient intensity, an stochastic path is followed while collecting a sample of the indirect illumination for the hit point. Casting many paths per pixel we obtain an approximation for the final color of the pixel. The main problem of this method is the slow convergence to a valid noiseless solution.

2.6.1 The rendering equation and random walks

The *path-tracing* can be included in the class of illumination algorithms that solve the rendering equation (see section 2.3.5) using random walks.

Random walks are a Monte Carlo common tool to solve second kind Fredholm integral equation [73, 36, 44]. For instance, they are used in global illumination and radiosity [13, 27] to solve the *rendering equation* using the path-tracing algorithm [43].

Besides *path-tracing*, other algorithms as *distribution ray-tracing* [24], *bidirectional path-tracing* [52, 89], *photon map* [41] and *Metropolis light transport* [90] are main global illumination techniques that use random walk.

2.6.2 The algorithm

In the path-tracing technique (see Fig.2.5) an image is computed by tracing paths from the eye (or observer position) trough the pixels that compose the image plane towards the surfaces of the scene. Once the hits are found by tracing the primary rays, a sample of the radiance arriving to the eye through the ray is computed using the rendering equation (see section 2.3.5). The most simple algorithm to compute the radiance value is to apply a basic and straightforward Monte Carlo integration scheme (see section 2.1) to the rendering equation in its hemisphere form:

$$L(x \to \theta) = L_e(x \to \theta) + \int_{\Omega_x} L(x \leftarrow \psi)\rho(x, \theta \leftrightarrow \psi)\cos(\psi, N_x)d\omega_\psi \tag{2.19}$$

The integral is evaluated by Monte Carlo by generating N random directions $\psi_\imath$ over the hemisphere Ω_x, distributed according to some probability density function $p(\psi_\imath)$:

$$\langle L_r(x \to \theta)\rangle = \frac{1}{N}\sum_{\imath=1}^{N} \frac{L(x \leftarrow \psi_\imath)\rho(x, \theta \leftrightarrow \psi_\imath)\cos(\psi_\imath, N_x)}{p(\psi_\imath)} \tag{2.20}$$

The cosine and the BRDF are evaluated easily by accessing the scene description, but $L(x \leftarrow \psi_\imath)$, the incident radiance at x, has to be evaluated. Since

$$L(x \leftarrow \psi_\imath) = L(r(x, \psi_\imath) \to -\psi_\imath), \tag{2.21}$$

we need to trace the ray leaving x in direction $\psi_\imath$ through the environment to find the closest intersection point $r(x, \psi_\imath)$. At this point, another radiance evaluation is needed. Thus, we have a recursive procedure to evaluate $L(x \leftarrow \psi_\imath)$, and a path, or a tree of paths, is traced through the scene.

2.6.3 Direct and indirect illumination

With previous algorithm, the path only gets to transport some lighting if at any time it hits an emissive surface. As the light sources are usually small compared to the other surfaces, many of the radiance evaluations yield to zero. This causes very slow convergence.

The solution of this problem comes from separating direct and indirect computation of the illumination. The reflected radiance term of the rendering equation can be split in two parts: a term that describes the direct illumination due to the light sources and one that describes the indirect illumination. At any hit, a ray can be shot in the direction of a light source to account for visibility and thus direct lighting. Additionally the path is continued recursively to account for indirect lighting. If by chance this path hits a light source, its emissive radiance contribution is not used (though a direct light ray is shot in anycase and possibly hit the same object or other light source).

By sending paths to light sources explicitly, accurate pictures are obtained much faster.

2.6.4 Termination conditions: Russian Roulette

The recursive path described in the simple stochastic path-tracing algorithm needs a stopping condition to ensure a finishing algorithm. This condition must not introduce any bias in the computation and a correct solution has to be obtained. A possible technique is to cut the path at a fixed length. This puts an upper bound on the amount of rays that need to be traced, but important light transport might have been ignored and leads to a biased solution.

Russian Roulette is a technique that addresses the problem of keeping the lengths of the paths manageable, but at the same time leaves room for exploring all possible paths of any length. The technique works as follows. At a hit point, we decide at random whether to terminate or follow the path with probability equal to α, that we will call *absorption probability.* If α is small, the recursion will continue many times, and the final estimator will be more accurate. If α is large, the recursion will stop sooner, but the estimator will have a higher variance. We can choose α to be a value directly related to $1 - albedo$ (as the hemispherical reflectance of the material of the surface) of the point, thus, dark surfaces will absorb the path more easily, while lighter surfaces have a higher chance of reflecting the path. This corresponds to the physical behavior of light incident on these surfaces.

Another possibility is to accumulate the albedos of visited hit points and stop when the accumulated albedo is below a given threshold.

2.6.5 Some discussion

The main drawback of these algorithms based on Monte Carlo random walks is however the high number of paths needed to obtain an acceptable result. As the variance of the estimators is proportional to N^{-1} with N the number of paths, it makes necessary the use of many paths, of the order of millions, to obtain an acceptable noiseless image. This is still more dramatic in an animation computation, due to the high number of frames to be computed. Thus achieving some sort of path reusing (see section 2.8) can reduce the computational cost.

2.7 Other Global illumination techniques

Ray-tracing and path-tracing are eye-driven (named also *gathering*), this is, as only the light that arrives to the eye is taken into account to "see" the scene, at first thought it does not make sense to simulate the paths of the light that arrive somewhere else different than the eye. Then the paths of the light are simulated inversely: the paths that the light has to be done to arrive finally to the eye are tracked. But there are some situations in which it is difficult to compute the indirect lighting, as when the light sources are somewhat hidden and the random paths find hits that seldom have a direct link to the light source. There are also some light effects as caustics that are difficult to compute with eye-driven paths.

For these reasons, there is a collection of algorithms that actually simulate the paths of the photons from the light sources and save in some way their positions, intensity and incoming directions when they hit the objects in the scene. In photon mapping (see [41, 42]), the algorithm is divided in two stages. First, the photon tracing that simulates the paths of the light from the light sources, and stores the photon information of the scene, commonly in a kd-tree structure. Then, the photon gathering, that finds what photons are seen in every pixel and using a technique called "density estimation", computes an approximation of the radiance to the eye.

In bi-directional path-tracing[52] there is no need of saving the positions for all photons traced from the light sources. The idea is to find a stochastic path of the photon from the light, and another path of a possible photon that arrives to the eye. The different segments of both shooting and gathering paths can be combined resulting in several paths that a photon can trace to arrive to the eye, each with its own probability and giving a sample of the solution. These samples are combined by multiple importance sampling (see section 2.1.4).

2.8 Reuse of paths

Bidirectional path-tracing [52, 89], previously presented in section 2.7, can be considered as the first attempt to reduce the cost by reusing paths in the context of global illumination. They consider gathering paths from the eye through the image pixels and shooting paths from the light sources, and connect the hit points of shooting and gathering paths using *shadow (visibility) rays.*

In the radiosity context, Besuievsky et al [14] used the same set of lines to expand direct illumination from different light-source positions, which were packed in a bounding box. Lines crossing this box expanded the power of all intersected positions. This method had a drawback: lines would be wasted if the source positions were not tightly packed. Also Besuievsky et al [16, 15] presented the multi-frame lighting method, valid for animations with objects that move in known-in-advance trajectories. In this approach, all the frames are merged in a single one, with the moving objects replicated as many times as frames in the animation.

2.8.1 Reuse of gathering paths in path-tracing

The idea of the reuse of full paths and for different states (i.e., pixels, patches or light sources) was first presented by John Halton in [35] in the context of the random walk solution of an equation system.

This technique was applied by Bekaert et al. in the context of path-tracing in [12] (see Fig. 2.5), combined with multiple importance sampling [89] to avoid bias. Pixels were grouped in tiles, and paths belonging to a pixel in the tile were reused for the other pixels in the tile from the second hit point of the path. A speed-up of one order of magnitude was reported for fairly complex scenes.

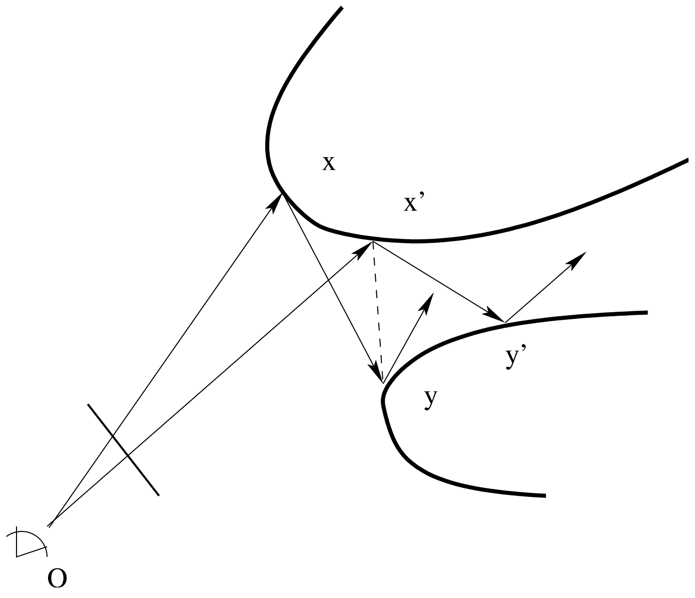


Figure 2.5: Reusing a path, in the context of the path-tracing algorithm, from the second hit point, y, for a single observer O, creating thus the new path $O, x', y, \ldots$ at the cost of the visibility test $vis(x', y)$

Sbert et al [76] presented a theoretical framework valid to reuse paths in gathering random walks, and outlined applications to radiosity and global illumination.

2.8.2 Reuse of shooting light paths

Path-reuse has been applied to light source animation in [77, 79]. Both papers apply the idea of path reuse but from the point of view of the shooting paths from moving light sources. The first shot is then connected to previously computed paths and the samples weighted by multiple importance sampling. In [77] it is done using random walk algorithm for radiosity. In [79], a fast light source animation algorithm based on the virtual light sources illumination method, due to Wald et al [92], is presented. They reused paths in all frames (and not only in the frame in which paths were obtained). This algorithm presented a speed up close to the number of frames of the animation and interactive rates are achieved.

2.8.3 Reuse of hits for neighbor eyes

Havran et al. [38] presented the reuse of paths in a walk-through, that is, when the observer changes position. Paths cast from one observer position were reused for other neighbor positions. Their technique admitted motion blur and they applied it in the context of bidirectional path-tracing. Although obtaining a high speed-up, the method remained biased as the samples were not weighted with the respective probability. Also Havran et al [37] presented a technique that aimed at exploiting temporal coherence of ray cast walkthroughs. They reused ray/object intersections computed in the last frame of the walkthrough for accelerating the ray casting in the current frame.

2.9 Summary

We have seen in this chapter some of the concepts and previous work needed to understand the core of this thesis and at the same time we have situated the context of the subsequent chapters.

We have briefly reviewed basic mathematical concepts as Monte Carlo integration. Then we have summarized the physics of the light in the context of optics and how to model it to do computations and create algorithms that simulate the behavior of the light. Then we have seen some of the classic global illumination algorithms: radiosity, ray-tracing and path-tracing. Besides the importance that these algorithms have in the field of rendering, we thought necessary to include them here because the obscurances technique, that is the basic concept that guides this dissertation, is programmed under the environment of both radiosity and ray-tracing. In some cases it is also compared to the results of true global illumination techniques as path-tracing and radiosity. Finally a brief introduction to the algorithms that reduce cost by reusing paths in the context of global illumination is presented.

Chapter 3

From obscurances to ambient occlusion: A survey

This chapter aims to survey the previous and related work for the obscurances and similar techniques.

3.1 Introduction

In 1998, Zhukov et al.[96] presented the paper "An Ambient Light Illumination Model". It introduced the *obscurances*, an efficient technique that achieved in a much less costly way some of the features that only global illumination techniques had presented before. Those features included the realistic appearance of objects diffusely illuminated, presenting a darker aspect in those zones that are more occluded. The obscurances technique was presented to be used in videogame environments, thus precomputed and stored in texture maps attached to the objects.

In 2002, Landis[53] and Bredow[17] presented in the SIGGRAPH course "RenderMan in Production" a technique that they had been using in the movies *Pearl Harbor* and *Stuart Little 2*, respectively, and they called it **ambient occlusion**. This technique was clearly based on the obscurances concept by Zhukov et al. but ported to production rendering for movies that mixed live footage with computer generated imagery (CGI), that is, programmed and rendered in a ray tracing environment as, for example, Renderman.

Since then the concept has popularized under the name of ambient occlusion and it is included as a shader in most commercial renderers. Nowadays, many of the movies that include CGI use ambient occlusion as a cheap way to simulate diffuse ambient illumination in substitution of previously common "bounce lights". Ambient occlusion effect, used altogether with environment lighting and some other tricky effects (as caustics) are the state of the art for illumination in production rendering.

Meanwhile, in academic environments obscurances and ambient occlusion techniques are still under research in both movies and videogames facets. Many different techniques to compute ambient occlusion have appeared. Some of these techniques are aimed to solve particular cases as self-occlusion for an object, occlusion between different objects, indirect illumination for closed scenes, sky-dome illumination, and changes of ambient occlusion for moving or deformable objects. Some others are aimed to accelerate the computation itself by taking advantage of the good features of the obscurances concept, or by adapting the algorithms to be used with graphics hardware. Finally, some improvements to those techniques are made.

In this article we survey the state of the art for the techniques of obscurances and ambient occlusion. First, in next section, we explore two techniques that appeared some years before the obscurances and use the same idea. Then in section 3.3 we explain the basic concepts and its characteristics. Then we take a look at the different improvements and accelerations for these techniques (sections 3.4 and 3.5). Next some comparative tables are shown (section 3.6) and we take a look at some current examples of the use of the techniques in the industry (section 3.7). Finally we present our conclusions.

3.2 Previous work

3.2.1 Shading particle systems

Many years before obscurances or ambient occlusion were invented, a similar idea was used in [70] for the rendering of grass and trees in the first animated short movie "The adventures of André and Wally B.". They rendered the vegetables using a kind of particle system to avoid the huge amount of memory (for that time) that a complete polygonal model of a tree needs. They used three components of light for the illumination of the trees, that they called ambient, diffuse and specular shading. These concepts were used slightly different as what they mean nowadays. Diffuse shading was a representation for the direct lighting coming from the Sun. Ambient shading was a representation for the light coming from the environment, i.e., indirect lighting including a minimum constant lighting. In the case of the diffuse (direct) and ambient (indirect) shading for trees, the lighting was negative-exponentially attenuated for the particles that represented the leaves, depending on the distance to the boundary of the tree in the direction to the position of the Sun in the first case, and to the closest boundary in the second case, both using a random component to cause the effect of light occlusion for the leaves in a tree, some more illuminated and some less (see fig. 3.1). The idea is quite similar to obscurances, as the more hidden is a surface, the darker it is seen. A modern approach for trees and plants can be found in [39] and is reviewed in section 3.5.4.

3.2.2 Accessibility

The aim and use of Miller's approach [65] is different from obscurances, though its results may be somewhat related. Miller defines the "accessibility of an object" as a property of the surfaces of an object that can not be reached by a "spherical probe" as a simplification for a sandpaper (to polish) or a cloth (to clean the dust). Thus, the more hidden parts of the objects remain uncleaned or unpolished and can be shaded in a different way (different color and/or texture). Though the idea and the proposed solution is completely different, in some cases the resulting images look similar to those computed with ambient occlusion, specially when the shading of the unreached surfaces is darker than the accessible ones. It is not by chance that if we change slightly this concept to obtain the "accessibility of light" we get the ambient occlusion concept.

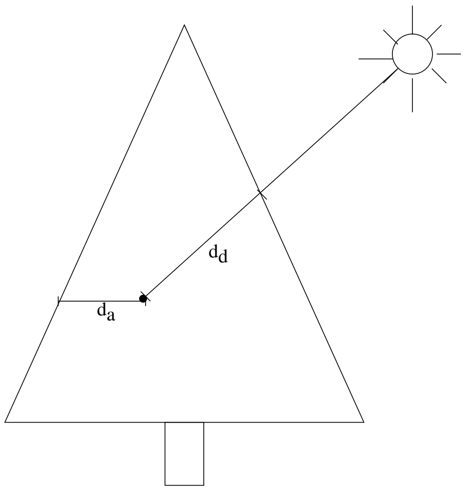


Figure 3.1: In [70], direct and indirect illumination are attenuated using distances d_d and d_a respectively and disturbed by a random component.

3.2.3 Extended Ambient Term

In [19] the Extended Ambient Term is presented. As in classic ambient term, this extended version ignores geometric occlusion conditions to compute an approximation to indirect illumination, but adds the idea of computing not a single ambient term, but a set of them.

The polygons of the scene are classified depending on their orientation in a given number C of classes (this number will usually be 6 according to an orthogonally oriented cube) and the distribution of unshot power of the classes is then estimated by solving a small system of linear equations. If a polygon is not orthogonally oriented, it will be assigned different weights for each class. The unknowns are the indirect lighting colors (ambient terms) for each class, and the wanted solution.

3.3 Theoretical basis

3.3.1 Observing reality

Let us take a look at the illumination of the objects in the real world. Imagine we are in an environment where the illumination is mostly diffuse and indirect, as if a clear white wall dominated the scene or we were in open air in a cloudy day. In these cases, the objects that are more hidden are seen darker, as the indirect light that comes from mostly everywhere is occluded by other objects. We just have to look between the keys of our keyboard or the piece of road under a car in a cloudy day to appreciate this effect.

Modeling this effect, that we will call *obscurances* from now on, following [96], is much more simple and much less costly than global illumination. In global illumination we need to simulate the interaction of light between all objects of our scene. The obscurance effect can be considered as a pure geometric property of every point in our scene: we just need to evaluate the *hiddenness* or occlusion of the point by considering the objects around it.

3.3.2 The obscurances illumination model

Let us move on again to the virtual world. The obscurances are introduced in [96] and [40]. They first assume there is no specific light source in the virtual scene, but a diffuse ambient light that comes from everywhere.

The obscurance of a point P is defined as the integral over the hemisphere centered on the point P of a function ρ of the distance d from the point P to the nearest object in all directions ω of the hemisphere, cosine weighted.

$$W(P) = \frac{1}{\pi} \int_{\omega \in \Omega} \rho(d(P, \omega)) \cos\theta d\omega \tag{3.1}$$

where

- $\rho(d(P, \omega))$: function with values between 0 and 1, and giving the magnitude of ambient light incoming from direction ω
- $d(P, \omega)$: distance of P to the first intersected point in direction ω
- θ: angle between direction ω and the normal at P
- $1/\pi$ is the normalization factor such that if $\rho() = 1$ over the whole hemisphere Ω then $W(P) = 1$.

The function ρ is a monotone increasing function of the distance d. It is defined for all positive values and results in 0 when d is 0. From 0 to a determined value d_{max}, the function increases from 0 to 1, and for values greater than d_{max} the returned value is 1. This means that we only consider a limited environment around the point P and beyond it we will not take care of the occlusions. The shape of the function ρ is shown in figure 3.2.

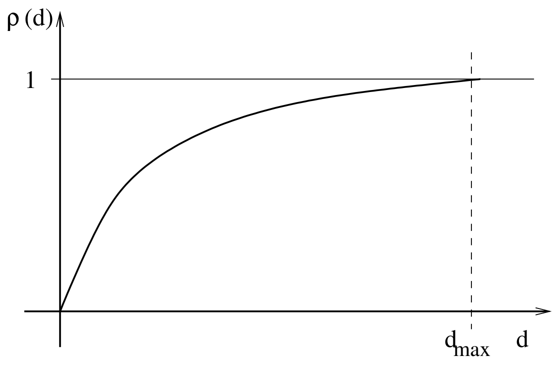


Figure 3.2: Shape of function $\rho(d)$.

In this way, the integral function $W(P)$ captures mathematically the geometric properties of the environment of the point P. If we take a look at the extreme cases, an obscurance value of 1 means that the point is completely open (not occluded) and a value of 0 would mean that it is completely occluded. This would be a very strange case, there exist points with an obscurance value of 0, but as they are completely occluded, we can not see them.

The d_{max} (maximum distance) is the main parameter that controls the indirect illumination computed with obscurances. The user chooses the value of d_{max} depending on the quantity of shadow she/he needs for the scene. It should be in concordance with the relative sizes of the objects with respect to the scene and with the size of the scene itself. It should be much larger, for example, if we compute a view of a stadium than if we compute a close-up of a foot and a ball.

In sections 4.2 and 4.3 you can find a deeper study on the ρ function and the d_{max} parameter.

Obscurances with light sources

The obscurances illumination model is thought to be used to simulate the indirect lighting caused by diffuse interreflections between objects in a fast and simple way. The direct lighting has to be computed apart and in an independent way. Fast, simple and known techniques to compute direct lighting are commonly used. We can take any of them and add the direct lighting results to the indirect lighting computation.

For more realistic results, the indirect lighting computed with obscurances has to correlate with physically realistic indirect lighting. In particular, an image of a scene computed with any global illumination technique, as path tracing, and an image of the same scene with the same camera and light sources computed with obscurances have to look similar, specially in average intensities.

For this reason, the obscurance value of a point has to be used in the following way to obtain its indirect illumination:

$$I(P) = R(P) \times I_A \times W(P) \tag{3.2}$$

This is, the obscurance at that point is multiplied by the diffuse reflectivity $(R(P))$ at the point and by an average intensity value (I_A) of the whole scene.

Ambient light

The average ambient intensity (I_A) is computed assuming that light energy is distributed in a uniform way among the whole scene and illuminates all objects with the same intensity. In this way I_A is:

$$I_A = \frac{R_{ave}}{1 - R_{ave}} \times \frac{1}{A_{total}} \sum_{\imath=1}^{n} A_\imath \times E_\imath \tag{3.3}$$

where A_{total} and $A_\imath$ are the total area and the area of each patch respectively, $E_\imath$ is the emittance of the patch and R_{ave} is the average reflectivity of all patches weighted by the area:

$$R_{ave} = \frac{1}{A_{total}} \sum_{\imath=1}^{n} A_\imath \times R_\imath \tag{3.4}$$

3.3.3 Ambient occlusion as a simplification for obscurances

The obscurances as described previously have inspired a family of techniques that nowadays are implemented in most render software packages commonly used by the animation and videogame industry.

It all started in a course about the widely known and used software package RenderMan at SIGGRAPH 2002, where two of the speakers talked about a technique that they called *ambient occlusion* and had been using each one in a different movie that same year. Hayden Landis [53] from *Industrial Light & Magic* used the ambient occlusion for self-shadowing of objects to "attenuate properly the surfaces not fully exposed to the environment". The examples were the planes from the movie *Pearl Harbor.* They got the unoccluded lighting from a blurred environment map looked up in a direction that averaged the unoccluded zone of the surroundings of the hit point and called that direction *bent normal.*

The ambient occlusion defines the percentage of occlusion of a point on a surface according to equation (3.5).

$$AO(P) = \frac{1}{\pi} \int_{\omega \in \Omega} V(P, \omega) \cos\theta d\omega \tag{3.5}$$

In the same course and a few hours later, Bredow[17] presented the use of the obscurance effect for the more hidden zones of the *Pishkin building*, a fictional skyscrapper in the movie *Stuart Little 2*, and computed it by putting a large area light source over the building and another one less intense under it, saving the results in textures to reuse it in different frames.

The next year in another course at SIGGRAPH Christensen [21] did an in-deep presentation following previous year Landis talk, centered on how to apply the algorithms on Renderman. Some very good ideas are introduced for the computation of ambient occlusion, as taking advantage of low frequency changes to do a sparse computation and interpolate the results. Some adaptive sampling is also used. The maximum distance parameter as used in the original paper [96] and the introduction of the *cone angle* as "the fraction of hemisphere above a point that should be taken into account" are also the add-ons of this course presentation. *Color bleeding* is also treated in this talk, but presented apart from ambient occlusion as single bounce global illumination. This is different than the method presented in section 4.1, where obscurances and color bleeding are presented together as a whole. In this talk, Christensen also reused results between frames in an animation, by using an *irradiance cache* (or *occlusion cache*) file.

The poster presented by Méndez et Al. at EUROGRAPHICS 2003 [62] applied the original obscurances idea to the ray-tracing context, thus including all aspects that ambient occlusion lacks: color bleeding, maximum distance and the obscurance function. This is developed as part of the core of this thesis in section 6.1 of chapter 6.

Differences between ambient occlusion and obscurances

Ambient occlusion as presented in the three presentations at SIGGRAPH represents a simplification of the obscurances as understood in previous section in two ways. First, the distance of the ray intersection to the original point is not taken into account, and the function just results in 0 (intersects) or 1 (does not intersect). In a more conceptual way we could think of ambient occlusion as a simple percentage of openness of a point, and in obscurances the secondary interreflections are taken into account and some lighting intensity is added, this is, diffuse indirect lighting is actually taken into account (see figure 3.3).

Second, the ambient intensity and average reflectivity parameters are normally not used, and some ambient parameter is adjusted in an empirical way. It is easy to understand that physical accuracy is not really important in production rendering and even less in videogames, as we only need to find realistic and visually pleasant images. Only in other kind of applications as for example for architecture or interior design, accurate lighting is important. This is why ambient occlusion has become so popular in CGI and it is becoming widely used in videogames too.

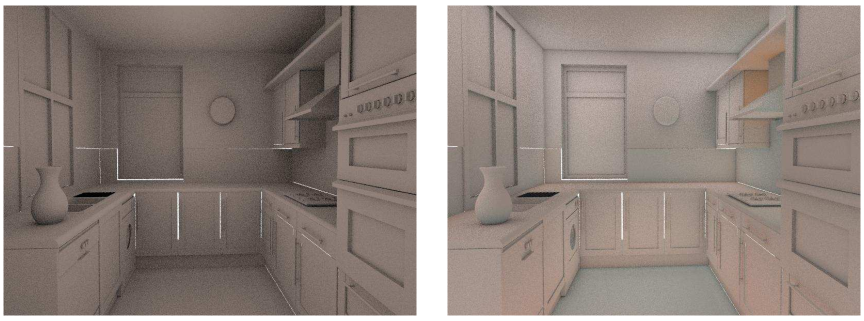

(a) Ambient occlusion. (b) Obscurances.

Figure 3.3: A comparison between ambient occlusion and obscurances with color bleeding.

3.3.4 Comparisons

Obscurances and ambient occlusion vs. constant ambient term

- Obscurance and ambient occlusion techniques are capable to simulate the ambient light distribution in an environment in a non-constant way and make possible to render realistic images.
- They increase realism by adding contact shadows and self occlusions for objects.
- Constant ambient term techniques are faster.

Obscurances and ambient occlusion vs. global illumination

- Obscurance and ambient occlusion techniques are faster than global illumination ones.
- Obscurance and ambient occlusion do not compute lighting in a physically accurated way.
- The indirect illumination is computed in a visually realistic way without the need for computationally expensive methods as radiosity or ray-tracing-based methods for global illumination.
- Pre-visualization of scenes without light sources is possible.
- As they are computed for a limited environment of a point, they allow the recomputation of small parts of the scene for moving objects.
- Their computation is pure geometrical, so changes in direct lighting or average ambient intensity do not affect the obscurances or AO computation.
- The direct and the indirect illumination are decoupled.

Obscurances vs. ambient occlusion

- The use of a continuous function of the distance in obscurances to account for indirect lighting gives better results than ambient occlusion and allows the addition of color bleeding.
- The average ambient intensity and average reflectivity are used in obscurances to search for a more accurate physical lighting.
- Both work in open environments.

3.4 Computing methods

3.4.1 Ray-casting

To compute the obscurances or the ambient occlusion of a point we need to evaluate the occlusions in the environments of the point. One way to do it is by sending rays from the point and make them intersect with the scene[1].

Occlusion of a point

Obscurances integral can be evaluated using Monte Carlo evaluation (see section 2.1). Using $\frac{1}{\pi}\cos\theta$ as probability density function (*pdf*), this is, casting rays from P with uniform direction weighted by cosinus, the obscurances integral (3.1) can be evaluated using $nrays$ samples as:

$$W(P) = \frac{1}{nrays} \times \sum_{j=1}^{nrays} \rho(d_j) \tag{3.6}$$

The number of rays sent is given by the user or by some method (if adaptive sampling is used, for example, it can be different for every point). Once a ray is sent, an intersection test with the scene in a distance less or equal than d_{max} is done. If the intersection test is positive, the ρ function is applied to the distance from the original point to the intersection point. If no intersection occurs, ρ returns value 1. The resulting value of the obscurances will be the average of the values obtained by all rays cast.

This technique is used by those approaches that need to evaluate the obscurances for a specific point. These points can be either the hits of a ray-tracing algorithm as in [63, 56, 53] or the vertices of an object. For the vertices some GPU approach as in [18] is more commonly used. Alternatively, the obscurances can also be computed for the center of a triangle or face [48].

Obscurances of a patch

In some cases [96, 40, 60] the obscurances of a piece of surface (a patch) are needed instead of a point. In this case we are evaluating a four dimensional integral instead of a two dimensional one. The idea remains the same that when computed for a point, except for the origins of the rays cast that are randomly chosen among the points of the piece of surface that represents the patch.

[1]A good algorithm for ray-scene intersection is needed for our purposes, specially if it is faster for near intersections. We can find lots of literature on the subject as part of the explanation on ray-tracing algorithms. See section 2.5.

3.4.2 Keeping the values

The ray-traced approaches normally keep the values in image space or use them directly to compute the final color of the pixel, stored in the final image. In the cases that the value has to be reused, it can be used in several images [58] or kept in a special texture called obscurance map [96, 21] or an atlas (one texture for the whole scene or for a specific object) as in [61]. It can also be kept in the vertex or in the triangle [29].

3.4.3 GPU

In the later years the power and computation capability of the specialized hardware for graphics have grown incredibly faster and faster. First, graphics cards specialized in matrix operations for projections appeared, and later, programmable graphics hardware for massive parallel computations and specific memory for textures have arisen. Almost every old and new algorithm for graphics (and even other algorithms for more general purpose) is revised and tuned to be accelerated and fit in the newest hardware. Obscurances are no exception.

Projecting shadows from lights

A first attempt to compute ambient occlusion aided by graphics hardware was made by Pharr in the first GPU Gems book [68]. The method is purely based on Landis work [53], saving the amount of occlusion and average unoccluded vector for each triangle. First a software ray-traced method is shown. Later a novel hardware accelerated approach using shadow maps is explained. This hardware method is different than the software ray-traced one in the sense that only an up vector for every point is used instead of the surface normal. The idea is that the object is surrounded by a kind of sky light hemisphere that consists in a set of directional lights pointing at the object (see fig. 3.4). To calculate accessibility, the shadow contributions of each light are averaged in a floating point accumulation buffer. The proposed future work at the end of the chapter was later addressed by Kontkanen et al. [47], that we will see in section 3.5.3.

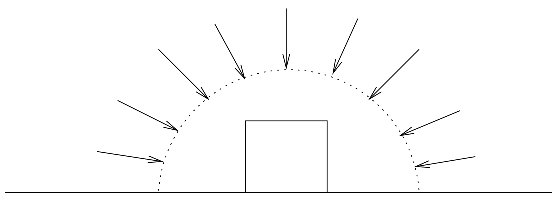


Figure 3.4: Schema of skylight ambient occlusion: light comes from all directions above the object.

Use of buffers and occlusion queries

In [74] the ambient occlusion (or, as they name "first-order approximation of the rendering equation") is computed for every vertex. The scene is rendered into the depth buffer from a directional light source point of view. The vertices are rendered again as points and an occlusion query against the depth buffer is performed. This is done for every element of a set of directional light sources surrounding the scene. A visibility matrix is used to store the data.

In [29], Franklin finds the “accessibility values” for every vertex or for every face. The idea is to use *occlusion queries* by rendering the object from the point of view of the face, looking up and directionally (5 renders per face) and counting the pixels that present occlusion. The technique presented here is useful for objects (self-occlusion) and floor.

Depth peeling

This technique is presented in [61]. The basic idea of the depth peeling technique is to extract visibility layers from the scene in order to do some computation between them. We can see the pixel image resulting from depth peeling as being equivalent to tracing a bundle of parallel rays through the scene where each pixel corresponds to a ray in the bundle. Each of these rays may intersect several surfaces in the scene, and through depth peeling we can discover all of the intersections in the form of image layers.

The computation of the obscurances is divided into two phases. In the first phase, layers are obtained using depth peeling. In the second phase, the obscurances between each pair of layers are computed and the result is added and averaged in the correspondent obscurance map position. Both phases use GPU programming. This is developed as part of the core of this thesis in section 5.2 of chapter 5.

Ambient occlusion fields

Kontkanen et al. presented in [48] a technique to compute ambient occlusion where a field that encodes an approximation of the occlusion caused by an object is precomputed in the surrounding space of it. The components (a, b and c) of an inverse quadratic function $1/(ax^2 + bx + c)$, the center of the object and the radius from this center to the convex hull is precomputed and stored in a cube map for every object so that an spherical cap approximation of the ambient occlusion can be computed in real time in graphics hardware (see fig. 3.5). The technique is valid for inter-object ambient occlusion, but not for self-shadowing or deformable objects.

The same authors published these ideas in chapter 2.4 of the ShaderX4 book [49] but more focused on the shader programs for graphic cards.

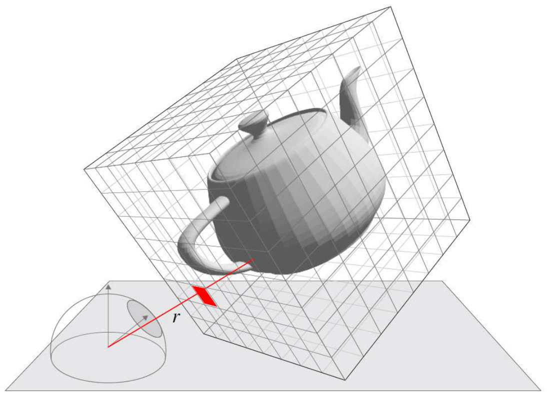


Figure 3.5: A cube map is used for every object to store the necessary parameters to compute ambient occlusion in *ambient occlusion fields* [48].

3.5 Advanced features

3.5.1 Color bleeding

Until now the obscurances have been explained as a factor that multiplies the whole light spectrum, lowering its intensity if there are objects around the point. But in actual radiosity there is an effect, called *color bleeding*, that consists in the perception that the objects around another object that has intense coloration get dyed with this color. Adding this effect to obscurances is practically for free. This contribution to obscurances is explained in [60] and it is part of the core of this thesis and is developed in section 4.1 of chapter 4.

3.5.2 Moving objects

Mendez et al. [60] took advantage of the fact that the obscurances are computed locally within a limited distance to recompute the obscurances of a moving object and the objects around it. They keep in memory the list of patches that need to be updated from frame to frame. There are patches of three kinds: One, the patches of the moving object itself. Second, the patches that are within the maximum distance around the moving object considering its position in the actual frame. And third, the patches around, but for the position in the next frame. The obscurances of the patches of the dynamic list, have to be updated for the moving objects for each frame position. As only a small portion of the scene is updated, real-time rates can be achieved. This is part of the core of this thesis and is developed in section 5.1 of chapter 5.

In [54], Malmer et al. used a 3d texture containing the ambient occlusion values, interpreted as the solid angle that the object, as occluder, projects to every 3d-texel. The 3d texture is precomputed around an object, and in real time ambient occlusion is computed between objects. An improved version saves also the direction (cone) and is useful for self-shadowing.

3.5.3 Deformable objects

The technique presented in [18] by Bunnell works for dynamic and deformable objects that need to compute self ambient occlusion on the fly in a videogame context. The trick consists in considering the vertices of the object as small portions of surface, each one defined by the position of its center, its normal and area. These small surfaces can be projected to the other ones to account for the occlusions. The technique works with environment lighting and bent normals. The computation is made element-to-element, so it is n-square, and the performance has to be improved by constructing a hierarchy grouping surfaces and using a representative group for large distances. A few bounces of indirect lighting can also be computed with the same technique.

Kontkanen and Aila presented in [47] an fast approximated solution for animated characters. The idea is to precompute the ambient occlusion of some poses of the character and find a correspondence to the parameters for these poses, mainly the angles of the vertices. They compute the correspondence matrix and the solution for any other pose is found by interpolating linearly via the linear equation matrix.

Kirk and Orikan [45] present a method very similar to the previous one but compressing the data used for computation, thus saving memory and reducing computation time.

3.5.4 Trees and plants

Obscurances have been applied to trees more or less directly in [30] and [59]. In [30], the depth peeling approach (see section 5.2) has been used to precompute obscurances and use them in real time for tree models simplified with leave clusters. In [59], a ray-traced approach taking into account the leaves translucency has been used.

Hegeman et al. [39] present a quite interesting approximation of ambient occlusion for trees and grass. Trees are approximated to spheres or ellipsoids and the ambient occlusion formula is computed considering this shape as if it were full of blocking elements (see fig. 3.6). The result is that the more inner an element (leave) is, the darker it appears. It considers skydome ambient occlusion (normal always up). Inter-object occlusion (between trees, or between trees and floor) is done using the solid angle approximation. A different formula is used with grass.

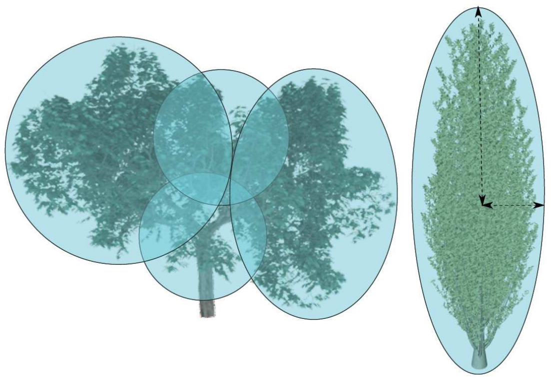


Figure 3.6: Trees are approximated for ellipsoids and an adapted formula to compute ambient occlusion for ellipsoids filled with blocking elements is used in [39].

3.5.5 Volumetric *vicinity shading*

Stewart presented an algorithm to compute obscurances for volumetric data in [84] and dubbed it *vicinity shading*. The objective was to compute an occlusion value for each voxel of a 3D volume of densities aiming to use it for a later rendering of any of its isosurfaces. Instead of computing first the surface (using any classic algorithm as marching cubes) and then applying to the surface any technique to compute occlusions, the volumetric data is used directly to compute occlusions, changing the definition of occlusion to *finding a voxel with equal or more density in a given direction*. The normalized gradients are used as the normals of the surfaces. To reduce cost, instead of computing the occlusions for every voxel independently, global directions are treated.

3.5.6 Light animation and frame optimization

In the context of global illumination, if we have to compute a series of frames of an animation in which the camera and the objects in the scene are still and only the light sources move, all the lighting computation has to start over from frame to frame.

In the context of obscurances, Mendez et al. [56] took advantage of one of its properties, that is the decoupling between direct and indirect illumination. In the case of light animation, they only needed to compute indirect illumination once and reuse it in all frames and only direct illumination is recomputed for every frame. Besides the animation of the light, other effects as reducing or increasing the intensity of the light or changing its color could be incorporated, as only the recomputation of the ambient intensity and the average reflectivity were needed once for every frame. This is part of the core of this thesis and is developed in section 7.3 of chapter 7.

In [37], Havran et al. presented a method to reuse diffuse lighting information by reprojecting the hit in various neighbor frames in a camera walktrough. This method can apply perfectly to obscurances. Thus, both strategies to gain efficiency for camera and light animation with obscurances could be combined in a single algorithm and the results were presented in [58] by Mendez et al., thus obtaining a big set of frames at once that can be used to create different animations to see, for example, the animation of light from different points of view, do the walkthrough with different direct light conditions, or see both animations in a single movie. This is part of the core of this thesis and is developed in section 7.4 of chapter 7.

3.5.7 Non-diffuse features

When obscurances were integrated to ray-tracing algorithms in [62] specular and glossy effects could be easily incorporated by adding to the indirect lighting computed with obscurances the illumination of the reflected or refracted rays as in classical ray-tracing algorithms. Despite the introduction of these effects, the computation of the obscurances itself only took account of the diffuse reflective illumination. In [59], Mendez et al. studied how the obscurances should behave when they are computed in an environment with non-diffuse materials, like specular or refractive ones. They proposed to extend the original algorithm that computes the obscurances with color bleeding in a way that it could cope with other kind of materials and the interactions between them, such as perfect specular surfaces, refractive and translucent objects. This is part of the core of this thesis and is developed in section 6.2 of chapter 6.

3.5.8 Monte Carlo optimization

The obscurance method requires to take samples over the hemisphere to compute the Monte Carlo integral of the obscurance function. The efficiency of the obscurance computation is thus related to the sampling technique used.

Mendez et al. [57] compared four different sampling techniques: uniform random, quasi-Monte Carlo with random offset, systematic [50, 22, 85, 78] and stratified sampling. Both systematic and stratified showed an improvement of 50% in efficiency with respect to uniform sampling, although systematic sampling presented a highly irregular distribution of the error. Halton sampling with random offset resulted in still a higher improvement, so they found this last one the best strategy to compute obscurances. This is part of the core of this thesis and is developed in section 4.5 of chapter 4.

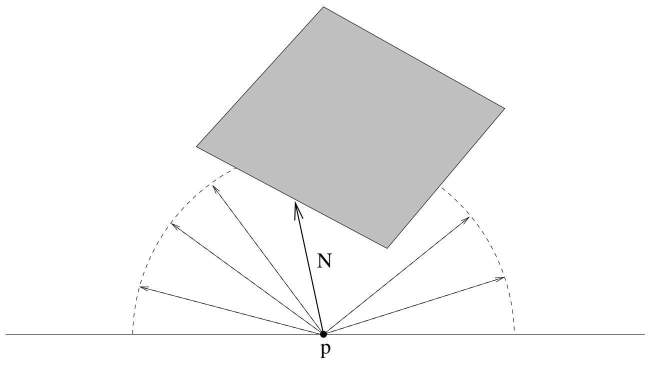


Figure 3.7: The *bent normal* N is the average direction of the unoccluded portion of the hemisphere over point p. It is used to find the color of the unoccluded zone of the environment. But in some cases, like the one showing this image, the average direction is occluded and another solution has to be used, as finding directly the contributions of the unoccluded directions.

3.5.9 Bent normal and environment mapping

The *bent normal* is used from the first appearance of ambient occlusion by Landis [53] and its immediate followers [21, 68, 18] and even in some of the late improvements of the technique [48, 54].

It consists in saving the average direction of the unoccluded portion of the hemisphere over every point at which the ambient occlusion is computed. This value is then used to find, via environment mapping, the color of the light that comes from this unoccluded hole. One possible problem here appears when the computed average direction coincides with a direction occluded by some object as it is the case when unoccluded area of the hemisphere forms a rim next to the *horizon* of the hemisphere. In this case the blocking object is just above the surface and the *bent normal* points directly to the occluder as in fig. 3.7. Christensen [21] addresses this problem by averaging several unoccluded values of the environment map, instead of using only the *bent normal* one.

When using the *bent normal*, it is assumed that no light comes from the occluders, not even the indirect lighting. This is why this concept makes no sense within the obscurances idea, where the indirect lighting is actually taken into account wherever it comes from.

3.6 Comparison tables

In this section we overview most of the techniques previously referenced in this survey using tables for comparison. In each table we mark which techniques are related to which features. Each table refers to one or various aspects of the techniques.

3.6.1 Where is the value computed and saved

The next table compares all studied articles with respect to where the obscurances or ambient occlusion values are computed and saved.

- **Surface**: The value is computed (and possibly stored) for a piece of surface called *patch* or for each triangle of an object.

- **Vertex**: The value is computed (and possibly stored) for every vertex of an object.
- **Hit**: The value is computed for every hit point of a pixel ray in a ray-tracing algorithm. The value is usually stored per pixel in image-space.
- **Texture**: The value is saved in some kind of texture, obscurance map, occlusion map, cube map or 3D texture.
- **Vol**: The value is saved in a volumetric data set.

Technique	Surface	Vertex	Hit	Texture	Vol
Zhukov98-ALIM* [96]	X			X	
Bredow02-ROF [17]			X	X	
Landis02-PRGI [53]			X		
Iones03-FRLVG* [40]	X			X	
Mendez03-RTOCB* [60]	X			X	
Christensen03-GIAT [21]			X		
Stewart03-VSEPVD [84]					X
Mendez03-ORTEV [63]			X		
Mendez03-ORT* [62]			X		
Pharr04-AO [68]	X				
Mendez04-CHSTO* [57]			X		
Mendez04-CLAOG* [56]			X		
Sattler04-HAAOC [74]		X		X	
Bunnell05-DAOIL [18]		X		X	
Kontkanen05-AOF [48]	X			X (cube)	
Kontkanen06-AOF [49]				X (cube)	
Malmer06-FPAOPS [54]				X (grid)	
Mendez06-RTOCB [61]	X			X	
Mendez06-OGE* [59]			X		
Mendez06-ERLCA* [58]			X		
Kontkanen06-AOAC [47]		X		X	
Franklin06-HBAO [29]	X	X			
Hegeman06-AOT [39]	X		X		
Kirk07-RTAOD [45]		X		X	

3.6.2 Object and scene consideration

The next table compares all studied articles with respect to the characteristics of the objects or the scenes that are considered.

- **Self-shadow**: The technique takes self-shadowing (intra-object occlusions) into account.
- **Inter-objects**: The technique computes occlusions between different objects.
- **Closed**: The technique can compute obscurances or ambient occlusion in a closed scene.
- **Sky**: The technique computes skydome ambient occlusion, considering ambient lighting coming from the sky.

Technique	Self-shadow	Inter-objects	Closed	Sky
Zhukov98-ALIM* [96]	X	X	X	
Bredow02-ROF [17]	X			X
Landis02-PRGI [53]	X			
Iones03-FRLVG* [40]	X	X	X	
Mendez03-RTOCB* [60]	X	X	X	
Christensen03-GIAT [21]	X	X	X	
Stewart03-VSEPVD [84]	X			
Mendez03-ORTEV [63]	X	X	X	
Mendez03-ORT* [62]	X	X	X	
Pharr04-AO [68]	X			X
Mendez04-CHSTO* [57]	X	X	X	
Mendez04-CLAOG* [56]	X	X	X	
Sattler04-HAAOC [74]	X			
Bunnell05-DAOIL [18]	X			
Kontkanen05-AOF [48]		X		
Kontkanen06-AOF [49]		X		
Malmer06-FPAOPS [54]	X (some)	X		
Mendez06-RTOCB [61]	X	X	X	
Mendez06-OGE* [59]	X	X	X	
Mendez06-ERLCA* [58]	X	X	X	
Kontkanen06-AOAC [47]	X			
Franklin06-HBAO [29]	X	Floor		Floor
Hegeman06-AOT [39]	X	X		X
Kirk07-RTAOD [45]	X			

3.6.3 GPU and dynamics

The next table compares all studied articles pointing out if they use GPU and/or can consider dynamic objects.

- **Classic GPU**: The technique uses some non-programmable GPU algorithm (like projections).

- **Prog. GPU**: The technique uses some modern GPU with programmable vertex or fragment shaders.

- **mov. obj.**: Obscurances or ambient occlusion can be recomputed in real-time for solid moving objects.

- **def. obj**: Obscurances or ambient occlusion can be recomputed in real-time for deformable objects.

Technique	classic GPU	prog GPU	mov. obj.	def. obj.
Zhukov98-ALIM* [96]				
Bredow02-ROF [17]				
Landis02-PRGI [53]				
Iones03-FRLVG* [40]				
Mendez03-RTOCB* [60]			X	
Christensen03-GIAT [21]				
Stewart03-VSEPVD [84]				
Mendez03-ORTEV [63]				
Mendez03-ORT* [62]				
Pharr04-AO [68]	X			
Mendez04-CHSTO* [57]				
Mendez04-CLAOG* [56]				
Sattler04-HAAOC [74]	X		X	X
Bunnell05-DAOIL [18]		X	X	X
Kontkanen05-AOF [48]		X	X	
Kontkanen06-AOF [49]		X	X	
Malmer06-FPAOPS [54]		X	X	
Mendez06-RTOCB [61]	X	X		
Mendez06-OGE* [59]				
Mendez06-ERLCA* [58]				
Kontkanen06-AOAC [47]			X	X
Franklin06-HBAO [29]	X	X		
Hegeman06-AOT [39]		X		
Kirk07-RTAOD [45]			X	X

3.6.4 Optimizations

The next table compares all studied articles with respect to the optimizations they use to compute occlusions faster.

- **Precomp**: Some precomputation of the values is done and stored for later reuse.

- **Frame opt**: Some reuse of information is done between frames of an animation.

- **Sparse**: Values are computed sparsely and interpolated.

- **Adaptive**: Some kind of adaptive sampling is made.

Technique	Precomp	Frame opt	Sparse	Adaptive
Zhukov98-ALIM* [96]	X			
Bredow02-ROF [17]	X			
Landis02-PRGI [53]				
Iones03-FRLVG* [40]	X			
Mendez03-RTOCB* [60]	X			
Christensen03-GIAT [21]		X	X	X
Stewart03-VSEPVD [84]	X			
Mendez03-ORTEV [63]				
Mendez03-ORT* [62]				
Pharr04-AO [68]				
Mendez04-CHSTO* [57]				
Mendez04-CLAOG* [56]		X		
Sattler04-HAAOC [74]				
Bunnell05-DAOIL [18]				
Kontkanen05-AOF [48]	X			
Kontkanen06-AOF [49]	X			
Malmer06-FPAOPS [54]	X			
Mendez06-RTOCB [61]				
Mendez06-OGE* [59]				
Mendez06-ERLCA* [58]		X		
Kontkanen06-AOAC [47]	X			
Franklin06-HBAO [29]	X			
Hegeman06-AOT [39]	X			
Kirk07-RTAOD [45]	X			

3.6.5 Add-ons

The next table compares all studied articles with respect to the use of additional techniques that help improve results.

- **Env. map.**: An environment mapping is used to acquire color.

- **B. normal**: An averaged non-occluded direction (bent normal) is computed to account for the environmental color.

- **Cone**: A maximum angle is used to account for occlusions, that joined to the normal (or the bent normal) forms a cone.

- d_{max}: A maximum distance is used to account for occlusions.

- **function**: A monotone increasing function of the distance is used to compute the obscurance value.

Technique	Env. map.	B. normal	cone	d_{max}	Func
Zhukov98-ALIM* [96]				X	X
Bredow02-ROF [17]					
Landis02-PRGI [53]	X	X			
Iones03-FRLVG* [40]				X	X
Mendez03-RTOCB* [60]				X	X
Christensen03-GIAT [21]		X	X	X	
Stewart03-VSEPVD [84]				X	X
Mendez03-ORTEV [63]				X	X
Mendez03-ORT* [62]				X	X
Pharr04-AO [68]	X	X			
Mendez04-CHSTO* [57]				X	X
Mendez04-CLAOG* [56]				X	X
Sattler04-HAAOC [74]	X				
Bunnell05-DAOIL [18]	X	X			
Kontkanen05-AOF [48]		X		X	
Kontkanen06-AOF [49]				X	
Malmer06-FPAOPS [54]	X	X	X	X	
Mendez06-RTOCB [61]				X	X
Mendez06-OGE* [59]				X	X
Mendez06-ERLCA* [58]				X	X
Kontkanen06-AOAC [47]					
Franklin06-HBAO [29]					
Hegeman06-AOT [39]					X
Kirk07-RTAOD [45]					

3.7 Impact

Nowadays most commercial renderers include some kind of ambient occlusion, normally in the form of an extra shader. We will study briefly in section 3.7.1 what software packages include it and how it can be used. Next in sections 3.7.2 and 3.7.3 we will enumerate some of the most important movies and videogames that have included ambient occlusion among its features.

3.7.1 Software packages

Mental Ray

Mental Ray is a high quality production renderer usually included in third party software packages as ***Maya*** or ***3DStudio Max***.

Mental Ray includes ambient occlusion in the form of a shader from version 3.1+, and names it *dirtmap*. It uses raytracing to determine the amount of occluding geometry in the neighborhood of a rendered point, and outputs the result as a blend of two colors (usually black and white, but the *unoccluded color* can be a lookup to a environment map). You can modify the number of samples, the sampled radii, and a number of other settings to create a plethora of effects, like simple dirt, single-pass ambient occlusion, or even faked sub surface scattering. See, for examples and tutorials, [8, 9, 3].

Renderman

As we have told before, Landis [53] and Christensen [21] popularized ambient occlusion in their respective talks at SIGGRAPH in 2002 and 2003, and their talks were about ***RenderMan***, the popular renderer made and used by *Pixar* from their beginnings. Using predefined functions as `gather()` and `occlusion()` and setting the parameters, the ambient occlusion can be easily computed in RenderMan.

Maya and 3D Studio Max

Both ***Maya*** and ***3D Studio Max*** are very complete 3D software packages aimed to modeling, lighting and animating, but for renderer purposes both include an external renderer such as the previously discussed Mental Ray or RenderMan, so ambient occlusion shader is integrated in the renderers.

Blender

Blender is a free 3D animation program and has become popular in the open source community. It incorporates a kind of skydome ambient occlusion.

VRay

Ambient occlusion was available as a plugin for VRay from 2004 [10] and is included in the renderer from version 1.5 [6] and named *VRayDirt*.

Cinema4D

Cinema4D does not incorporate ambient occlusion directly, but it does in a separate module called **Advanced Renderer**. Alternatively a plugin can be obtained [1].

Others

Other more specific shaders as ***QuteMol*** [4] for rendering molecular structures and ***ShadeVis*** [5] for cultural heritage rendering not only use ambient occlusion but present it as its main attraction.

3.7.2 Movies and TV

According to the first tutorials that named ambient occlusion, the former movies where the technique was used were *Pearl Harbor*, *Jurassic Park III* and *Stuart Little 2*.

From then, almost five years ago, ambient occlusion has changed from being the exception to become the standard. Nowadays few are the movies that use CGI and do not take advantage of the benefits (speed and realism) of ambient occlusion. This year, Academy Award winner for special effects, *Pirates of the Caribbean: Dead Man's Chest*, used the technique for its CGI characters [72]. Other recent examples are next Disney presentation, *Meet the Robinsons*, and the later Pixar movies *Cars* [20] (see fig. 3.8) and *Ratatouille*.

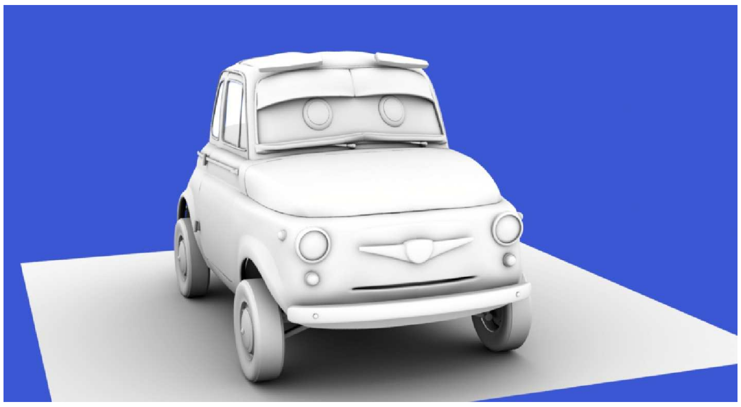


Figure 3.8: Ambient occlusion for *Luigi the Fiat 500*, a character from the movie cars ©2006 Disney/Pixar.

3.7.3 Videogames

We have seen that ambient occlusion has become very popular in movies. On the contrary, despite the obscurances technique was originally designed to work with games, there are not so many games actually using it. Recent games that use ambient occlusion (*TimeShift*, *Ghost Recon Advanced Warfighter*) compute it in the game production process for characters and scenarios and store the resulting static textures to use them in real-time while playing.

3.8 Summary

We have surveyed in this chapter the origins and evolution of two intimately related concepts: the obscurances and the ambient occlusion. Both techniques rely on the idea that, visually, the more hidden a point of a surface is, the darker it is seen, and in practice they are computationally inexpensive with respect to global illumination techniques and they achieve similar visual realism (though not physically accurate).

After the basic definitions for the obscurances we have explored how ambient occlusion appeared in the context of production rendering for movies and we have compared both techniques looking for similarities and differences. Next we have summarized the improvements and several techniques to accelerate the computation that different authors have worked out during the later years. Besides, the good features of obscurances have inspired new techniques as the *frame array* [58].

Lately, ambient occlusion has become popular in the movie industry, since the technique is included in most commercial renderers. The concept has become so popular that even has an entry in the Wikipedia [7]. Concerning videogames, it is included in some of them in the *baked* form, this is, precomputed and saved in still textures that are used in real time by characters and scenarios.

Nowadays, the most challenging issue for obscurances and ambient occlusion is to find a really fast technique to modify their values for dynamic objects in real time for videogames. Some attempts have been made, but no released game has still included this feature. Next generation consoles may change this.

Chapter 4

Improving obscurances

In the previous chapter the obscurances and the ambient occlusion techniques have been introduced and compared. In this chapter we look more deeply into the obscurances concept by adding some improvements, studying different options for actual implementations of the technique and introducing some problems that might occur if we use obscurances.

In section 4.1 the effect of color transfer between surfaces is added to the original obscurances equations. In section 4.2, some different functions ρ for the obscurances computation are analyzed and in section and 4.3 we study the effect of different values for the maximum distance parameter d_{max}. Next in section 4.4 a problem of the obscurances and its possible solution is introduced. Next in section 4.5 we study different methods to sample the hemisphere for the obscurances and study their relative efficiencies. Finally, in section 4.6, a different algorithm to account for average intensity and average reflectivity of the scene is introduced.

The techniques explained in this chapter are introduced in following papers:

- *Real-Time Obscurances with Color Bleeding* [60] (sections 4.1 and 4.4),
- *Obscurances for Ray-Tracing (Extended Version)* [63] (sections 4.2 and 4.3),
- *Comparing Hemisphere Sampling Techniques for Obscurance Computation* [57] (section 4.5) and
- *Combining Light Animation with Obscurances for Glossy Environments* [56] (section 4.6).

4.1 Adding color bleeding

Originally the obscurances were introduced as a factor that multiplies the whole light spectrum, lowering its intensity if there are objects around the point. But in actual radiosity there is an effect, called *color bleeding*, that consists in the perception that the objects around another object that has intense coloration get dyed with this color. Adding this effect to obscurances is practically for free.

4.1.1 Modifying the formulas

We modify equation (3.1) by adding a reflectivity term $R(Q)$ for each differential of solid angle that corresponds to the color (diffuse reflectivity) of points Q of objects around P:

$$W(P) = \frac{1}{\pi} \int_{\omega \in \Omega} R(Q)\rho(d(P,\omega)) \cos\theta d\omega \tag{4.1}$$

When no surface is hit at a distance less than d_{max} in direction ω the obscurance takes the value of R_{ave} as computed in (3.4).

Thus, the obscurances factor is reduced by a reflectivity factor. This has to be compensated in the ambient intensity factor (I_A) of the indirect lighting. In other words, we have to take in to account again the direct light intensity.

Ambient intensity will be, in this way:

$$I_A = \frac{1}{1 - R_{ave}} \times \frac{1}{A_{total}} \sum_{\imath=1}^{n} A_\imath \times E_\imath \tag{4.2}$$

The Monte Carlo version of the integral is:

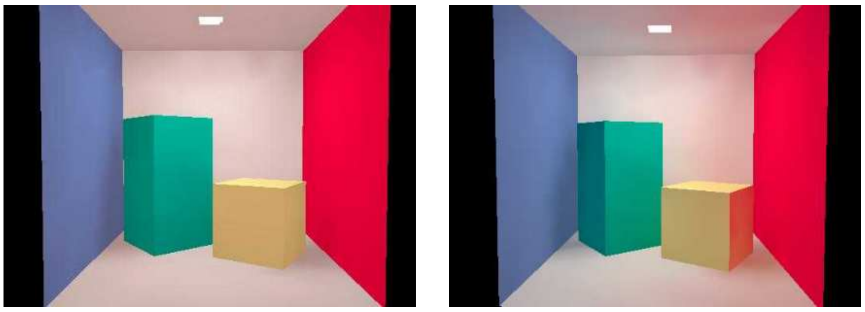
$$W(P) = \frac{1}{nrays} \times \sum_{\jmath=1}^{nrays} \rho(d_\jmath) R_{int} \tag{4.3}$$

R_{int} is the reflectivity of the point of the scene that the random ray intersects, and it will be R_{ave} when there is no intersection.

4.1.2 Results

The cost for obscurances computation is less than one third of the cost for secondary illumination computation in the two radiosity algorithms used, shooting Random Walk and Hierarchical Monte Carlo. As stated before, the computation takes advantage of the fact that only a d_{max}-neighboorhood of the patch has to be examined for intersection. The total number of available rays is distributed in the obscurance computation according to the area distribution.

In fig. 4.1a we show the Cornell Box scene computed with the obscurances without color bleeding, while in fig. 4.1b we have used the improved algorithm.



(a) Obscurances without color bleeding.

(b) Obscurances with color bleeding.

Figure 4.1: These images show the Cornell Box scene with obscurances. The left one without color bleeding and the right one with color bleeding.

Color bleeding is clearly visible, adding a lot of realism to the image.

4.2 The different ρ functions

There are different families of functions that give us an approximation to the shape of the ρ function seen previously in figure 3.2. We have selected some of them to study which one gives us the best results

- **Exponential**:

$$\rho(d) = 1 - e^{\frac{d}{d_{max}}} \tag{4.4}$$

- **Square root**:

$$\rho(d) = \sqrt{\frac{d}{d_{max}}} \tag{4.5}$$

- **Constant 0**:

$$\rho(d) = 0 \tag{4.6}$$

The shape and effect of each function can be appreciated in figure 4.2. The exponential function (row (1.)) present a darker effect and less color bleeding than the square root function (row (2.)). The constant function does not have the desired shape and does not depend on the distance (d) parameter, but it can be considered as a simplified version of the technique presented here and is used under the name of *ambient occlusion* in several commercial renderers, such as *Photorrealistic Renderman* by *Pixar*[53, 21]. In row (3.) of figure 4 we can see that the use of this constant function results in a darker image and does not present color bleeding.

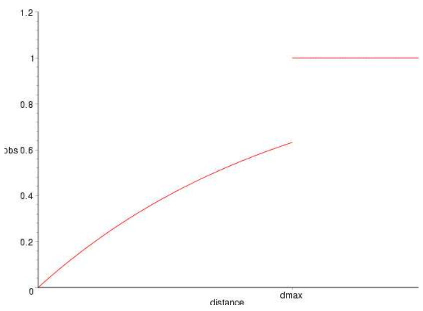
Any of the two former functions may be useful for our purposes. We choose the second one, the square root, as it is more simple and easier to compute than the exponential one.

4.3 The maximum distance (parameter d_{max})

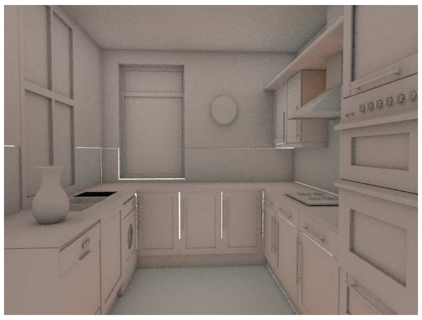
Once the $\rho()$ function is selected (see section 4.2) several parameters influence the behavior of the obscurances. One parameter is the maximum distance selected, d_{max}. If the ray-tracing routines used (to trace the rays to intersect objects and thus compute obscurances) are accelerated by a voxelized structure of the scene (see section 2.5.3 in chapter 2), shortest rays allow for a faster computation. In this way, a small distance d_{max} will allow for a faster computation, while a longer one will take into account more darkening (and color bleeding) effects. In Fig.4.3 we see four obscurance images with different d_{max}. In Fig.4.3.a $d_{max} = 0.2$m[1], and the computation time is 219 sec.[2], and in Fig.4.3.d $d_{max} = 2.0$m, and the computation time is 314 sec. The other two images show values in between, as expected. For the purpose of our comparisons we selected the $d_{max} = 1.0$m. This value is chosen in an empirical way, as we have perceived that a d_{max} value between a quarter and half the size of the bounding box of the scene give better results than bigger or smaller values.

[1]Our model of the kitchen is approximatedly between 2 and 3 meters long

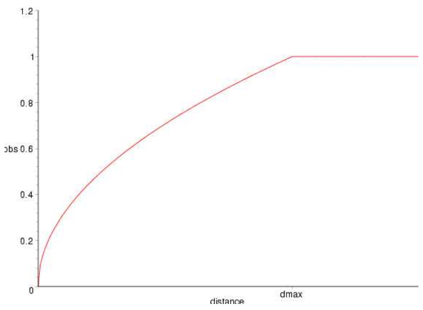
[2]Images are 800×600 pixels and computed in a Pentium 4 1.6 Ghz with 1 Gb RAM.



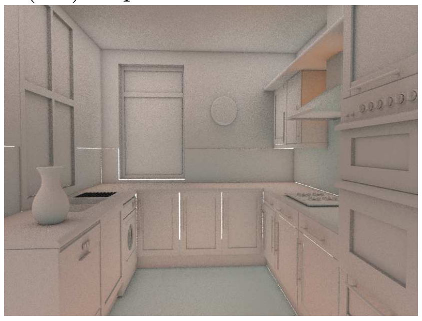
(1.a) Shape of the exp function.



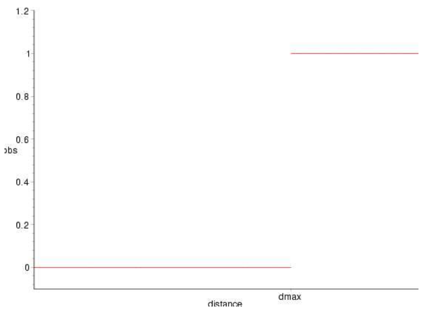
(1.b) Exp function obscurances.



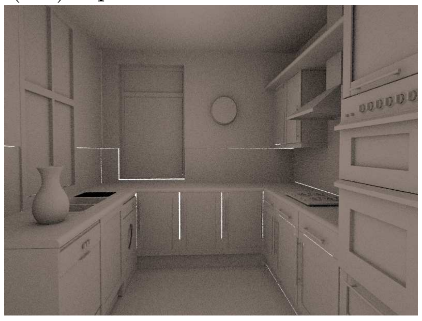
(2.a) Shape of the sqroot function.



(2.b) Sqroot function obscurances.

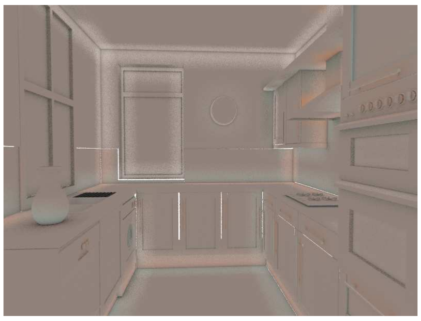


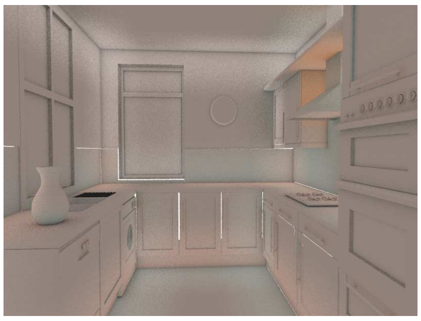
(3.a) Shape of the const function.

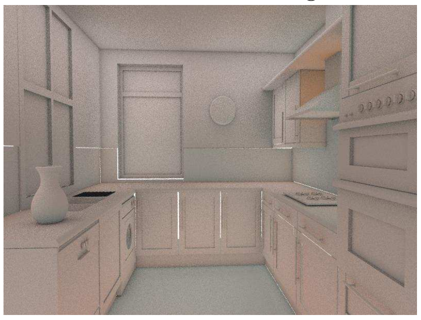


(3.b) Ambient occlusion.

Figure 4.2: The use of different functions to calculate the obscurances can be appreciated. In (1) the exponential function is used. In (2) the function used is the square root one. And in (3) a stair function equal to 0 when $d \leq d_{max}$ and to 1 when $d > d_{max}$ is used, this one is called *ambient occlusion* on several commercial renderers. Here we take $d_{max} = 1.0m$. The kitchen model is approximately 2.0m to 3.0m long. See fig. 4.3 for a comparison of different d_{max}.


(a) Maximum distance 0.2 meters. 219 secs. rendering.


(b) Maximum distance 0.5 meters. 241 secs. rendering.


(c) Maximum distance 1.0 meters. 264 secs. rendering.

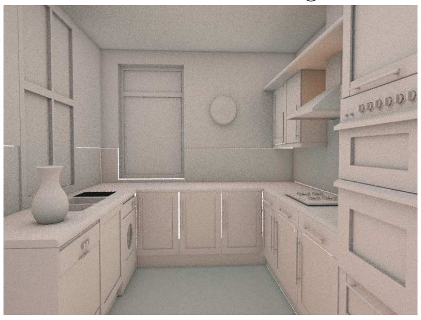

(d) Maximum distance 2.0 meters. 314 secs. rendering.

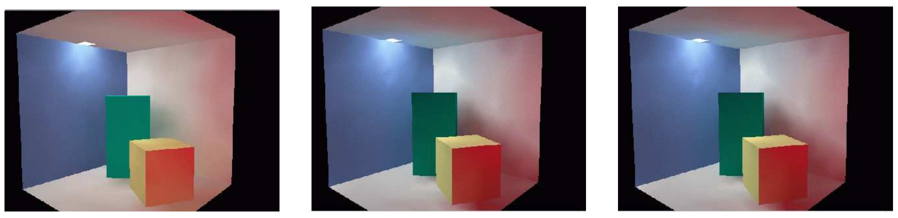
Figure 4.3: Comparison of the images of obscurances with different maximum distances. This parameter allows us to control the locality of the calculation. Due to the voxelization of the scene, when the distance decreases, the calculation is faster. Image resolution is 800 x 600 pixels.

4.4 Important secondary reflectors problem

Although the obscurances technique with color bleeding[3] presented in previous sections gives a good visual approximation to radiosity in most of the cases, there are some configurations for which it fails. These configurations happen when the light sources are very close to a surface, so that it becomes a very important secondary reflector. In other words, when the light is not distributed more or less uniformly in the scene.

An example is seen in figures 4.4a and 4.4b. In fig. 4.4a we have the obscurances solution, while in fig. 4.4b the correct Hierarchical Monte Carlo radiosity solution is shown. As in this scene there are very important secondary reflectors (the light source points directly to the blue wall), the obscurances fail to give an accurated representation of radiosity, in which the global ambient light of the scene appears more blueish.

A possible solution is to further expand the direct illumination, at the expense of an increased computational cost. This is, the most important secondary reflectors will redistribute the received direct illumination. This simple solution gives very good results for most of the light positions. In fig. 4.4c, direct illumination arriving at the wall patches near the source has been redistributed. Compare fig. 4.4c, with fig. 4.4a, old obscurances, and fig. 4.4b, radiosity solution. Observe in the new obscurance solution, fig. 4.4c, the color bleeding of the blue wall against the ceiling and white wall as in the radiosity solution, fig. 4.4b.



(a) Obscurances (b) HMC (c) Improved obs

Figure 4.4: (a) An image computed with obscurances with the problem of important secondary reflectors (b) The same image computed with Hierarchical Monte Carlo radiosity (c) Obscurances with direct illumination expansion.

4.5 Noise reduction using different sampling techniques

The obscurance method requires to take samples over the hemisphere to compute the Monte Carlo integral of the obscurance function. The efficiency of the obscurance computation is thus related to the sampling technique used.

We compare in this section four different sampling techniques: uniform random, quasi-Monte Carlo with random offset, systematic and stratified sampling.

[3]From now on, any time we name the obscurances it will be assumed that they include color bleeding.

4.5.1 Hemisphere sampling

To obtain a cosine weighted random direction over the hemisphere we sample twice a random variable uniformly distributed in the unit interval (i.e. calling the *drand()* function), obtaining ξ_1 and ξ_2. The direction (ϕ,θ) is then given by:

$$\begin{cases} \phi = 2\pi\xi_1 \\ \theta = \arcsin\sqrt{\xi_2} \end{cases} \tag{4.7}$$

Instead of using a pseudo-random number generator to obtain ξ_1 and ξ_2, we can use deterministic or quasi-Monte Carlo sequences, like Halton one [67]. Quasi-Monte Carlo sequences distribute the samples more regularly over the domain. To avoid bias, the values in the sequence can be added to the same random offset (Cranley-Patterson rotation [46]).

4.5.2 Systematic Sampling

Systematic sampling [50, 22, 85, 78] is a classical Monte Carlo technique that has been used for years in some fields, notably in Stereology [25, 26, 33]. In systematic sampling a uniform grid is translated by a random offset giving the sampling points to probe the target function. As systematic sampling is based on regular sampling, we obtain cheaper samples than those obtained with independent uniform random sampling. It can be proved that for certain kind of functions the variance in systematic sampling decreases faster than in pure Monte Carlo sampling [25]. The drawback is that systematic sampling produces systematic error when the domain is somehow regular.

To take n_1 x n_2 systematic samples on the hemisphere, we proceed as follows [33]. We consider first n_1 x n_2 partitions of the hemisphere, and take $t_1 = \frac{2\pi}{n_1}$ and $t_2 = \frac{1}{n_2}$ as the periods for longitude and latitude respectively. We sample twice a random variable uniformly distributed in the unit interval, obtaining ξ_1 and ξ_2.

The directions are then computed as follows:

$$\begin{cases} \text{for } i = 0 \text{ to } (n_1 - 1) & \phi_i = t_1(\xi_1 + i) \\ \text{for } j = 0 \text{ to } (n_2 - 1) & \theta_j = \arcsin\sqrt{t_2(\xi_2 + j)} \end{cases} \tag{4.8}$$

4.5.3 Stratified Sampling

Stratified sampling [83, 80] is based on this same idea of dividing the domain into subdomains of equal probability, but the samples are chosen in a different way. Instead of choosing a random offset, we get a random sample from each subdomain. For a certain kind of functions, the variance of stratified sampling with one sample per stratum also decreases faster than in pure Monte Carlo sampling [83]. To take n_1 x n_2 stratified samples on a hemisphere, we proceed as follows.

We sample n_1 x n_2 pairs of random variable uniformly distributed in the unit interval, obtaining pairs (ξ_i and ξ_j), $i = 1, \ldots, n_1$, $j = 1, \ldots, n_2$. The directions are then given by:

$$\begin{cases} \text{for } i = 0 \text{ to } (n_1 - 1) & \phi_i = t_1(\xi_i + i) \\ \text{for } j = 0 \text{ to } (n_2 - 1) & \theta_j = \arcsin\sqrt{t_2(\xi_j + j)} \end{cases} \tag{4.9}$$

4.5.4 Results

We have used two scenes, kitchen and box, and a series of five sample numbers, 2x2, ..., 6x6, to test uniform random, systematic, stratified and Halton sampling. This last one is done with random offset, as used in [66], because pure Halton sampling results in an unacceptable pattern. In fig. 4.5a and 4.5b we present charts of Mean Square Error (MSE) versus time, for the average of 10 executions. From these graphs we see that both systematic and stratified perform very similar and overcome uniform random. However, the systematic nature of the error in systematic sampling (see fig. 4.6) is an important drawback of this technique. We see also from fig. 4.5a & 4.5b that Halton sampling with random offset is the most efficient technique. See fig. 4.7 for a visual comparison of all four methods.

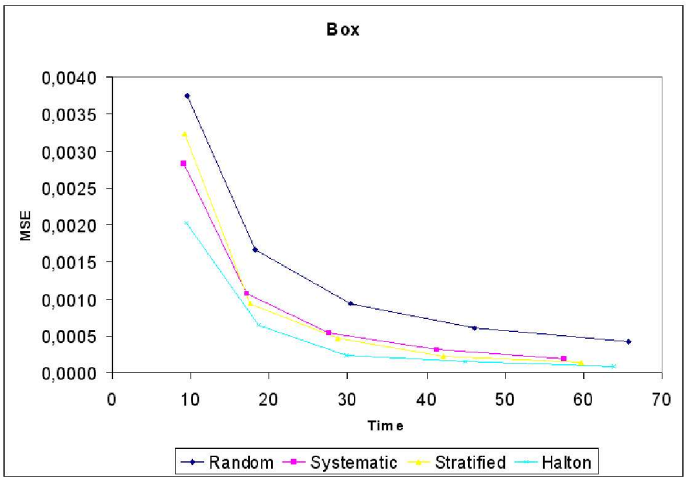
4.6 Improving ambient light computation

Equation (3.3) describes how ambient intensity is computed, as a draft approximation to what actual ambient intensity could be. It uses equation (3.4) for average reflectivity. In both equations, the total area of the scene is taken into account. It means that we make the assumption that the light energy gets distributed around the scene and illuminates with the same intensity not only every object but every part of every object, and that secondary lighting occur in the same way. When light emitters are positioned in the center of the scene and illuminate most of it, the ambient intensity given by equation (3.3) is a good estimation.

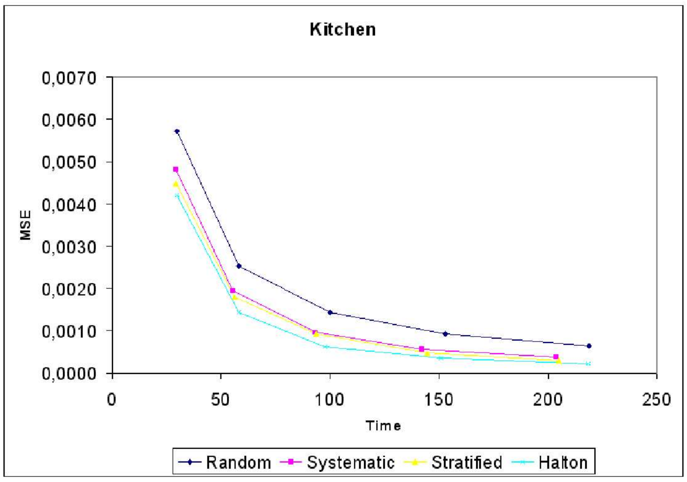
But in most scenes these assumptions do not apply. Normally, a noticeable part of the surfaces of the scene are hidden to the light, i. e., totally occluded by other objects, for example, the objects inside a cupboard, the inner part of an opaque vase and everything it could contain, etc. As we can see, normally, more than half the surfaces of the scene should not be taken into account in that *total area* term. On the other hand, if light sources illuminate with an important portion of the total power directly to colored objects, previously computed average reflectivity (3.4) from the scene would be biased with respect to the actual one.

To solve this problem, I_A and R_{ave} can be more accurately estimated by shooting a few rays from the light sources beforehand, and follow their paths while computing stochastically the parameters we need. In this way ambient intensity, average reflectivity and the area reached by the light are easily computed. When we compute a series of frames of an animation, if the number of samples is not big enough, a problem of noise in the form of flickering appears between images. A filter is used to reduce this flickering. We will compute ambient intensity for every frame, and use it to multiply the obscurances factor and diffuse color to obtain indirect lighting.

This approach is used when animating light sources (section 7.3 in chapter 7) as for every frame different values of ambient lighting and average reflectivity are needed. If light sources power changes among frames or light movement causes changes in occlusion conditions, ambient intensity will also change and result in beautiful effects in the animation. In fig. 4.9 these effects can be appreciated.

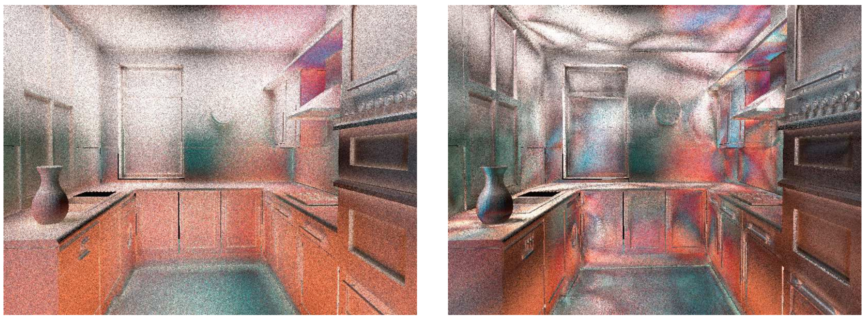


(a) Chart of Box scene.



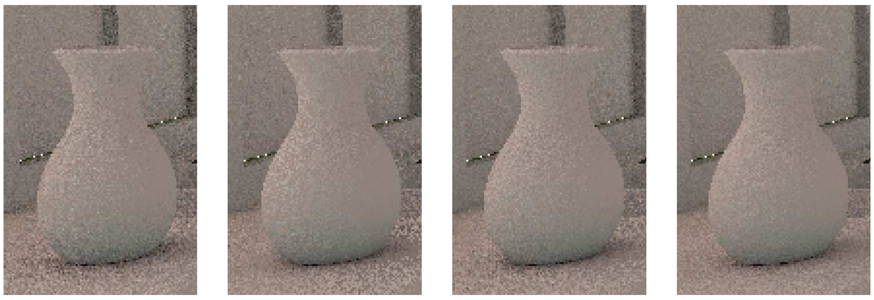
(b) Chart of Kitchen scene.

Figure 4.5: Charts of the comparison of efficiency for both box and kitchen models. Computation time in seconds for an image is measured in X axis, and Y axis measures the Mean Square Error, averaged for all pixels, for the computation of the obscurances. Series of 4, 9, 16, 25 and 36 samples, are measured for each technique (random, systematic, stratified and halton) and model (box and kitchen).



(a) MSE map for random. (b) MSE map for systematic.

Figure 4.6: These images represent maps of Mean Square Error (MSE) for (a) uniform random and (b) systematic sampling obscurances. Systematic variation of error in (b) is clearly appreciated.



(a) Random. (b) Systematic. (c) Stratified. (d) Halton.

Figure 4.7: Detail from the kitchen model image, for a visual comparison of all four sampling methods; (a) uniform random, (b) systematic, (c) stratified and (d) quasi-Monte Carlo with random offset.

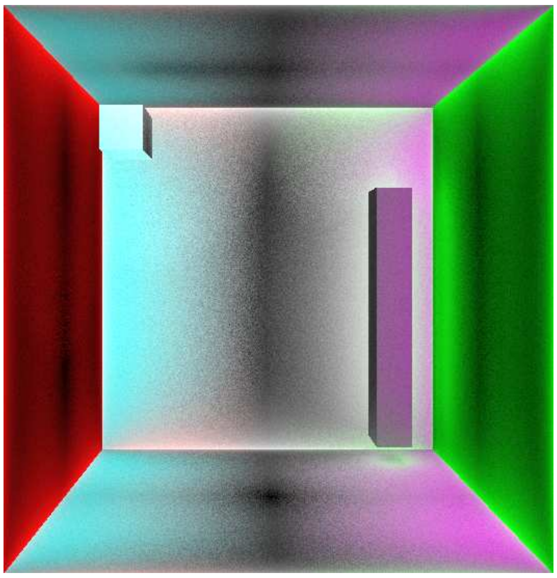
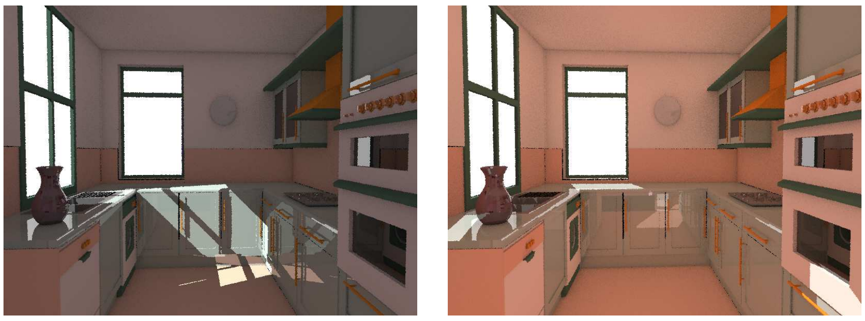


Figure 4.8: Color map for the sample variances for the obscurance computation. We can appreciate that different parts of the image show different variance values.



(a) Greenish ambient term.

(b) Reddish ambient term.

Figure 4.9: These two images show different ambient intensities for indirect lighting when different directional lights are applied. Image (a) shows greenish ambient and image (b) shows reddish ambient lighting.

4.7 Summary

In this chapter we have presented various improvements for the computation of obscurances. First we have added color bleeding by modifying slightly the original equations and obtained much more realism with no added cost.

Some different possibilities for the ρ function have been studied and we have concluded that the *square root function* $\rho(d) = \sqrt{\frac{d}{d_{max}}}$ is the one that works best for our purposes. Different values for the *maximum distance parameter* d_{max} of the obscurances have been tested, concluding that we obtain best visual results when d_{max} has a value between 1/3 and 1/2 of the scene.

The problem of the strong secondary reflectors has been presented and a possible solution has been introduced, but this solution increases the cost of computing the obscurances.

Then we have compared four sampling methods to compute obscurances, uniform sampling, systematic, stratified and Halton with random offset. Both systematic and stratified have shown an improvement of 50% in efficiency with respect to uniform sampling, although systematic sampling presents a highly irregular distribution of the error. Halton sampling with random offset has resulted in still a higher improvement, so this is the technique usually chosen to compute obscurances.

Finally a new method to compute the average ambient intensity and the average reflectivity is introduced. These values, combined with the obscurances, are used to compute the final value of the indirect illumination and they are considered unique for the scene at a given time. When light sources move or change intensity, as we will see in section 7.3 of chapter 7, this method will be useful to compute different intensities an reflectivities for every frame.

These studies and techniques to improve the computation of the obscurances are general and applicable to most obscurances implementations. The methods used in more particular implementations (videogames, production rendering and animations) are studied in next chapters.

Chapter 5

Obscurances in diffuse environments: videogames

As seen in chapter 3 the idea of obscurances was first thought to be used in videogame environments, by precomputing the obscurance values and saving them in obscurance maps, using these maps in real-time while navigating the scene. As obscurances are computed locally, we show in section 5.1 that they can be recomputed in real-time in the environments of a moving object and we present an algorithm to do that.

The advantage of obscurances with respect to radiosity in this context is that they are much faster to compute, and can even be accelerated by using GPU techniques. In section 5.2 we use the depth peeling technique to compute the obscurances for objects and scenes, using programmable shaders of modern GPU cards.

The techniques explained in this chapter are introduced in following papers:

- *Real-Time Obscurances with Color Bleeding* [60] (section 5.1 and
- *Real-Time Obscurances with Color Bleeding (GPU Obscurances with Depth Peeling)* [61]. (section 5.2).

5.1 Real-time animation of objects

5.1.1 The problem

Although the computation of the initial obscurances is not real time, we can update them in real-time for a small number of polygons.

The initial computation of obscurances may need several seconds to compute depending on the scene complexity and the number of rays we send per patch. Once we have the obscurances computed we can use them in obscurance maps in real-time. In global illumination, if an object of the scene moves, the illumination of the object influences the illumination of the rest of the scene. But in the case of obscurances, as they are computed locally within a limited distance, if a small object of the scene moves we only need to recompute the obscurances of the moving object and the objects around it, within the distance given by the parameter d_{max}. This computation is much faster and we can achieve real-time. The algorithm works as follows.

```
for all p in patches do
    computeInitialObscurances(p)
    if ray from p hits moving object then
        Store p in list of influenced patches
    if p is part of moving object then
        Store all hit patches in list of influenced patches
```

Algorithm 1: Program that creates the initial list of influenced patches.

```
for all p in moving object do
    RecomputeObscurances (p)
    Store hit patch in list of influenced patches
    Store hit patch in new list of influenced patches
for all p in list of influenced patches do
    RecomputeObscurances (p)
    if ray from p hits moving object then
        Store p in new list of influenced patches
List of influenced patches:=new list of influenced patches
```

Algorithm 2: Program that uses the previous list of influenced patches while computing a new one.

5.1.2 The algorithm

We need to keep in memory the list of patches that need to be updated from frame to frame. There are patches of three kinds: One, the patches of the moving object itself. Second, the patches that are within the maximum distance around the moving object considering its position in the actual frame. And third, the patches around, but for the position in the next frame.

We store initially in a list the patches influenced by the dynamic objects, i.e., the patches that have been reached by rays from a patch in a dynamic object in the initial obscurances calculation. When the object moves to a new position the obscurances of this list have to be recalculated. Obviously, also the obscurances of the patches of the moving object have to be recalculated, and this causes the update of the list of influenced patches. Lastly, the obscurances of these patches have to be recalculated, ending the whole updating process.

The algorithm thus starts by creating the list of influenced patches (algorithm 1). Once computed, and when an object moves to a new position, the obscurances are recomputed (algorithm 2).

5.1.3 Implementation and results

Figs. 5.1 and 5.1 show images captured from a demo videogame done within the game engine Crystal Space [2] to which real-time obscurances as previously described have been incorporated. In the web page `http://ima.udg.es/~amendez/thesis/` some more images and videos can be found. In the three demonstrative videos we achieve rates up to 7 frames per second (computed with a 900 Mhz AMD3) in a room with 4016 patches and recomputing about 150 of them.

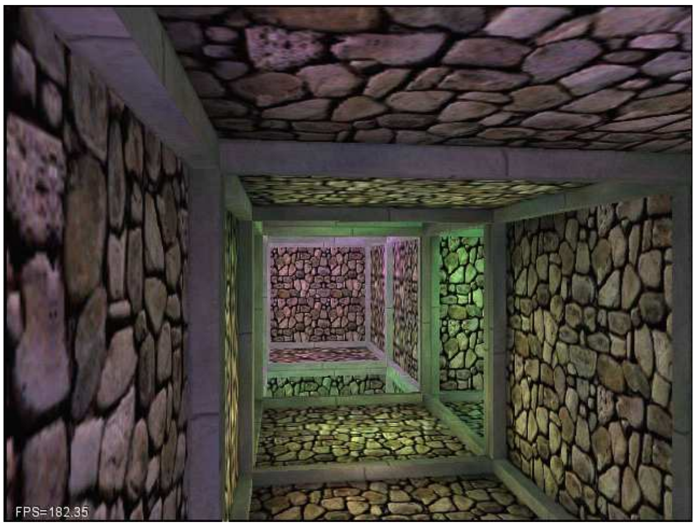
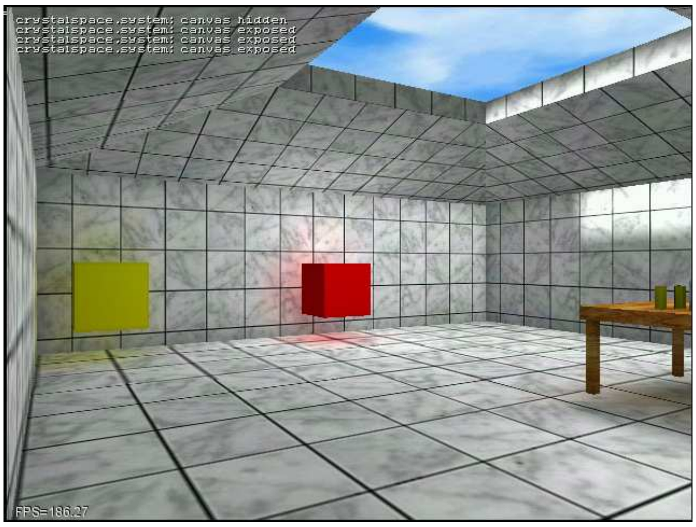


Figure 5.1: A captured image of a scene of a demo videogame in *Crystal Space* engine with obscurances illumination added.



```
crystalspace.system: canvas hidden
crystalspace.system: canvas exposed
crystalspace.system: canvas exposed
crystalspace.system: canvas exposed
```

Figure 5.2: A second snapshot of a *Crystal Space* scene with obscurances. It demonstrates the yellow and red interreflection effects.

5.2 Use of GPU: Obscurances with depth-peeling

The basic idea of the depth peeling technique [28] is to extract visibility layers from the scene in order to do some computation between them. We can see the pixel image resulting from depth peeling as being equivalent to tracing a bundle of parallel rays through the scene where each pixel corresponds to a ray in the bundle. Each of these rays may intersect several surfaces in the scene, and through depth peeling we can discover all of the intersections in the form of image layers.

The computation of the obscurances is divided into two phases. In the first phase, layers are obtained using depth peeling. In the second phase, the obscurances between each pair of layers are computed and the result is added and averaged in the correspondent obscurance map position. Both phases use GPU programming.

5.2.1 From Global Lines to Depth Peeling

Global Lines

Sbert [75] demonstrated that casting cosine distributed rays from all patches in the scene is equivalent to casting global lines joining random points of the bounding sphere of the scene. Furthermore, it is also equivalent to casting bundles or parallel rays of random directions (see Figure 5.3). Bundles of parallel rays can be efficiently cast on the graphics hardware using the depth peeling algorithm.

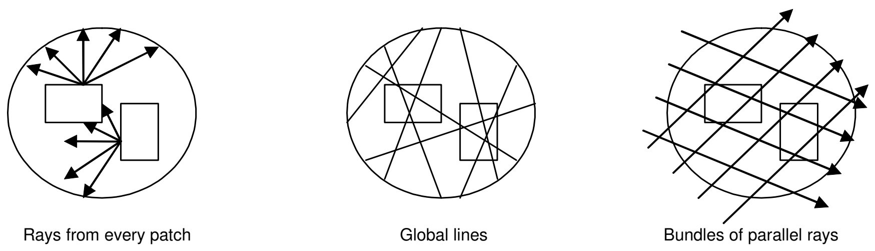


Figure 5.3: Different ray-tracing techniques for computing obscurances.

Depth Peeling

The basic idea behind the depth-peeling technique is to extract visibility layers from the scene in order to do some computation between them. In [28], the technique is used to achieve order-independent transparency. Global illumination [86, 34] has been done also with depth-peeling.

We can see the pixel image resulting from depth-peeling as being equivalent to tracing a bundle of parallel rays through the scene where each pixel corresponds to a ray in the bundle. Each of these rays may intersect several surfaces in the scene, and through depth-peeling we can discover all of the intersections in the form of image layers and not only the closest one obtained by the z-buffer algorithm.

5.2.2 GPU obscurances using depth peeling

Once we have chosen a random direction for the bundle, the computation of obscurances with depth peeling is divided into two phases. In the first phase, layers are obtained using depth-peeling. In the second phase, the obscurances between each pair of layers are computed and the result is added and averaged in the corresponding obscurance map position.

Depth Peeling

We assume that, in a pre-processing step, the scene is completely mapped to a single texture atlas. A texel of the texture atlas corresponds to a small surface area, that corresponds to obscurance patches. When a patch is referenced, we can simply use the texture address of the corresponding texel. The obscurance computation picks a random direction and carries out depth-peeling process in this direction. When we let the GPU to do it for us, we use an orthogonal projection, and from the sampled direction we render the scene setting the model-view transform to rotate the sample direction to the z axis.

We use the *pixel* (RGBA) of an image layer to store the patch identification, a flag indicates whether the patch is front-facing or back-facing to the camera and the camera to patch distance. Our pixel buffer is initialized with (-1.0,-1.0,1.0,1.0), giving us reasonable default values.

The facing direction of a pixel can be determined by using the cosine of the angle between the camera's -z vector and the normal vector of the patch. If the result is greater than 0, it is front-facing, otherwise it is back-facing. The cosine can be determined by using the z component of the dot product between the inverse transpose model-view matrix and the normalized normal vector of the patch.

As we store the pixels in a four-component float array (or the RGBA color), we use the first two components to store the patch ID ($RG \leftarrow (u, v)$), the third to store the cosine ($B \leftarrow \cos\alpha$), and the fourth component to store the distance between the camera and the patch ($A \leftarrow z$).

The vertex shader receives the vertex coordinates, the texture coordinates (in (u,v), identifying the texel), and the normal, and it generates the cosine and the transformed vertex position:

```
void main( float4 position      :  POSITION,
           float2 texCoord      :  TEXCOORD0, //Patch ID
           float4 Norm          :  NORMAL, //Patch Normal

           out float4 oposition :  POSITION,
           out float2 otexCoord :  TEXCOORD0,
           out float cosine     :  TEXCOORD1,

           uniform float4x4 modelView,
           uniform float4x4 modelViewInvTrans)
{
           oposition = mul(modelView,position);
           otexCoord = texCoord;

           //sample direction is rotated to (0,0,1)
           cosine = mul(modelViewInvTrans,Norm).z;
}
```

The fragment shader receives the interpolated texture coordinates of the fragment, the position (where z is the depth), the cosine and the interpolated texture coordinates of the patch. For the first layer, the depth does not need to be compared with the previous one. However, for all subsequent layers, we sample the previous layer using the texture coordinates and discard it if the depth of the previous layer (the fourth component of the sample) is closer to the camera than the actual fragment thus getting the peeling effect (Figure 5.4).

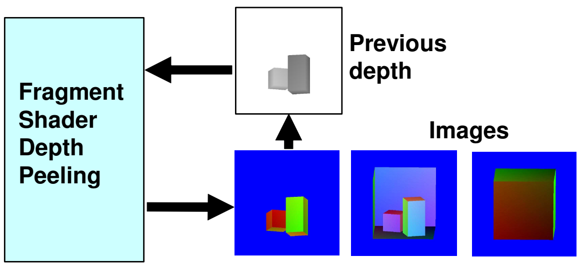


Figure 5.4: Schema of the depth peeling with GPU.

This rendering step is repeated until all pixels are discarded. The images of all the rendered layers define all ray-surface intersections (Figure 5.5).

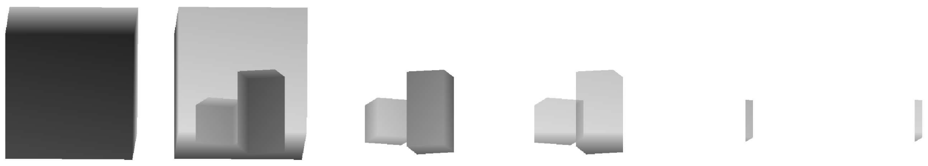


Figure 5.5: Six different image layers showing depth information for each pixel for the Cornell Box scene.

```
void main(float4           position :  WPOS,
          float2           texCoord :  TEXCOORD0, //Patch ID
          float            cosine   :  TEXCOORD1, //Patch Orientation

          out float4       color    :  COLOR, //Patch ID + Orientation + Depth

          uniform sampler2D ztex, //previous depth image
          uniform float    res, //resolution of projection window
          uniform float    first) //is first layer?
{
   if( first == 0.0 ) // not first -> peel
   {
      float depth = tex2D(ztex,position.xy/res).a; //last depth
      if (position.z < (depth + 0.000001)) discard; //ignore previous layers
   }
   color.rg = texCoord;
   color.b = cosine;
   color.a = position.z; //new depth
}
```

Obscurances

For each pair of consecutive layers, the obscurance formula is computed.

We configure the camera to obtain a one-to-one mapping between pixels and texels. The size of the viewport is set to the same resolution as the obscurance map, starting from (0,0), with an orthogonal projection from -1 to +1 in both dimensions.

Now each pair of consecutive images is taken from the texture memory and sent to the graphic pipeline as a stream of points of size 1.0 (render to vertex array). This way we can update a single position in the target buffer for each element of the image. This will generate a pair of point streams A and B that are merged together and sent to the Vertex Shader. Stream A is sent as vertex positions and stream B as texture coordinates. As we generate the streams in both images in the same way, points at the same position in streams A and B are at the same position in consecutive images, thus may see each other in the sampling direction and transfer energy consequently (Figure 5.6).

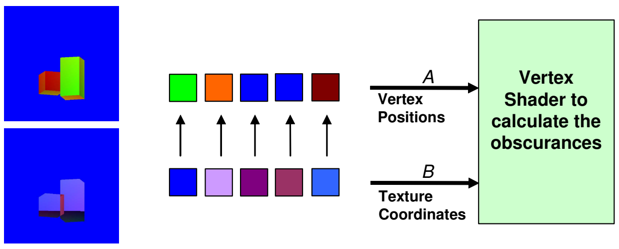


Figure 5.6: Two consecutive layers (left) generate two streams of points carrying patch ID's (middle) that are merged together and processed by the vertex shader (right).

The obscurance computation needs to be done bidirectionally but we cannot generate two values in different positions of the target buffer in a single pass and thus we have to do a two-pass transfer. In the first pass, we update the patches in the projection that generated stream A using the information in pixels of stream B (Figure 5.6). In the second pass the streams are exchanged, thus the same set of shaders are used in both directions.

If a patch in stream A cannot see the corresponding patch in stream B, the vertex carrying this patch is eliminated by moving it out of the view frustum. If patches see each other and the difference between their distances to the camera is less than d_{max} , then the transfer is done. If the distance is greater or transfers with the background, then the patch gets the ambient reflectivity. When the transfer process is done, we generate vertex coordinates to update the position in the obscurance map that corresponds to the patch identified by the two first components of the current pixel element in stream A.

The vertex shader needs to generate vertex coordinates in homogeneous clip space. The desired position is encoded as the patch ID but is in a normalized form (as 2D texture coordinates are in the range [0..1]). The following formula computes which homogeneous clip coordinates we need to generate to obtain the desired normalized window coordinates given a camera and a viewport set as explained earlier:

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} 2x_w - 1 \\ 2y_w - 1 \\ 2z_w - 1 \\ 1 \end{pmatrix} \tag{5.1}$$

Note that the clipping process only keeps the fragment if $-w_c \leq z_c \leq w_c$. As we define a w_c as 1, the clipping process will only keep the fragment when $-1 \leq z_c \leq 1$. If $z_c = 2.0$ then $z_w = 1.5$ and the fragment is out of the view frustum and is discarded. If we set $z_c = 0.0$ then $z_w = 0.5$, thus the vertex is kept by the clipping process. So we can use the z_c value as a way to accept or discard vertices. The vertex shader for the obscurance transfer process is:

```
void main( float4 A   :  POSITION, //x,y = ID; z = Orientation; w = Depth;
           float4 B   :  TEXCOORD0, //x,y = ID; z = Orientation; w = Depth;

           out float4 oposition :  POSITION,
           out float2 pA        :  TEXCOORD0, //Texture coordinates of A.
           out float2 pB        :  TEXCOORD1, //Texture coordinates of B.
           out float  distance  :  TEXCOORD2, //Distance.

           uniform float        direction) //Switch of direction
{
   //If patches see each other, i.e.  both exist and one is front facing and
   //the other is back facing.
   //Direction tells if we are transferring in the camera -z direction or +z.
   //A.r contains the patch ID. If it contains -1.0 means that it does not
   //belong to the scene.
   //A.b contains the cosine.
   if(((direction == 0) && (A.r != -1.0) && (A.b < 0.0)
        && ((B.b > 0.0) || (B.r == -1.0)))
      || ((direction == 1) && (A.r != -1.0) && (A.b > 0.0)
         && ((B.b < 0.0) || (B.r == -1.0))))
   {
      pA = float2(A.r, A.g); //Set the texture coordinates for A.
      pB = float2(B.r, B.g); //Set the texture coordinates for B.
      //Create vertex to update desired position.  z = 0.0 => kept by clipping
      oposition = float4((pA * 2.0) - float2(1.0, 1.0), 0.0, 1.0);
      //Calculate distance.  If patch in stream B not in scene => distance = 1.0
      distance = (texCoord.b != 1.0)?  abs(B.a - A.a) :  1.0;
   }
   else //If there is no transfer move out from the view frustum.
   {
      // z = 2.0 to get the id ignored by clipping.
      oposition = float4( 0.0, 0.0, 2.0, 1.0 );
      p1 = p2 = float2(1.0, 1.0);
      distance = 0.5;
   }
}
```

The fragment shader just applies the obscurance formula $\rho = \sqrt{d/d_{max}}$ if $d < d_{max}$ and 1 otherwise.

```
void main( float2 pA : TEXCOORD0, //Texture coordinates of A.
           float2 pB : TEXCOORD1, //Texture coordinates of B.
           float  distance :  TEXCOORD2,

           out float4 ocolor :  COLOR,

           uniform sampler2D reflectivity,
           uniform float     dmax,
           uniform float3    ambient)
{
   if(d>=dmax) ocolor.rgb = ambient; //If distance > dmax, add ambient
   //else we apply the obscurances formula
   else ocolor.rgb = tex2D(reflectivity,pB).rgb * sqrt(distance/dmax);
   ocolor.a = 1.0;
}
```

Figures 5.7, 5.8 and 5.9 show the results of applying our algorithm to models of the *De Espona* library. We show respectively the obscurances map, obscurances with direct illumination, and direct illumination with constant ambient term. Observe the quality of the illumination obtained with obscurances.

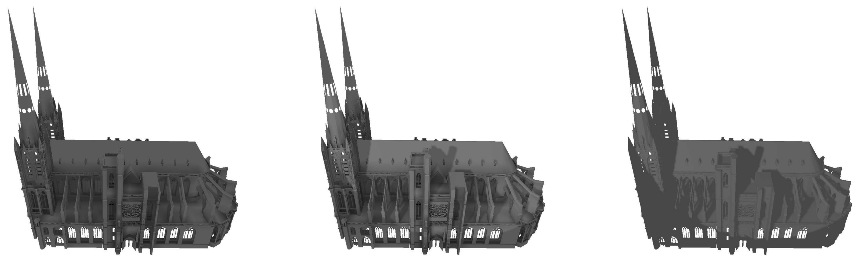


Figure 5.7: Cathedral model, 193180 polygons, obscurances computed in 38 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.

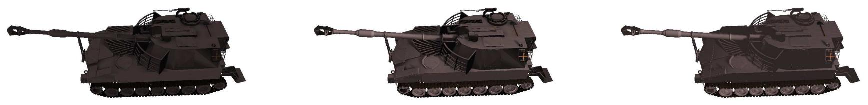


Figure 5.8: Tank model, 225280 polygons, obscurances computed in 38 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.

5.3 Summary

In this chapter we have shown how obscurances can be computed for real-time diffuse environments like videogames and virtual reality. Two different techniques have been applied.

First we have demonstrated that obscurances can be updated at interactive rates for a solid moving object and its close environment thanks tho the locality of the computation of obscurances.

The second approach is useful to precompute initial obscurances for objects or scenes using a GPU version of the depth peeling technique.

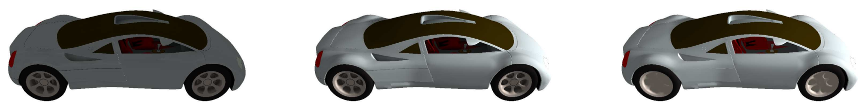


Figure 5.9: Car model, 97473 polygons, obscurances computed in 32 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.

Chapter 6

Obscurances in non-diffuse environments

Until now we have computed the obscurances for every patch of the scene without taking care of the point of view, as we have only dealt with diffuse environments, that reflect to the eye the same illumination no matter its position. There are, though, some environments and materials in which the perceived illumination depends on the point of view, this is, the relative position of the eye and the object with respect to the light. We will use ray-tracing techniques to generate images with obscurances in which the illumination of the scene depends on the point of view (i. e., the scene contains objects with non-diffuse materials) and to include specular and translucent effects (see section 2.5 and [31]).

In this chapter we first envisage how the obscurances can be used plugged in ray-tracing-like algorithms (section 6.1) and in another section (6.2) we study how we can modify the obscurances concept to deal with non-diffuse materials.

The techniques explained in this chapter are introduced in following papers:

- *Obscurances for Ray-Tracing* [62], (section 6.1),
- *Obscurances for Ray-Tracing (Extended Version)* [63] (section 6.1) and
- *Obscurances in General Environments* [59] (section 6.2).

6.1 Obscurances with ray-tracing

The value of the obscurances integral (3.1) is computed for the point hit by a ray cast to the scene from the eye through a pixel. The main advantage of the ray-tracing approach is that we can add to our rendered images some effects that are easily obtained with ray-tracing, like specular reflections and gleams.

We compute the lighting of the hit point by sending three kind of rays. The direct lighting is computed by sending rays to the light sources and computing its contribution if they are visible. The diffuse indirect lighting is computed using the obscurances. Finally, for specular objects additional rays are cast to find the reflected (or refracted, if it is the case) object, finding its illumination in a recursive way.

6.1.1 Algorithm

A ray is traced in the usual way from the viewpoint through a pixel, hitting a point P of some surface in the scene. Then from P, three kind of rays can be traced:

- **Shadow rays**: N_{dl} direct illumination rays are traced to the light sources from the hit points. Stochastically a random point of a random light source of the scene is selected and a shadow ray is traced from the hit point to the selected point. If both points see each other, contribution of the light source to the hit point is computed. Figure 6.1.c shows this contribution.

- **Obscurance rays**: N_{obs} rays are stochastically traced with a $\cos\theta$ distribution to compute the obscurance of point P, this is, we solve by Monte Carlo the integral in (4.1). In this way we compute an estimator of indirect light for this point by multiplying the obscurance term by the diffuse reflectivity at the point and the ambient intensity. Figure 6.1.a shows the obscurances contribution, that multiplied by the reflectivity color of the objects (fig. 6.1.b) results in the indirect light contribution (fig. 6.1.d).

- **Specular rays**: When the hit point corresponds to a glossy or specular surface a path is followed and the reflected contribution is computed in a recursive way according to usual ray-tracing. This is shown in fig. 6.1.e.

The final illumination is then given by adding the direct lighting, specular effect and diffuse effect represented by the obscurances multiplied by ambient light. Following the kitchen example, we finally get the image in fig. 6.1.f.

6.1.2 Results

The ray-traced obscurances have been implemented in the SIR system [55].

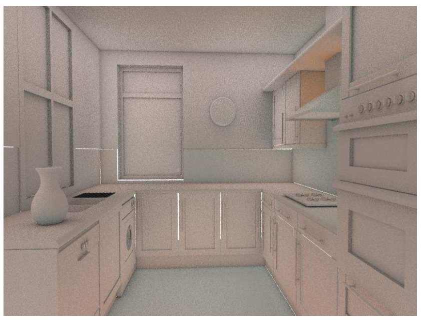
All images have been rendered on a Pentium 4 1.6 Ghz with 1 Gb RAM. We have used in the obscurances computation 8 rays per pixel and 5 rays per hit point, that makes a total of 40 rays per pixel. The kitchen scene is composed of 28937 triangles featuring 203 objects.

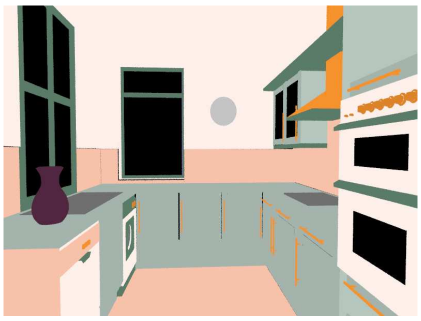
Besides the kitchen (fig. 6.2), two more models have been used to test our algorithm. One represents the inner part of an aircraft, including seats, windows, a screen, etc. It is composed of 184687 triangles featuring 461 objects and it is shown in fig. 6.3. The other model represents a set of stairs, banisters, columns, etc., that could be the main entrance of a castle or a theater. It has 121032 triangles featuring 580 objects and is shown in fig. 6.4.

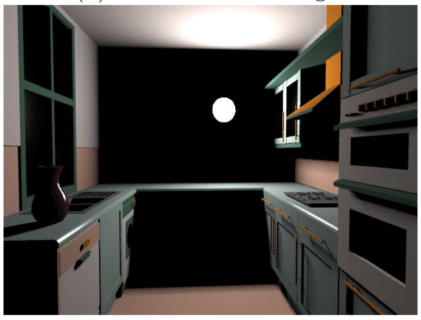
Comparison to Path Tracing

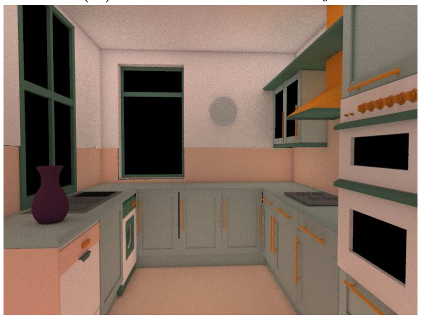
Here we compare our method to Path Tracing (PT), implemented also in the SIR [55]. Thus both methods share the same ray-tracing routines allowing for unbiased comparison. In fig. 6.5 some other views of the kitchen are shown and compared to PT. In fig.6.6 images rendered from the other models (aircraft and stairs) and a different illumination of the kitchen are also shown and compared to PT.

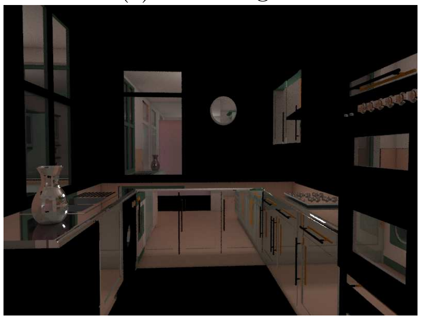
When rendering all these images, in the ray traced obscurances method, for every pixel 40 rays are used to get the direct light, other 40 rays are traced to


(a) Obscurances image.


(b) Diffuse reflectivity.


(c) Direct light.


(d)=(a)x(b) Indirect light.


(e) Specular surfaces.

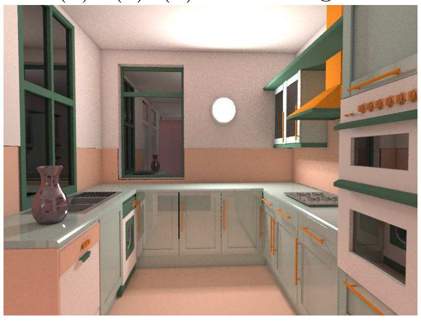

(f)=(c)+(d)+(e) Final image.

Figure 6.1: These six images show the contribution that each kind of ray of our rendering algorithm give to the final image and how it is computed. The first image (a) shows our main contribution: the obscurances. The second (b) image shows the diffuse color of each object in the scene. By multiplying both images we get indirect light image (d). Direct light is shown in (c). Adding the contribution of the specular surfaces (e) to (c) and (d) we get the final image (f). All the images of the kitchen scene shown here have a resolution of 800x600 pixels.

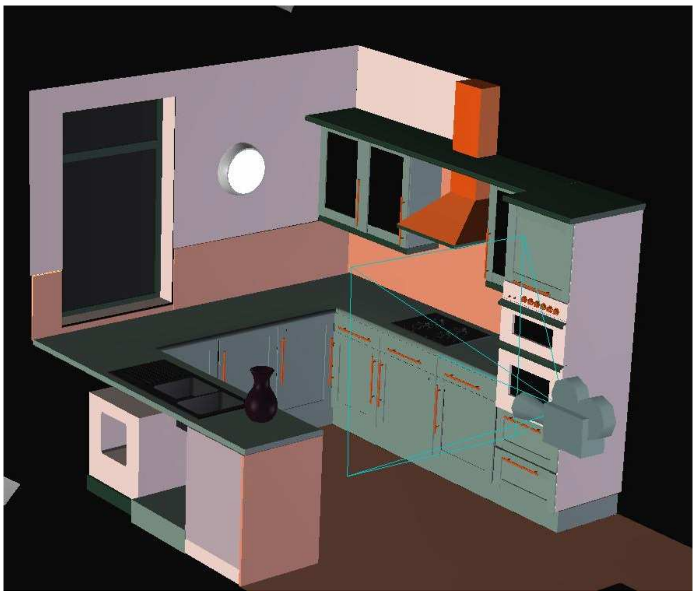


Figure 6.2: Kitchen model showing the camera position.

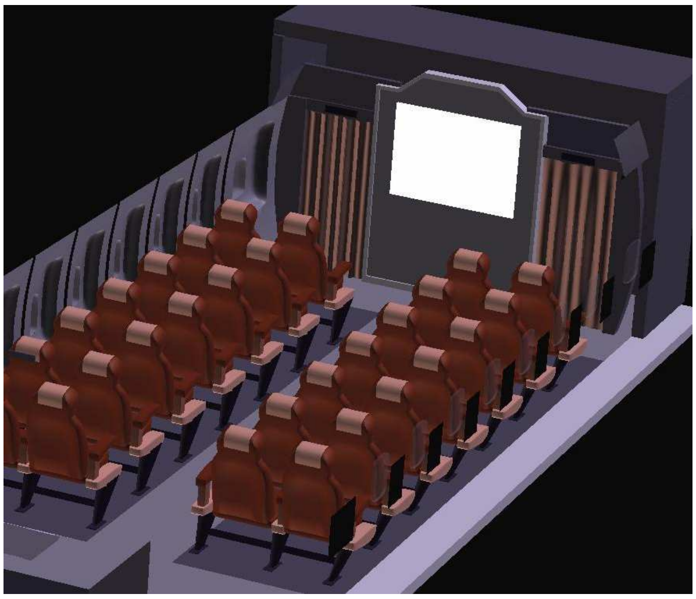


Figure 6.3: Aircraft model.

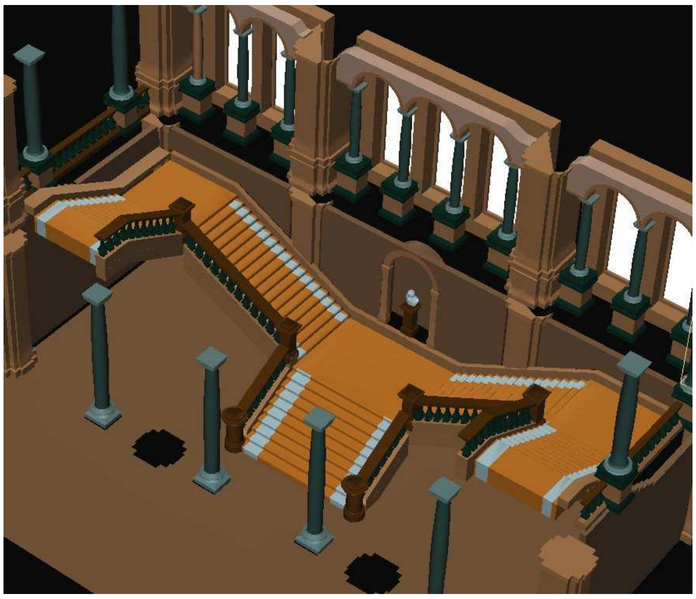


Figure 6.4: Stairs model.

get the obscurances, and if a ray hits a non-diffuse surface one additional ray is cast to account for specular effects up to six levels of depth if applyable.

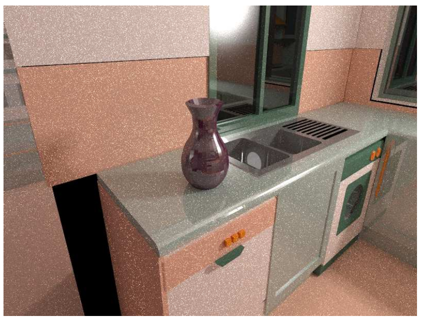
To achieve the same error for direct light in the path tracing method, 40 rays are also used per pixel. Each of these rays account for the first level of a path that will continue up to 6 levels of depth to get the indirect light. And same as before, when a non-diffuse surface is hit an additional ray is cast to get the specular effects.

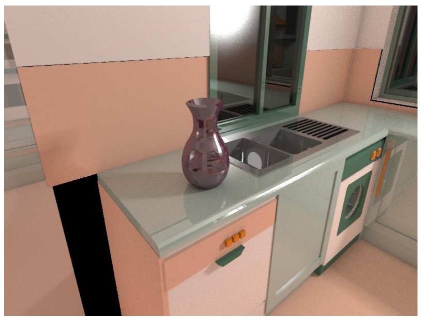
There is no surprise then that, for every pair of images, when a visual comparison is done, direct illumination looks the same and indirect illumination is perceived with much more noise in PT than with obscurances, even when PT images take almost 10 times more to compute than the obscurances ones.

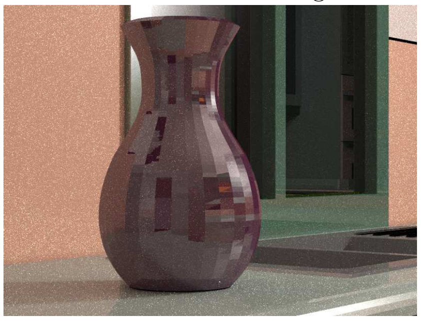
To perceive much better the differences between both methods, fig. 6.7 show the final images of aircraft model in a bigger presentation: path tracing, fig. 6.7a, and obscurances based, fig. 6.7b. Apart from the noise level, we can see the difference between a true global illumination solution, PT, and our obscurances based solution. In the PT solution, (figure 6.7a), the objects near the light source (in this case we can not see the light source beause it is behind the camera) are lighted due to the ceiling and other walls that act as secondary sources. On the other hand in the obscurances image, the seats and walls in the rear part of the aircraft image are more illuminated due to the ambient light term. A solution to this problem would require to store the incoming lighting of these secondary emitters.

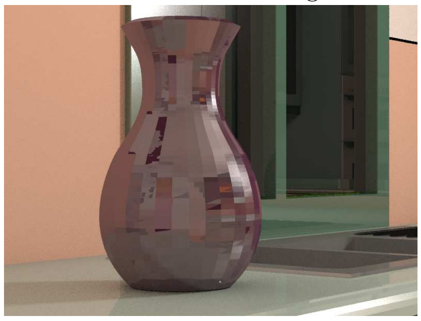
6.2 Obscurances in non-diffuse environments

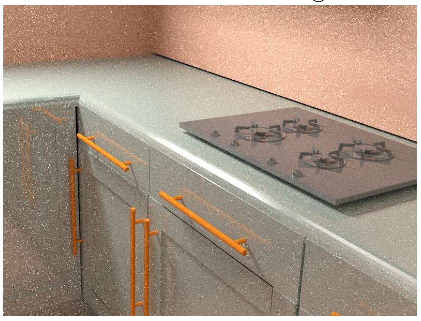
Despite the introduction of specular and glossy effects, the computation of the obscurances itself only takes account of the diffuse reflective illumination. In this section we study how the obscurances should behave when they are computed in an environment with non-diffuse materials, like specular or refractive ones.


(a.i) Path tracing.
11596 sec. rendering.


(a.ii) Ray traced obscurances.
1318 sec. rendering.


(b.i) Path tracing.
17788 sec. rendering.


(b.ii) Ray traced obscurances.
2210 sec. rendering.


(c.i) Path tracing.
14579 sec. rendering.

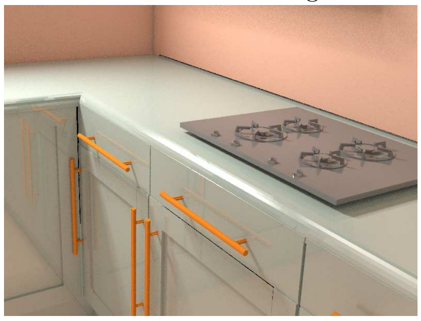
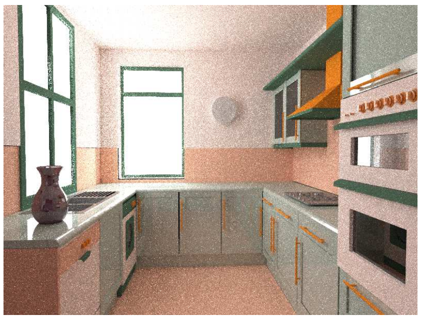
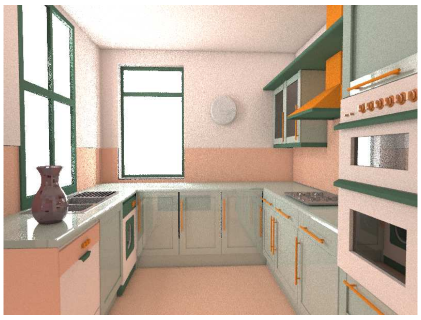
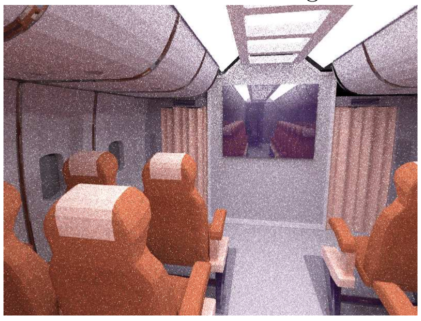
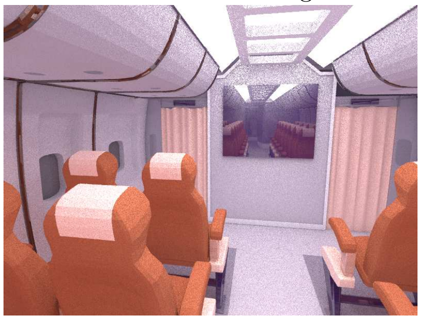

(c.ii) Ray traced obscurances.
1624 sec. rendering.

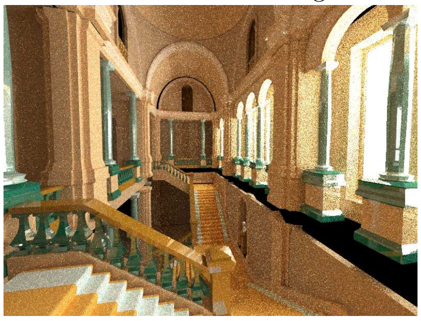
Figure 6.5: Multiple views of the kitchen scene compared with path tracing results. Image resolution is 800 x 600 pixels.


(a.i) Path tracing.
6510 sec. rendering.


(a.ii) Ray traced obscurances.
655 sec. rendering.


(b.i) Path tracing.
14309 sec. rendering.


(b.ii) Ray traced obscurances.
1495 sec. rendering.


(c.i) Path tracing.
11180 sec. rendering.

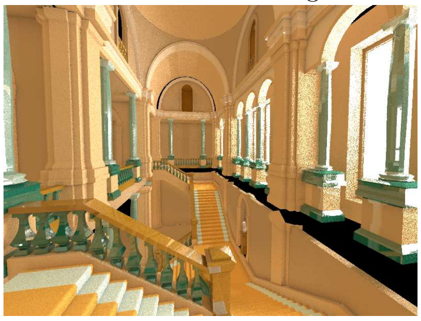
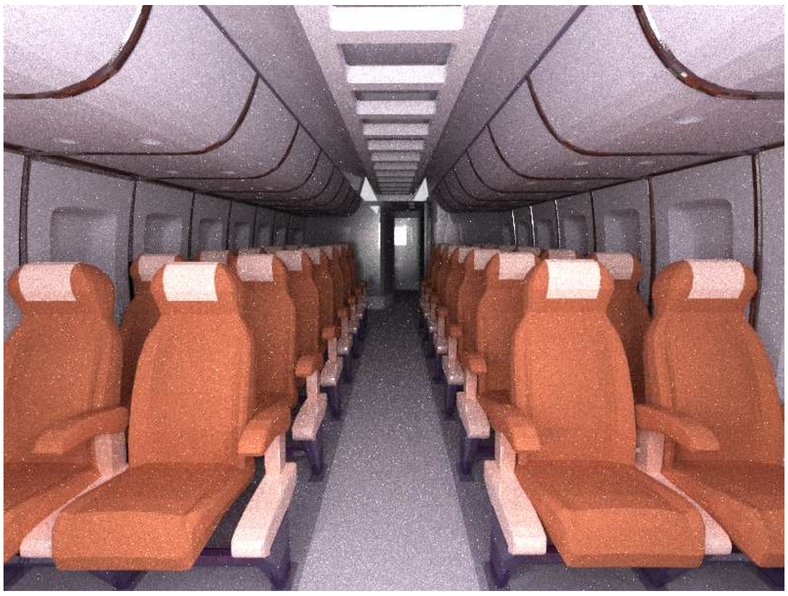

(c.ii) Ray traced obscurances.
858 sec. rendering.

Figure 6.6: The kitchen with daylight illumination and views of the aircraft and the stairs, all compared with path tracing results. Image resolution is 800 x 600 pixels.


(a) Path tracing. 10190 sec. rendering.

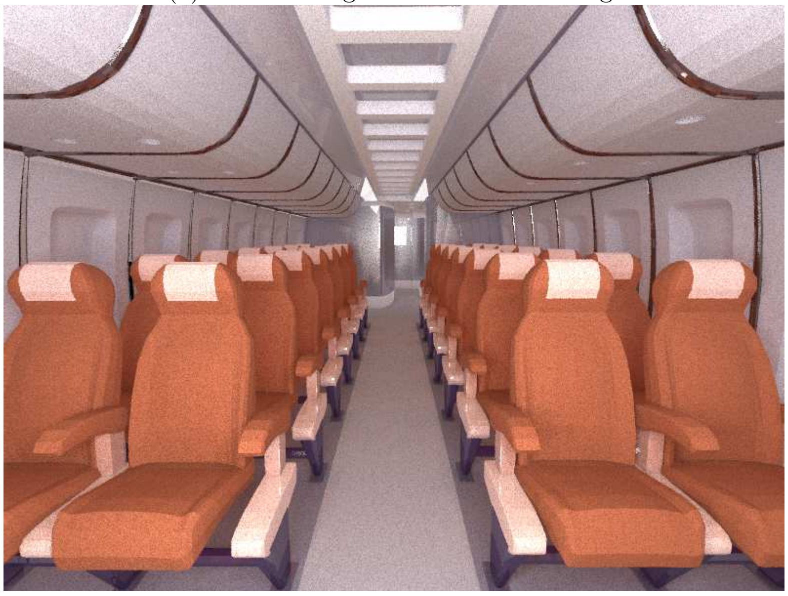

(b) Ray traced obscurances. 1000 sec. rendering.

Figure 6.7: Final aircraft images in more detailed presentation. Obscurances image is computed 10 times faster and presents significantly less noise than path tracing image.

We propose to extend the original algorithm that computes the obscurances with color bleeding in a way that it can cope with other kind of materials and the interactions between them, such as perfect specular surfaces, refractive and translucent objects.

The obscurances are always computed for a point in a diffuse surface, or at least for a point in a surface that has part of its BRDF diffuse. The difference is when the objects around this point have specular properties. The main obscurances assumption is that the more occluded is a point, the darker it will appear, this is why we apply the ρ function to the *distance* parameter. But when the point is enclosed by specular (either reflective or refractive) objects we have to think different, because indirect lighting can come reflected from the rest of the scene and, in consequence, the point considered would not appear so dark as if it were enclosed by diffuse objects. Some alternatives to achieve these effects are discussed.

Another concept of obscurances (*inner obscurances*) is introduced for diffuse translucent materials and the special case of plants and trees with translucent leaves is studied.

6.2.1 Obscurances in diffuse environments

In this section we will be showing our results with the example of the vase view of our kitchen model presented with different material properties.

Figure 6.8 presents an image of a vase made of a white diffuse material. The indirect light is decoupled from direct light and is shown in figure 6.9. The image of the obscurances values is shown in figure 6.10. Direct illumination is computed by selecting random points on the light surface and testing visibility. The specular effects are computed by following the corresponding reflected or refracted rays.

All images presented in this paper are made of 800×600 pixels and are computed on a Pentium 4 running at 1.6 Ghz with a memory of 2Gb. For each pixel, 40 obscurance rays are sent.

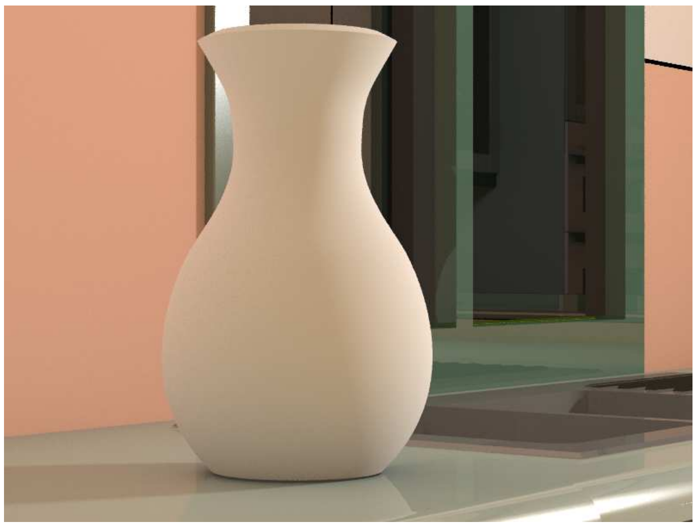


Figure 6.8: Model of a vase made of white diffuse lambertian material with indirect illumination rendered using obscurances technique. It is a 800×600 pixels image and it takes 1563 seconds to compute.

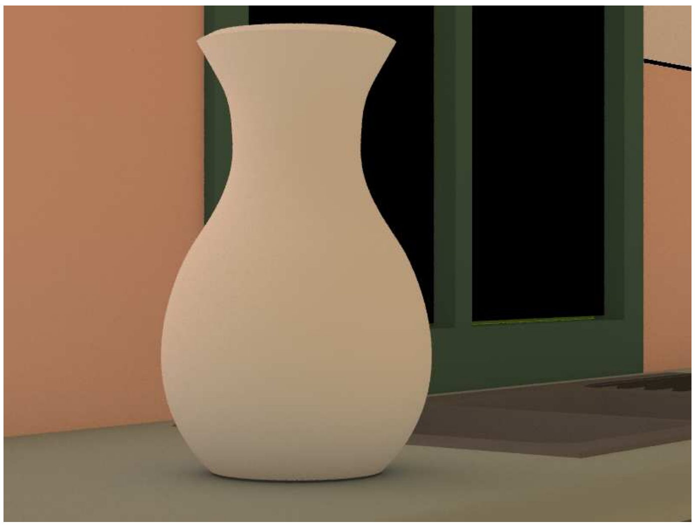


Figure 6.9: Indirect light is computed by multiplying the diffuse color, the ambient intensity and the obscurances. Adding direct light and specular effects to this image, we get image in figure 6.8

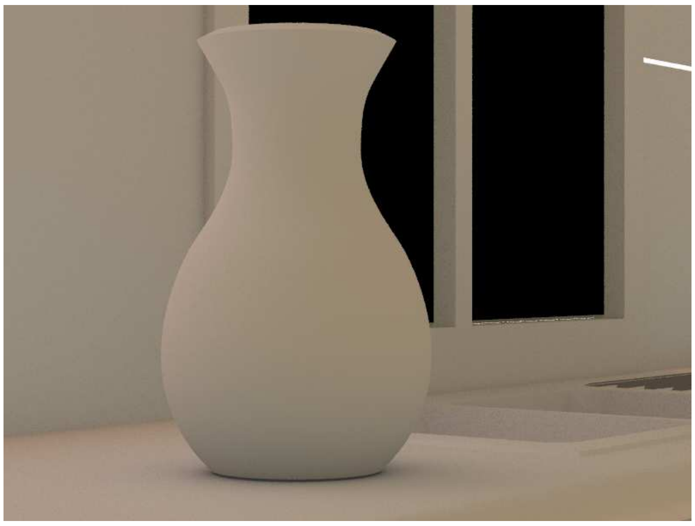


Figure 6.10: Map of the obscurance values for the diffuse vase image.

6.2.2 Generalization to other materials

The obscurances concepts and formulae seen until now work well for diffuse environments, this is, when the object where the point P is located and the objects around it are all made of diffuse reflective materials. But in general environments, materials can have many different reflective and refractive behaviors. These behaviors are modelled with different BRDFs and BTDFs. Therefore we are going to study how the obscurances computation should be modified according to the basic cases for these BDFs different than diffuse: perfect specular material, transparent material, and translucent material.

We can consider that a point P from a perfect specular object has no "own" color, as it takes the color of the object seen from the reflected direction on the surface. For the same reason, it has no "own" obscurance value. We take the color and obscurance value from the first diffuse object found by the recursive reflections of the viewing ray. When we think of perfect refractive (transparent) materials, we follow the same reasoning, but now considering the refracted ray directions.

If point P is on a translucent object, the indirect light comes from the inside of the object and it should be a function of the *thickness* of the object. In this case, the classic obscurance method does not make sense. We propose a similar method, that we call *inner obscurances*, and it is presented in section 6.2.3.

But even when we consider point P to be on a surface from a lambertian reflective object, the interaction of the obscurance rays with the rest of the non-diffuse materials has to be taken into account. In the following subsections we study the changes on the obscurances computation when the objects around the diffuse point P have other material properties.

From the obscurances definition (4.1) we have defined Q as the point seen from P in direction ω. If no object is found in direction ω or it is further than d_{max}, we consider $R(Q)$ to be the average reflectivity color (R_{ave}) and $\rho() = 1$. If the object is found at a distance less than d_{max}, and if it is a lambertian material we take its reflective color as $R(Q)$ and we compute ρ according to the shape in figure 3.2 (we will usually take $\rho(d) = \sqrt{d/d_{max}}$).

But what happens if the object, where Q is located on, is not lambertian?

What if the object is perfectly specular?

Technically, a pure specular material has no "own" color. The color of a specular object in a certain point depends mainly on the "viewing" direction, taking the color of another object seen at a direction given by the law of reflection. In figure 6.11 an image of a perfect specular vase is shown.

In this way we can clearly see what value $R(Q)$ should have taken in equation (4.1) when Q is located on a specular object. As from the definition, $R(Q)$ is the color seen from P in direction ω, we will take the color of the first diffuse object hit by the corresponding reflected direction ω'. Figure 6.12 shows this situation.

We have to discuss also the use of the ρ function. The main obscurances assumption is that the more occluded is a point, the darker it will appear, this is why we apply the ρ function to the *distance* parameter. But when the point is enclosed by specular objects we have to think different, because indirect lighting can come reflected from the rest of the scene and, in consequence, the point considered would not appear so dark as if it were enclosed by diffuse objects.

We have considered three possible solutions. In the first solution, use ρ the same way as with diffuse lambertian materials, i. e., taking into account just the distance from P to Q, and as $R(Q)$, the color of the first hit diffuse material following the corresponding reflected ray. The schema of this solution, the image of obscurances and final image are shown in the first column of figure 6.13. But we can see that this first solution does not fulfill our expectations of less darkness around the base of the vase. That is because we use the same *distance* parameter as if we were considering a diffuse vase. In a second solution, we can consider the *distance* parameter as the sum of the distances of all the corresponding reflected rays needed to find a diffuse surface. On the one hand we

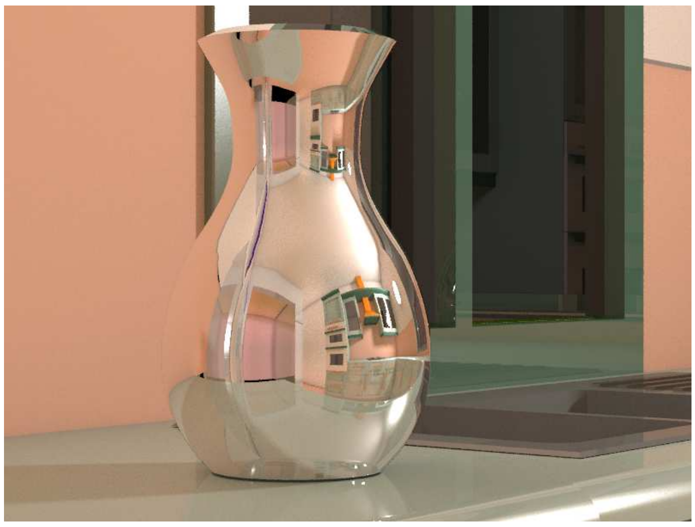


Figure 6.11: Vase made of perfect specular material with obscurances computed as in figure 6.13b. It is a 800 × 600 px image and it takes 2108 seconds to compute.

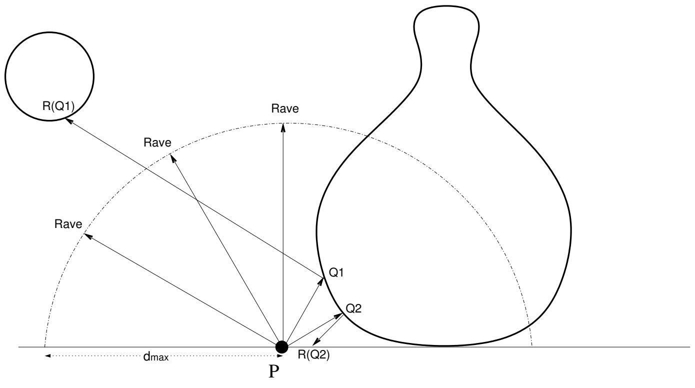


Figure 6.12: Schema of the reflected obscurance rays. Obscurances of the diffuse point P are computed by sending cosinus distributed rays that query the space around P. If some ray finds a specular surface, it follows the corresponding reflected path.

obtain less dark obscurances (and this is what we are looking for). On the other hand, as a collateral effect, we gain computation time, as we can stop sending rays when the sum of the distances overcomes d_{max}. Schema and results are shown in column b of figure 6.13. In any case, we have to cast additional rays when we find a specular object, and this increases the computational cost. This is why we suggest a third option that comes with no additional cost, though visual results (figure 6.13, third column) are not so pleasant. In this last case, we use directly $R(Q) = R_{ave}$ and $\rho = 1$.

Comparing visually these three solutions, the one that gives best results is the second one, but if time is an important issue the third one is faster (1693 seconds in the example of figure 6.13) compared to the two former ones (2179 and 2108 seconds, respectively).

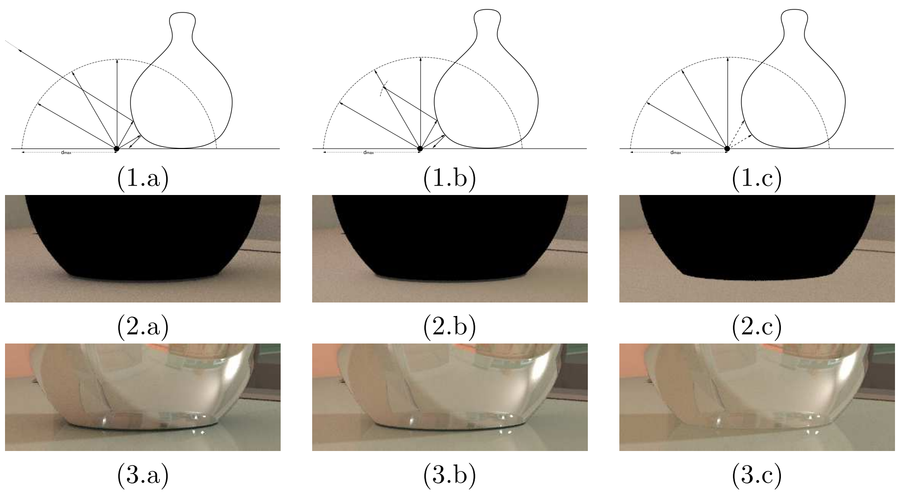


Figure 6.13: The base of the vase is here detailed with three different computations of the obscurances when the surrounding object is perfectly specular. The first row shows the schema of the way obscurances are computed, the second row presents only the obscurances and the third row presents final images. The complete image ($800 \times 600px$) of the first column takes 2179 seconds to compute. For the image in column b it takes 2108 secs. The third image takes 1693 secs.

What if the object is perfectly transparent?

The problem here is similar to the specular case, but with refracted directions instead of reflected ones. Light rays change their directions when traversing through different dielectric materials following the Snell's law of refraction. This is well studied, known and programmed in early versions of ray-tracing (see [24]). Figure 6.14 shows an image of a transparent vase with an index of refraction of 1.5 with respect to the air. In figure 6.15 this situation is shown.

We have similar problems and therefore we propose similar solutions as in previous subsection. This is, an obscurance ray cast from a point of a diffuse object near a transparent object, takes $R(Q)$ as the color of the first hit point of a diffuse material following the refracted (and/or reflected, if it is the case) rays. The *distance* parameter in $\rho(d)$ can be taken into account also in the same way as the three different solutions previously seen in previous subsection.

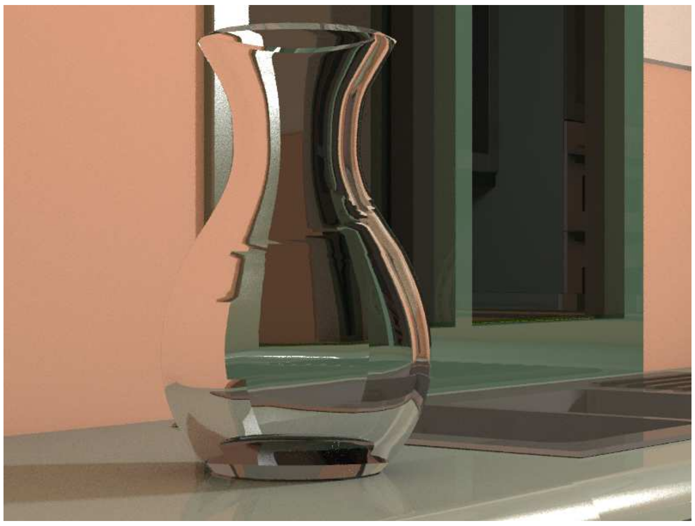


Figure 6.14: Vase made of transparent material like crystal. It is a 800×600 px image and it takes 2403 seconds to compute.

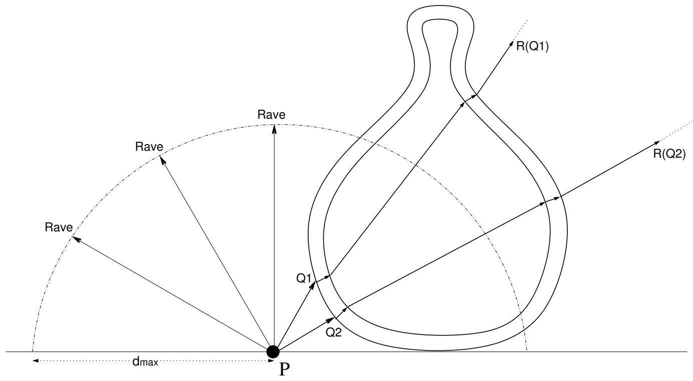


Figure 6.15: Schema of the refracted obscurance rays. Obscurances of P are computed by sending cosinus distributed rays that query the space around P. If some ray finds a transparent surface, it follows the corresponding refracted path.

What if the object is translucent?

A completely different problem is presented here, and depending on the kind of translucency we are considering we can have many different subproblems and complexities. The main feature of a translucent object is that light gets into it and many things can happen to a photon once inside the object, including scattering, backscattering, attenuation or simply traversing the object (see [81]). For simplicity we will only consider attenuation and distribution of photons as they get into the object.

Taking all this into account, what should be the color and intensity seen from P at point Q onto a translucent object? This is difficult to answer, specially if we want to adapt it to the obscurances philosophy and preserve the decoupling of direct and indirect lighting, because the light coming from inside the object comes mainly from the direct light pointing to it from the other side. Certainly it does not depend on the reflectivity color $R(Q)$ (remember we are talking of pure translucent objects) and it does not depend on any function of the *distance*. One of the parameters we could think of is an *average inner indirect lighting* for every translucent object, computed using its geometrical properties and its attenuation parameters (possibly an attenuation color and a maximum distance). The computation of these parameters are out of our focus by now and remains as future work.

A provisional solution is to use *black*, R_{ave} or any value in between as $R(Q)$. As no function of the distance is needed, we take $\rho = 1$.

6.2.3 Obscurances of a translucent material

As we have said before, we consider here a simplified problem of translucent materials considering only the light absorption of the material, not the diffuse scattering or back scattering. Of course the original concept of the obscurances does not apply but we can use the same concept of space querying (inside the object, in this case) and maximum distance to compute a similar concept, that we can dub *inner obscurances*, and are shown in figure 6.17. A similar idea but with a different solution is the *vicinity shading* used by Stewart [84].

We can define the *inner obscurance* as:

$$I(P) = \frac{1}{\pi} a(P) R_{ave} I_A \int_{\omega \in \Omega} \gamma(d(P,\omega)) \cos\theta d\omega \tag{6.1}$$

where

- $\gamma(d(P,\omega))$: function with values between 0 and 1, and giving the magnitude of ambient light incoming from direction ω. It is a special function useful for translucent objects and is equivalent to $1-\rho$.
- $d(P,\omega)$: distance from P to the boundary in direction ω, from the inside of the object.
- θ: angle between direction ω and the normal at P
- I_A: ambient light intensity. It is multiplied by R_{ave} because no color bleeding is used inside the object.
- $a(P)$: Absorption coefficient.

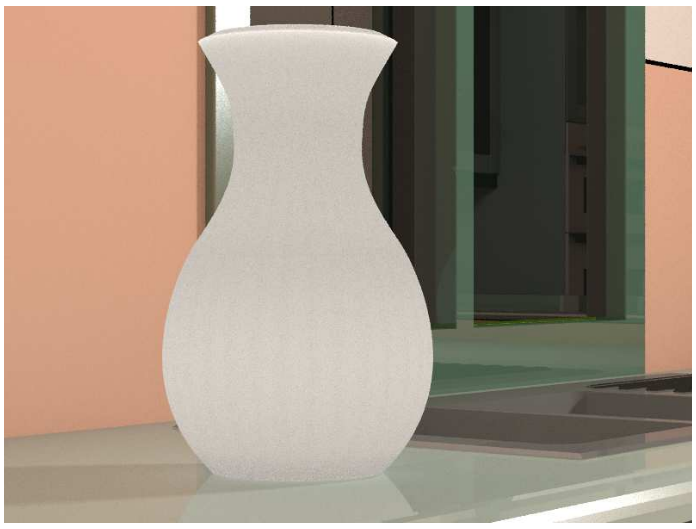


Figure 6.16: Vase made of translucent material like wax. It is a 800×600 px image and it takes 1433 seconds to compute.

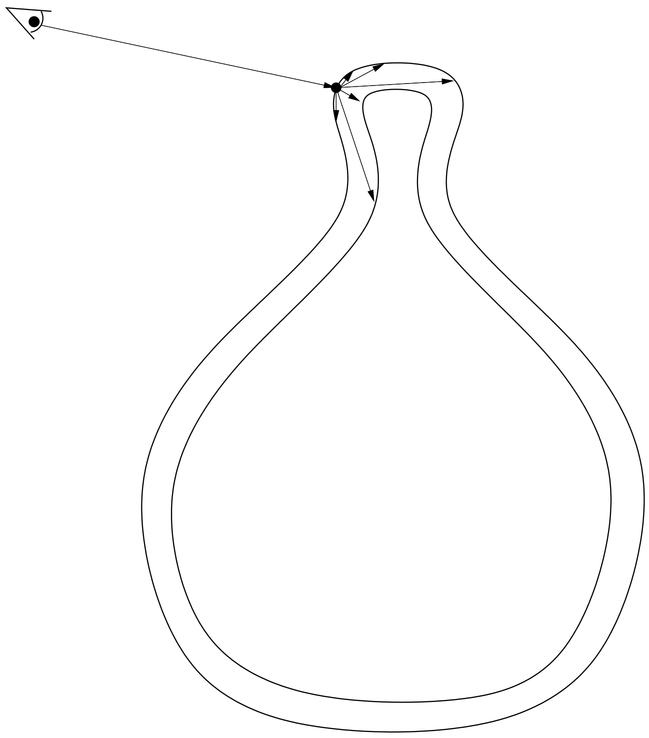


Figure 6.17: Schema of the computation of the *inner obscurance value.* From the inner part of the objects its *thickness* is queried to obtain the inner obscurances value.

- $1/\pi$ is the normalization factor such that if $\gamma() = 1$ over the whole hemisphere Ω then $I(P)$ is $a \times R_{ave} \times I_A$

Here a different maximum distance d_{max} is defined for each different translucent material, giving the idea of the distance at which the light energy is completely absorbed by the material. The function γ can be defined as $\gamma(d) = 1 - \sqrt{d/d_{max}}$.

Figure 6.16 shows our vase as made of a perfect translucent material. Though it is directly illuminated, this kind of material does not reflect the direct light in any way, as it is supposed to fully *get into* the object and scatter. We only see our approximation of the indirect light getting out of the object as if direct light had come to it from all directions. The slimmer parts of the object appear more intensely illuminated than the thicker parts.

One specific case of translucency can be found in complex objects like trees and plants. Normally the leaves are so slim that they are represented with a single polygon with texture (or sometimes we represent many leaves in a polygon, forming *clusters*). We modified our algorithm to deal with simple objects made with single textured polygons, and applied it to trees and plants. In this case we do not take into account the inner and outer spaces of an object, and in consequence the *inner obscurances* do not make sense. We simply compute the obscurances of the backface of a leave and multiply them by the translucency factor. We finally add the back and front obscurances, and the final result is a slightly more clear tree. Figure 6.18 shows our result with a lemon tree.

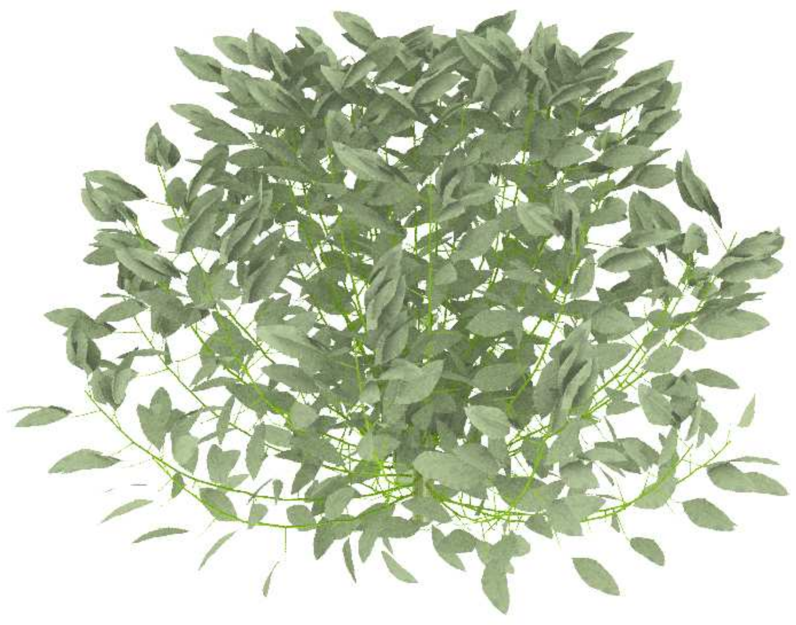


Figure 6.18: This method can be used with good results with models of plants and trees. It is a 800×600 px image and it takes 894 seconds to compute. We have to note that it is a simpler model than the kitchen one.

6.2.4 What about the other BRDFs and BTDFs?

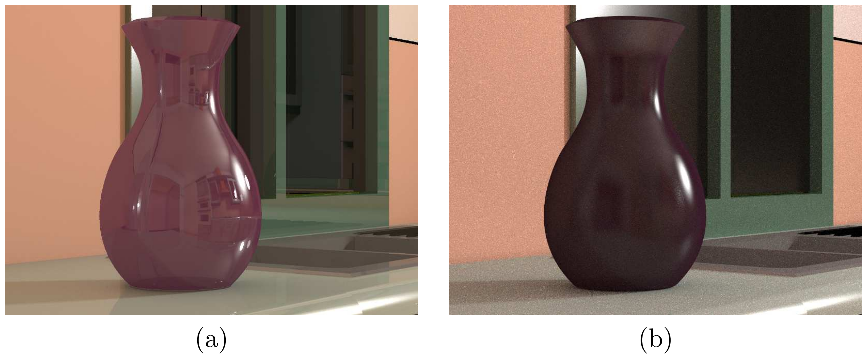


Figure 6.19: Image of the vase with a combination of basic BRDFs and BTDFs. Left: using a combination of the different strategies to compute obscurances (7723 seconds). Right: a global illumination path tracing image of the same vase (19431 seconds).

So far, we have only explained the basic cases. But we can consider any material as partly diffuse, partly specular, partly transparent and partly translucent. Though reality is much more complex than these simple cases or even a linear combination of these, a wide range of materials can be modelled with this method.

We consider in our examples every material as made of a percentage of each four basic materials. Thus, a maximum of four different obscurances computations per hit have to be done using the appropriate algorithm for each case. The problem comes with the specular and transparent cases, that require sending additional rays to follow the paths of the viewing rays, and possibly (if a material has both specular and transparent properties) casting two paths per hit, leading to a quadratic increase of the number of rays sent. To reduce this problem we could choose only one of the two possible ways at each impact hit, using importance sampling. Also, a Russian roulette technique can be used to stop recursion.

Figure 6.19a shows an image of a vase made of a material that combines all BRDFs and BTDFs seen in this paper. Image in figure 6.19b is a path-tracing (global illumination) version of the same model. Note that it presents much more noise.

6.3 Summary

In this chapter we have first presented how the obscurances can be included in a ray-tracing environment to obtain realistic looking ray-traced images. This technique is much faster than ray-traced global illumination techniques, such as path tracing, and resulting images present much less noise. The technique presented here can be used as a fast editing tool or as the final image.

In a second section we have introduced how the obscurances technique can be extended to deal with general environments. The different algorithms and techniques for different basic BRDFs and BTDFs have been introduced. In the case of translucent objects a measure of the inner thickness of an object from a point, an equation based on the obscurances integral but for the interior of translucent objects, has been introduced. Also, a simple model of obscurances for the special case of the leaves of trees has been tested. Finally one possible way to combine these basic material properties has been shown.

We have shown that obscurances contribute to realism not only for diffuse environments, but for general environments too.

Chapter 7

Animations: reuse of information between frames

In this chapter we study how we can save computation time when computing a series of frames in an animation and some of the elements of the scene move, as the light sources or the camera.

In the next section we will study a technique to reuse information between neighbor frames in camera animation. The technique presented here is general and useful for several techniques that compute radiance of a hit point, including path tracing and obscurances. Using examples of path tracing, an unbiased solution is presented. Next, in section 7.2, we will apply the same technique to obscurances, but simplified, as obscurances are always computed for diffuse materials. A technique to reuse indirect lighting (computed with obscurances) between frames with still camera and moving light sources is presented in section 7.3. Finally (section 7.4) both camera and light sources animation are combined in a single algorithm, leading to a new concept that we will call *frame array*; a multi-dimensional array of frames that can be navigated to form movies that present different combinations of animations of light and camera.

The techniques presented in this chapter are introduced in following papers:

- *Reusing Frames in Camera Animation* [64], (section 7.1),
- *Efficient Rendering of Light and Camera Animation for Navigating a Frame Array* [58] (sections 7.2 and 7.4) and
- *Combining Light Animation with Obscurances for Glossy Environments* [56]) (section 7.3).

7.1 Camera animation: reuse of illumination between neighbor frames

In global illumination an image can be computed by tracing paths from the eye (or observer position) trough the pixels that compose the image plane towards the surfaces of the scene. In the path-tracing technique (see section 2.6), from the hit point in the scene a random walk is followed, gathering the energy at every new hit point. The main drawback of these Monte Carlo random walks is the high number of paths needed to obtain an acceptable result. This is still more dramatic in an animation computation, due to the high number of frames to be computed. Thus achieving some sort of path reusing can reduce the computational cost.

To obtain an animation or a sequence of frames in a global illumination framework with production quality, we need to cast many rays per pixel. Each frame has to have high accuracy to avoid both noise in the frame and flickering from frame to frame. An efficient solution to reduce this cost has been presented for camera animation [38]. The first hit of the ray cast from the eye through a pixel is reprojected to neighbor eyes and if there is visibility, the incoming illumination to the hit multiplied by the corresponding BRDF is averaged to previous results. Although very computationally efficient, this solution is biased, as it does not take into account the different probability densities that generated the different contributions to a pixel.

We propose one solution for this problem based on using multiple importance sampling [89], and show that for diffuse surfaces the results of [38] are correct, and for neighbor eyes that are very near from each other the bias is not noticeable. We test our solution with an animation using the path-tracing algorithm for global illumination, and compare it with a classic independent solution and the previous unweighted, biased, technique. Although the results are shown in this section for the path-tracing algorithm, the validity of our technique is general.

7.1.1 Frame reuse

In this section the theoretical framework of our algorithm is introduced. We first introduce the basic *native* estimators, then we show how we can estimate the radiance from a different eye and finally how the radiance estimators obtained with paths from different eyes can be combined by multiple importance sampling.

Estimating the *native* radiance

In global illumination we are interested in the integrals $L^o(i) = \int_{A_i} L(p)dp$ where A_i is the area of pixel i, and $L(p)$ is the radiance from visible scene point x that reaches the observer at point o through point p in pixel i . Introducing a change of integration variable [87] we obtain

$$L^o(i) = \int_S L(x \to o)G(o,x)V_i(x)\frac{\cos^3\theta_i}{f^2}dx, \qquad (7.1)$$

where the integration extends over all scene surface points S, θ_i is the angle between the normal of the screen plane and direction $\omega(x \to o)$ at the center of the pixel i, and f is the focal distance, i.e. the distance from o to the plane of the screen. $V_i(x)$ takes the value of 1 if x is visible through the pixel i and 0 otherwise. The geometric factor $G(o,x)$ is defined as

$$G(o,x) = vis(o,x)\frac{\cos(N_x, \omega(x \to o))}{d^2(x,o)} \qquad (7.2)$$

where $vis(o,x)$ is 1 if x and o see each other and 0 otherwise, N_x is the normal at point x, $\omega(x \to o)$ is the direction from x to o, and $d(x,o)$ is their distance. The radiance $L(x \to o)$ comes from the global illumination equation [43]:

$$\begin{aligned} &L(x \to o) = L^e(x \to o)+ \\ &\int_\Omega \rho(\omega^{in}, x, x \to o)L(x,\omega^{in})\cos(N_x,\omega^{in})d\omega^{in} \end{aligned} \qquad (7.3)$$

where $L^e(x \rightarrow o)$ is the self emitted radiance, $\rho(\omega^{in}, x, x \rightarrow o)$ is the bidirectional reflectance distribution (brdf) function at point x, incoming direction ω^{in} [1] and outgoing direction $x \rightarrow o$. $L(x, \omega^{in})$ is the incoming radiance to x in direction ω^{in}.

Substituting (7.3) into (7.1), and dropping constant terms and self-emission [2], we obtain

$$L^o(i) = \int_S \int_\Omega G(o,x) V_i(x) \rho(\omega^{in}, x, x \rightarrow o) L(x, \omega^{in}) \cos(N_x, \omega^{in}) d\omega^{in} dx \quad (7.4)$$

Primary estimator $\widehat{L^o(i)}$ for $L^o(i)$ is obtained by selecting a point x with probability $p_i^o(x)$ and then direction ω^{in} with probability $p(\omega^{in}; x, x \rightarrow o)$. An unbiased estimator for $L(x, \omega^{in})$, $L(\widehat{x, \omega^{in}})$, can be obtained by any suitable technique, for instance by the random walk path-tracing technique. Thus

$$\widehat{L^o(i)} = \frac{G(o,x) V_i(x) \rho(\omega^{in}, x, x \rightarrow o) L(\widehat{x, \omega^{in}}) \cos(N_x, \omega^{in})}{p_i^o(x) p(\omega^{in}; x, x \rightarrow o)} \quad (7.5)$$

With importance sampling we select probabilities

$$p_i^o(x) \propto G(o,x) V_i(x)$$

and

$$p(\omega^{in}; x, x \rightarrow o) \propto \rho(\omega^{in}, x, x \rightarrow o) \cos(N_x, \omega^{in})$$

In this case the estimator becomes:

$$\widehat{L^o(i)} = a(x, x \rightarrow o) \Omega_i L(\widehat{x, \omega^{in}}) \quad (7.6)$$

where Ω_i (solid angle subtended by pixel i) and $a(x, x \rightarrow o)$ (albedo) are the probabilities normalization constants. Estimator (7.5) is the unbiased *native* estimator for a pixel, that is, the one obtained by sending rays from the observer through the pixel.

Estimating the radiance from a different eye

Consider now a different observer o' (see Fig.7.1). This observer will see x through a different pixel, j. We can obtain a (biased) estimator for $L^{o'}(j)$ reusing the value obtained for the radiance at x with estimator $L(\widehat{x, \omega^{in}})$ (this comes to reusing the path from x supposing the estimator is a random walk, see Fig. 7.1). The estimator for pixel j from eye o' obtained with a ray started from eye o is given by the following expression:

$$L^{\widehat{o',i}}(j) = \frac{G(o',x) V_j(x) \rho(\omega^{in}, x, x \rightarrow o') L(\widehat{x, \omega^{in}}) \cos(N_x, \omega^{in})}{p_i^o(x) p(\omega^{in}; x, x \rightarrow o)} \quad (7.7)$$

Note that the probabilities used in the denominator are the native probabilities used to find point x from eye o through pixel i, but they should be normalized

[1]In fact, the true incoming direction is $-\omega^{in}$, but we use the opposite one to keep the reciprocity in the brdf.

[2]Self emission can be easily dealt with separately.

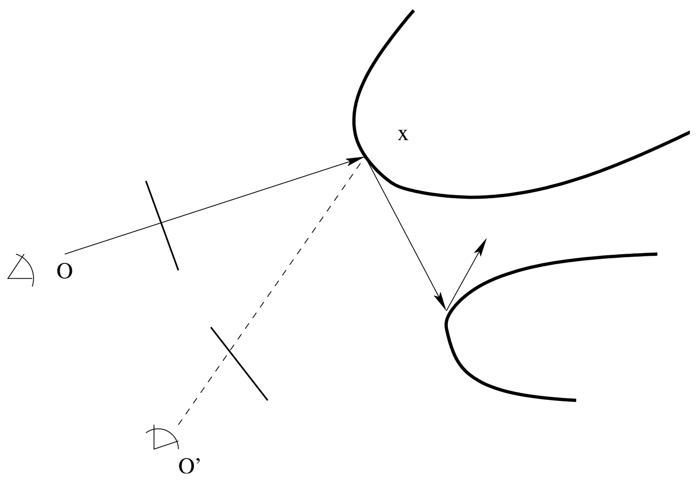


Figure 7.1: Reusing the path from observer O for observer O', at the cost of the visibility test $vis(O', x)$.

with respect to pixel j as seen from eye o'. We have then to normalize $p_i^o(x)$ with respect to the new eye. Let us drop the assumption that probability $p_i^o(x)$ depends on pixel i as seen from o and through which the ray was generated (this is an approximation, considering pixels on a spherical screen). The corresponding normalization condition should be fulfilled

$$\int_S p^o(x) V_j(x) dx = 1 \tag{7.8}$$

where $V_j(x) = 1$ if point x is visible from o' trough pixel j and 0 otherwise. Suppose now that we distribute eye rays from o with probability proportional to $G(o, x)$. To obtain probabilities $p^o(x)$ we have to find the normalization constant given by the integral:

$$I = \int_S G(o, x) V_j(x) dx \tag{7.9}$$

Integral (7.9) can be interpreted as the solid angle from o that *sees* the portion of the scene seen from the solid angle subtended by pixel j from eye o' (Ω_j), see fig. 7.2. Observe that when $o = o'$ integral (7.9) is equal to Ω_j. Integral (7.9) is not known and computing it (using Monte Carlo integration) would require sending a lot of rays from o and comparing the visible or unoccluded hit points from o' to the total unoccluded+occluded. Lacking this information, we make the assumption that we have no visibility problems. Thus, given hit x from o obtained with probabilities proportional to $G(o, x)$, and taking into account that without occlusions $\Delta\Omega_j = G(o', x)\Delta S$ and $\Delta I = G(o, x)\Delta S$, the normalization constant (7.9) is approximated by

$$I \approx \frac{\Omega_j G(o, x)}{G(o', x)} \tag{7.10}$$

Expression (7.10) is used to normalize p_i^o and estimator (7.7) (having dropped the dependence on the pixel i) thus becomes

$$\begin{aligned} \widehat{L^{o'}(j)} &= \frac{G(o',x)\rho(\omega^{in},x,x\to o')\widehat{L(x,\omega^{in})}\cos(N_x,\omega^{in})}{\frac{G(o',x)}{\Omega_j G(o,x)} G(o,x) p(\omega^{in};x,x\to o)} \\ &= \frac{\Omega_j \rho(\omega^{in},x,x\to o')\widehat{L(x,\omega^{in})}\cos(N_x,\omega^{in})}{p(\omega^{in};x,x\to o)} \end{aligned} \tag{7.11}$$

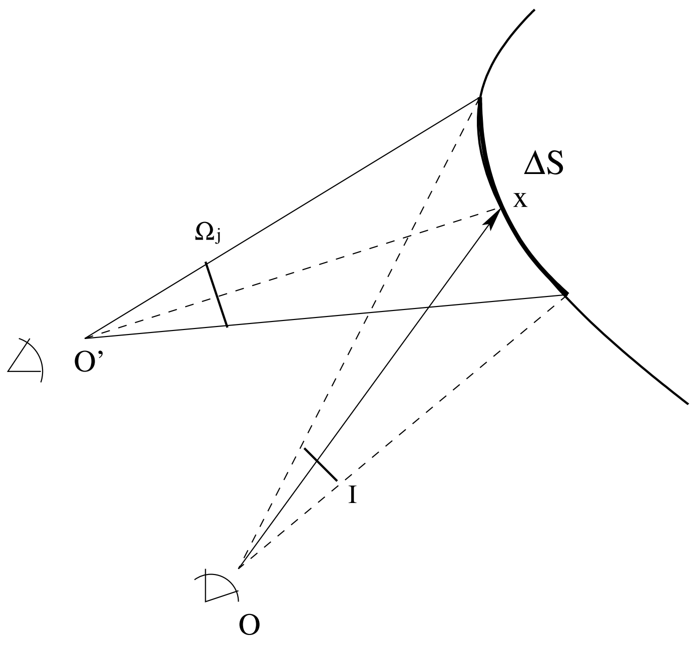


Figure 7.2: Here is shown in a graphical way the interpretation of equation (7.9). Ω_j is the solid angle subtended by pixel j, and I is the solid angle through which eye o sees what eye o' can see through Ω_j.

and for a diffuse hit point ρ (and thus neither p) depends on the incoming eye ray, thus it becomes simply

$$\widehat{L^{o'}(j)} = \Omega_j \widehat{L(x)} \tag{7.12}$$

where $\widehat{L(x)}$ is the estimated radiance from hit point x.

Combining paths

In [38] an unweighted combination of estimators of kind (7.5) and (7.7) was done, resulting in a biased estimator. We present now a strategy that gives an unbiased estimator. For each pixel and frame, we keep accumulated radiance value and native ray estimators (among many other data useful for our computation, see memory cost analysis in section 7.1.2) generated with probability $p_i^o(x)p(\omega^{in}; x, x \to o))$. When hits from neighbor frames can be reused for this pixel (suppose estimator $\widehat{L^{o,j}(i)}$, generated with probability $p_j^{o^j}(x^j)p(\omega^{in,j}; x^j, x^j \to o^j))$, we combine them using multiple importance sampling with the native estimator. This gives the new unbiased estimator:

$$\widehat{L^o(i)} = \sum_j \frac{p_j^{o^j}(x^j)p(\omega^{in,j}; x^j, x^j \to o^j)\widehat{L^{o,j}(i)}}{\sum_k p_k^{o^k}(x^j)p(\omega^{in,j}; x^j, x^j \to o^k)} \tag{7.13}$$

We show now how this estimator is applied.

For the sake of simplicity, and without loss of generality, consider only two observers O_1 and O_2[3]. In this case estimator (7.13) becomes estimator (7.14)

[3]For a clearer explanation we also drop here the albedo and solid angle Ω

for observer O_1, using the approximation (7.12) for the estimator and also (7.9) for the normalization constant in the weights. We have taken the approximation that all pixels subtend the same solid angle. Remember also that the visibility boolean function is included in the G function. Consider first the particular case for diffuse hit pixels.

$$\begin{aligned} L(O_1) &= \frac{G(O_1,x_1)}{G(O_1,x_1)+G(O_2,x_1)\frac{G(O_1,x_1)}{G(O_2,x_1)}} L(x_1) \\ &+ \frac{G(O_2,x_2)}{G(O_1,x_2)\frac{G(O_2,x_2)}{G(O_1,x_2)}+G(O_2,x_2)} L(x_2) \\ &= \frac{1}{2}L(x_1) + \frac{1}{2}L(x_2) \end{aligned} \tag{7.14}$$

Thus approximation (7.10) for the normalization constant leads to the simple unweighted estimator when we deal only with diffuse hits, and this is why Havran et al. solution [38] works well for diffuse surfaces.

For the non-diffuse general case, using again approximation (7.10) allows us to eliminate all G terms, and using estimator form (7.11) we obtain

$$\begin{aligned} L(O_1) &= \frac{\rho(\omega^{in,1};x_1,x_1\rightarrow O1)L(x_1,\omega^{in,1})\cos(N_{x_1},\omega^{in,1})}{p(\omega^{in,1};x_1,x_1\rightarrow O1)+p(\omega^{in,1};x_1,x_1\rightarrow O2)} \\ &+ \frac{\rho(\omega^{in,2};x_2,x_2\rightarrow O1)L(x_2,\omega^{in,2})\cos(N_{x_2},\omega^{in,2})}{p(\omega^{in,2};x_2,x_2\rightarrow O1)+p(\omega^{in,2};x_2,x_2\rightarrow O2)} \end{aligned} \tag{7.15}$$

where $L(x_1,\omega^{in,1})$ and $L(x_2,\omega^{in,2})$ are the incoming radiances to x_1 from direction $\omega^{in,1}$ and to x_2 from direction $\omega^{in,2}$.

7.1.2 Implementation

Algorithm

Once we hit a point in the scene from the eye, we can reuse this information for all the other frames in our camera animation, but only if the point is visible from the other eyes. For this reason and also because of memory restrictions, we are only interested in reusing the hit with the closest neighbor frames, as the probability of being visible is much higher.

We will consider two phases in the computation of our animation frames. The first one is the *hit harvest* (see algorithm 3), where we find the native hit points, compute the rest of the gathering path, connect every hit point with the other eyes to see if they can be reused, and finally, if it is the case, we add a pointer to it in the list of outer hits of the corresponding pixel. Meanwhile, additional probabilities and reflectances needed for the computation are kept in memory. The second phase is the *image computation* itself, in which we use all the stored information, including the native and outer hits for every pixel to compute the final pixel color.

As a few seconds animation involves hundreds of frames, it is not feasible to keep all them in memory at the same time. We have two possible strategies. The first one is to reuse a hit in the n previous and subsequent frames keeping in memory all information needed for the final computation, that will be done once we know we are not reusing more hits for that frame, that is, the $(actual - n)$th frame. But as we can see in equations (7.15) and (7.13), for every hit that will be used in a pixel computation, we need to use all probabilities in combination with all the eyes that have been used in the other hits for that pixel. This means keeping in memory also information for frames previous to the $(actual - n)$th, or recompute these probabilities every time we need them. This is a waste of time or a waste of memory.

```
for i = firstFrame to lastFrame do
    currentEye = getEye(i)
    for all pixel in images[i] do
        hit=traceRay(currentEye, pixel)
        for j = firstFrame to lastFrame do
            reuseHitinImage(hit,images[j])
```

Algorithm 3: The algorithm for the *hit harvest* phase, considering only the group of neighbor frames.

The second strategy consists in considering a group of $2n+1$ neighbor frames and reuse every native hit in all the other frames in the group, no matter if it is the first one, the last one or the one in the middle. When we are done for the group, instead of moving to the next $2n+1$ frames, we can move just one frame, overlapping $2n$ frames, but without deleting the previous results for the frames that are still active. This previous results can be simply averaged with the new ones. This is the strategy we follow.

Cost analysis

Now we analyze the relative cost of the animations with and without reuse. Suppose we cast n_r rays per pixel and reuse n_f frames at once. The cost of tracing an eye ray is c_e, the cost of computing the illumination at the hit point in the scene is c_l, and the cost of a visibility test is c_v. In the case of no reuse, the cost of computing n_f frames is $n_f n'_r(c_e+c_l)$. In the reusing case the cost is $n_f n_r(c_e+c_l)+n_f(n_f-1)n_r c_v$, where the second term in the sum accounts for reusing all rays. In the optimal case a ray through a pixel would be equivalent to a native ray, and we compare thus the cost of two animations with equal number of rays per pixel, that is, $n'_r \simeq n_f n_r$. The relative cost for this case is:

$$\begin{aligned} \frac{n_f n_f n_r(c_e+c_l))}{n_f n_r(c_e+c_l)+n_f(n_f-1)n_r c_v} &= \\ \frac{n_f n_r(c_e+c_l)}{n_r(c_e+c_l)+(n_f-1)n_r c_v} &= \\ \frac{n_f(c_e+c_l)}{(c_e+c_l)+(n_f-1)c_v} & \end{aligned} \tag{7.16}$$

This last expression has the limiting value $\frac{(c_e+c_l)}{c_v}$ when n_f tends to infinite. Supposing $c_l \gg c_e$, limit efficiency will be $\frac{c_l}{c_v}$.

Observe that the above limit efficiency is an upper bound, as on the one hand not all rays will be able to be reused, and on the other hand the variance associated with a reusing frames estimator is higher than with an independent one, because in the independent estimator we have the benefit of importance sampling.

A second, and very important, independent increase in efficiency comes from the reduction in flickering from frame to frame. This reduction is due to that reuse of paths for different frames correlates the computations for all them. And in the way we have constructed our algorithm there is no flickering shown when passing from a group of reused frames to the following one, as we interleave them.

Memory use

We need a huge amount of memory to keep track of all our computation. We have to keep not only all the images for the current group being computed (including native reflectances and hits), but also the lists of outer hits for every pixel and all possible combinations of probabilities and reflectances per pixel and frame.

Same as before, suppose we use n_h hits per pixel and reuse n_f frames at once. Images are made of $w \times h$ pixels, so we have a total number of pixels $n_{pix} = w \times h \times n_f$.

For every pixel we keep the final color and the number of samples to add and average every result. We also need the list of outer hits for every pixel. The total nodes of outer hits are $n_{nod} = n_h \times n_{pix} \times (n_f - 1)$ and are distributed in lists among all the pixels. Every node contains four integers: the image number, the pixel (w and h) and the number of sample, thus it can point towards the data we need to reuse from another image. For every native hit in a pixel we have to keep the native direct and indirect gathered radiance, cosinus weighted, and a n_f-vector containing all probabilities and reflectances if we combined the hit with the rest of eyes.

Just to see the numbers in a concrete example, if we are using 2 samples for every one of the 800×600 pixels and reusing groups of 7 images, we get 3360000 pixels ($800 \times 600 \times 7$), each containing a final color (3 floats), the number of samples (one integer) and a pointer to the list of nodes. We have 40320000 nodes ($2 \times 800 \times 600 \times 7 \times 6$), four integers and a pointer each. We also have the 6720000 native radiances to reuse (direct and indirect, 3 floats each) and 47040000 ($2 \times 800 \times 600 \times 7^2$) probabilities (1 float) and reflectances (3 floats). If we assume each float, integer and pointer is 4 bytes, we need a total memory of 1787520000 bytes for this structure. That's almost all the memory of our 2Gb pentium 4.

7.1.3 Results

We have applied the proposed algorithm to an animation computed using path tracing. The frame resolution is 800×600 pixels. We rendered 48 frames, for a 2 seconds animation[4]. For every pixel, we used 2 samples and reused them in groups of seven neighbor frames. As these groups are overlapped, we get a maximum average in number of samples of 98 (2×7^2). The actual average is less than that (between 85 and 90) because some reuses are lost (they can lie out of frame or be hidden by other objects) and it mainly depends on distances between different neighbor eyes, i.e. the smoothness of the camera movement. If camera movement is not smooth or the number of neighbor frames to reuse is too high this ratio decreases, and noticeable differences of noise between different parts of the same image might appear. In our example, as our camera movement corresponds to a zooming of a glossy phong brdf vase, the pixels in the center of the images get more samples than those lying near the borders. This can be interpreted as an advantage, as perception focuses in the center of the image when zooming, some kind of perception driven sampling is performed. The first frames and the last ones present more noise because they cannot be overlapped with previous (in the case of the first ones) or subsequent (for the last ones) groups of frames. Computing time was approximately 40 hours, for a PentiumIV with 2 Gigabytes of memory. That is 50 minutes per frame. A single

[4]Animations can be found in `http://ima.udg.es/~amendez/thesis/`.

path tracing image without reuse and with 98 samples per pixel takes more than 5 hours to compute. It is more than 6 times faster, and it can be even faster if we reuse more frames. If we compare this animation with the one computed with Havran et al. method [38], we can appreciate almost no differences. This is because we have reused very few frames and they are very close to each other.

We have computed a second animation with a higher number of reusing frames to prove more clearly the differences between the methods. In this case one hit is reused in 17 frames. We used 2 samples per hit, so we get a maximum average in number of samples of 578 (2×17^2). The actual average is about 500 due to loss of reuses. Due to memory restrictions, resolution is now reduced to 320×240 pixels. Computation time is about 16 hours. This is 20 minutes per frame, almost 8 times faster than the computation time of a single path tracing image with 500 samples per pixel (155 minutes). If we look at Havran et al. version of the same animation we clearly appreciate more noise in the vase in the form of glittering.

In Fig. 7.3 we show the same frame (the middle frame in our second animation) obtained with three different computations. In the first one (image a) an image with no reuse has been computed using 500 samples per pixel. It takes more than two and a half hours to compute. Image b) shows biased Havran et al. [38] version for reuse of frames. Time computation results in about 18 minutes per frame. Image c) is the result of our unbiased version. It takes 20 minutes to compute, a little more time than b), but we need much more memory. Both images b) and c) present more noise than image a) near the border due to loss of reuses. Image b) presents more noise than image c) in the vase and other glossy objects. Diffuse objects look the same in images b) and c).

In Fig. 7.4, details for the vase are shown. First image a) is computed with no reuse and 15 samples per pixel. Image b) is biased and computed with the Havran et al. [38] version and image c) is computed with our unbiased method. Both are computed with 2 samples per pixel and reuse of three fairly separated frames, i. e., a maximum average in number of samples of 18 (2×3^2). We clearly see much more noise in image b).

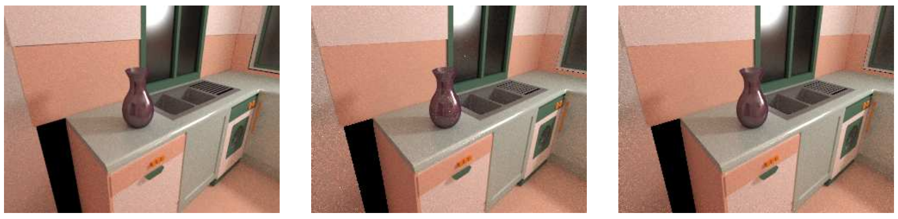

(a) (b) (c)

Figure 7.3: Here we show the same frame (the middle frame in our animation) obtained with three different computations. In the first one (image a) an image with no reuse has been computed. It takes more than 2,5 hours to be computed. Image b) shows Havran et al. [38] version for reuse of frames and takes 18 minutes to compute. Image c) is the result of our unbiased version and takes 20 minutes. The three images look very similar but there are some differences besides computation time that are clearly visible in fig. 7.4. The unbiased c) version uses much more memory. Images b) and c) presents noise near the border due to loss of reuses, and image b) presents noise in glossy objects due to biased computation.

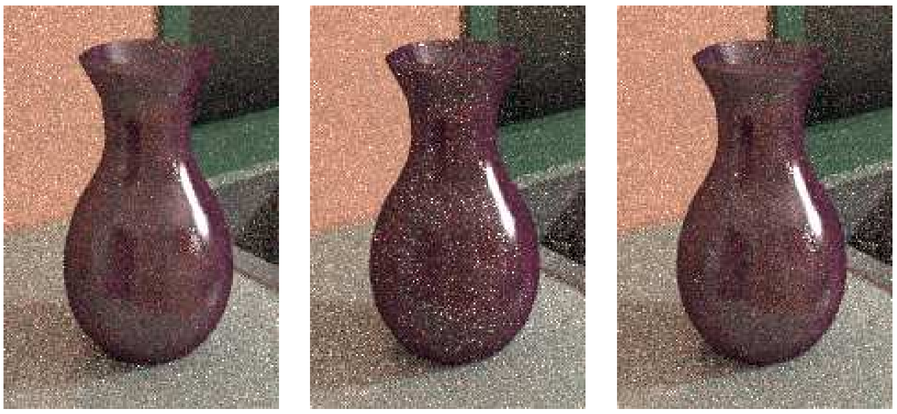

(a) (b) (c)

Figure 7.4: The differences between the methods are clearly appreciated for non-diffuse materials when we reduce the computation time (noise is higher, consequently) and the separation between frames increases. Here we see the details of the vase for the 800×600 image when reusing only 3 frames fairly separated. First image a) is computed with no reuse. Image b) is biased and computed with Havran et al. [38] version and image c) is computed with our unbiased method. We clearly see much more noise in image b).

7.2 Camera animation with obscurances

As seen in previous section, in [38], an architecture to reuse the information computed for one frame to other frames is presented. From the eye, a ray is cast through a pixel to find a hit point in the scene (named *native hit* to distinguish it from the *outer hits* found through other eyes) and a sample of the radiant flux to the eye is taken using any suitable technique as path tracing or bidirectional path tracing. If the point is from a diffuse surface, the radiance information for that point can be reused in the surrounding frames by reprojecting the point (supposing visibility) and finding the corresponding pixel, where the sampled values are simply averaged. Fig. 7.1 shows the idea. In [64] the same idea is discussed under the point of view of path reuse using multiple importance sampling, as different generation probabilities are involved for non-diffuse BRDF's with different points of view. A new algorithm is presented then to deal with non-diffuse materials, but it requires big amount of memory as all radiances, probabilities, and other data to be combined have to be saved until the final computation of the image.

Now suppose we want to compute a series of frames in a camera animation using ray-tracing with obscurances instead of a true global illumination technique. In this case the direct lighting of a point in a diffuse surface (or the diffuse part of its BRDF, as we divide the BRDF in its pure diffuse and pure specular part) and the obscurances for indirect lighting can be reused in multiple frames using Havran et al's technique. To deal with the specular part of the BRDF's we do not have the choice of reusing it as the probability of following the path starting from another eye is zero in the case of pure specular materials. In consequence, the pixels that show specular objects will need more samples or they will have less quality than the diffuse ones.

7.3 Combining light animation with obscurances

In the context of global illumination, if we have to compute a series of frames of an animation in which the camera and the objects in the scene are still and only the light sources move, all the lighting computation has to start over from frame to frame.

In the context of obscurances, we can take advantage of one of its properties, that is the decoupling between direct and indirect illumination. In the case of light animation, we only need to compute indirect illumination once and reuse it in all frames and only direct illumination is recomputed for every frame.

7.3.1 Animation of light sources

Instead of computing every frame from scratch, we can take advantage of the fact that neither the obscurances nor the hit points have to be recomputed. This is possible if the camera and all objects in the scene are fixed, and only lights move from frame to frame. Thus we compute first the obscurances and store the hit points and incoming directions of the eye rays. Then, for each frame direct illumination and specular effects are computed using shadow and specular rays using the stored hit points and directions. This has the side effect of increasing frame-to-frame coherence, eliminating flickering.

If we take, for example, images of fig. 6.1 in chapter 6, images (a) and (b) are the same for all frames. As we have seen in section 4.6, to compute indirect light in image (d), ambient light factor can be estimated independently for every frame. Thus, only direct light (c), specular effects (e), and an ambient light factor, have to be recomputed when light sources move or change intensity with respect to the next frame. We will compute ambient intensity for every frame, and use it to multiply the obscurances factor and diffuse color to obtain indirect lighting.If light sources power changes among frames or light movement causes changes in occlusion conditions, ambient intensity will also change and result in beautiful effects in the animation. In fig. 4.9 these effects can be appreciated.

7.3.2 Results

The ray-traced obscurances with light animation have been implemented in the SIR system [55]. Three types of light sources have been implemented: point, directional and area light sources, but only point and directional have been animated, because area light sources are always associated to an object of the scene, and moving objects are not treated yet in this context. Point and directional lights are animated using classic keyframes.

Three movies[5], where the three models seen in section 6.1 (figures 6.2, 6.3 and 6.4) are used to show our results. The movie corresponding to the aircraft presents a 2 seconds animation (48 frames) where the point light runs along the aircraft cabin from the camera position to the tail. Obscurances computation take 462 seconds, and 484 seconds the rest of the animation, what makes an average of 10.1 seconds per frame (19.7 seconds per frame if we take account of obscurances).

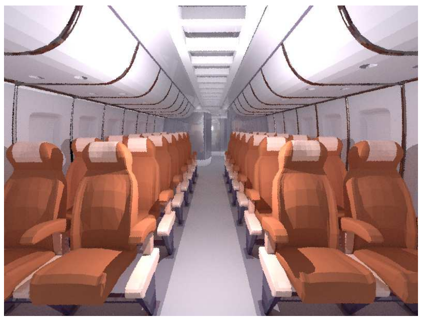
The movie of the kitchen presents a 10 secons animation (240 frames) in which directional light follows a direction from the windows given by the two polar angles of the corresponding keyframe, thus simulating the movement of

[5]The movies can be downloaded from `http://ima.udg.es/~amendez/thesis/`

the sunlight. Different ambient term intensities for each frame are clearly appreciated. On the one hand, we get a brighter image when more light gets into the room, which we take directly proportional to the cosinus of the angle between the window and the light ray. On the other hand the ambient intensity color is directly related to the colors of the objects onto which the light rays hit. Obscurances computation take 264 seconds, and 1960 seconds the rest of the animation, what makes an average of 8.16 seconds per frame (9.27 seconds per frame if we take account of obscurances).

The third movie follows the same idea than the previous one but with the stairs model. In the animation we see the directional light from the seven windows moving in a sun-like way. It takes 144 frames (6 seconds). Obscurances computation take 290 seconds, and 1490 seconds the rest of the animation, what makes an average of 10.35 seconds per frame (12.36 seconds per frame if we take account of obscurances).

In fig. 7.5 we see two frames, each one of a different movie. All images have a resolution of 800×600 pixels and have been rendered on a Pentium 4 1.6 Ghz with 1 Gb RAM. We have used in the obscurances computation 40 rays per pixel.


(a) Aircraft point light animation.

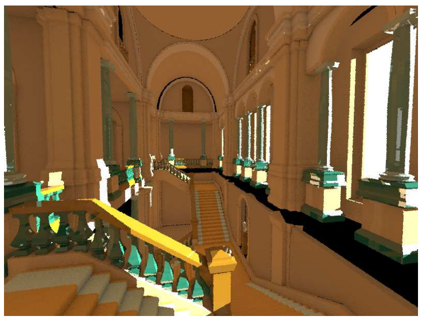

(b) Stairs directional light animation.

Figure 7.5: One frame from each animation of the aircraft and the stairs. The aircraft animation presents point light animation. The stairs movie shows the sun-like movement of directional lights.

7.4 Light and camera animation to navigate a frame array

Both strategies to gain efficiency for camera and light animation with obscurances presented in subsections 7.2 and 7.3, respectively, can be combined in a single algorithm, thus obtaining a big set of images at once that can be used to create different animations to see, for example, the animation of light from different points of view, do the walkthrough with different direct light conditions, or see both animations in a single movie. This multi-dimensional set of images will be named *frame array*.

7.4.1 Combination of camera and light animation

Until now, we have seen on the one hand how to reuse information for neighbor frames in a camera animation (section 7.2) and on the other hand how to animate lights reusing the indirect lighting from a fixed camera position (section 7.3). From now on we will combine both solutions in a unique algorithm.

The main idea is to compute simultaneously all combinations of camera and light animation, this is, for every eye position we will compute the images for all light positions. This is represented in figure 7.6. Of course this results in a high number of images but if we want to generate movies with different combinations of the light and camera animations, we clearly save time.

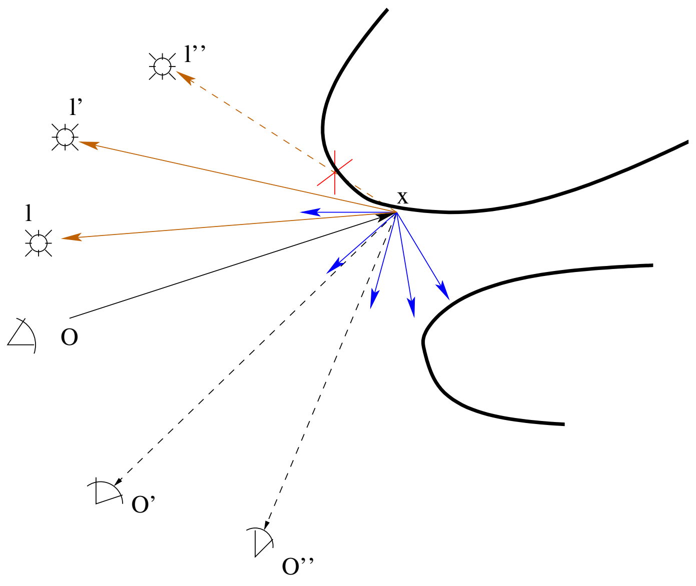


Figure 7.6: The eyes (O, O' and O") are the different positions of the animation of the camera. The suns (l, l' and l") represent the different positions of the light animation. The obscurances value for the hit point x is computed with the dark rays leaving x, and it can be reused for the different camera positions as well as for the different light positions.

We keep in memory only the images to reuse, this is, for the current eye position, the indirect light image and an array of N_l direct and specular images, being N_l the number of light positions. For every neighbor eye in which to reuse the current results, we keep the indirect light image and the array of direct light images.

The algorithm (see Alg. 4) goes as follows. As part of the initialization process we compute the average light intensity of the scene for every light position. Then, for every pixel in every frame in the camera animation we get as many hits in the scene as samples we need for pixel. Once we have a hit we compute the obscurance value for the indirect color, the direct lighting and the specular values if the BRDF of the object has specular component. These specular and direct values have to be computed for every light position. The specular values are the ones that add more computation time to our algorithm, as a new ray is sent in the reflection direction, a new hit is found and its illumination computed in a recursive way. But the direct lighting values are very fast and easy to compute, specially when we have a point light or a directional light, as we have only to compute visibility in one direction for each light position.

The specular values can not be reused, so we keep them directly as final values. We reuse the values for the different direct lighting positions and the

```
for all l in lights do
    ambient[l]=computeAmbientIntensity(l)
for i = 1 to numFrames do
    currentEye = getEye(i)
    for all pixel in images[i] do
        hit=traceRay(currentEye, pixel)
        obs=getObscurances(hit)
        indirectcolor=obs*hit.diffuse
        if isSpecular(hit) then
            specularcolor=getSpecularColor(hit)
        for all l in lights do
            directcolor[l]=getDirectLight(hit,l)
        for k = (i-$N_r$) to (i+$N_r$) do
            if i==k then
                setImageValues(images[k], directcolor, indirectcolor, specular-
                color)
            else
                reuseValuesInImage(images[k], directcolor, indirectcolor)
    computeFinalImage(i-$N_r$)
for i=(numFrames-$N_r$+1) to numFrames do
    computeFinalImage(i)
```

Algorithm 4: The algorithm for the combination of camera and light animation.

obscurances values in the N_r previous and the N_r subsequent frames, as the neighbor eyes are the ones more likely to have visibility with the hit. As they are diffuse color values, they can be directly averaged with the previous results in the corresponding image.

When we have treated all pixels in the current frame, and before we keep treating the ones of the next frame, we have one frame for which we will not reuse any more values, so we can compute its final values, i. e. multiply the indirect color image and the corresponding average intensities and add them to the respective direct and specular images of the array. Then save all images with different direct light positions to disc. At the end of the computation, all remaining images have to be finally computed and saved to disc the same way.

Cost analysis

From [64] we know the relative cost of the camera animations with and without reuse. The cost of tracing an eye ray is c_e, the cost of computing the illumination at the hit point in the scene is c_l, and the cost of a visibility test is c_v. The relative cost has the limiting value $\frac{(c_e+c_l)}{c_v}$ when n_f tends to infinite. This makes sense, as we gain the time of computing all the illumination at the hit point at only the cost of the visibility to a neighbor eye. If we separate the different contributions of direct, indirect (obscurances) and specular lighting ($c_l = c_d + c_{ind} + c_{sp}$), and we only consider pure specular lighting (as it will be the case in our examples) the relative cost has to be modified to $\frac{c_e+c_l}{c_v+c_{sp}}$ as the pure specular lighting can not be reused at all.

On the other hand, the acceleration obtained due to the reuse of indirect diffuse illumination is the number of frames this illumination is reused, n_{fr}, times the relative cost of computing indirect diffuse illumination respective to the cost of computing all illumination (this is direct+indirect+ specular), $\frac{c_{ind}}{c_l}$.

As both accelerations are orthogonal (i.e., independent), total acceleration will be the product of both relative efficiencies $\frac{c_e+c_l}{c_v+c_{sp}} \times n_{fr}\frac{c_{ind}}{c_l} \approx n_{fr} \times \frac{c_{ind}}{c_v+c_{sp}}$, when $c_e \ll c_l$.

In conclusion, we save time in both ways, on the one hand by reusing lighting information between frames of different camera positions, and on the other hand by reusing the indirect lighting computation for all positions of the light. The final acceleration obtained will heavily depend on the relative cost of computing the specular lighting, i.e. the depth of recursion and relative number of pixels that show specular objects.

Application

At the end of our computation we get as result a set of images combining each camera position with each light position. This is very useful when the animator wants to see the same camera animation under different light conditions, check the same animation of the light from different points of view or combine both light and camera animations in the same or different movies. Figure 7.7 shows clearly this application. Note that, though the directions and senses of the grey arrows in the figure are the most common cases, the only restriction for navigating the frames is to move only one step at a time in any direction, so a movie that follows the path of the dotted blue arrow, for example, is perfectly possible.

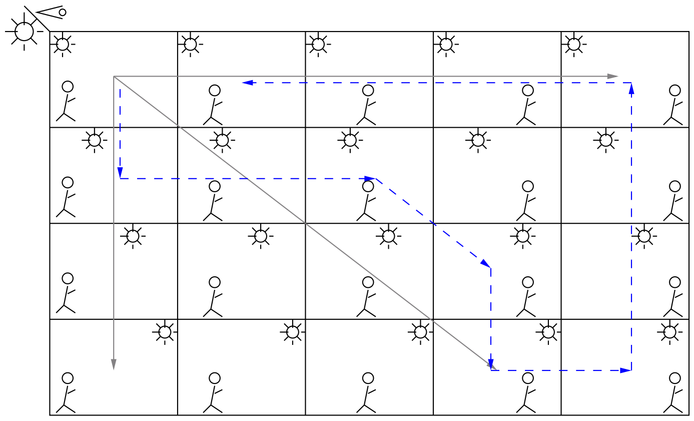


Figure 7.7: The algorithm results in a matrix of images that we can navigate. The rows are images with changes in camera animation and the columns show the images with changes in light animation. We can generate movies with camera animation for different light positions (horizontal arrow), see the animation of light from different points of view (vertical arrow), or both animations simultaneously (diagonal). We have also the freedom to move along any direction as, for example, following the dotted blue arrow.

7.4.2 Results

To show the validity of our method we have computed three animations[6]. All of them are computed in a Pentium IV PC with 1.8 Gh and 2 Gb of RAM. Two

[6]The animations can be found at `http://ima.udg.es/~amendez/thesis/`

native rays per pixel are cast to find hits and five obscurance rays are cast per hit. One ray per hit is cast to find direct light. If the hit object is specular (or has a partly specular BRDF) a path is followed to find the reflected illumination in a recursive way. The diffuse (direct and indirect) illumination per hit is reused in the three previous and three subsequent frames for the camera animation, what makes a total of seven frames to reuse, this is, a maximum theoretical number of samples (if the hit is seen by all eyes to reuse) of 70 obscurances samples and 14 direct light samples (per each light position) per pixel. The resolution for all images is 400×300 pixels.

The first movie shows the compound animation of a camera and a point light in the cabin of an aircraft. The camera animation by itself would take eight seconds (this is, 196 frames) and the light moves for 2 seconds (48 frames). The final movie takes 10 seconds and we can appreciate that in the first 2 seconds the camera is fixed and the light is moving. The next 2 seconds the camera moves and the light stays still. The last 6 seconds both camera and light are animated. Of course in this case we do not use all 9216 (196×48) images generated , but it is worth having the freedom to generate any other movie involving these two animations. Figure 7.8 shows part of the possibilities we have. They took almost 18 hours to compute, an image every 7 seconds. A single image of the aircraft model without any reuse and only one light position, using 70 rays per pixel for obscurances computation and 14 rays per pixel for direct light computation, takes 260 seconds, more or less, depending on the camera position and the number of pixels that show specular objects.

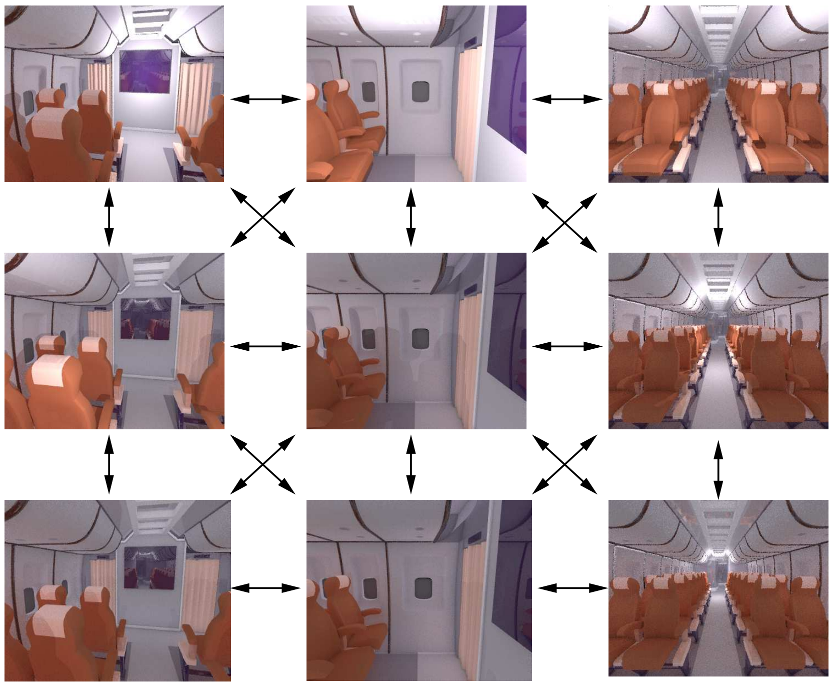


Figure 7.8: Here we see a few frames of the animation of the aircraft model. The arrows show a few of the multiple possibilities we have to generate different movies with our set of images.

The second movie shows the animations of the camera moving through a kitchen (12 seconds, 288 frames) combined with an animation of a directional light coming from the windows and changing its angle for two seconds (48 frames). Same as before, it shows only a part of the possible combinations for all 13824 images generated (288×48, that took almost 32 hours to compute, an image every 8.2 seconds). Note the changes in the average ambient color and intensity. A single image of the kitchen model without any reuse and only one light position, using 70 rays per pixel for the obscurances computation and 14 rays per pixel for direct light computation, takes around 320 seconds.

The third movie is an animation of a camera traveling over the stairs of a theater (6 seconds, 144 frames) combined with an animation (2 seconds, 48 frames) of the angle of a directional light from the seven windows. It also shows the changes in ambient intensity. All 6912 images (144×48) took a bit more than 14.5 hours, 7.6 seconds per image. A single image of the stairs model without any reuse and only one light position, using 70 rays per pixel for the obscurances computation and 14 rays per pixel for direct light computation, takes around 230 seconds.

7.5 Summary

In this chapter we have presented efficient techniques to reuse illumination between frames.

First we have presented an efficient unbiased method to combine frames in camera animation. It consists in reusing the incoming radiance information of a hit point for the neighbor frames of the animation. The different probabilities are taken into account and multiple importance sampling technique is used to correctly combine the different samples. Our method makes the difference when using non-diffuse materials, because the diffuse ones distribute reflected rays with equal probabilities in all directions, and when the separation between reusing frames increases. In diffuse cases or when reusing frames are very close, other methods, though biased, can work fine. The new method has been demonstrated with an animation of a camera in a scene that contains a vase with a glossy brdf, computing the global illumination using the path-tracing technique. The main drawback of our method is the large amount of memory needed for the computation.

The previous technique can be simplified when using obscurances. As they are computed for diffuse surfaces, the illumination is equally distributed among the hemisphere and equal probabilities can be used, thus no multiple importance sampling is needed. This is presented in another section.

Then we have presented the combination of the obscurances technique with moving light sources. Since direct and indirect illumination are decoupled, animation of source lights can be added with no extra computation time for obscurances. Direct illumination is computed for every frame but we save computation time as the hard part of the indirect illumination is computed only once. For indirect illumination, only the ambient light term is recomputed for every frame by shooting a few rays from the light sources, and follow their paths while computing stochastically the parameters we need. The technique presented here can be used as a fast editing tool or as the final image. Light animation can help in light design and to convey shape and features of the objects in the scene.

Finally we have seen an algorithm that combines the two previous introduced transversal techniques that reuse information between frames in an animation. On the one hand, the camera animation reuses the diffuse illumination of a hit between neighbor frames by reprojection, and on the other hand, using the obscurances technique to simulate global illumination direct and indirect illumination can be decoupled, and the animation of light can be computed reusing the indirect lighting information. Besides the advantage of joining both techniques, our algorithm is capable to generate images for all combinations of both animations, giving freedom to generate different movies from the same set of images, a multi-dimensional array of images that we dub *frame array*. Considering the time of computation per image, the acceleration obtained with our algorithm is lineal with the number of light positions, with a multiplicative constant which depends on the relative cost of indirect and specular lighting. In our experiments with several scenes and 48 light positions the speed-up was between 35 and 40 relative to computing every frame from scratch.

When computing images for production rendering, lots of high quality images are needed and the cost of computing only one of these images is very high. These technique presented here help saving time by reusing information between frames.

Chapter 8

Conclusions and future work

We have presented in this dissertation some improvements and applications that take advantage of the good properties of the obscurances, a technique that mimics and simplifies global illumination, and that presents good realistic results, though it is not physically accurate. Also, for the computation of frames of an animation, some techniques to reuse information and save computation time have been introduced.

In this chapter we summarize the contributions of this dissertation in section 8.1, then in section 8.2 we present the list of publications that support this work, and finally the lines for future work are described in section 8.3.

8.1 Contributions

8.1.1 Survey

In chapter 3 we have presented a survey on obscurances and similar techniques as ambient occlusion. Ambient occlusion has become a widely known technique to simulate realistic effects in illumination, specially in recent animated movies.

8.1.2 General improvements of obscurances

In chapter 4 some general improvements for obscurances have been presented. The main contributions in this chapter are:

- **Color bleeding**: We modified slightly the obscurances equations to deal with the exchange of coloration between surfaces with no added cost.
- **Obscurances function** ρ: We have tested different ρ functions concluding that the square root one ($\rho(d) = \sqrt{\frac{d}{d_{max}}}$) is the one that fits best to our purposes. The different **maximum distances** d_{max} have been tested and we get the best results when d_{max} is between one third and half the size of the scene.
- **Hemisphere sampling**: We have done a study on different techniques to choose the hemisphere samples and we concluded that the most efficient one is to choose numbers from Halton series with a random offset to avoid correlation.

- **Important secondary reflectors**: A problem that the obscurances technique presents has been addressed. If there are important secondary reflectors, as when the light source points directly to a wall, the obscurances fail to compute a similar illumination as global illumination algorithms like radiosity. Our solution consists in expanding direct illumination, though it increases computational cost.

- **New ambient intensity**: A new technique to find average ambient intensity and average reflectivity of the scene is tested. This technique give good results when computing frames for animation of light sources.

8.1.3 Improvements and new techniques for obscurances in real-time environments

In chapter 5 two new algorithms for obscurances in videogame environments are presented:

- **Moving objects**: Thanks to the limited reach to a maximum distance of the obscurances computation, the obscurances of the patches of a moving object and its surroundings can be updated at interactive rates.

- **Depth peeling with GPU**: The precomputation of the obscurances for objects and scenes is accelerated using a GPU approach of the depth peeling algorithm.

8.1.4 Generalization of obscurances to non-diffuse environments

In chapter 6 the obscurances are introduced in general environments (with objects that have materials with diffuse and non-diffuse properties), first embedded in a ray-tracing-like algorithm and later modifying its computation to deal with non-diffuse BRDFs and BTDFs:

- **Obscurances in ray-tracing**: The obscurances are plugged in a ray-tracing algorithm. The classic ambient term to account for indirect illumination is substituted by the obscurances computation at every hit point of the scene. The results are compared to stochastic path tracing, a global illumination algorithm, and we present similar results at one tenth of the computation time.

- **Obscurances for general environments**: A study on the obscurances algorithm is performed to deal with non diffuse environments. New algorithms are proposed for the basic types of BRDFs and BTDFs (specular, transparent and translucent) and a combination of the basic materials is tested. Also, a measure of the illumination coming from a diffuse object based on its thickness is introduced, and a test for the special case of trees is done.

8.1.5 Reuse of information between frames

In chapter 7 our research is addressed to the field of path reuse for animations:

- **Reuse of hits in camera animation**: We demonstrate that a frame in a walkthrough (camera animation) can reuse its illumination of the scene in the neighbor frames, even for view-dependent BRDFs. The hits obtained from one frame are reprojected to neighbor frames and, if the other eyes can see the point, its illumination can be reused in an unbiased way, by weighting correctly the different generation probabilities using multiple importance sampling.

- **Reuse of indirect light in light source animation**: When the camera is still and the light sources move, the indirect illumination can be reused in all frames using obscurances, as they allow decoupling between direct and indirect illumination. The average intensity and average reflectivity of the scene are computed for every frame, leading to a beautiful effect of color-changing between frames.

- **Frame array: combining light and camera animation**: The two previous results can be combined in a single algorithm. All combinations of light and camera animations are computed at the same time at a small part of the cost of computing all single images separately. Thus we create the concept of *frame array*, a set of images that the user can navigate to test a light animation from different camera positions, see the same walkthrough with different lightings, or combine both animations in a single movie.

8.2 Publications

The core of the research for the thesis is developed in two journal articles, four papers of proceedings, one book chapter, one technical report and one poster. In addition, one article is currently submitted to a journal. Following, we present their references and the contribution that each one has done to this dissertation.

- Àlex Méndez Feliu, Mateu Sbert, Jordi Catà
  Real-Time Obscurances with Color Bleeding
  Proceedings of 18th Spring Conference on Computer Graphics SCCG 2003
  Bratislava, Slovakia, April 2003.
  Pp. 171–176, ACM Press, New York, NY, USA, 2003
  Best presentation award

  - Obscurances are extended with color bleeding
  - Important secondary reflector problem is introduced
  - Interactive frame-rates are accomplished for the update of the obscurances for moving objects

- Àlex Méndez Feliu, Mateu Sbert, László Neumann
  Obscurances for Ray-Tracing
  EUROGRAPHICS 2003 Poster Presentation
  Granada, Spain, September 2003.

  - Obscurances in ray-tracing environments are presented

- Àlex Méndez Feliu, Mateu Sbert, László Neumann
  Obscurances for Ray-Tracing (extended version)
  Research report IIiA 03-09-RR

Institut d'Informàtica i Aplicacions
Universitat de Girona, Desembre 2003.

- The poster is extended with a more detailed explanation and the studies on function ρ and the maximum distance d_{max}.

- Àlex Méndez Feliu, Mateu Sbert
  Comparing Hemisphere Sampling Techniques for Obscurance Computation
  Proceedings of the International Conference on Computer Graphics and Artificial Intelligence 3IA 2004
  Limoges, France, May 2004.

  - Four sampling methods of the hemisphere for the obscurances computation are compared.

- Àlex Méndez Feliu, Mateu Sbert
  Combining Light Animation with Obscurances for Glossy Environments
  Computer Animation and Virtual Worlds 2004; 15(3-4): 463–470
  John Wiley & Sons, July 2004.
  The Very Best Papers of CASA 2004, Geneva, Switzerland

  - Thanks to the decoupling of direct and indirect illumination, the computation of a series of frames for moving light sources is accelerated.

- Àlex Méndez Feliu, Mateu Sbert, Jordi Catà , Nicolau Sunyer, Sergi Funtané
  Real-Time Obscurances with Color Bleeding (GPU Obscurances with Depth Peeling)
  ShaderX 4, Chapter 2.6
  Charles River Media, January 2006.

  - A new technique for computing obscurances with depth-peeling and GPU is presented.

- Àlex Méndez Feliu, Mateu Sbert, László Szirmay-Kalos
  Reusing Frames in Camera Animation
  Journal of Winter School of Computer Graphics, ISSN 1231-6972, Vol. 14, 2006
  Plzn, Czech Republic, January 2006.

  - A solution to reuse hit radiance information between frames in an unbiased way is presented.
  - It is shown that for diffuse materials and for very close neighbor frames, the biased solution is valid.

- Àlex Méndez Feliu, Mateu Sbert
  Obscurances in General Environments
  Proceedings of Graphicon 2006
  Novosibirsk, Russia, July 2006.

  - A study for the modification of the obscurances algorithm in non-diffuse environments is done.

- Àlex Méndez Feliu, Mateu Sbert
  Efficient Rendering of Light and Camera Animation for Navigating a Frame Array
  Proceedings of Computer Animation and Social Agents CASA 2006
  Geneve, Switzerland, July 2006
  - The combination of two previous techniques to reuse information between frames is efficiently done.
  - The *frame array* concept is introduced.
- Àlex Méndez Feliu, Mateu Sbert
  From obsurances to ambient occlusion: A survey
  Submitted to Visual Computer
  - It is the survey corresponding to chapter 3.

8.3 Future work

Many paths are open for future directions of research. In this section we present them.

8.3.1 Sampling

Adaptive sampling

As the error in computing the obscurances is unevenly distributed (see fig. 4.8), depending on the relative positions of the objects of the scene, we plan to use an adaptive sampling strategy, this is, using more samples where they are more needed, according to some oracle function (variance [69], f-divergences [71]). In this strategy, batches of samples are progressively cast and the homogeneity of the obtained obscurances is examined. An heuristic is then devised to continue or stop sampling.

Sparse sampling

The obscurances have usually a low variation along the surfaces. Christensen in [21] takes advantage of this and performs a sparse computation of ambient occlusion. This technique can be combined with adaptive sampling for the pieces of surface with a higher variation of the values.

8.3.2 Ambient term

We could combine the obscurances with other ambient term methods, studying the use of average ambient intensity, possibly using different terms depending on the position and/or orientation of the point.

Orientation-depending ambient term (Extended Ambient Term)

The Extended Ambient Term (EAT) by Castro et al. is presented in [19]. Obscurances can be easily combined with EAT, using this value in substitution for the average ambient intensity term (I_A). The different I_A can be used depending on the orientation of the patch or the normal at the hit point for which we compute the illumination.

Position-depending ambient term

The different ambient terms can also be computed for every element of a 3D-structure of the scene, i. e. for every voxel of a voxelization or for every leave in a kd-tree. This improvement can be used for the animation of objects in the scene. In addition to the real-time change of the obscurances computation presented in section 5.1, the ambient term for the object also changes, emphasizing the color bleeding effects and darker lighting for the zones more hidden to direct lighting. This approach would take more into account the actual lighting of the scene.

Time-depending ambient term

In section 4.6 we have explained how we can use time-depending ambient term to change ambient intensity and color for every frame in an animation with moving light sources. The algorithm presented works, but it requires some research for improvement.

We have two problems with this algorithm, on the one hand the flickering noise between frames and, on the other hand, the difficulty to compute the actual area of the scene where the light arrives. We solve those problems in a roughly way. The noise is solved directly by taking more samples, but some kind of noise filtering between the values would lead to better results. The problem of the area reached by the rays is solved using a rule of thumb, a multiplicative term is put to obtain a natural observed result (not too dark and not too bright, or the most similar to accurate global illumination methods). More research has to be done in this last issue to find an automatic way to obtain the area reached by the light.

8.3.3 Complex models

Plants and trees

For complex models with a huge number of polygons or a complex structure as trees and plants, a simpler model for obscurances could be devised. For example, Hegeman et al. [39] compute ambient occlusion for trees simplifying the model to one or several ellipsoids and using a specific adapted equation for the occlusion computation.

Hair, skin and cloth

Human hair or animal fur, besides having complex structures and posing a similar problem to plants and trees, present anisotropic material properties.

Human skin is complex also, but in another level. Its surface is irregular, presents imperfections, and it has a certain level of translucency. In [51], Krishnaswamy et al. present a model that mimics the actual composition of the skin by programming several layers of skin interaction with the light.

Cloth has, besides the complexities of previous seen models, an added level of complexity, as several types of cloth can be manufactured and different computational models can be constructed for each one. The different textiles to make cloth present different challenges to rendering. For example, silk and some synthetic textiles as nylon are anisotropic, fur and wool present similar problems as hair, etc.

In this way, the adaptation of the obscurances computation model should be devised for each one of these complex models, and in the case of cloth, a previous classification of the models is necessary.

Participative media

Participative media as clouds, fog and smoke present added problems to rendering. Their special characteristics of not having a limited boundary and their particular distribution of the illumination of a scene make them difficult to illuminate and render.

If the participative media is evenly distributed among the space in front of the camera, and the illumination is mainly diffuse (it is the case of fog and submarine rendering), the simple trick is to blur the object visualization as long as it gets away from the camera. But for clouds and smoke, and when intense light sources are in the scene, rendering these *objects* is a difficult problem.

We can study the different problems that participative media presents for obscurances and adapt their computation to their particular problems.

Other BDFs

The images computed with obscurances in non-diffuse environments can be improved by applying other lighting effects not yet considered.

Translucency of objects is a much more complex concept than as programmed in section 6.2.3. For a more accurate computation of translucency, the different scattering models have to be taken into account to know how much light gets into an object, scatters, and then gets out and in which directions to influence the obscurances around. The ideas to solve this can be similar to the ones for participative media, but taking into account that in these cases we can not *get into* the object.

There are some luminescence effects as phosphorescence (the light energy is not immediately bounced off the surface material, but kept and released some time later, i. e. the BDF is also a function of time) or fluorescence (where the wavelength of the outgoing lighting is longer than the ingoing, i. e. it changes color, specially when ultraviolet lighting is involved and the object seems to increase its lighting intensity), that are not quite explored in rendering and obscurances could be adapted to them. Other effects as caustics can also be explored.

8.3.4 GPU

Most of the recent work in the graphics community take advantage of the great possibilities that modern GPU cards present. Obscurances and ambient occlusion are no exception. See chapter 3 for several examples.

In particular, the animation of objects in real-time scenarios seen in section 5.1, that in this dissertation is presented as only *interactive* (i. e. between five and ten frames per second can be achieved, but not twenty-four, as needed for real-time) could easily be *real-time* with the effort to apply vertex and fragment shader programming.

8.3.5 Reuse of information

Reuse of frames: camera

Future work will be addressed to increase the efficiency of our approach using coherence in visibility computation, by guessing on the one hand the visibility for one observer from the results for neighbor observers and on the other hand by using an acceleration schema similar to [38]. Combination with an adaptive sampling technique, i.e., using more native samples for those pixels that come with not enough outer hit samples, because they are occluded by other objects or because they lie near the bounder of the image. The algorithm could easily be adapted to deal with stereo images.

Different concepts to keep the obscurance values in the scene can be explored, as irradiance gradients by Ward and Heckbert [94] or a kd-tree structure as in photon mapping [42].

8.3.6 Frame Array

Further research for the frame array structure for reusing information for multi-dimensional independent animations of the elements of a scene will be explored.

The main problem to solve is the huge amount of memory used to keep all the information needed to compute the images. The number of simultaneous images to compute is limited due to memory restrictions, and a few secons of animation require hundreds of images to compute for each dimension. In this dissertation (see section 7.4) 2-dimensional frame arrays have been programmed. Adding a third dimension (as a second light source, or a moving object) overloads memory capacity.

In this way, an optimization of the memory has to be done. One possibility is to optimize the algorithm to save images to disk as soon as they are computed, with the objective to keep the minimum amount of images in memory at the same time. A second possibility is to use some kind of compression for the images.

Once solved the memory problem, future research for higher dimensional frame arrays can be addressed. The camera walkthrough is always one of these dimensions. The second dimension will be a moving light source. Further dimensions, as a second animated light source, could be addressed. It does not make sense to use a second dimension for a second path of the camera animation, as two camera paths can be joined and considered as only one path, i. e. the camera positions are always considere as one, and only one, dimension. Another possibility would be to consider a rigid moving object as the third dimension, and this would bring new challenges of research.

References

[1] Ambient occlusion plugin for cinema4d. http://www.keindesign.de/stefan/cinema/ao.html. Active by March 2007.

[2] Crystal space 3d. http://www.crystalspace3d.org/. Active by March 2007.

[3] Mental ray - ambient occlusion shader. http://www.christopher-thomas.net/pages/free_tutorials/tut_ambient_occl%usion_shader/ct_tut_mentalray_ambientocclusion_shader.htm. Active by March 2007.

[4] Qutemol web page. http://qutemol.sourceforge.net/. Active by March 2007.

[5] Shadevis page. http://vcg.sourceforge.net/tiki-index.php?page=ShadeVis. Active by March 2007.

[6] V-ray 1.5 technology preview. http://www.vray.info/features/vray1.5_preview/#dirt. Active by March 2007.

[7] Wikipedia: Ambient occlusion. http://en.wikipedia.org/wiki/Ambient_occlusion.

[8] Ambient occlusion shader - mental ray for maya. http://www.deathfall.com/article.php?sid=2703, july 2003.

[9] 3ds max 7: Mentalray: Ambient occlusion: Using ambient occlusion to enhance render details. http://www.everflow.com/support/tutorials/AO/Everflow_AmbientOcclusion1%.html, May 2005.

[10] Vray ambient occlusion 2.0. http://plugins.angstraum.at/vrayao/index.htm, Jul 2006.

[11] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the Spring Joint Computer Conference*, pages 37–45, 1968.

[12] Philipe Bekaert, Mateu Sbert, and John H. Halton. Accelerating path tracing by re-using paths. *Proceedings of Workshop on Rendering*, pages 125–134, 2002.

[13] Philippe Bekaert. *Hierarchical and Stochastic Algorithms for Radiosity.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, 1999.

[14] Gonzalo Besuievsky and Xavier Pueyo. A dynamic light sources algorithm for radiosity environments. *Computer Animation and Simulation '98*, pages 13–28, 1998.

[15] Gonzalo Besuievsky and Xavier Pueyo. Animating radiosity environments through the multi-frame lighting method. *Visualization and Computer Animation*, pages 93–106, 2001.

[16] Gonzalo Besuievsky and Mateu Sbert. The multi-frame lighting method: A monte carlo based solution for radiosity in dynamic environments. *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)*, pages 185–194, 1996.

[17] Rob Bredow. Renderman on film, july 2002. Course 16: RenderMan in Production. ACM SIGGRAPH 2002 Course Notes.

[18] Michael Bunnel. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter Dynamic Ambient Occlusion and Indirect Lighting, pages 223–233. Addison-Wesley Professional, 2005.

[19] Francesc Castro, Lászlò Neumann, and Mateu Sbert. Extended ambient term. *Journal of Graphics Tools*, 5(4):1–7, 2000.

[20] Per Christensen. Ray tracing for the movie cars. `http://www.sci.utah.edu/~wald/RT06/papers/raytracing06per.pdf`. Symposium on Interactive Ray Tracing 2006, Salt Lake City, Utah, USA.

[21] Per Christensen. Global illumination and all that, july 2003. Course 9: RenderMan, Theory and Practice . ACM SIGGRAPH 2003 Course Notes.

[22] W.G. Cochran. *Sampling Techniques.* John Wiley and sons, New York, 1977.

[23] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis.* Academic Press Professional, Boston, MA, 1993.

[24] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. In *Computer Graphics (ACM SIGGRAPH '84 Proceedings)*, volume 18, pages 137–145, July 1984.

[25] L. M. Cruz-Orive. On the precision of systematic sampling: a review of matheron's transitive methods. *Journal of Microscopy*, 153(3):315–333, March 1989.

[26] L. M. Cruz-Orive. Systematic sampling in stereology. In *Bull. Intern. Statis. Instit. Proceedings 49th Session*, 55, pages 451–468, Florence, 1993.

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

[28] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 2001.

[29] Dustin Franklin. *ShaderX 4*, chapter Chapter 2.3: Hardware-Based Ambient Occlusion, pages 91–100. Charles River Media, 2006.

[30] Ismael García, Mateu Sbert, and László Szirmay-Kalos. Tree rendering with billboard clouds. In *In Proceedings of Third Hungarian Conference on Computer Graphics and Geometry*, pages 9–15, 2005.

[31] Andrew S. Glassner. *Introduction to Ray Tracing.* Academic Press, New York, NY, 1989.

[32] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the Interaction of Light Between Diffuse Surfaces. In *Computer Graphics (ACM SIGGRAPH '84 Proceedings)*, volume 18, pages 212–222, July 1984.

[33] X. Gual-Arnau and L. M. Cruz-Orive. Systematic sampling on the circle and on the sphere. *Advances in Applied Probability (SGSA)*, (32):628–647, 2000.

[34] Toshiya Hachisuka. *GPU Gems 2*, chapter High-Quality Global Illumination Rendering Using Rasterization. Addison-Wesley Professional, 2005.

[35] John H. Halton. Sequential monte carlo techniques for the solution of linear systems. *Journal of Scientific Computing*, 9:213–257, 1994.

[36] J. Hammersley and D. Handscomb. *Monte Carlo Methods.* Chapman and Hall, London, 1979.

[37] Vlastimil Havran, Jiri Bittner, and Hans-Peter Seidel. Exploiting temporal coherence in ray casted walkthroughs. In *Proceedings of the Spring Conference on Computer Graphics 2003 (SCCG 2003)*, April 2003.

[38] Vlastimil Havran, Cyrille Damez, Karol Myszkowski, and Hans-Peter Seidel. An efficient spatio-temporal architecture for animation rendering. In *Proceedings of Eurographics Symposium on Rendering 2003.* ACM SIGGRAPH, June 2003.

[39] Kyle Hegeman, Simon Premože, Michael Ashikhmin, and George Drettakis. Approximate ambient occlusion for trees. In *Proceedings of ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, Redwood City, CA, USA, March 2006.

[40] Andrey Iones, Anton Krupkin, Mateu Sbert, and Sergey Zhukov. Fast, realistic lighting for video games. *IEEE Computer Graphics and Applications*, 23(3):54–64, May/June 2003.

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

[42] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping.* A. K. Peters, Natick, MA, 2001.

[43] James T. Kajiya. The Rendering Equation. In *Computer Graphics (ACM SIGGRAPH '86 Proceedings)*, volume 20, pages 143–150, August 1986.

[44] M. H. Kalos and P. A. Whitlock. *Monte Carlo Methods.* John Wiley & Sons, 1986.

[45] Adam G. Kirk and Okan Arikan. Real-time ambient occlusion for dynamic character skins. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, April 2007.

[46] T. Kollig and A. Keller. Efficient multidimensional sampling. *Computer Graphics Forum*, 21(3):557–563, 2002.

[47] Janne Kontkanen and Timo Aila. Ambient occlusion for animated characters. In Thomas Akenine-Möller Wolfgang Heidrich, editor, *Rendering Techniques 2006 (Eurographics Symposium on Rendering)*. Eurographics, jun 2006.

[48] Janne Kontkanen and Samuli Laine. Ambient occlusion fields. In *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, Washington, District of Columbia, USA, April 2005.

[49] Janne Kontkanen and Samuli Laine. *ShaderX 4*, chapter 2.4: Ambient Occlusion Fields, pages 101–108. Charles River Media, 2006.

[50] P. R. Krishnaiah and C. R. Rao. *Handbook of statistics*, volume 6, sampling. Elsevier, Amsterdam, 1988.

[51] A. Krishnaswamy and G.V.G. Baranoski. A biophysically-based spectral model of light interaction with human skin. *Computer Graphics Forum*, 23(3):331–340, 2004.

[52] Eric P. Lafortune and Yves D. Willems. Bi-directional Path Tracing. In H. P. Santo, editor, *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, pages 145–153, Alvor, Portugal, December 1993.

[53] Hayden Landis. Production-ready global illumination, july 2002. Course 16: RenderMan in Production. ACM SIGGRAPH 2002 Course Notes.

[54] Mattias Malmer, Fredrik Malmer, Ulf Assarson, and Nicolas Holzschuch. Fast precomputed ambient occlusion for proximity shadows. *Journal of Graphics Tools*, 2006.

[55] Ignacio Martín, Frederic Pérez, and Xavier Pueyo. The sir rendering architecture. *Computers & Graphics*, 22(5):601–609, 1998.

[56] Àlex Méndez-Feliu and Mateu Sbert. Combining light animation with obscurances for glossy environments. *Computer Animation and Virtual worlds*, 15(3-4):463–470, July 2004.

[57] Àlex Méndez-Feliu and Mateu Sbert. Comparing hemisphere sampling techniques for obscurance computation. In *Proceedings of the International Conference on Computer Graphics and Artificial Intelligence (3IA 2004)*, Limoges, France, May 2004.

[58] Àlex Méndez-Feliu and Mateu Sbert. Efficient rendering of light and camera animation for navigating a frame array. In *Proceedings of Computer Animation and Social Agents (CASA 2006)*, Geneve, Switzerland, July 2006.

[59] Àlex Méndez-Feliu and Mateu Sbert. Obscurances in general environments. In *Proceedings of Graphicon 2006*, Novosibirsk, Russia, July 2006.

[60] Àlex Méndez-Feliu, Mateu Sbert, and Jordi Cata. Real-time obscurances with color bleeding. In *Proceedings of Spring Conference on Computer Graphics (SCCG 2003)*, Bratislava, Slovakia, April 2003.

[61] Àlex Méndez-Feliu, Mateu Sbert, Jordi Catà, Nicolau Sunyer, and Sergi Funtané. *ShaderX 4*, chapter Chapter 2.3: Real-Time Obscurances with Color Bleeding (GPU Obscurances with Depth Peeling), pages 121–133. Charles River Media, 2006.

[62] Àlex Méndez-Feliu, Mateu Sbert, and László Neumann. Obscurances for ray-tracing. In *EUROGRAPHICS 2003 Poster Presentation*, Granada, Spain, September 2003. Poster.

[63] Àlex Méndez-Feliu, Mateu Sbert, and László Neumann. Obscurances for ray-tracing (extended version). Technical Report IIiA 03-09-RR, Institut d'Informatica i Aplicacions, Universitat de Girona, Girona, Spain, December 2003.

[64] Àlex Méndez-Feliu, Mateu Sbert, and László Szirmay-Kalos. Reusing frames in camera animation. *Journal of Winter School of Computer Graphics*, 14(1–3):97–104, January 2006. ISSN 1213-6972.

[65] Gavin Miller. Efficient algorithms for local and global accessibility shading. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques SIGGRAPH'94*, pages 319–326, July 1994.

[66] I. Neulander. Image-based diffuse lighting using visibility maps. SIGGRAPH 2003 Technical Sketch, July 2003.

[67] Harald Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[68] Matt Pharr and Simon Green. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, chapter Dynamic Ambient Occlusion and Indirect Lighting, pages 279–292. Addison-Wesley Professional, 2004.

[69] W. Purgathofer. A statistical method for adaptive stochastic sampling. In *Proceedings of Eurographics '86*, volume 11, pages 157–162, Lisbon, Portugal, August 1986.

[70] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques SIGGRAPH'85*, pages 313–322, July 1985.

[71] Jaume Rigau, Miquel Feixas, and Mateu Sbert. Refinement criteria based on f-divergences. In *Proceedings of the Eurographics Symposium on Rendering*, pages 260–269, Leuven, Belgium, June 2003.

[72] Barbara Robertson. Shades of davy jones. `http://features.cgsociety.org/story_custom.php?story_id=3889`. Active by March 2007.

[73] R.Y. Rubinstein. *Simulation and the Monte Carlo Method.* Wiley Series in Probabilities and Mathematical Statistics, 1981.

[74] Mirko Sattler, Ralf Sarlette, Gabriel Zachmann, and Reinhard Klein. Hardware-accelerated ambient occlusion computation. In *Vision, Modeling, and Visualization 2004*, pages 331–338, November 2004.

[75] Mateu Sbert. *The Use of Global Random Directions to Compute Radiosity: Global Monte Carlo Techniques.* PhD thesis, Universitat Politecnica de Catalunya, Barcelona, Spain, 1997. Available from `http://ima.udg.es/~mateu`.

[76] Mateu Sbert, Philippe Bekaert, and John Halton. Reusing paths in radiosity and global illumination. *Monte Carlo Methods and Applications*, 10(3–4):575–586, 2004.

[77] Mateu Sbert, Francesc Castro, and John Halton. Reuse of paths in light source animation. In *Proceedings of Computer Graphics International 2004 (CGI '04)*, pages 532–535. IEEE Computer Society, June 2004.

[78] Mateu Sbert, Jaume Rigau, Miquel Feixas, and László Neumann. Systematic sampling in image-synthesis. In *Proceedings of ICCSA 2006 (LNCS 3980)*, May 2006.

[79] Mateu Sbert, László Szecsi, and László Szirmay-Kalos. Real-time light animation. *Computer Graphics Forum (Eurographics 2004 Proceedings)*, 23(3):291–299, September 2004.

[80] P. Shirley. Discrepancy as a quality measure for sample distributions. In *Proceedings of Eurographics '91*, 1991.

[81] Robert Siegel and John R. Howell. *Thermal Radiation Heat Transfer, 3rd Edition.* Hemisphere Publishing Corporation, 1992.

[82] Francois Sillion and Claude Puech. *Radiosity and Global Illumination.* Morgan Kaufmann, San Francisco, CA, 1994.

[83] Ilya M. Sobol. *Monte Carlo numerical methods.* Ed. Science, Moscow, 1973.

[84] A. James Stewart. Vicinity shading for enhanced perception of volumetric data. In *IEEE Visualization*, October 2003.

[85] D. Stoyan, W. S. Kendall, and J. Mecke. *Stochastic Geometry and its Applications.* John Wiley and sons, Chichester, 1987.

[86] László Szirmay-Kalos and Werner Purgathofer. Global ray-bundle tracing with hardware acceleration. In G. Drettakis and N. Max, editors, *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop '98)*, pages 247–258. Springer Wien, 1998.

[87] László Szirmay-Kalos, Mateu Sbert, Roel Martínez, and Robert F. Tobler. Incoming first-shot for non-diffuse global illumination. In *Spring Conference on Computer Graphics*, Budmerice, Slovakia, 2000. Available from `http://www.fsz.bme.hu/~szirmay/puba.htm`.

[88] T. Ullmann, B. Bruderlin, D. Beier, and A. Schmidt. Adaptive progressive vertex tracing in distributed environments. In *Ninth Pacific Conference on Computer Graphics and Applications, 2001. Proceedings.*, pages 285–294, 2001.

[89] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation.* PhD thesis, Stanford University, December 1997. Available from `http://graphics.stanford.edu/papers/veach_thesis`.

[90] Eric Veach and Leonidas J. Guibas. Metropolis light transport. In *Computer Graphics (ACM SIGGRAPH '97 Proceedings)*, volume 31, pages 65–76, 1997.

[91] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at `http://www.mpi-sb.mpg.de/~wald/PhD/`.

[92] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Interactive global illumination using fast ray tracing. In *Rendering Techniques 2002 (Proceedings of the Thirteenth Eurographics Workshop on Rendering)*, June 2002.

[93] B. Walter, G. Drettakis, and S. Parker. Interactive Rendering using the Render Cache. In *Proc. of the 10th Eurographics Workshop on Rendering*, pages 235–246, 1999.

[94] G. Ward and P. Heckbert. Irradiance gradients. In *Proceedings of the Third Eurographics Workshop on Rendering*, pages 85–98, May 1992.

[95] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 32(6):343–349, June 1980.

[96] Sergei Zhukov, Andrej Iones, and Grigorij Kronin. An ambient light illumination model. In G. Drettakis and N. Max, editors, *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop '98)*, pages 45–56. Springer Wien, 1998.