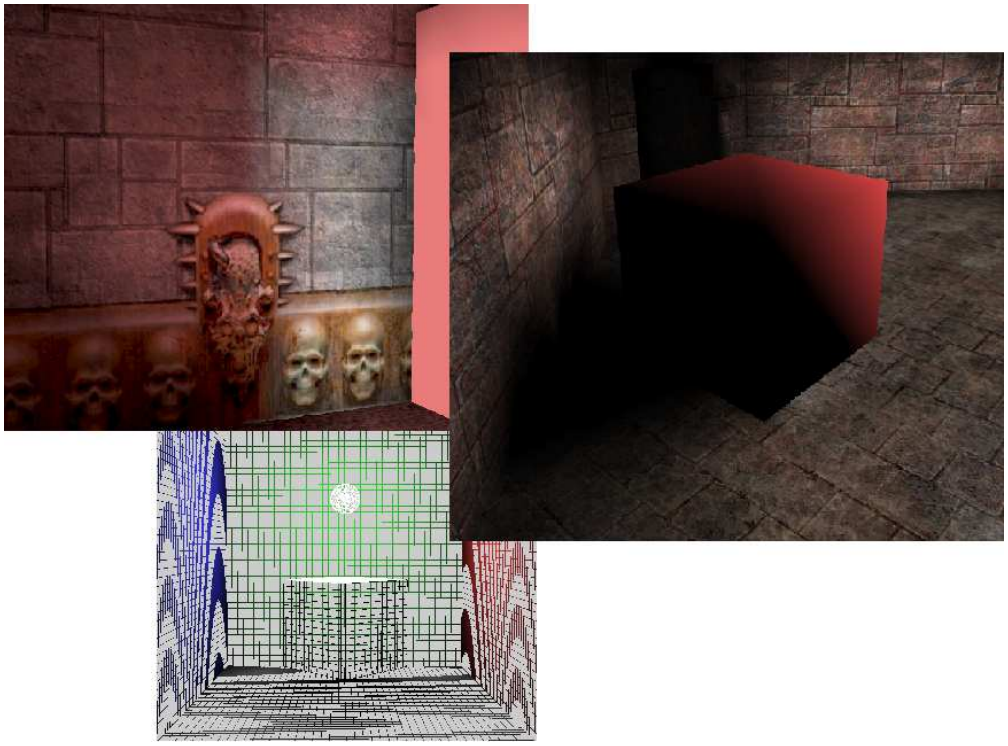
# The $\pi$-engine

## PART 2

Dynamic illumination in 3D engines using real-time photon mapping




| Group E4 206b | Supervisors |
|---|---|
| Dennis Kristensen | Claus B. Madsen |
| Thorsten Jørgen Ottosen | Olav Bangsø |

Faculty of Engineering and Science
Aalborg University

**Department of Computer Science**

**TITLE:**

**The $\pi$-engine**, PART 2, Dynamic illumination in 3D engines using real-time photon mapping.

**SEMESTER PERIOD:**

DAT6,
1st of February 2003-
20th of June 2003

**PROJECT GROUP:**

E4-206b

**GROUP MEMBERS:**

Dennis Kristensen, snicki@cs.auc.dk
Thorsten Jørgen Ottosen, nesotto@cs.auc.dk

**SUPERVISORS:**

Claus B. Madsen, cbm@cvmt.auc.dk
Olav Bangsø, bangsy@cs.auc.dk

**NUMBER OF COPIES:** 7

**NUMBER OF PAGES:** 77

**TOTAL NUMBER OF PAGES:** 82

**SYNOPSIS:**

This report describes the application of real-time photon mapping to enhance realism of lighting and to automate illumination work in interactive 3D applications.

The first part of this report reviews the essential theory about light and real-time photon mapping whereas the second part discusses the implementation and results in detail.

This report looks into two major problems of real-time photon mapping: to identify and remove performance bottlenecks and to reduce fluctuation and improve visual quality.

The main conflict is between image quality and real-time constraints. The key to improve quality is to increase photon and polygon count. However, increasing either of these decreases performance.

The conclusion summarizes important guidelines that must be followed to enable real-time photon mapping and describes problems that are subject to future work.

# Preface

*If there is any situation worse than having no documentation, it must be having wrong documentation.*
—Bertrand Meyer

Our report is entitled "The $\pi$-engine" which is short for "The photon illumination engine". We remind the reader that this report is the second of two; the first report dealt with analysis and implementation of a test scenario, and this report solves major problems laid out in the first.

We expect the reader to be familiar with C++ since many algorithms are presented in C++ code or C++ like pseudo-code.

The spin-offs of this project will be an open source Quake 3 BSP loader for Open Scene Graph, an easy to understand photon map implementation and an `Exact_photon_map` class which can be used to verify other implementations against. Everything can be downloaded from `http://www.cs.auc.dk/~nesotto/pie`.

We have tried to make the report as self-contained as possible. When this is not possible, we provide references and try to be as specific as possible by including page numbers. Note that no hyphens are used to break WWW-addresses in the bibliography; any hyphen will therefore be part of the address itself. *Italics* are used to emphasize importance whereas technical terms or important aspect are written in a **bold font** when they are defined. When we discuss issues related to real-time graphics and 3D hardware, we refer to an OpenGL context if nothing else is mentioned. However, most of the principles are ubiquitous and exist in other APIs as well.

We would like to thank the following people. Thanks to Henrik Wann Jensen for providing us with the implementation of his photon map. Thanks to Frank Suykens and Bent Dalgaard Larsen for our correspondence. We also appreciate the helpful people on the Open Scene Graph mailing list. Thanks to Bjarne Stroustrup for creating C++ and to Alex Stepanov for developing the Standard Template Library.

Dennis Kristensen

Thorsten Jørgen Ottosen

# Contents

# 1 Introduction

*The greatest challenge to any thinker is stating the problem in a way that will allow a solution.*
—Bertrand Russel

The main subject of this report is photon mapping and its applications in a dynamic real-time context. In particular we investigate dynamic indirect light. This chapter introduces real-time photon mapping and sketches the problems addressed in this report.

We begin with a short presentation of the lighting phenomena that real-time photon mapping can help simulate. The main idea of real-time photon mapping is then described followed by the goals of the project. In the end we give an overview of the rest of the report. If the reader is new to computer graphics we recommend that he also reads the introduction to our first report [OK02].

## 1.1 Lighting in 3D engines

Traditionally, 3D graphics engines have separated the rendering of static and dynamic light. **Static light** is light originating from light sources with static properties whereas **dynamic light** originates from light sources with variable properties like origin and direction. Static and dynamic light have again been separated into indirect light and direct light. **Direct light** refers to the light received directly from light sources. **Indirect light** is the light received from other surfaces (that is, reflected light). **Global illumination** is the combination of both direct and indirect light (notice that we use the terms *light* and *illumination* interchangeable). In real-time graphics shadows are treated separately for performance reasons. Figure 1.1 shows how important the indirect illumination is.


Figure 1.1: A room rendered with a) full global illumination, b) indirect illumination, c) direct illumination [VMKK00].

**Texture mapping** is used to attach images to models for improved visual quality without adding complexity to the geometric model. A **light map** is a texture map that represents how light strikes a surface, and it is widely used to create static indirect light and aspects of static direct light such as shadows. Contrary to photo-realistic rendering, indirect dynamic light is not simulated in real-time rendering since it relies on a global analysis of the scene which is too costly. Two of the effects that are due to indirect illumination are color bleeding and caustics. Color bleeding can be seen when e.g. a indirectly lit white wall appears reddish because a red object is situated close to it (see Figure 1.2 on the right), and an example of caustics is when a surface is lit by a magnifier (see Figure 1.2 on the left). The lack of indirect illumination is acknowledged as a shortcoming in current real-time rendering by David Kirk, NVIDIA [Sta03]:

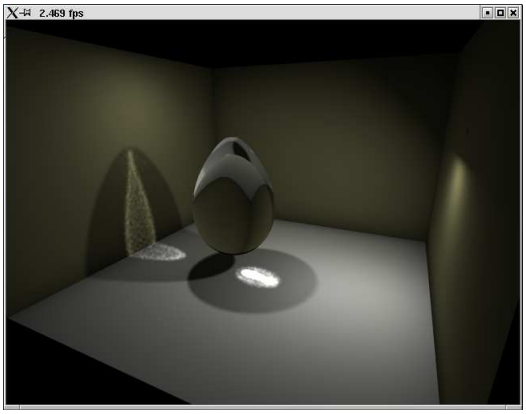
> ... we've moved from texture-mapped rendering to programmable pixel shading in pursuit of cinematic realism, but we aren't quite there yet. In particular, real-time 3D graphics rendering is not yet capable of global illumination.

We hope to move one step closer to this goal by using real-time photon mapping.

## 1.2 Real-time photon mapping

**Photon mapping** is a technique that can enhance the quality and speed of the ray tracing rendering technique. It is particularly useful for producing lighting effects such as caustics, shadows and color bleeding [Jen01, xv].

Photon mapping is a two pass algorithm where the rendering step is improved by using extra light information generated in a preprocessing step. In **real-time photon mapping** we can summarize the two steps as follows:



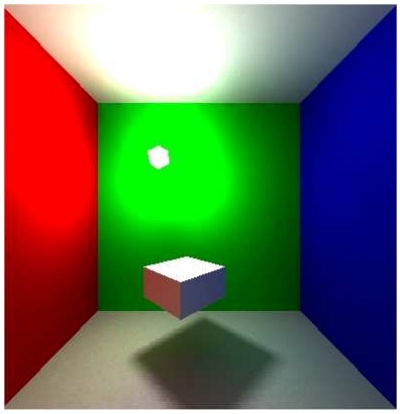


Figure 1.2: Left: An example of caustics. This scene with a glass egg consists of 4000 triangles [WKB+02]. Right: An example of color bleeding; notice how the colors of the walls appear on the sides of the hanging box.

1. Photon tracing:
   (a) emit photons from light source;
   (b) scatter photons diffusely.
2. Rendering:
   (a) use the photons to estimate the light each object receives;
   (b) blend the texture color of the object with the light estimate.

During **photon tracing** photons are emitted from the light sources in the scene. When a photon hits a surface, the photon is saved in a suitable data structure called the **global photon map** (or just the **photon map**). The photon is then split up into several new photons that are traced recursively one or two times. During **rendering** the photon map is used to estimate the light each vertex on each object receives. We say that we make an **irradiance estimate** for each vertex. Finally the light estimates are blended with the texture color.

The basic optimization idea of the technique is to calculate the irradiance estimate for each vertex in the scene and let the graphics hardware interpolate the color values in between vertices. Real-time photon mapping can be implemented in other ways than using hardware to interpolate between vertex colors, but when we say real-time photon mapping, we refer to this definition unless stated otherwise. The important observation that justifies this interpolation scheme is that the indirect illumination on diffuse surfaces often change slowly over the surface [WRC88, 3]. It is important to realize that this is not the case when illumination changes abruptly near shadow boundaries and caustics.

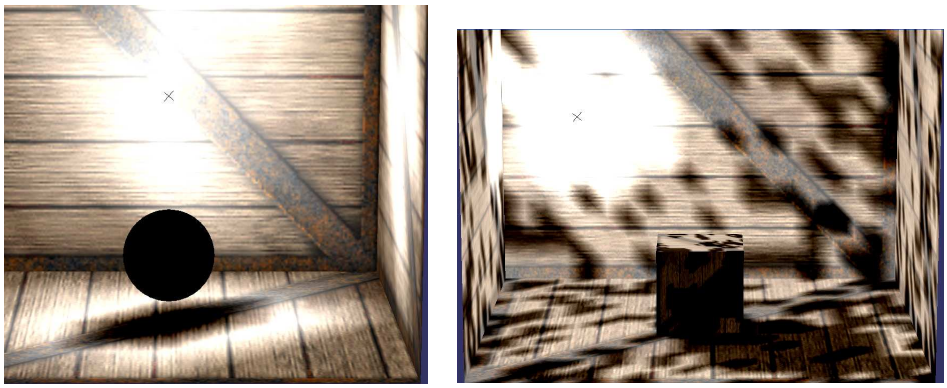


Figure 1.3: The results from our first report. The most apparent problems are the speckled appearance and the white saturation.

A similar method is used in **irradiance caching** where irradiance estimates are precomputed for each photon position and estimates between these positions are interpolated [Jen01, 140]. In general the interpolation of diffuse illumination is often used to increase performance in interactive contexts. This can be seen in a recent survey by Damez et al. where many of the methods use some form of interpolation [DDM03].

## 1.3 Goals

In this project we will incorporate photon mapping in a standard 3D engine. The motivation for adding photon mapping to 3D engines was described in [OK02, 10]. The main motivation is to add dynamic illumination and to avoid many special cases. The **long term goal** is to maintain framerates of at least 30 FPS while simulating dynamic light.

The problems that need to be overcome in real-time photon mapping can roughly be categorized as either performance related or visual quality related [OK02, 79f]. By **performance** we mean the execution time whereas **visual quality** refers to the realism of the rendered frames and the amount of fluctuation between the frames. The requirement of both high realism and high performance constitute opposing demands. Therefore we must find a compromise between visual quality and performance.

Most performance improvements are irrelevant if the image quality cannot be made sufficiently good. What we presented in our last report was far from satisfactory (see Figure 1.3). Thus our first goal is:

**Goal 1:** *Improve the image quality and reduce fluctuation to an acceptable level.*

If we cannot do this, then what use is it to improve performance? It is also

imperative to have a realistic scene to test the engine on. If the scene is not (at least a little) realistic, then it will be impossible to determine which problems that are performance bottlenecks. Hence, the second goal is:

**Goal 2:** *Build a dynamic scene that can work as a test scenario.*

But requiring that the scene is dynamic is not enough:

**Goal 3:** *Assemble the 3D engine so important modules are implemented and functioning in a realistic manner.*

By *realistic* we mean that e.g. the intersection testing module cannot use a linear search scheme, but should rely on some spatial sorting of the scene graph. If we do not have a realistic implementation of the different modules, then we can only guess what the performance bottlenecks might be.

Using photon mapping could simplify illumination effects, but on the other hand it might complicate the engine itself:

**Goal 4:** *The integration of photon mapping and the real-time 3D graphics engine should be as easy as possible.*

This implies that we should be able to handle scenes of the same size as before the integration. When the engine has matured to fulfill goal 4, it will be necessary with some tests:

**Goal 5:** *Test different methods to scatter photons throughout the scene and find out which methods that should be preferred in the engine.*

It will also be imperative to review what would be most important to do next:

**Goal 6:** *Identify areas that clearly seems to be performance bottlenecks.*

The most pressing performance problems are predicted to fall into two broad categories: intersection testing and irradiance calculation. Our last report identified three main areas that still needs further work [OK02, 60ff]:

1. Faster photon map data structure.
2. Faster intersection testing.
3. Optimal photon distribution.

We will test and profile all three areas to identify those areas that need further improvement.

In our first report we did not put focus on any special visual effect, but wanted to look at both shadows, color bleeding and caustics [OK02, 10f]. Admittedly this scope is too big and we narrow it to the following:

**Goal 7:** *Keep the focus on dynamic indirect lighting with color bleeding in diffuse environments.*

## 1.4 Overview of report

**Chapter 2. Illumination theory.** We start with an introduction to light phenomena and continue with a description of physical models of light. This will give us the necessary background information needed to understand photon mapping.

**Chapter 3. Real-time photon mapping.** Here we give a detailed description of the real-time photon mapping algorithm. In particular, we describe how illumination from the photon map is combined with the texture color. We describe several enhancement techniques and discuss how photon scattering is best performed.

**Chapter 4. Implementation overview.** This chapter introduces the scene graph library that we have used as the foundation of the $\pi$-engine. We show the basic control flow of the application and discuss parameters that can tweak the photon mapping algorithm. We also describe how testing is done throughout the rest of the report.

**Chapter 5. Implementation details.** The collaboration of the fundamental classes in the $\pi$-engine is explained followed by an in-depth view of the most important classes. We discuss several performance and quality related issues that have been or should be dealt with.

**Chapter 6. Results.** In this chapter we explain several tests and test parameters. The tests are concerned with both visual quality and performance issues. This chapter will also give an impression of the status of the $\pi$-engine.

**Chapter 7. Conclusion.** In the conclusion we look back at the work of two semesters. We review what this particular report has contributed with and describe directions for future work.

# 2 Illumination theory

*I believe that global illumination will become the norm. Direct illumination renderers like prman have reached the limit of the realism they can produce in terms of lighting.*
—Kaveh Kardan, Square USA

To make a realistic simulation of light it is important to understand the physical models that are used to describe light. In order to simulate light interaction in computer graphics we use an illumination model that encapsulates as much of the physical model as possible. At the same time it should be simple enough to make it tractable for use in computer graphics applications. An **illumination model** (or **reflection model**) uses parameters such as light source properties, material properties, and surface geometry to describe light-surface interaction. Illumination models can be separated into global and local illumination models. In a **local illumination model** only direct light is considered. In a **global illumination model** both direct and indirect light is considered

This chapter starts with a description of light sources and lighting effects. Then we discuss physical properties of light followed by reflection models. We end the chapter with the fundamental equation that all global illumination algorithms strive to solve.

## 2.1 Lighting effects overview

Lighting effects or phenomena depend on the light source properties as well as the material properties and geometry of the surface where the light-

ing effect occurs. Therefore we discuss these properties before the lighting effects.

### 2.1.1 Basic definitions

In real-time 3D computer graphics we are usually restricted to three types of light sources: directional lights, point lights, and spot lights [Ebe01, 100]. A **directional light source** is assumed to be located infinitely far away such that the light rays are parallel—a classical application is to approximate sunlight. **Point light sources** emit light *uniformly* in all directions. A **spot light source** only emits light within a cone and the light can be distributed uniformly or with focus. Other light source attributes include the color and intensity of the light, and point lights and spot lights should—if realism is a concern—have their illumination attenuated with distance from the light source. Despite its caveats (see [Wat00, 421ff]), the ubiquitous color representation is the RGB model.

If a light source is visible from a surface, the surface will be hit by **direct light** (or **direct illumination**). **Reflection** means the return of light from a surface, and **indirect light** (or **indirect illumination** or **ambient light**) is light that has been reflected one or more times. We sometimes use the term **bounce** instead of reflection. Of course, a directly lit surface is also hit by indirect light.

When light is emitted from a light source, we need to be able to describe how it scatters in the scene. Most reflection models deal with at least two types of reflections: diffuse and specular. **Diffuse reflections** model the reflection of light scattered in all directions whereas **specular reflections** models mirror-like reflections. The reflective properties of most materials can be described by a combination of these two components. A surface that reflects light *uniformly* in all directions is called **perfectly diffuse** (or **Lambertian**) whereas one that is completely specular is denoted as a **perfect mirror**.

### 2.1.2 Lighting effects

Indirect illumination is an important global illumination effect which has a profound impact on the perceived realism. This is because many of the effects described below stems from indirect illumination.

A **Glossy reflection** can be seen as a combination of a diffuse and a specular reflection and results in a blurred reflection. Some materials such as paper and wood reflects more light in shallow angles—this is known as the **Fresnel effect**.

**Color bleeding** is the transfer of color between objects caused by the reflection of indirect light. Real-time graphics might be able to simulate color bleeding by using colored lights. We already saw an example of color

bleeding in Figure 1.2. Notice that there is very little visible color bleeding on the floor; in general the direct illumination is much stronger than the indirect.

When light traveling through a transparent medium encounters a boundary leading into another transparent medium, a part of the ray is reflected and a part enters the second medium. The part that enters the second medium is bent at the boundary and is said to be **refracted** [SBJWJ00, 1113]. The light interaction between specular and diffuse surfaces can produce caustics. A **caustic** appears on a diffuse surface when light is concentrated by a specular reflection or refraction. An example of caustics produced by refraction was shown in Figure 1.2. Notice that refraction produces darker areas around the caustics. Refraction can also disperse light as when it passes through a prism.

Both direct and indirect illumination can form a shadow. A **shadow** is an area that is only partially illuminated due to blockage of light by an opaque object called the **occluder**. If a point is not directly lit at all, it is part of the **umbra** of the shadow. If a point on the shadow is lit by a portion of the light source, it is part of the **penumbra** of the shadow. A **hard shadow** consist of pure umbra points whereas a **soft shadow** has its umbra surrounded with a penumbra. It follows that the typical diffuse point light source cannot produce soft shadows.

## 2.2 Lighting terminology

Several different physical models of light exist. In computer graphics the ray optics model is predominant. **Ray optics** (or **geometrical optics**) involves the study of the propagation of light with the assumption that light travels in a straight line as it passes through a uniform media. Ray optics can be used to simulate most visual effects including reflection and refraction. We discuss the radiometry terminology for describing ray optics.

### 2.2.1 Notation

In the following $x, x'$ will denote surface locations, $\vec{n}$ will be the normal vector at $x$, and $\vec{\omega}, \vec{\omega_i}$ are unit vectors that represents the reflected and incoming direction, respectively. $\vec{\omega_r}$ and $\vec{\omega_s}$ represents the refracted and specular reflected direction, respectively.

The set of all possible directions is the unit sphere $\Omega_{4\pi}$ that has a solid angle of $4\pi$ steradian. A hemisphere $\Omega_{2\pi}$ covers $2\pi$ steradian. Associated with a direction is the differential solid angle, $d\omega$, that is used for integration over finite solid angles. Notice that we do not use a curly "d" ($\delta$) even though we describe partial derivatives; this notation is commonly used [Jen01] [Suy02].

| Symbol | Description | Unit |
|---|---|---|
| $Q$ | radiant energy | $[J]$ |
| $Q_\lambda$ | spectral radiant energy | $[J]$ |
| $\Phi$ | radiant flux | $[W]$ |
| $\Phi_\lambda$ | spectral radiant flux | $[W/m]$ |
| $B(x)$ | radiosity of surface location $x$ | $[W/m^2]$ |
| $E(x)$ | irradiance at surface location $x$ | $[W/m^2]$ |
| $L(x, \vec{\omega})$ | radiance at surface point $x$ in direction $\vec{\omega}$ | $[W/(m^2 \cdot sr)]$ |

Table 2.1: Overview of symbols used in radiometry.

In spherical coordinates the directions are represented as $\vec{\omega} = (\theta, \phi)$ and $\vec{\omega_i} = (\theta_i, \phi_i)$. Moreover, we sometimes use subscripts $i$ and $r$ to mean the *incoming* and the *reflected* of some concept. Note that all vectors are assumed to be normalized unless stated otherwise.

### 2.2.2 Radiometry

In **radiometry** the basic quantity is the **photon** which describes a quantum of electromagnetic radiation. Symbols related to radiometry are listed in Table 2.1. A photon with wavelength $\lambda$ has an energy $e_\lambda$. The **spectral radiant energy** of $n_\lambda$ photons with the same wavelength is defined as $Q_\lambda = n_\lambda e_\lambda$.

**Radiant energy** is the quantity of energy propagating onto, through or emerging from a specified surface of given area in a given period of time [Jen01, 13]. We calculate it as the energy of a collection of photons

$$Q = \int_0^\infty Q_\lambda \, d\lambda \quad [\mathrm{J}] \ , \tag{2.1}$$

that is, we integrate the spectral radiant energy for all possible wavelengths. Flux is commonly used to denote the rate of transfer of particles or energy across a given surface. With **radiant flux** (or **power** or **flux**) we denote the flow of radiant energy per time given by

$$\Phi = \frac{dQ}{dt} \quad [\mathrm{W}] \ , \tag{2.2}$$

that is, the quantity of energy transferring through a surface or region of space per time.

**Radiosity**, $B$, is the radiant flux leaving a surface whereas **irradiance**, $E$, is the radiant flux arriving at a surface. If a surface does not absorb or transmit light, then $B = E$. Irradiance is given by

$$E(x) = \frac{d\Phi}{dA} \quad \left[\mathrm{W/m^2}\right] \ . \tag{2.3}$$

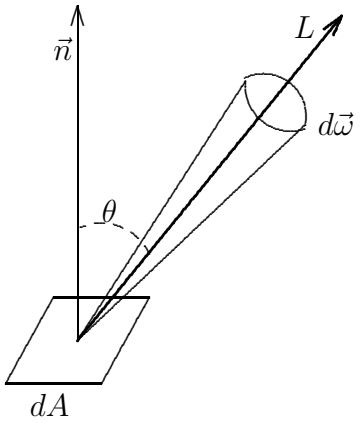



Figure 2.1: The radiance, $L$, is defined as the radiant flux per unit solid angle, $d\vec{\omega}$, per unit projected area, $dA$.

**Radiance** can be thought of as the number of photons arriving (or leaving) per time at a small area from a given direction, and it can be used to describe the intensity of light at a given point in space in a given direction (see Figure 2.1). Formally, radiance is the radiant flux per solid angle per projected area given by

$$L(x, \vec{\omega}) = \frac{dE(x)}{\cos\theta \, d\vec{\omega}} = \frac{d^2\Phi}{\cos\theta \, d\vec{\omega} \, dA} \tag{2.4}$$

that is, the area and solid angle density of radiant flux. The cosine factor in the denominator expresses that the surface area is foreshortened and the effective surface area is $\cos\theta dA$. In vacuum, an important property of radiance is that it is constant along a line of sight, that is, the photons are not dispersed, do not loose energy and do not disappear entirely—this is used by all ray tracing algorithms [Jen01, 15].

## 2.3 Light scattering

Now that we have an overview of the lighting terminology, we are interested in describing light-surface interaction. In simplified situations light-surface interaction can be described by well-known physical laws. If we consider arbitrary reflection properties of materials, however, the important question arises how the reflection properties can be represented. In computer vision as well as computer graphics the **bidirectional reflectance-distribution function** (BRDF) is used as the fundamental tool to describe reflection characteristics.

### 2.3.1 The BRDF

Informally, a BRDF describes how much of the light that comes in from one direction goes out in another direction. The fraction of incident light that is reflected by a surface is called the **reflectance** and is denoted $\rho(x)$—the remainder is either transmitted or absorbed. Normally the BRDF is dependent on the wavelength of the incoming light, but in the following we omit such concerns. An approximation could be to use a BRDF for each of the RGB-components.

Formally, the BRDF defines the relationship between differential reflected radiance and differential irradiance [Wu03, 6]. By Equation 2.4 we have that

$$f_r(x, \vec{\omega_i}, \vec{\omega}) = \frac{dL_r(x, \vec{\omega})}{dE(x, \vec{\omega_i})} = \frac{dL_r(x, \vec{\omega})}{L_i(x, \vec{\omega_i})(\vec{\omega_i} \cdot \vec{n})d\vec{\omega_i}} \quad \left[\mathrm{sr}^{-1}\right] \tag{2.5}$$

If we know the incident radiance field at a surface location, we can compute the reflected radiance in all directions. This is done by rearranging equation 2.5 and integrating the incident radiance $L_i$:

$$L_r(x, \vec{\omega}) = \int_{\Omega_{2\pi}} f_r(x, \vec{\omega_i}, \vec{\omega}) dE(x, \vec{\omega_i}) \tag{2.6}$$

$$= \int_{\Omega_{2\pi}} f_r(x, \vec{\omega_i}, \vec{\omega}) L_i(x, \vec{\omega_i})(\vec{\omega_i} \cdot \vec{n}) d\vec{\omega_i} \tag{2.7}$$

An important property of the BRDF is **Helmholtz's law of reciprocity** which states that the BRDF is independent of the direction in which light flows:

$$f_r(x, \vec{\omega_i}, \vec{\omega}) = f_r(x, \vec{\omega}, \vec{\omega_i}) \tag{2.8}$$

This is a fundamental property that makes it possible to trace light paths in both directions. Another physical property of the BRDF is that it is less or equal to 1 due to energy conservation. A surface (which is not an emitter) cannot reflect more light than it receives.

The BRDF is itself a simplification of more complex models since we assume that light striking a surface is reflected at the same surface location. For example, the BSSRDF can be used to simulate translucent materials like milk, marble and skin [Jen01, 18f].

In this project we mostly deal with Lambertian surfaces. For a Lambertian surface the reflected radiance is constant in all directions regardless of the irradiance. As a consequence a point on a Lambertian surface is equally bright from any view direction. This gives the constant BRDF $f_{r,d}(x) = \rho(x)/\pi$ [Jen01, 21]. The radiance is therefore by Equation 2.6

$$L_r(x) = f_{r,d}(x) \int_{\Omega_{2\pi}} dE(x, \vec{\omega_i}) = f_{r,d}(x) E(x) \tag{2.9}$$

## 2.4 The rendering equation

A basis for all global illumination algorithms is found in the **rendering equation** which can be used to compute the outgoing radiance at any surface location in the model. It states that the outgoing radiance, $L_o$, is the sum of the emitted radiance, $L_e$, and the reflected radiance, $L_r$:

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + L_r(x, \vec{\omega}) \quad . \tag{2.10}$$

By using equation 2.7 to compute the reflected radiance we get the rendering equation as it is often used in ray-tracing algorithms:

$$L_o(x, \vec{\omega}) = L_e(x, \vec{\omega}) + \int_{\Omega_{2\pi}} f_r(x, \vec{\omega_i}, \vec{\omega}) L_i(x, \vec{\omega_i})(\vec{\omega_i} \cdot \vec{n}) d\vec{\omega_i} \quad . \tag{2.11}$$

$L_i(x, \vec{\omega})$ originates from outgoing radiance on other surfaces which reveals the recursive nature of the equation. The integral is solved numerically by tracing rays over the hemisphere and calculating the outgoing radiance at the surfaces that they intersect [Suy02, 9f].

## 2.5 Summary

Many different lighting phenomena exist and we have seen how they are affected by different types of light sources and material properties (Section 2.1). The light-surface interactions that lead to these phenomena are modeled in an illumination model. Local illumination models only model direct light whereas global illumination models model both direct and indirect light.

In Section 2.2 we described radiometry which is a lighting terminology where the basic quantity is the photon. The two most important concept in radiometry is irradiance and radiance. Irradiance can be thought of as the number of photons arriving per time at a particular location from all directions. Incoming radiance can be described as the irradiance originating from a particular direction.

BRDFs are used to describe how much of the light that comes in from one direction is reflected in another direction (Section 2.3). The most important property of the BRDF is Helmholtz's law of reciprocity which allows global illumination algorithms to trace light paths in both directions. Outgoing radiance on a diffuse surface is the product of the diffuse BRDF and the irradiance.

In Section 2.4 we described the rendering equation which provides a mathematical model for computation of outgoing radiance. The equation can be solved numerically by ray-tracing; incoming radiance is computed by recursively computing the outgoing radiance from other surfaces.

# 3 Real-time photon mapping

*One of us recalls producing a "random" plot with only 11 planes, and being told by his computer center's programming consultant that he had misused the random number generator: "We guarantee that each number is random individually, but we don't guarantee that more than one of them is random."*
—[PVTF02]

Photo-realistic photon mapping is a full global illumination algorithm that can be used to solve the rendering equation in a way that includes complex simulation of indirect illumination. Conceptually, the method is ordinary Monte Carlo ray tracing with extra light information stored in photon maps. **Monte Carlo ray tracing** approximates the rendering equation by tracing a large amount of randomly generated rays throughout the the scene for many recursions. This is computationally very expensive and a single image can take hours to render. In this respect there is a long way before we can generate tens of images per second.

In this chapter we review the basic photon mapping algorithm and methods we use to enable it to run in real-time. Our presentation emphasizes the cut down real-time version that we use, but we summarize how it deviates from the normal photo-realistic rendering method. Two techniques from the photo-realistic version are introduces; we shall later evaluate their relevance to real-time rendering. We describe how the irradiance estimate can be blended together with the textures of the scene to produce the final image. Then we investigate how photons should be traced throughout the scene, and quality improvement techniques are discussed. For the rest of this report we assume that the reader has a basic understanding of real-time graphics (an overview can be found in [OK02, 39ff]).

## 3.1 The algorithm

Photon mapping is a simple two pass algorithm consisting of a photon tracing pass and a rendering pass. In real-time photon mapping we do not make a full photon tracing pass for each frame, but distribute calculations out on several frames to speed up the algorithm. An important aspect of our work is to enable global illumination in a *dynamic* context. When light sources and objects are animated, it can lead to problems when photons are stored for more than one frame. One could consider to make an invalidation scheme that detected which photons to remove and re-emit (see e.g. [DBMS02]). Such a scheme is probably of less value if the scene is full of action—then the entire photon map should be invalidated. Until we can afford re-shooting the entire photon map for each frame, we only shoot a fraction of the photons for each frame. What is important to this strategy is that it should work reasonable as long as the frame-rate is high, so the photon map is re-filled several times per second.

Before we explain the photon mapping algorithm, it will be necessary with a short description the photon. In photon mapping a **photon** is defined by its power, position and incoming direction. In reality a photon has a particular wavelength which is perceived as a certain **color** by the eye. When many photons of the same wavelength reach the eye, we see the same color, but with a larger **intensity**. In photon mapping (and in computer graphics in general) the concept of color and intensity merges into one component, namely the power represented as an RGB vector. In the **RGB vector** the relationship between the three components define the color and the length of the vector or the sum of the components defines the intensity. Therefore a single photon represents a collection of real photons with the same wavelength.

### 3.1.1 Photon tracing

**Photon tracing** works in the same way as ray tracing, except for the fact that photons propagate power whereas rays gather radiance—this is important since the photon-surface interaction can be different than ray-surface interaction [Jen01, 60]. Photon tracing of a single photon can be described in five steps:

1. **Emit:** Choose photon origin and direction from light source. This procedure depends on the type of light source, see Section 3.4. The power of the light source is distributed evenly among all emitted photons.

2. **Intersect:** Trace photon until the first surface intersection. If no intersection is found, the next photon can be traced.

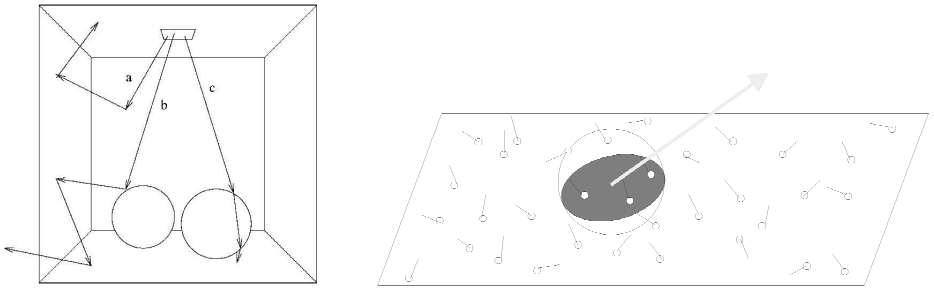



Figure 3.1: Left: Photon paths in a scene with a specular sphere on the left and a glass sphere on the right: (a) two diffuse reflections followed by absorption, (b) a specular reflection followed by two diffuse reflections, (c) two refractions followed by absorption [JCS01, 20]. Right: Gathering radiance in a sphere. The gray area is the area of the circle inside the sphere that is used to find the irradiance [JCS01, 30].

3. **Store:** Store the photon containing the point of intersection and the incoming direction of the photon in the photon map.

4. **Reflect:** Create reflected photons. First the power of the reflected photon must be calculated by scaling the power of the incoming photon with the diffuse reflectance $\rho(x)$ and the diffuse BRDF $f_{r,d}$ [Jen01, 61]—this is how color bleeding is accounted for. To make it simple we use a reflectance of $1$ and assign one BRDF for an entire surface. To quickly cover the scene with photons, the photon is split into several lower-powered photons which are dispersed diffusely on the hemisphere.

5. **Recurse:** Goto step 2 until some predefined depth is reached. This depth is usually 1 or 2.

Figure 3.1 shows three different photon paths. A **photon path** denotes the light path traveled by a photon until the photon cease to exist. While the photons are traced throughout the scene, the photons must be stored so that they can be easily retrieved later. A **kd-tree** data structure is used to store the photons because it can be represented compactly as an array and because it is quite fast to search for $k$ photons in. If $n$ is the number of photons stored in the photon map (which we denote as the **size** of the photon map), the $k$ nearest photons of a point can be found in $O(\lg n + k)$ time [Jen01, 69]. Such a search is called a **nearest neighbor search**. Because photons are stored in a geometry-independent data structure, photon mapping scales well with complex scenes [Suy02, 106].

### 3.1.2 Rendering

In normal polygon rendering all unculled polygons are passed directly to the graphics hardware which draws the polygons on the screen. To take advantage of the photon map we intervene the normal rendering pipeline between culling and drawing. The additional steps in this pass are:

1. **Irradiance estimate:** The irradiance at a vertex can be estimated by direct gathering or final gathering.

   (a) **Direct gathering**: This process is depicted in Figure 3.1 on the right. Conceptually, a nearest neighbor search for $k$ photons is conducted at a vertex by expanding a sphere until $k$ photons is found or until the search radius exceeds a predefined maximum. We call this distance for the **maximum search radius**. The irradiance is the sum of the collected radiance divided by the area in which the photons were found.

   (b) **Final gathering**: This process is used to mask radiance inaccuracies in the photon map by approximating the irradiance by a huge amount of radiance estimates. Final gathering gives better results, but it is much slower to compute. The procedure is explained in detail in Section 3.2.2.

2. **Radiance reconstruction:** The irradiance is then interpolated between vertices by graphics hardware. Combined with the BRDF of the surface and (optionally) direct illumination from the light sources, the radiance is computed for each pixel.

### 3.1.3 Comparison with the photo-realistic algorithm

To make the algorithm faster we use far fewer recursions, fewer photons, disable final gathering and use several simplifications. First of all, the we only simulate Lambertian BRDFs which means, for example, that we cannot get caustics. In normal photon mapping there are two or three photon maps: the global photon map, the caustic map and the volume photon map. The caustic map stores photons that have been specularly reflected or refracted at least once whereas the volume map is used when participating media like smoke must be simulated.

Second, we simulate diffuse reflection by splitting up a photon which is actually the opposite of the photo-realistic version where fewer photons are usually reflected. Instead of reflecting all photons with lesser power, only a reflectance dependent fraction of the photons is emitted with full power; this is called Russian Roulette [Jen01, 61].

Our radiance construction is quite different since it is both simpler and relies on hardware interpolation. Ignoring the volume map, a normal radiance construction is done using ray tracing where the irradiance at a single point is computed as the sum of the direct light, the specularly reflected light, a direct gathering in the caustics map and final gathering from the global map [Suy02, 115].

### 3.1.4 The radiance estimate

The radiance estimate at a given surface location is calculated using the photon map as

$$L_r(x,\vec{\omega}) \approx \frac{1}{\pi r^2}\sum_{p=1}^{k} f_r(x,\vec{\omega}_p,\vec{\omega})\Delta\Phi_p(x,\vec{\omega}_p) \tag{3.1}$$

where the first term is the circle area ($r$ is the distance to the most distant photon), $f_r$ is the BRDF, $\Delta\Phi_p$ is the power of the photon and $\vec{\omega}_p$ is the incoming direction. This formula can be derived from Equation 2.7 (see [Suy02, 110f]). The factor $\frac{1}{\pi r^2}$ is **area estimate** for a sphere; if we use other volumes to gather radiance in, we might have to change the area estimate. When we only consider Lambertian surfaces, the radiance estimate becomes

$$L_r(x) \approx f_d(x)\,\frac{1}{\pi r^2}\sum_{p=1}^{k}\Delta\Phi_p(x,\vec{\omega}_p) \tag{3.2}$$

where the factor

$$\frac{1}{\pi r^2}\sum_{p=1}^{k}\Delta\Phi_p(x,\vec{\omega}_p) \tag{3.3}$$

is an approximation of the irradiance at $x$ which we denote as the **irradiance estimate**.

The accuracy of the estimate depends on $n$ and $k$—ideally $n$ should be very large (perhaps millions) and $k$ should be relative low (perhaps hundreds). The formula and radiance gathering itself is based on several assumptions:

- the geometry is locally flat,
- the photons are well distributed throughout the scene,
- no photons from other surfaces are present.

If the geometry is not locally flat, the area estimate becomes wrong. In theory one should simply use a very little radius, but in practice it might be difficult unless the photon map stores a huge amount of photons. If photons are not well distributed, the power carried by the photons will not represent the real power.

The area estimate can also be wrong for another reason: near edges and corners large parts of the sphere will be outside geometry. The error can be quite significant and the effect is known as **boundary bias** [Suy02, 112].

Photons that belong to entirely different surfaces or areas far away are said to **leak** into the irradiance estimate. On those surface the illumination could be different thus introducing a bias. **Surface leaking** is defined as leaking from surfaces with a different normal or translated surfaces with a similar normal, and we call the corresponding bias for **surface bias**. Using a large maximum search radius near shadow boundaries and inside caustics can also include photons from a differently lit area even though the surface is flat; this is denoted **distance leaking** and it results in an **distance bias** in the radiance estimate.

Corners and edges are the most problematic regions in photon mapping since all three types of bias can be introduced here. Section 3.5 discusses bias in detail.

## 3.2 Improvements

In this section we describe two techniques that are normally used in photorealistic rendering: a maximum search radius heuristic and final gathering. The radius heuristic promises to remove distance bias and improve performance of the radiance estimate whereas final gathering can potentially produce much better rendering results.

### 3.2.1 Maximum search radius

The maximum search radius and the maximum search count ($k$) controls the behavior of the radiance estimate. While $k$ is relatively scene independent, the search radius is not. If the radius is too high, we search too much of the kd-tree which impede performance and introduces distance bias. If the radius is too low, we get an inaccurate radiance estimate. It would therefore be beneficial to automate the process of finding a good maximum search radius in a scene independent manner [Suy02, 116f].

If a radiance estimate is low, it means we have searched in a area with low photon density. In this area there is a high change of distance leaking and a smaller radius would have been preferred. If the scene in general has a low photon density, then we should in general use a relatively small radius. The question now is when a radiance value can be labeled as low.

A sort of average radiance estimate can be constructed by considering all photons:

$$L_{avg}(x,\vec{\omega}) \approx \frac{1}{\pi r_n^2} \sum_{p=1}^{n} f_r(x,\vec{\omega}_p,\vec{\omega}) \Delta\Phi_p(x,\vec{\omega}_p) \ . \tag{3.4}$$

where $r_n$ is the radius of the sphere enclosing all photons. We label a radiance estimate as low when it is below a certain small percentage $\alpha$ of the average radiance. The maximum search radius is reached when

$$L_r = \alpha L_{avg} \ . \tag{3.5}$$

We simplify both estimates by replacing $\Delta\Phi_p$ with an average photon power $\Delta\Phi_{avg}$ and by replacing the BRDF evaluation by an average diffuse BRDF $\rho_{avg}/\pi$. The equality becomes

$$\frac{k\ \Delta\Phi_{avg}\ \rho_{avg}/\pi}{\pi r_{max}^2} = \alpha \frac{n\ \Delta\Phi_{avg}\ \rho_{avg}/\pi}{\pi r_n^2}. \tag{3.6}$$

and therefore

$$r_{max} = \sqrt{\frac{k\ r_n^2}{\alpha n}} \tag{3.7}$$

A slightly different heuristic can be found in [JCS01]. Both heuristics show the same square root dependence on $n$ and are reported to perform well (especially with caustic maps).

### 3.2.2 Final gathering

Final gathering is used to mask errors in the radiance reconstruction from the global map [Suy02, 115]. The are two reasons for using final gathering: (1) the global photon map only needs to store a coarse approximation of the radiance in a scene [Suy02, 107] and (2) some surfaces are hard to disperse enough photons on (there exist solutions to this problem, but they are not easy to apply in a real-time context [SW00] [PP98]). Since final gathering can produce much better rendering results, it also gives us a hint about how good the real-time method can potentially become.

The principle behind final gathering can be seen in Figure 3.2. A **final gather** is conceptually a diffuse sampling of radiance over all incoming directions; each direction defines a **final gather ray** which intersects the scene and the outgoing radiance is calculated here. (Section 3.4 explains how the sampled directions should be generated). Once all radiance estimates are found, they are averaged together and multiplied with the local BRDF to produce the final radiance estimate. By taking an average we make the estimate independent of the number of final gather rays. Notice that the last step does not consider any area as it is done by Equation 3.1.

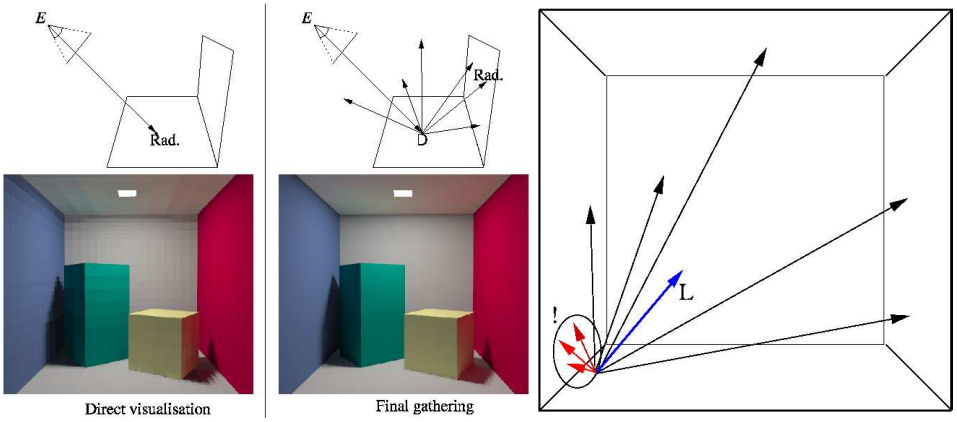



Figure 3.2: Left: A picture rendered with and without final gathering. Although these pictures are rendered using the radiosity algorithm, the effects are similar for photon mapping [Suy02, 40]. Right: Final gathering for a point near a corner. Many rays will hit the close-by surface. The errors in the radiance reconstruction in the encircled area may be visible in the final gather result.

The error on surfaces very close to the point where a final gather is performed may be visible in the accurate estimate—this case is shown in Figure 3.2. The wall close-by (red rays) covers a large part of the hemisphere with respect to the final gathering point. The error in the reconstruction in that area will have an important influence on the error in the estimate. The simple solution is to make a secondary final gather if the final gather point is within a certain (small) distance. In practice this should only be necessary for a small fraction of the rays.

Final gather rays are traced like normal rays with one exception. When the ray hits a light source directly or indirectly through specular bounces, it should not be used to make a radiance estimate. In the first case the radiance is accounted for by direct light sampling, and in the second case the radiance is included in the caustics map. Note that this is only done in photo-realistic rendering.

The number of final gather rays can be several thousands, and most of the rendering time is therefore used in final gathering. Apart from the search radius heuristic, two well known techniques can optimize final gathering; the first tries to minimize the number of nearest neighbor searches and the last tries to reduce the number of final gathers [Suy02, 116ff]. **Irradiance pre-computation** works by precomputing irradiance in all the photon positions. Simply put, a radiance estimate is calculated by multiplying the precomputed irradiance from the nearest photon with the local BRDF. The running time can be decreased by as much as a factor of six. **Irradiance caching** works by computing the final gathering irradiance estimate for a

selected number of points in the scene only and to interpolate these result for all points in between[Suy02, 118] [Jen01, 140]. The speedup can be a factor of one-hundred or more.

## 3.3 Rendering and blending

In this section we briefly discuss a couple of alternative ways to render the illumination stored in the photon map. We shall see two different approaches to radiance reconstruction.

### 3.3.1 Rendering

The overall goal for all global illumination algorithms is to calculate the exact radiance in the direction of the viewer for each pixel. We cheat a lot and only calculate the radiance in each visible vertex and let the graphics hardware interpolate all intermediate radiance values. (In computer graphics this radiance is represented by an RGB vector and although its more than a color, its common just to talk about a pixel color even though we mean the radiance.)

To calculate the radiance for a single vertex we calculate the *irradiance estimate* (not the radiance estimate) using the photon map and include it in a formula together with the color of the texture. In some sense the color of the texture encodes the perfectly diffuse BRDF and Equation 3.2 gives a simple way to combine the BRDF and irradiance. In effect the irradiance estimates are used to generate a light map which is then transformed in a pixel shader as described in the next section.

The speedup of this hardware interpolation is without doubt enormous compared to estimating the radiance many times for each pixel as it is done in photo-realistic rendering. Other sources recognize that hardware interpolation can be used for rendering diffuse surfaces [DDM03, 57ff].

An interesting approach to hardware utilization can be found in [Kel97]. Keller generates a particle approximation of the diffuse radiance in the scene using a technique similar to **quasi-Monte Carlo integration** which is simply Monte Carlo ray-tracing using quasi-random sequences—see Section 3.4 for a discussion. The particles are much like photons since they propagate power. The graphics hardware renders an image with shadows where each particle is used as a point light source. Global illumination is obtained by summing up the single images in an accumulation buffer.

We can actually choose to use the photon map in two ways: we can choose to store the direct light in the photon map or not. In photo-realistic rendering the direct illumination is often excluded from the photon map and the light sources are explicitly sampled with ray-tracing. If the scene is dominated by indirect illumination, storing the direct illumination in the

photon map can be adequate [Jen01, 89]. In our implementation we have made it easy to leave out the direct light and enable OpenGL lighting instead.

### 3.3.2 Blending

The final color of a pixel depends on many factors. In the standard rendering pipeline there is a fixed number of settings to control how, for example, the color of a fragment is combined with the color of a texture and lighting information to create the final pixel color. A **pixel shader** is a *user programmable* replacement of this process. We will now see how a pixel shader can be used to let the hardware blend the irradiance estimates with the texture colors.

The first method is the one we presented in our last report [OK02, 74f]. The idea is to use the irradiance estimate to simulate everything from shadows to caustics. An irradiance estimate with the value $[0,0,0]^T$ is mapped to a fragment as $[-1,-1,-1]^T$ and should represent completely darkness and an irradiance estimate with value $[1,1,1]^T$ is mapped to $[1,1,1]^T$ and should represent completely white saturation. If we denote the value of an irradiance estimate with $\overrightarrow{C_f}$ (fragment color), the final pixel color is

$$\overrightarrow{C_p} = 2\overrightarrow{C_f} - [1,1,1]^T + \overrightarrow{C_t} \tag{3.8}$$

where $\overrightarrow{C_t}$ is the color of the texture. For each component of the RGB vector the range of the sub-expression $2C_f - 1$ is $[-1,1]$ which makes it possible to darken a white texture ($C_t = 1$) completely and to make a caustic on top a black texture ($C_t = 0$). This texture transformation is easily implemented in a standard pixel shader. Note that $C_f = 0.5$ gives the texture color.

While this method gives full flexibility, it also has some drawbacks. The most apparent problem is in saturated areas which appears completely white. Normally we should not allow irradiance estimates above $0.5$ unless there is a caustic (which we do not yet handle). In a test we tried to keep the estimates under $0.5$ by normalizing the estimates with the highest estimate from the previous frame. Unfortunately, there is always a few estimates that are high which makes the majority of the estimate too low. The visual result is that only a small area close to the light source is illuminated.

Since our last report we have narrowed our scope to exclude caustics and therefore a more conventional light map approach without the above problems can be taken. The conventional approach is to multiply the color of a light map with the color of the texture, that is,

$$C_p = C_f C_t \tag{3.9}$$

for each RGB component [WP01, 314] [Joh03, 316]. This formulation is better because it is physically plausible according to Equation 3.2. As stated

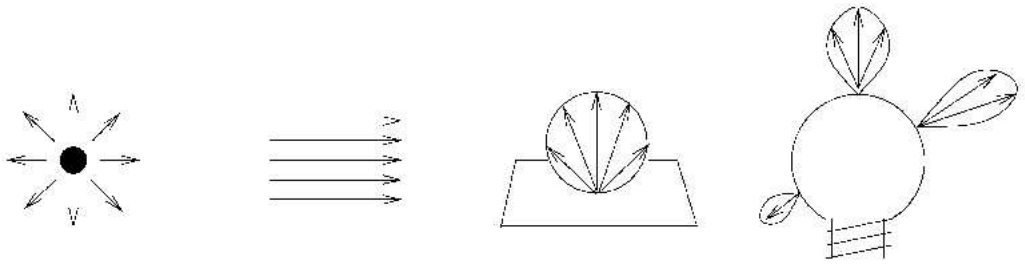


Figure 3.3: Light emission profiles for different types of light sources. [JCS01, 16]

in the previous section, $\overrightarrow{C_f}$ is the result of the (perhaps interpolated) irradiance estimate and $\overrightarrow{C_t}$ can be thought of as the Lambertian BRDF. The effect is that the texture color is decreased or perhaps unmodified which gave rise to the term **dark mapping** [Oud99, 15]. So we can still create shadow effects by this transformation whereas we have lost the ability to brighten geometry because the range of each component of $\overrightarrow{C_f}$ is normally clamped to $[0, 1]$. If there exist some way to extend this range, it might be possible to create white saturation.

## 3.4 Photon scattering

When photons are traced throughout the scene, it is important that the distribution of the photons in the photon map approximates the actual radiance distribution. In this section we first discuss the general methods used to scatter the photons and then we describe how they can be improved. Our primary aim is to generate directions for two purposes: a direction to emit the photon from a light source and a direction for diffuse scattering.

### 3.4.1 Simple scattering

The light sources we use are point light sources or spot lights. The emission profile for these and other types can be seen in Figure 3.3. For a diffuse point light source photons are normally shot randomly in all possible directions. To emit photons from a spherical light with a given radius we first pick a random position on the surface. Then we pick a random direction on the hemisphere above this point.

The hemisphere on the spherical light is sampled in the same manner as an ordinary diffuse reflection. Given two uniformly distributed random numbers $\xi_1 \in [0, 1]$ and $\xi_2 \in [0, 1]$ we find a random diffuse reflected direction as

$$\vec{\omega}_d = (\theta, \phi) = (\cos^{-1}(\sqrt{\xi_1}), 2\pi\xi_2) \tag{3.10}$$

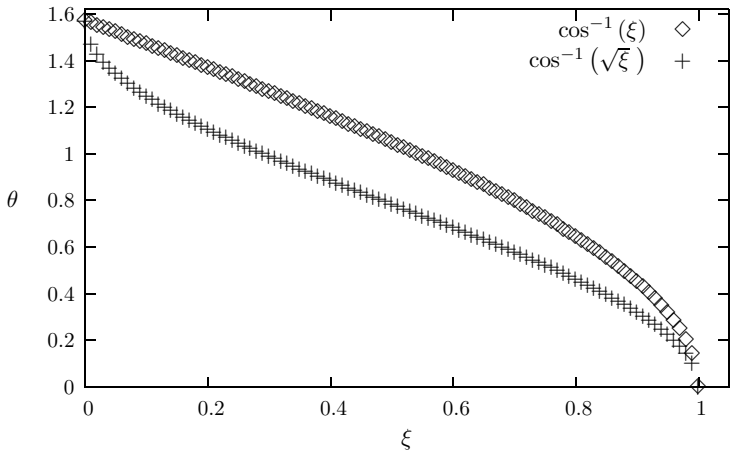



Figure 3.4: Different distributions of $\theta$. Notice how the square-root shifts the directions towards the $\xi$-axis. A steeper slope means that the angular density of $\theta$ is low; this is the case for small and large values of $\xi$. However, the surface area on the sphere is much smaller for small values of $\theta$.

using spherical coordinates [Jen01, 22]. A logical question is why is $\theta$ not computed as $\cos^{-1}(\xi_1)$? If so, the angular density of $\theta$ would be relatively high at angles away from the normal. By taking the square root we shift the higher density towards the normal. Figure 3.4 shows the two distributions. A relevant discussion can also be found in [II01].

It may also sound strange that diffuse scattering does not have uniform density over the hemisphere. This is simply because the receiver (whether another surface or the eye) sees the projected area of the light source [Jen01, 57]. If we could trace infinitely many photons, it would not be necessary to take the projected area into account because then a radiance estimate would be extremely accurate. But as long as we work with a finite and small photon map size we cannot rely on the geometric properties to account for the projected area. Using the inverse cosine can be seen as a way to emit more photons in the most important directions; the surfaces directly above a certain point should in general receive more light because they in general will be facing the tangent plane in the point. In final gathering we must also take this fact into account, but from another perspective. The directions of final gather rays should be generated using Equation 3.10.

For perfectly diffuse light sources we need a way to sample sphere directions uniformly. In this context **uniformly** means that the probability that the point is in a region depends only on the area of the region and not its location on the sphere. **Discrepancy** can be used to measure how uniform a distribution is [Shi91, 5]. Discrepancy provides a single number that indicates something about the overall quality of a set of sample points:

```
Vec3 direction;
do
{
  direction[0] = 2 * uniform_01() - 1;
  direction[1] = 2 * uniform_01() - 1;
  direction[2] = 2 * uniform_01() - 1;
}
while ( direction.length() > 1 );
```

Figure 3.5: Rejection sampling of a single direction in a unit sphere [Jen01, 57]. `uniform_01()` should return uniformly distributed random numbers $\in [0, 1]$.

```
for( int k = 1; k < N + 1; ++k )
{
  float h  = -1 + 2 * ( k - 1 ) / float( N - 1 );
  theta[k] = arccos( h );
  if( k == 1 or k == N )
    phi[k] = 0;
  else
    phi[k] = ( phi[ k - 1 ] + 3.6 / sqrt( N * ( 1 - h * h ) ) )
             mod ( 2 * pi );
}
```

Figure 3.6: The Saff-Kuijlaars method. The code shows how to generate $N$ uniformly distributed points on a sphere in polar coordinates [SK97].

a low discrepancy means that the distribution is very uniform whereas a high value means that the distribution is poorly uniform. (Several formal definitions of discrepancy are given in [Shi91] and [SKP98]).

The two standard techniques for sampling directions on a sphere are rejection sampling and explicit sampling. **Rejection sampling** works by generating random points inside the unit cube until the point is also within the unit sphere; the technique is shown in Figure 3.5. An elaborate discussion (albeit not a proof) of why this technique gives a uniform distribution can be found in [PVTF02, 294f]. **Explicit sampling** maps the random numbers to the surface of the sphere by for example randomly sampling the angles of a spherical mapping [Jen01, 57].

Other methods directly generates $N$ points on a sphere. We have implemented the method shown in Figure 3.6. In lack of a better name we call it the **Saff-Kuijlaars** method [SK97]. By inserting the minimum and maximum $k$ we can see that $h$ assumes discrete values in the interval $[-1, 1]$; this means $\theta$ lies in the range $[\pi, 0]$. The sampling of $\phi$ gives $0$ for the two extremes of $\theta$ which corresponds to the two poles on the sphere. In Figure 3.7 we can see how points are distributed with $N = 500$. The plot to the left unveils problems as nearly no points are present in a band around the sphere where $\phi$ is close to $0$ or $2\pi$.

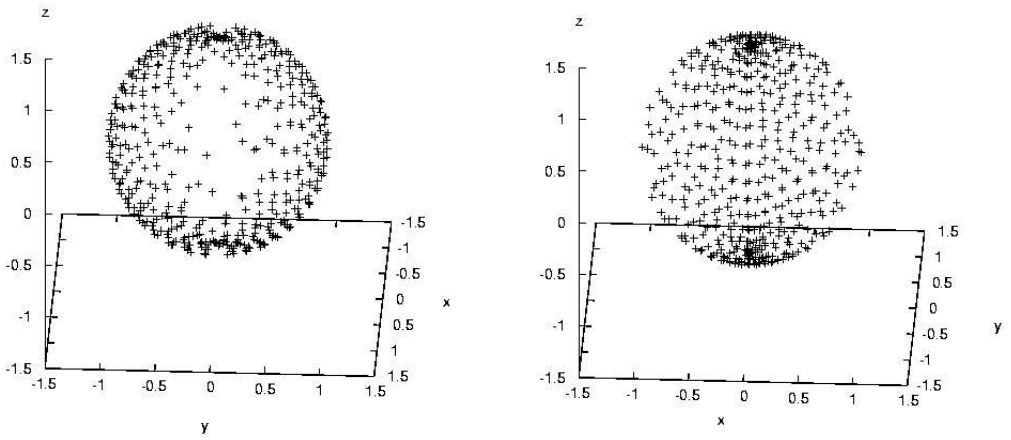



Figure 3.7: Points computed by the algorithm in Figure 3.6 with $N = 500$. Left: The method has problems around the $y = 0$ plane where $\phi$ is close to $0$ or $2\pi$. Right: The points seems evenly distributed in the other directions.

### 3.4.2 Improved scattering

Let us discuss how we can improve rejection sampling. In a real-time context it is not enough that the random number generator produces uniformly distributed numbers, but we must require that the discrepancy is low even for a small number of samples. This is a natural consequence of the relatively small amount of photons that we can afford to trace. There exist at least two solutions to this: stratified sampling and quasi-random sequences.

If we somehow can subdivide the surface of the sphere into patches having approximately the same area, we could simply pick a random sample within each patch. This process of spreading out the samples is called **stratified sampling** [Jen01, 155] [PVTF02, 321f]. We can include stratification into Equation 3.10:

$$\vec{\omega}_d = (\theta, \phi) = (\cos^{-1}\left(\sqrt{\frac{j-\xi_1}{M}}\right), 2\pi\frac{i-\xi_2}{N}), \quad j \in [1;M], i \in [1;N] \quad (3.11)$$

where $j, i, M$, and $N$ are integers. $M$ and $N$ defines the subdivision of the hemisphere, and although the subdivision is clearly not optimal (the surface patches have different area) it is much better than naive random sampling. One should always prefer to increase the number of patches rather than to use more samples within larger patches [Suy02, 20]. Geodesic dome constructions also provide a useful way to partition the sphere into relatively uniform patches (see [BB82, 492f]). One problem with the stratification is that we have to decide in advance how many samples that we need.

A **quasi-random sequence** is a sequence of $n$-tuples that fill $n$-space more uniformly than uncorrelated random points; several methods can be

found in [PVTF02, 313ff], [SKP98, 4], and [KK02]. The sequences exhibit two important properties:

1. The points can be generated on demand so that we do need to know a priori how many points we want.

2. The points have a very low discrepancy.

Despite their name, there is nothing random about these sequences. If we use quasi-random numbers when using rejection sampling or diffuse sampling of the hemisphere (Equation 3.10), the result should include implicit stratification [Jen01, 148]. It is important to remember that each coordinate is generated by a different quasi-random sequence [PVTF02, 316].

Empirical studies suggests that at least five times as quick convergence can be achieved with sampling based on quasi-random sequences compared to random sampling, and it can be significantly better in Lambertian scenes [PVTF02, 318f]. For some very accurate purposes, a little real pseudo-randomness should be added to the sequences to avoid patterns in caustics [Jen01, 148].

## 3.5 Bias reduction

The traditional irradiance estimate is done by collecting photons within a sphere which can lead to surface and distance leaking. As seen on Figure 3.9, surface leaking especially happens at edges and corners. Before we discuss these two problems, we briefly describe how distance bias can be avoided.

Distance bias is particular noticeable on surfaces with radiance discontinuities. Radiance discontinuities cause a sharp change in the density of photons. In the case that the real radiance drops close to zero, the reconstructed radiance falls off as $1/r^2$ where $r$ is the distance to the radiance discontinuity [Suy02, 113].

Filtering is the traditional technique used to sharpen caustics [JCS01, 32], but we expect that the technique can sharpen shadow boundaries as well. A **filter** assigns weights to each photons in the estimate whereby some photons will contribute more than others in the final estimate. A simple weighting criteria is for example the distance from the photons to the estimation point. We shall not investigate filtering further; different methods can be found in [Jen01, 80ff] and [Suy02, 113f].

### 3.5.1 Surface bias

We will now examine four different approaches that can be used to avoid surface leaking by detecting and removing leaked photons before they are

collected. One might consider postponing this filtering process until the $k$ nearest photons have been found, but by experiments we found that it makes the irradiance estimate very unstable. The simplest and least effective method is to use the incoming direction of the photon (1). The more complicated solutions exchange the sphere with other geometric primitives. The simplest solution here is to use a sphere slice (2), a cylinder is only slightly more complicated (3) whereas an ellipsoid is the most complicated (4). No matter what scheme that is used, it must be relatively fast since it will be used heavily during nearest neighbor searching.

A simple operation used in the following is the projection of a vector onto another. The **projection** of $\vec{w}$ onto $\vec{v}$ is a new vector $\vec{p}_{w,v}$ in the direction of $\vec{v}$:

$$\vec{p}_{w,v} = \vec{u}_v |\vec{w}| \cos\theta = \vec{u}_v \frac{\vec{v} \cdot \vec{w}}{|\vec{v}|} \tag{3.12}$$

where $\vec{u}_v$ is a unit vector in the direction of $\vec{v}$.

The incoming direction of the photon is used to exclude photons from the backside of thin surface. The test is run after the photon is known to be within the sphere. The method works like hidden surface removal by computing the dot product between the normal and the incoming direction: if the result is negative, the angle between the two vectors must be more than 90 degrees and the photon must belong to the opposite hemisphere. While this method is cheap to calculate, it cannot exclude many photons at orthogonal surfaces in corners and at edges.

We have instead added an extra check to the standard sphere approach. In Figure 3.8 the implementation of the method is shown. Besides being inside a sphere we further require a photon to be within a small distance in the direction of the surface normal. This is done by projecting the vector from the search location to the position of the photon onto the normal. Since we know that `search_location_normal` is normalized, Equation 3.12 tells us that we get the length directly by a dot product. Conceptually this makes us search only in a small slice of the sphere. The gain is that we remove almost all surface leaking. Unless the scene contains very thin objects we can omit the incident direction check because the slice is just made thinner than the thinnest object. This also means that we can leave out the incident direction in the photon data structure.

It often is suggested to use a cylinder instead [JCS01, 32]. The result of the filtering will be similar to the sphere slice, but it will require a few more instructions. Instead of calculating the length of the projection, one should calculate the projection of `to_project` onto the normal (cf Figure 3.8). Then one should create the vector from the projection to the position of the photon. If the length of this vector is less than the radius of the cylinder, the photon must be within the cylinder.

```
bool check_inclusion( float squared_dist, const Photon& p ) const
{
  assert( equal( search_loaction_normal.length(), 1 ) );
  bool in_sphere = squared_dist < max_squared_radius;
  if ( not in_sphere )
    return false;

  Vec3  to_project       = p.position() - search_location;
  float projected_length = to_project * search_location_normal;
  bool  in_slice = projected_length < sphere_slice_size and
                   projected_length > -sphere_slice_size;
  return in_slice;
}
```

Figure 3.8: Algorithm for determining if the photon lies within a thin slice in the sphere. `operator*()` computes the dot product.

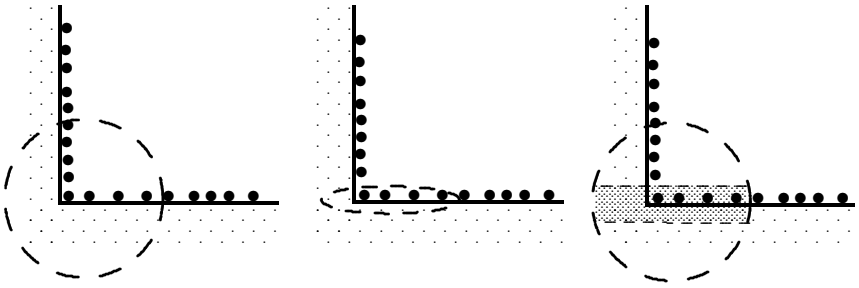


Figure 3.9: Using the sphere to locate photons in corners can include wrong photons. An ellipsoid is a poor choice. Using a slice of a sphere is fast an includes approximately the same photons as an ellipsoid.

An ellipsoid is also mentioned as an alternative volume [Jen01, 79]. To check if a photon is included, we consider the points of an ellipsoid with center $(x_0, y_0, z_0)$ and semi-axes $a, b, c$:

$$\frac{(x-x_0)^2}{a^2} + \frac{(y-y_0)^2}{b^2} + \frac{(z-z_0)^2}{c^2} \leq 1 \quad . \tag{3.13}$$

A photon shall contribute to the irradiance estimate if the inequality holds for its position. While this computation is relatively simply it will only work for axis-aligned ellipsoids; if that is not the case the test will be far more expensive. Therefore the ellipsoid is a poor candidate.

### 3.5.2 Boundary bias

In photo-realistic rendering boundary bias is not a serious problem since it can be masked by final gathering and since the maximum search radius

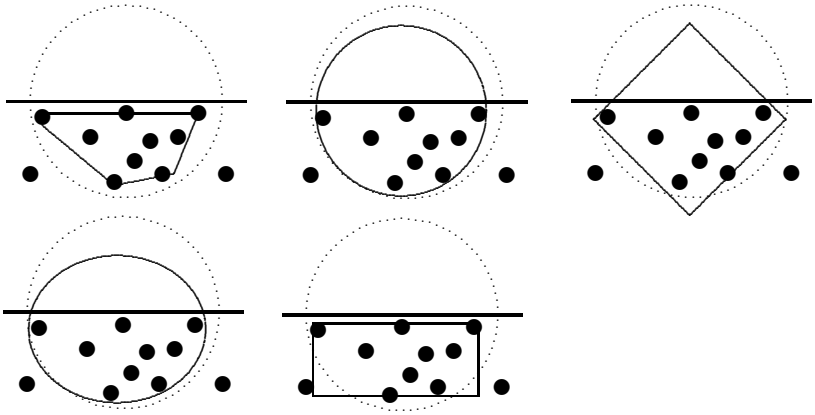


Figure 3.10: Comparison of different area estimation techniques. Starting from upper left corner: A convex hull, a bounding sphere, an axis-aligned bounding box, an oriented bounding ellipsoid, and an oriented bounding box. The convex hull gives the best results.

can be made much smaller than in real-time rendering. In real-time photon mapping, on the other hand, it is quite apparent. Basically the problem stems from a wrong area estimate near edges and corners, because up to $\frac{3}{4}$ of the search volume will reach out into open space (in a three-way corner). If we had access to geometrical description of the scene, it would be possible to compute the exact area [HP01]. Unfortunately, we only have a list of polygons from which it is hard to extract such a description.

What we can do is to fit some kind of bounding volume around the photons that we intend to use for the irradiance estimate. Figure 3.10 compares several bounding volumes. The simplest solution is to use axis-aligned bounding volumes. While this works fine on axis-aligned geometry it fails miserably on other surfaces (we tried it). To create oriented bounding volumes it will be necessary to conduct a principal component analysis. A **principal component analysis** is a statistical method used to find the axis that is naturally aligned to a set of points [Len02, 183ff]. This procedure requires computing the roots of a cubic polynomial and solving a homogeneous linear system for each of the roots.

A well-know method is to calculate a convex hull of the points in the estimate. We have not investigated how expensive it will be, but first the points need to be projected down into the plane defined by the normal. Then the convex hull algorithm can begin.

## 3.6 Summary

In Section 3.1 we stated that photon mapping consists of photon tracing and rendering. A key optimization is to distribute photon tracing over several frames. Furthermore, we only handle diffuse surfaces, make diffuse reflections by splitting a photon into several new photons, and rely on hardware interpolation of irradiance values between vertices. Furthermore, we do not support caustics or volume rendering so separate maps for these effects are not used. To compute the irradiance estimate it is customary to use a sphere to locate photons in. This makes it possible for photons to leak into the estimate.

As described in Section 3.2, a simple heuristic can be used to specify a fairly scene independent search radius, reduce distance bias, and perhaps make the radiance estimate faster to compute. Photon mapping can be extended with final gathering; at the expense of more rendering time, the results can be much better.

Section 3.3 described how the illumination stored in the photon map can be blended with the texture. Graphics hardware is used to render diffuse illumination via vertex color interpolation and this is one of the key optimization of several real-time techniques. The first blending approach makes it possible to visualize colors from black to white whereas the second allows only from black to the texture color.

Section 3.4 explained how photons should be emitted from different light sources and how diffuse scattering should be done. To get a good distribution of photons throughout the scene we rely on either stratification of the photon directions or quasi-random sequences.

Several techniques used to reduce bias in the irradiance estimate was explained in Section 3.5. Filtering can remove distance bias. Surface bias can be removed effectively by using a sphere slice instead of a sphere when gathering radiance; other approaches give similar results, but are more expensive to compute. Boundary bias can be resolved by computing a more accurate area estimate. To use bounding volumes effectively, it will be necessary with a costly principal axis analysis. The most accurate solution would be to create a convex hull. We believe that both of these solutions will be far to expensive for real-time photon mapping.

# 4 Implementation overview

*If you can't write it down in English, you can't code it.*
—Peter Halpern, Brooklyn, New York

A real-time graphics engine is a complex piece of software with many non-trivial responsibilities [OK02, 2f]. In this respect it is important that photon mapping can be easily integrated with existing engines. Therefore this chapter presents the status of our implementation with focus on the integration of real-time photon mapping.

We start with an introduction to the implementation and its most important modules. Then we show the core flow of control to give an overview of what takes place when rendering a single frame. A presentation of the many parameters of the photon mapping algorithm follows. In the end we describe how we test and introduce the three test scenes that we will use in the coming chapters.

## 4.1 Engine overview

We first describe the modules that are partly or completely implemented. The modules are listed below with a description of their desired functionality and status.

- **Agent:** An **autonomous agent** (or **agent**) is simply a computer controlled character. Different agents should reside in this module together with functionality to control them and their artificial intelligence. Currently only the functionality for controlling the player or observer is implemented and agents are restricted to follow a predefined path.

- **BSP:** Loading of scenes stored in Quake 3's BSP tree format is handled in this module. BSP tree stands for **binary space partitioning tree** and refers to the fact that each node in the tree splits the remaining geometry into two subtrees according to an arbitrary **sorting plane**—each child contains everything on a particular side of the sorting plane [Len02, 202f]. BSP trees are used in 3D engines to accelerate hidden surface removal, to ensure correct back-to-front rendering [Oud99, 10] and to accelerate spatial queries like intersection testing [Cha01, 39ff]. In this module extra functionality such as mesh refinement is also included. The resulting BSP trees are just like normal Quake 3 trees which have few polygons in the leaves.

- **Contrib:** This module stores the libraries we have not coded ourselves. The most important library is Open Scene Graph (OSG) which we have built everything on top of. OSG is a scene graph framework with good culling performance and easy integration of pixel shaders.

- **Input/Output:** All external communication is handled in this module. It includes configuration file read/write, keyboard/mouse interaction and display configuration.

- **Math:** Common mathematical primitives such as matrices, vectors and planes as well as functions on these.

- **Photon mapping:** The photon mapping functionality resides in this module. This important module will be described thoroughly in this and the next chapter.

- **Scene:** Camera and scene management is implemented as a thin wrapper around the Open Scene Graph library. This is were the general scene graph rendering is combined with our photon mapping functionality.

- **Shader:** Collection of different pixel shader programs. Most important is the program used to combine texture and fragment color.

Many other modules exist in a normal engine, but time constraints means that we have to do with a minimal engine. Some of the larger modules that almost certainly exists in normal engines are kinematics, physics simulation and artificial intelligence [WP01] [Len02] [Rab02]. What is quite important is that these modules should not affect the integration of photon mapping. The only impact such modules should have is performance related.

## 4.2 Open Scene Graph and BSP trees

We will now give an overview of the functionality we use from Open Scene Graph to re-build Quake's BSP trees. The nodes in a scene graph can be divided into grouping nodes and drawable nodes. **Grouping nodes** are nodes that can contain other grouping or drawable nodes. This kind of class hierarchy is an example of the **composite pattern** [GHJV94, 163ff].

**Drawable nodes** can contain one or more primitives. A **primitive** encapsulates one of the `DrawArrays` or `DrawElements` classes which are thin wrappers around OpenGL functions. It is worth noticing that OSG only generates bounding volumes around grouping nodes and that we only use the BSP tree structure to build the scene graph. OSG will then automatically benefit from the spatially sorted geometry when maintaining a bounding volume hierarchy for the scene. A **bounding volume hierarchy** is a rooted tree where each node contains a bounding volume of its children [Cha01, 30ff]. The actual naming of the classes representing the nodes can be seen in Table 4.1.

| Sub-class | Super-class | Description |
|---|---|---|
| `Group` | `Node` | General group/internal node |
| `Geode` | `Node` | Leaf node for grouping drawables |
| `Geometry` | `Drawable` | Node for grouping Primitives |
| `DrawArrays` | `Primitive` | Primitives for array data |
| `DrawElements<type>` | `Primitive` | Primitives for indices in array data |

Table 4.1: Some of the Open Scene Graph classes for building scene trees

Open Scene Graph does culling down to and including the `Drawable` level. This means that if a single triangle of a `Drawable` is in the view frustum, then all triangles from that `Drawable` will be drawn. On the other hand, if the bounding volume of a grouping node is outside the view frustum then all its descendents can be culled immediately.

When the geometry in a BSP tree is refined we extend the array data in the primitives. This leads to `Drawable`s with more triangles, but not in more `Drawable`s. This will in turn lead to more unculled triangles outside of the view frustum. As a result we will make more irradiance estimates than necessary. The best solution would be to rebuild the tree after it is refined although we have not implemented this scheme.

## 4.3 Control flow

In this section we present an overview of the control flow of the program. This will show how an engine that incorporates photon mapping needs to be different from a normal engine. We will use a top-down approach where

```
1   program pie
    {
      setup_scene();

      while( true )
6     {
        update_scene();
        cull_scene();
        draw_scene();
      }
11  )
```

```
function setup_scene()
{
   turn_off_opengl_light();
   load_bsp_level();
   refine_geometry();

   for_each( vertex_array )
      expand_color_array();

   for_each( texture )
      calculate_bleeding_color();
}
```

Figure 4.1: Left: The universal render loop. Right: The modified initialization.

each function is expanded separately and the new functionality is written with a red font.

At the outermost level the engine looks like any other (see Figure 4.1 on the left). However, the scene needs to go through several preprocessing steps which is shown in Figure 4.1 on the right. In line 3 we turn off OpenGL lighting since we normally replace all lighting with a custom pixel shader. All lighting is controlled by the photon map and the photon dispersion, and we do not want OpenGL light to interfere. In line 4-5 the scene is loaded and a second copy of the scene is made. One of the scenes is refined to make that scene fit for rendering (see Section 5.4). This highly detailed scene is necessary to use the irradiance estimates properly—or else the interpolation distance between vertices will be too big. The low-detail scene is kept for intersection testing. In line 7-8 we ensure that there is a one-to-one correspondence between a vertex and its color so we can set the color of a vertex during the draw stage. This expansion need only take place in the detailed scene. In line 10-11 the bleeding color of each texture is calculated. During photon tracing we need to know what color e.g. a wall will bleed with. Here we simply compute the average color of the texture. This is of course a very simple approximation to the BRDF, but under the assumption of a completely diffuse environment it should be reasonable. This feature is not implemented yet, but for testing purposes we can manually add a bleeding color to an object.

Ideally all changes to the scene graph should happen during the update stage. In Figure 4.2 on the left the new version of the update stage is shown. The first part is not interesting since it is standard functionality which takes care of transforming dynamic objects in the scene. However, `emit_photons()` is interesting because it effectively encapsulates all photon tracing, and it is explained in detail in Section 5.2. Note that the update of the scene cannot happen after photon emission since photons can be invalidated by dynamic changes.

```
function update_scene()
{
  for_each( node )
    update();

  for_each( light source )
    emit_photons();
}
```

```
function draw_scene()
{
  for_each( unculled drawable )
    for_each( vertex )
      color =
        irradiance_estimate( vertex );
    draw();
}
```

Figure 4.2: Left: The new update function. Right: The new draw stage.

In `cull_scene()` we do nothing special whereas `draw_scene()` has been modified (see Figure 4.2 on the right). Normally the draw stage does not modify the scene tree due to potential multi-threading issues, but for our purpose it is essential that irradiance estimates are made after culling to avoid estimates at vertices outside of the view frustum. The nested loop is the new functionality which updates the color of the vertices. Again, this update should only happen to the detailed scene.

## 4.4 Customizing photon mapping

In this section we give an overview of the many parameters that can be changed in the application. By giving an overview of the these parameters we hope it becomes easier for the reader to comprehend when they are mentioned in the following chapters. Chapter 6 describes the parameters in detail. The parameters fall in three main categories:

1. Scene specific: parameters that control the detail of the scene and the format of the scene graph.
2. Photon map specific: parameters that determine the size of the photon map and behavior of the balancing and the irradiance estimate.
3. Photon tracing specific: parameters that control when photons should be stored and how the diffuse scattering should be done.
4. Miscellaneous: This category includes parameters like enabling the maximum search radius heuristic and whether final gathering should be used.

Common for all these parameters are that they are specified in a configuration file and can therefore be modified without recompiling the application.

Figure 4.3 shows how some of the parameters are used to determine the size of the photon map. A short explanation to some of the parameters follow. `store_direct_light` is a boolean flag that controls if photons should be stored at the first reflection. `diffuse_reflections` is

```
int Photon_map::photons_reflected( int level ) const
{
  if( level == 0 ) return 0;
  return photons_from_light * pow( diffuse_reflections, level ) +
         photons_reflected( level - 1 );
}

int Photon_map::size() const
{
  int direct    = store_direct_light * photons_from_light;
  int indirect  = photons_reflected( recursion_depth );
  return ( indirect + direct ) * snowball_size * accumulation_frames;
}
```

Figure 4.3: Algorithm that determines the size of the photon map. `photons_reflected()` calculates the number of reflected photons for `level` reflections.

the number of new photons a single photon is split up into during a reflection. `recursion_depth` describes the number of diffuse reflections. `snowball_size` is used to store several photons in the photon map even though only one photon arrived at some surface; we call this for the **snowball feature**. `accumulation_frames` is simply the number of frames that photon emission should be distributed over.

## 4.5 Testing

In this section we first discuss our view on how testing should carried out. The following chapters will repeatedly refer to tests that we have made and assessed. It would be unfortunate if we did not describe the premises for those tests to the reader. In the end the individual test scenes are presented.

### 4.5.1 Test strategy

The first one should know is the hardware platform. All tests were run on a Pentium IV 3 GHz CPU with 512 MB DDR RAM. The graphics card is a GeForce 4 Ti4200 with 128 MB DDR RAM. The CPU has 512 KB Level 2 cache.

Second, one should know how we test in general. At the top level there is two kinds of tests: performance tests and visual quality tests. Performance tests are also of two kinds: isolated tests and non-isolated tests.

An **isolated test** only runs the feature under test and the test must be repeated a decent number of times. If the feature runs very fast, the test should run for at least a second a couple of times to remove possible OS overhead. A **context dependent test** tests a feature when it runs as an integrated part of the whole application. The test is done by exchanging an old

implementation of a feature with a new and keeping everything else the same. The test must clearly explain what state the rest of the application runs in if it could affect the evaluation of the test.

Whenever it is possible, one should prefer an isolated test. A context dependent test might be important if we need to know if the new optimized feature was worth the trouble. The new feature might be much faster, but it might not have a great impact on overall performance. However, the best way to determine in advance if some feature is a performance bottleneck is to use a profiling program.

Sadly the visual quality test often relies on subjective assessments, but the reader can decide for himself by running the binaries; they can be downloaded from `http://www.cs.auc.dk:/~nesotto/pie/`. In a couple of cases we have generated a photo-realistic rendering of the same scene which makes it easier to assess our real-time generated images.

To remove any ambiguities regarding the description of performance tests we always give the relative running time with the original time as index $1$. For example, if an old test takes $2$ seconds and the new test takes $1.5$ seconds we say the new running time is $0.75$ or $75\%$ of the original. We might also express this as the new test is $25\%$ faster or that the running time has been reduced by $25\%$. We refrain from the opposite comparison, that is, we never say that the old test is $33\%$ slower. Furthermore, it might happen that the new running time is $1.10$ or $110\%$ of the original and we say that the new test is $10\%$ slower.

### 4.5.2 Test scenes

We have chosen three test scenes which will be used for different tests. This section shows how the test scenes look like and explains what we are going to test.

For the photo-realistic rendering we use RenderPark—an open source test-bed system for physically based photo-realistic rendering. RenderPark provides implementation of a wide variety of state-of-the-art ray-tracing and radiosity algorithms [BdLPM].

There are some limitations to the scene complexity that the reader should be aware of. Currently we cannot use more than one light source. This is mainly to keep experimentation simple until satisfactory result have been achieved. Implementation wise it would be trivial to extend this limitation. Let us discuss the scenes:

1. **The Cornell box**: Although geometrically simple, the Cornell box can still be useful for making some visual quality tests. The simplicity allows us to convert the scene data to VRML which can be loaded by RenderPark. In Figure 4.4 we can see a rendering of the box without

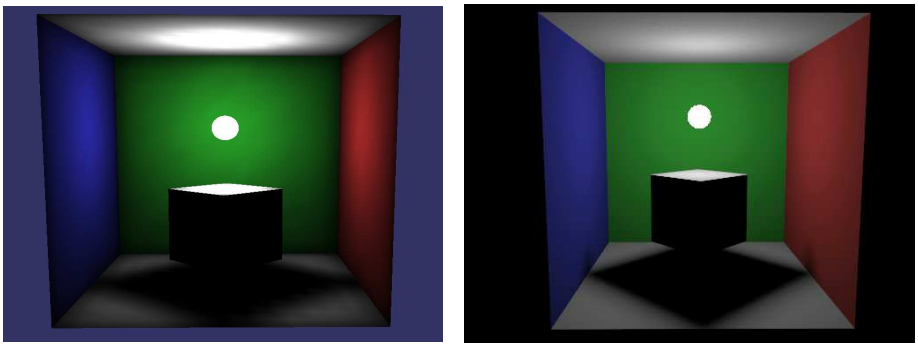


Figure 4.4: Left: A real-time rendering using $20,000$ photons. Right: A RenderPark rendering of the same scene.

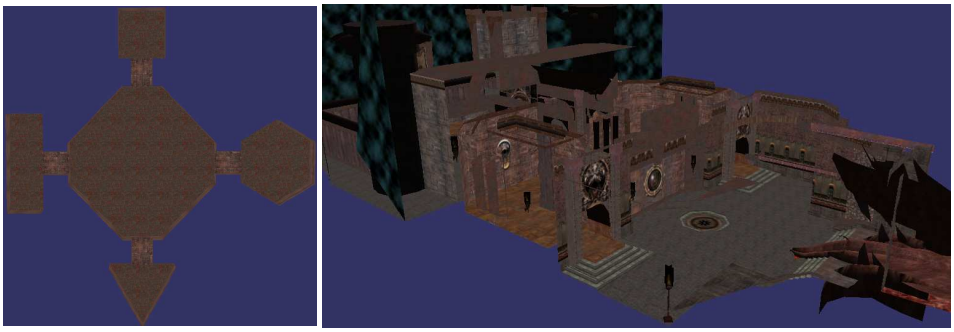


Figure 4.5: Left: View of the small Quake 3 test scene through the floor. Right: Overview of the normally sized Quake 3 level.

indirect light. Ignoring the light source, the scene geometry consists of 48 vertices.

2. **The small Quake level**: As the second test scene we will use a simple scene from Quake 3. With this level we can test all features of our implementation while maintaining an overview of the whole scene.

   As shown in Figure 4.5 on the left, the scene consists of five rooms: Four small rooms each connected to one larger room. In total there is 1107 vertices in the scene.

3. **The normal Quake level**: The third scene is a large level with many rooms. Most of the scene can be seen in Figure 4.5 on the right. It will be interesting to see how the $\pi$-engine performs on this realistically sized scene. The scene contains 16,065 vertices.

Ideally it would have been great to have a photo-realistic rendering of the two Quake scenes, but that task is complicated by the need for a RenderPark loader of the scenes.

## 4.6 Summary

In Section 4.1 we saw an overview of the modules in the engine. The BSP-loader can load Quake 3 levels and the use of the BSP structure is expected to improve the performance of intersection testing. Two other important modules are the photon mapping module and the scene module.

The scene graph node structure of OSG was explored in Section 4.2. Grouping nodes was used to make the inner nodes of the scene tree whereas primitive nodes encapsulated the OpenGL geometry itself. OSG will automatically maintain a bounding volume hierarchy based on the bounding volumes of grouping nodes.

Section 4.3 described the changes to the traditional rendering loop that are necessary to accommodate for photon mapping. At load time a detailed copy of the scene must be generated to allow for better irradiance interpolation; the low-detail scene is used for intersection testing only. In the detailed scene we also expand the color arrays to hold the irradiance estimates as the draw stage needs to calculate the irradiance estimate for each visible vertex. It is very important to calculate the vertex colors in the draw stage instead of the update stage. This way the irradiance estimate only has to be calculated for vertices within the view frustum.

As described in Section 4.4, the $\pi$-engine can be tweaked by many parameters which can be modified without recompilation. This provides effective means for experimenting. The most important parameters control the complexity of the scene, the way photons are stored and how the irradiance estimates are calculated.

Section 4.5 described how we intend to test performance and visual quality. Performance tests should preferably be isolated. Visual quality tests will to some extend rely on subjective assessments. Performance optimizations should be motivated by profiling data. We will use three scenes for testing purposes.

# 5 Implementation details

*Get your data structures correct first, and the rest of the program will write itself.*
—David Jones, Assen, The Netherlands

*When in doubt, use brute force.*
—Ken Thompson, Bell Labs

We have now seen how the $\pi$-engine works on a high abstraction level. In this chapter we describe in detail the core parts of the engine. Ideally the dependency between the scene module and the photon mapping module should be as low as possible. A low dependency will make it easier to integrate photon mapping with existing engines. As we will see in this chapter, it is indeed possible to make a design with few dependencies.

We begin with a high-level description of the scene and photon mapping modules illustrating how the different classes cooperate. Afterwards we review photon tracing and the photon map more thoroughly. In the end we discuss our use of the BSP scene-format from Quake 3. We discuss design tradeoffs, optimizations, and introduce configuration parameters along the way.

## 5.1 Class overview

We start with the small, but important photon data structure. In Section 3.1 we saw how the photon is used. In Figure 5.1 the actual photon data structure is shown. The reader should notice that the photon has a position, a power and a sort axis. The sort axis is set during balancing and used during nearest neighbor search to guide the search. The incoming direction

```
enum Axis { x, y, z };
class Photon
{
public: // foundation
  Photon();
  Photon( const Vec3& position, const Vec3& power,
          const Vec3& direction = Vec3() );
public: // inspectors
  const Vec3&  position() const;
  const Vec3&  power() const;
  const Vec3&  direction() const;
  Axis         sort_axis() const;
  float        sort_coordinate() const;
public: // modifiers
  void  scale_power( float factor );
  void  set_sort_axis( Axis );
private:
  Vec3  position;
  Vec3  power;
  Vec3  direction;
  Axis  sort_axis;
};
```

```
struct Photon {
  float pos[3];
  short plane;
  unsigned char
      theta, phi;
  float power[3];
};
```

Figure 5.1: The photon data structure. Left: The $\pi$-engine version. Right: Jensen's version [Jen01, 158]. The size of our class is $40$ bytes compared to $28$ bytes. In short, we favor simplicity, encapsulation, and speed whereas Jensen favors size.

of the photon is optional as seen by the default argument of the second constructor. As explained in Section 3.5, we can probably omit the direction if we use a sphere slice to locate photon in.

Compared to Jensen's structure we use more space (40 vs. 28 bytes). The main difference is the representation of the incoming direction where we use Cartesian coordinates instead of spherical coordinates. By using Cartesian coordinates we save expensive trigonometric operations used to convert between the two coordinate systems. This overhead can however be lowered by pre-computing look-up tables for the trigonometric functions.

The scene module contains functionality related to loading of the scene, light sources and management of the two different sets of geometry (as previously mentioned: one set for intersection testing and one for rendering).

The scene module hands over the sparse model and light sources to the photon mapping module—Figure 5.2 shows the relationship between the classes. The `Photon_tracer` class scatters photons throughout the scene and stores them in a `Photon_map`. The `Photon_tracer` makes use of three other classes:

- `Distribution`: Is used to generate directions on a sphere or hemisphere. To generate a direction on a hemisphere, `direction()` must be passed a normal vector. In standard object oriented manner we have implemented different strategies (see Section 3.4 for details).

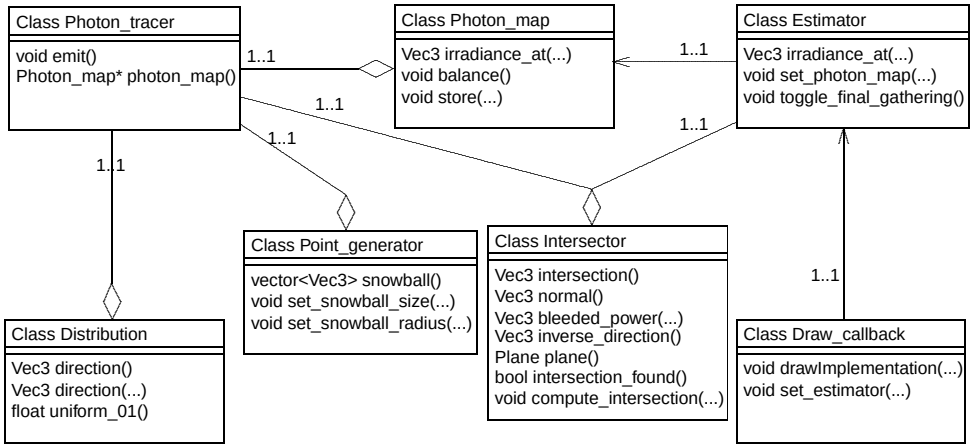



Figure 5.2: A simplified class diagram of the photon mapping module. If a rhomb is attached to a class, it means that a variable of that type is owned by the classes at other end of the line—for example, a `Photon_tracer` **aggregates** one `Distribution`. The numbers on the line indicates the multiplicity of the aggregation [Obj03, 91ff]. A line with an arrow means that the class—from which the line begins—is **associated** with an instance of the other class. In practice this means that it stores a pointer to the other class. An ellipsis denotes that arguments have been suppressed. At the implementation level most of these classes are abstract base classes in a class hierarchy.

- `Point_generator`: Is used to distribute several points in the intersection plane for use with the snowball feature. Currently we only use a random distribution of points, but stratified distributions can easily be added.

- `Intersector`: Is used to compute intersections and find the bleeding color at the intersection point. The class also keeps track of various information regarding the last intersection like normal and intersection plane.

The `Estimator` class stores a pointer to the `Photon_map` and uses the map to generate and cache irradiance estimates on demand. It also stores an instance of the `Intersector` class which is used during final gathering.

Finally, the `Draw_callback` stores a pointer to an `Estimator` to forward irradiance estimate requests to it. The `Draw_callback` class is installed in the scene tree and acts as the link between the photon mapping module and the scene module. The scene module applies `drawImplementation()` on each `Drawable` in the scene which in turn applies `Estimator::irradiance_at()` on each vertex.

## 5.2 The photon tracer class

The `Photon_tracer` is important enough to deserve a more thorough explanation. In Figure 4.2 on the left we saw how the `Photon_tracer` fits into the update stage of the rendering loop. In this section we explain the photon tracing and describe a technique that can reduce fluctuation and improve performance.

### 5.2.1 Photon tracing

Figure 5.3 shows detailed pseudo code for `emit()` and an explanation follows. At the top level each photon is emitted from the light source. The amount of emitted photons depends on the configuration file variable `photons_from_light`. If an intersection is found, additional reflections are traced until the maximum recursion depth. In the end the map is re-balanced (see Section 5.3.2 for details).

In `trace_photons_from_light()` the light source is queried for its position and a photon direction. From these two parameters we construct a line and the intersection of this line with the scene geometry is computed. In case an intersection is found, we save the photon if we want to include direct light. We do not call `Photon_map::store()` directly since we might generate several points from a single intersection using a `Point_generator`.

In `trace_photon_reflections()` we first calculate the new start point for the reflection. We cannot reuse the old intersection point since then we will get that intersection once again. The solution is to step a tiny fraction back in the direction the ray came from. Originally we went in the direction of the normal, but this lead to loss of photons in narrow corners because we simply went outside the scene geometry (see Figure 5.4). To calculate the next intersection we need to generate a reflected direction. This is done by calling `direction()` with the normal of the intersection; the normal is used to ensure we generate a direction on the hemisphere above the point of reflection. If we could find an intersection, we save the photon and trace the photon recursively.

### 5.2.2 Frame-coherent random numbers

To use **frame-coherent random numbers** means to reuse the random numbers that defined the photon directions over several frames. Simply put this means that many photon paths will be exactly the same; only those photons that are affected by moving objects will get a different photon path. The benefit of this method is twofold: (1) fluctuation can be reduced and (2) directions can be pre-generated. We discuss both aspects in the following.

```
void Photon_tracer::emit()
{
  for_each( photon )
  {
    trace_photon_from_light();
    if( intersector.intersection_found() and max_recursion_depth > 0 )
      trace_photon_reflections( max_recursion_depth );
  }
  map.balance()
}
```

```
void Photon_tracer::trace_photons_from_light()
{
  Vec3 begin = light.world_position();
  Vec3 dir   = light.trace_direction();
  intersector.compute_intersection( begin, dir );
  if ( should_store_direct_light and intersector.intersection_found() )
    save_photon();
}
```

```
void Photon_tracer::trace_photon_reflections( int depth )
{
  Vec3 begin = moved_point( intersector.intersection(),
                            intersector.inverse_direction() );
  for_each( diffuse reflection )
  {
    Vec3 dir = distribution.direction( intersector.normal() );
    intersector.compute_intersection( begin, dir );
    if( not intersector.intersection_found() )
      continue;
    save_photon();
    if ( depth - 1 > 0 )
      trace_photon_reflections( depth - 1 );
  }
}
```

Figure 5.3: Pseudo code for photon emission. Apart from handling of photon power this is almost the real code.

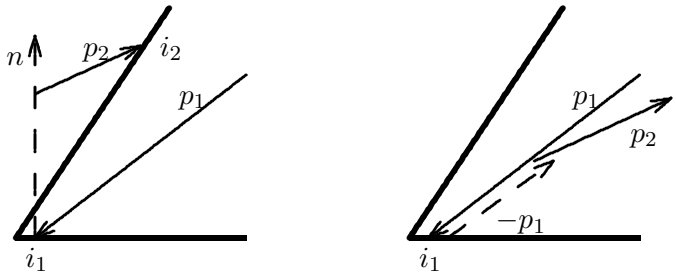



Figure 5.4: To avoid intersection with geometry in the start point we must move the start point away from the geometry. Left: Moving in the normal direction yields problems in corners. Right: Moving back along the incident direction is safe.

We can categorize animations based on their complexity. The simplest animation is when only the camera moves. In this situation we can reuse the photon map from the previous frame without shooting new photons if we assume that the camera is not connected to an occluding object—e.g., the camera might be attached to an avatar. The other extreme is when all light sources and objects are moving. This invalidates almost all photons in the map. However, none of these are the common case. It is often the case that only a few objects move in an otherwise static scene. When this is the case, it is advantageous to use the same photon path for the photons that are not influenced by the moving objects.

Christensen suggests an implementation that promises a $90\%$ reduction in fluctuation [JCS01, 91ff]. He uses a differently seeded random number generator for each emitted photon where the seed could simply be the number of the photon.

We have taken this idea further so it also improves performance. We observed that the generation of directions consumed as much as half of the time it took to render a frame. We observed that the program was $25$-$50\%$ faster by pre-calculating all the directions needed for a program run.

This is done by adding restrictions to the photon paths. In particular, we do not allow photon paths of different lengths. If for example all photons are reflected twice, two directions will be calculated for *each* photon with respect to a predefined reference normal. When a surface is hit, the reflected direction is found by rotating the pre-calculated direction. Notice that we always need to make this rotation, but we have saved the generation of directions which requires many expensive calls to the random number generator.

To remove the restriction of equal photon path lengths, we can calculate the number of direction that would have been used on a particular path. Then we can simple throw away this amount of directions.

## 5.3 The real-time photon map class

In this section we summarize how a real-time photon map is developed. We present the extra functionality that needs to be incorporated and we describe details of the balancing. Afterwards the irradiance estimate is discussed and we evaluate the benefits of using several smaller photon maps.

### 5.3.1 Changes to the photon map

The overall idea behind the `Realtime_photon_map` class is to store the photons for several frames instead of just one [Joz02, 20]. This will allow for a much smaller number of emitted photons per frame.

```
class Realtime_photon_map
{
  typedef vector<Photon>          Per_frame_photons_t;
  vector<Per_frame_photons_t>     photons;
  vector<Photon*>                 weak_photons;
  Nearest_queue                   nearest_photons;
  int                             current_frame;
  bool                            is_new_frame;
};
```

Figure 5.5: Important data structures in the `Realtime_photon_map` class.

The only previous work stems from [Joz02, 27ff], so let us first review the major changes that he found necessary. His idea is to statically allocate a fixed-sized array to store the photons for a single frame. The size should then be as large as the maximum number of photons that can be saved during one emission step. Since photons might be absorbed or disappear (because they do not hit anything), there might be empty slots. So he adds an "is-empty" flag to his photon data structure to accommodate for this.

Furthermore, he creates an accompanying array of indices that are used to keep track of which frames the photons belong to. These indices needs to be updated when the photon array is sorted during the balancing step.

Our implementation is similar in the sense that it needs to address the same problems, but it is different in the approach it takes to solve the problems. A simplified view inside our map shows the data structures in Figure 5.5.

Notice that we have removed the constraint that the maximum number of photons must be fixed by using a `vector` to hold the photons from a single frame. And we further allow a variable number of frames by using another `vector` called `photons`. This simplifies the implementation a great deal—in particular, we do not need to use an "is-empty" flag, and we do not need an array of indices to keep track which frame the photons belong to. The photons simply belong to a certain `vector` which is cleared and refilled as needed. The variable `current_frame` is used to keep track of which `vector` to store the photons in, and `is_new_frame` is used to indicate when to reset a `vector` and start refilling it with new photons.

We have still not explained how we make the many smaller containers function as one big. The answer lies in the variable `weak_photons` which (by storing pointers) acts a view of the entire map. This is illustrated in Figure 5.6. Before the balancing step these pointers need to be updated by storing the addresses of all the relevant photons.

It might seem more expensive to update all the pointers in addition to storing the new photons for a frame, but it can actually improve performance in some cases. The balancing process can be sped up by using an array of pointers since only pointers will be shuffled around [Jen01, 71].

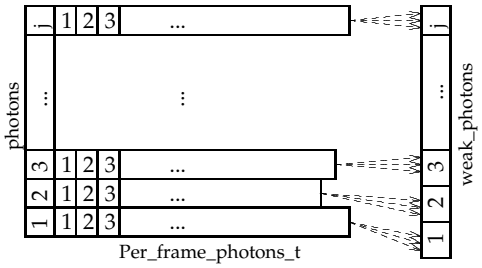



Figure 5.6: Structure of the real-time photon map. The `vector photons` contains photons from $j$ frames. Different frames might store a different amount of photons. `weak_photons` contains pointers to all of the photons.

However, it is more important to improve the running time of the nearest neighbor search since it will run many thousand times per frame. Therefore it might be a problem to store pointers since the nearest neighbor search will be slower because of extra indirections throughout the process. We made a quick test that replaced the array of pointers with real `Photon` objects, but the test did not show any noticeable difference. To reach a decisive conclusion a more thorough test would be necessary.

The performance of the real-time modified photon map is quite satisfactory: it is as fast a the implementation from [Jen01] despite its extra functionality.

### 5.3.2 Photon map balancing

The first step in `balance()` is to update `weak_photons` to get a consistent view the photons in the map and to initialize an axis aligned bounding box of all photons in the map. We then recursively sort the photons with `build_tree()` as shown in Figure 5.7.

First we check if we can stop the recursion. This is done when there is `stop_recursion_size` or less photons left whereby `stop_recursion_size` determines the maximum size of the leafs in the kd-tree. For each recursion step the largest axis of the photons is found (line 4) and we split across that axis. If for example $x$ was the largest axis, we use a plane parallel to the $y$-$z$-plane as the splitting plane. The largest axis can be found cheaply (in the `largest_axis()`) by maintaining the bounding box of the photons on each recursion level. This functionality is hidden inside `build_left/right_subtree()` which simply saves the bounding box state on the stack. Then the bounding box is updated and a recursive call to `build_tree()` is made. An simple alternative to the bounding box is to cycle through the axes, but after a few tests we concluded that it is an inferior technique.

The `nth_element()` call in line 6 sorts the first half of the considered

```
1   void build_tree( size_t from, size_t to ) {
      if ( to - from <= stop_recursion_size ) return;

      Axis sort_axis       = largest_axis();
5     const size_t  middle = middle_index( from, to );
      nth_element( &weak_photons[from], &weak_photons[middle],
                   &weak_photons[to] + 1, Axis_ordering( sort_axis ) );
      Photon* median       = weak_photons[middle];
      median_photon->set_sort_axis( sort_axis );
10    const size_t  left_to    = middle - 1;
      const size_t  right_from = middle + 1;

      if ( from < left_to )
        build_left_subtree( from, left_to );
15
      if ( right_from < to )
        build_right_subtree( right_from, to );
    }
```

Figure 5.7: The algorithm used to build the kd-tree. Our subdivision splits the array in the middle and sort each half recursively. Other implementations builds a heap instead [Jen01, 71f]. We are not aware of any performance differences of the two approaches, but our implementation is somewhat simpler.

photons along the `sort_axis`. `nth_element()` is the perfect choice here since the relative order in each half is unspecified—we just know that each element in the left range comes before each element in the right range (of course, some elements might be equal too) [ISO98, 556].

Notice that the sort axis is saved in the median photon so it can be used to guide the nearest neighbor search (line 9).

### 5.3.3 The irradiance estimate

The most interesting part of the irradiance estimate is how to find the $k$ nearest neighbors. This is described first followed by some optimizations.

After some initialization the $k$ nearest photons is found by the `find_nearest()` algorithm shown in Figure 5.8. The algorithm is also recursive and we start by checking the basis of the recursion in line 4. If the current sub-tree contains `stop_recursion_size` or less photons, we stop the tree traversal and linearly check if the photons should be included (line 5 and 6). We must search the leafs linearly since we cannot rely on the elements being sorted (since `balance()` stopped at this leaf size).

`add_photon()` checks if the photon should be included in the estimate as described in Section 3.5. If the photon passes the check, it is added to the `nearest_photons` container.

If the recursion can continue, we follow the sub-tree that includes the search location recursively (line 16-24). The `can_stop_search()` checks

```
1   void find_nearest( size_t from, size_t to ) const {
      const size_t photons_left = to - from + 1;

       if ( photons_left <= stop_recursion_size ) {
5        for ( size_t i = from; i <= to; ++i )
           add_photon( *weak_photons[i] );
         return;
       }

10     const size_t middle      = middle_index( from, to );
       const Photon& root       = *weak_photons[middle];
       add_photon( root );
       const size_t left_to     = middle - 1;
       const size_t right_from  = middle + 1;
15
       if ( should_search_left_subtree( root ) ) {
         find_nearest( from, left_to );
         if ( not can_stop_search( root ) )
           find_nearest( right_from, to );
20     } else {
         find_nearest( right_from, to );
         if ( not can_stop_search( root ) )
           find_nearest( from, left_to );
       }
25  }
```

Figure 5.8: The algorithm to find the nearest photons.

in line 18 and 22 handles the cases where candidate photons might not be present in both sub-trees. It simply computes the distance from the `root` to the search location along the sort axis; if the distance in that direction is more than the maximum search radius, we know that we do not have to search that sub-tree.

The `nearest_photons` container is an instance of the `Nearest_queue` class which is a modified `vector` implementing two optimizations:

1. No dynamic allocations are done during photon search. This was done because dynamic allocation during insertions was spotted as a performance problem early on.

2. A priority queue is maintained when `max_search_count` photons have been found [Jen01, 74]. There is no reason to build a queue before the vector is full and when it is, the queue is maintained with `push_heap()` and `pop_heap()` from the C++ Standard Template Library [ISO98, 561f]. If $k$ is the size of the queue, `push_heap()` requires at most $\lg k$ comparison whereas `pop_heap()` requires at most $2 \lg k$ comparison. Both functions needs to run whenever a new element is inserted, but the extra work makes it possible to change the search radius dynamically which can further speed up the searching.

Every recursive algorithm can be rewritten as an iterative algorithm. The iterative version is often faster than the recursive algorithm because of less

function call overhead. Christensen presents an iterative version which is reported to be $25\%$ faster compared to the recursive version [JCS01, 100]. The speedup of this algorithm is most valuable in photo-realistic rendering where the amount of nearest neighbor searches is orders of magnitude larger than in real-time rendering. We have not implemented the iterative algorithm, but keep it as an opportunity if the nearest neighbor search becomes a performance bottleneck.

### 5.3.4 Several photon maps

As we can see in Figure 5.2 all relationships are one-to-one. This has been done to keep the implementation simple, but a small discussion of when to break the one-to-one relationship to the photon map follows.

Although Christensen mentions that in general there is no reason to split up the photon map [JCS01, 108], there are two reasons for using several photon maps: (1) to speed up computations and (2) to avoid leaking of photons. The complexity of `Photon_map::balance()` is $O(n \lg n)$ and searching for the $k$ nearest photons takes $O(\lg n + k)$ time [Jen01, 69ff]. To obtain the irradiance estimate we further need to accumulate the power of the photons which takes $O(k)$ time—so the running time for the entire irradiance estimate is still $O(\lg n + k)$.

Let us assume that we can somehow split the global photon map into $m$ smaller maps with an average of $a = n/m$ photons in each and that this does not make the quality of the estimate worse; we can then compare the theoretical running times. Using $m$ maps our running time becomes $O(a \lg a)_1 + \cdots + O(a \lg a)_m = m\ O(a \lg a) = O(n \lg a)$ for balancing and $O(\lg a + k)$ for the irradiance estimate. There is no sum in the estimate since we assume (1) that we can locate all the photons in one map, and (2) that we can find the right map in constant time. Unless the map is absurdly large, $k$ will be the dominant factor and $\lg n$ will be insignificant.

In Table 5.1 we have compared the expected speedup for an estimate with $50$ photons. It shows that maximally $5$-$10\%$ can be saved in theory which supports Christensen's view. However, a recent empirical study shows that in practice there might be a big difference [LC03]. The method partitions the photon map into smaller photon maps according to the normal of surfaces—this has the additional benefit that most surface leaking can be avoided. Surfaces with a similar normal share the same photon map, and for a particular surface the estimate will never consider any of the other maps. This means one can spare the sphere slice calculation.

As we expected they see the largest speedup in the balancing process whereas the irradiance estimate is about $30$-$66\%$ faster. It is unknown why the irradiance estimate becomes so fast, but it might be because of better cache hit rates. Another explanation is that our analysis uses too small

| $n = 10^6$ | | |
|---|---|---|
| $a$ | $\lg a + 50$ | running time |
| $10^6$ | 69.9 | 100% |
| $10^5$ | 66.6 | 95% |
| $10^4$ | 63.3 | 90% |
| $10^3$ | 60.0 | 86% |

| $n = 10^4$ | | |
|---|---|---|
| $a$ | $\lg a + 50$ | running time |
| $10^4$ | 63.3 | 100% |
| $10^3$ | 60.0 | 95% |
| $10^2$ | 56.6 | 90% |
| $10^1$ | 53.3 | 84% |

Table 5.1: Expected running times of the irradiance estimate using several photon maps. Notice that we search for $50$ photons. In practice there is limit to how small one can make the average photon map size, $a$, since a value close to or below the number of photons to search for makes little sense.

values of $n$ and $k$ for the big-O notation to make sense. The hidden constants might be quite significant and *different* on $\lg n$ and $k$. A good example of how such constants affect the running time in real programs can be found in [Ale02a] and [Ale02b]. A third explanation might be that sub-trees are easier discarded by `can_stop_search()` since the photon positions saved in one photon map can be more spatially separated.

It has also been suggested to partition the photon map into several smaller maps that each stores photons from the cells used in the visibility system of the engine [Joh03, 319f]. The visibility system determines that only some of the scene is visible and we need only make lookups in the corresponding maps. This scheme fails if final gathering is required.

## 5.4 The BSP format

We have implemented loading of id Software's Quake 3 BSP format. This give us access to many test scenes with geometry optimized to contain a low polygon count [Pro00]. We first discuss how the geometry can be refined and then discuss a problem with the refined BSP tree.

### 5.4.1 Mesh refinement

The BSP scenes are composed of a minimal amount of triangles which are spatially sorted in a BSP tree. This is a good basis for fast intersection testing. The scenes, however, consist of large polygons which need to be refined to make the irradiance interpolation scheme reasonable accurate. A single **refinement** is done by splitting the triangle with the *globally* longest side into two such that the longest side is split in the middle. We refer to that triangle as the **largest triangle** even though its area might not be the largest. It is important to choose the globally largest side because we do not want to refine geometry which is already detailed enough, such as curved

```
class Triangle
{
  float           longest_edge_length;
  // ...
  bool operator<( const Triangle& other ) const
  { return edge_length < other.edge_length; }
};

typedef priority_queue<Triangle> Triangles_t;
Triangles_t triangles;
```

```
void refine( Node& node, int number_of_refinements )
{
  add_all_triangles_in_model( node, triangles );
  for_each( mesh_refinement )
    split_largest_triangle( triangles );
}
```

```
void split_largest_triangle( Triangles_t& triangles )
{
  Triangle t = tringles.top();
  triangles.pop();

  Triangle t2 = split_triangle( t );
  triangles.push( t );
  triangles.push( t2 );
}
```

Figure 5.9: The mesh refinement algorithm. A priority queue of all triangles is maintained such that the triangle with the *globally* longest edge always is on top. The algorithm works by always splitting the top triangle in two. The triangles are always sorted by the `priority_queue` according to `operator<()`.

objects. Pseudo code for the algorithm is shown in Figure 5.9. To test this aspect, `mesh_refinement` is used as a test parameter.

Referring to Figure 5.9, in `refine()` we store all initial triangles in the scene. This is done by storing pointers to `Drawable`s inside `Triangle` objects. What happens underneath in `split_largest_triangle()` is that these `Drawable`s are modified. The first two lines of `split_largest_triangle()` pick the largest triangle and remove it from the queue. Then `split_triangle()` modifies its argument so it contains one of the two new triangles whereas the second triangle is returned. When the two new triangles are inserted, the `priority_queue` will automatically sort itself such that the largest triangle is always on the top.

Different levels of refinement affects the number of irradiance estimates needed as well as fluctuation and visual quality. A higher degree of refinement leads to less fluctuation and better visual quality at the cost of performance. The fluctuation is decreased because the areas of instability due low photon density are reduced. Visual quality is enhanced because sharper irradiance discontinuities can be visualized.

When the geometry is loaded into OSG, all the geometry is converted

into the `GL_TRIANGLES` format such that no vertices are shared. This makes the refinement easier, but adds a lot of duplicated vertices. For three refinements of a single triangle one vertex can be duplicated as many as five times. This leads to estimation overhead because the same points will be estimated five times. We expect that there is two ways to solve this problem: (1) to use a cache scheme or (2) to transform the `GL_TRIANGLES` form into the `GL_TRIANGLES_STRIP` form. We have chosen to implement solution (1).

### 5.4.2 The estimate cache

For a realistic level of refinement each vertex can be duplicated more than 5 times. To avoid making extra irradiance estimates for each vertex we cache the first estimate for a given location in a given frame and use that estimate for subsequent estimates of the same vertex in the frame.

The cache is implemented using the `map` class from the C++ Standard Library. The implementation of the caching is shown in Figure 5.10. The vertices are used as keys, and a pair containing the irradiance estimate and the frame it was estimated in is used as the value of the map. It is worth noticing that we do not need fill the `cache` variable at load time since `map` automatically creates objects when they do not exist.

The map relies on the less-than operator to maintain a sorted tree of its keys. The tree is usually implemented as a red-black tree which gives optimal performance for a tree based solution [Sed98, 516]. However, sorting the elements is completely unnecessary in our case and therefore a hash map is a better candidate, but have not yet implemented it. We add `use_estimate_cache` as a test parameter to measure the performance gain obtained by using the estimate cache.

The culling system in OSG is optimized for culling speed rather than minimizing the amount of polygons to draw. This is normally a good compromise as modern graphics cards can draw a few extra polygons at nearly no cost. In our case, on the other hand, this means additional expensive irradiance estimates. To further minimize the amount of drawn polygons a sort of Potential Visible Set (PVS) system should be utilized, but we shall not investigate this further. General visibility processing is described in [WP01, 270ff] and [Ebe01, 411ff].

## 5.5 Summary

In Section 5.1 we described how the core classes of the $\pi$-engine cooperate. It is possible to make a design that minimizes the dependencies between the scene module and the photon mapping module. The photon mapping module needs to access scene geometry and light sources from

```
class Estimator
{
  typedef std::pair<Vec3, int>               Estimate_frame_t;
  mutable std::map<Vec3,Estimate_frame_t>    cache;
  Photon_map*                                photon_map;
  // ...
  Vec3 irradiance_at( const Vec3& position, const Vec3& normal ) const
  {
    Estimate_frame_t& e = cache[position];

    if( e.second < current_frame )
    {
      e.second = current_frame );
      e.first  = photon_map->irradiance_at( position, normal );
    }
    return e.first;
  }
};
```

Figure 5.10: Implementation of the estimate cache.

the scene module. The scene module must install a single class from the photon mapping module, the `Draw_callback`. The design of the photon mapping module is largely object oriented which makes it easy to substitute behavior of the algorithms.

The `Photon_tracer` class was explained in Section 5.2. We saw in detail how the photon emission is done and how the classes in the photon mapping module are utilized. Frame-coherent random numbers can be used to reduce fluctuation, and by pre-calculating all the random numbers we could make the overall running time 25-50% faster.

Section 5.3 described the `Realtime_photon_map` class. Compared to a normal photon map it needs a mechanism to store and manage the photons emitted during several frames. The implementation uses a `vector` for each frame and then another `vector` of pointers during balancing and searching. This scheme is actually cited in the literature as a performance improvement, but we keep a skeptical stance towards it. Compared to a normal photon map, the `Realtime_photon_map` is not noticeable slower. We discussed the advantage of using several photon maps to speed up irradiance estimates and balancing of the kd-tree. The irradiance estimate is by far the most important to optimize. In theory only very small performance gains are achievable, but empirical studies have given surprising results.

In Section 5.4 we stated that Quake 3's BSP format is a good basis for fast intersection testing. The scene must be refined to provide decent visual quality, and we showed a simple algorithm for refining the geometry where the globally largest side of a triangle is always split first. When the geometry is refined, the leafs of the tree contained too many triangles. The refinement generates duplicated vertices and therefore it is important to cache irradiance estimates for these vertices.

# 6 Results

[The First Rule of Program Optimization]
*Don't do it.*
[The Second Rule of Program Optimization—For experts only]
*Don't do it yet.*
—Michael Jackson, Michael Jackson Systems Ltd.

In this chapter we present test results. The test results fall in two categories, namely visual quality tests and performance tests. The important questions we try to answer in this chapter are:

1. How good can we make the visual quality regardless of the rendering time?
2. What type of photon scattering gives the best results?
3. Is it possible to make decent color bleeding and shadows?
4. How tessellated must the scene be to give satisfactory results?
5. What are the key performance bottlenecks?
6. How good quality can we get if the frame-rate must be 30 FPS?

Beside these questions, we will describe a long line of small test parameters to see how they affect the algorithm. We make most of our visual quality tests in the Cornell box. This includes color bleeding, shadow effects, and photon scattering. In the second scene, the small quake scene, we investigate tessellation level and performance. The third scene is mainly used together with the two other scenes to see how certain features work with different scenes.

We first describe the many test parameters and evaluate if their behavior is as expected. Visual quality is then discussed followed by performance measurements.

## 6.1 Photon mapping parameters

A parameter is presented like this: `foo`: `int > 0 [50]` which means that the parameter `foo` has the type `int` with the restriction that it must be positive and that a plausible value is $50$. When we describe the tradeoffs involved in a particular parameter, we always assume that all other parameters are held constant.

### 6.1.1 Scene parameters

*Parameter 1:* `BSP_tree`: `bool [true]`. At the scene level, we can specify whether or not to use the BSP tree. The impact we expect from enabling the BSP tree is that intersection tests will be faster for large scenes (see e.g. [HPP00] and [Cha01]). On the small Quake scene there is a noticeable speedup and the larger Quake scene benefits even more from the BSP tree.

Using the BSP tree in the small Quake scene is around $25\%$ faster when there is a minimal number of irradiance estimates (no refinements). On the large Quake level it is $75\%$ faster to use the BSP tree.

*Parameter 2:* `mesh_refinement`: `int >= 0 [10000]`. This option controls the number of polygons that are potentially drawn. This number indicates how many times the largest triangle is split into two. When we use a finer mesh we expect that rendering is slower, but that the visual result is better. To make a more scene independent parameter, this parameter should be exchanged with the length of the maximum triangle side that should be allowed. Then the refinement should simply continue until all sides are smaller than the specified value.

*Parameter 3:* `use_estimate_cache`: `bool [true]`. Determines whether irradiance estimates should be cached. This is a major optimization when scenes contain duplicate vertices. Dependent on the level of refinement, every vertex can be estimated up to $5$-$6$ times for each rendering. The overall speedup depends on the fraction of the total running time that is spent on calculating irradiance estimates.

### 6.1.2 Photon map parameters

*Parameter 4:* `max_search_count`: `int > 0 [50]`. This number simply specifies the maximum number of photons to include in the irradiance estimate. A large value will reduce the variance in the estimate [Suy02, 114], but it will also slow the process down since the search algorithm runs in $O(\lg n + k)$ time. If the photon map size is increased, we can also increase `max_search_count`, but at a lower rate [Suy02, 114]. This number has a large impact on the overall performance. In the Cornell box (in which inter-

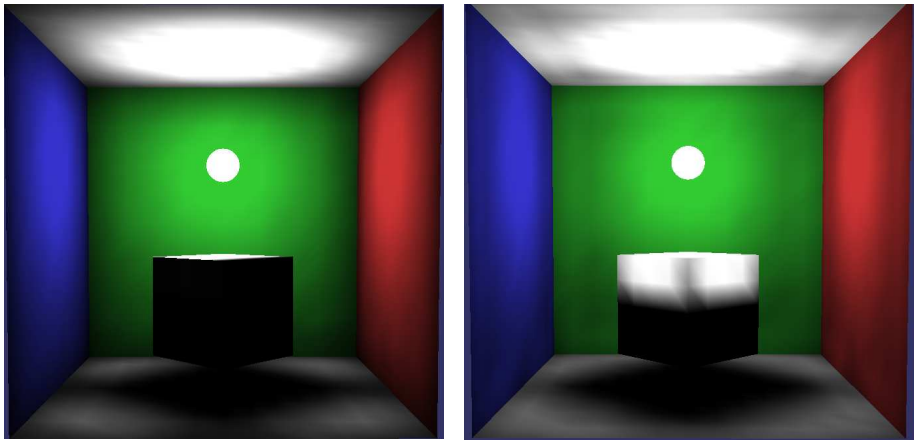


Figure 6.1: The Cornell box with (left) and without (right) use of the sphere slice feature. Even though we use a large search radius, surface leaking is effectively removed.

section testing is fast), moving from 200 photons down to 50 photons can make the engine 50% faster.

*Parameter 5:* `max_search_radius`: `float > 0 [scene dependent]`. The maximum search radius describes the initial radius for the sphere where photons should be collected. A large `max_search_radius` can introduce distance bias in the irradiance estimate. Even though we use a priority queue for updating the search radius during a search, the parameter has a great impact on performance. Therefore this parameter should be as small as possible. It is quite easy to see when the radius is too small because the image becomes speckled.

*Parameter 6:* `use_radius_heuristic`: `bool [false]`. If we enable the radius heuristic from Section 3.2, the `max_search_radius` parameter is disabled. We tried to use the technique on the three scenes and it worked reasonable. The technique is perhaps most useful when the light source is dynamic. In the small Quake scene we observed that the frame-rate could increase when the illumination moved such that it was stored on a smaller area. Although we are not completely certain that we can fine-tune $\alpha$ for all scenes, our first impression is good.

*Parameter 7:* `use_sphere_slice`: `bool [true]`. This parameter controls the use of the sphere slice feature from Section 3.5. This feature is really good at removing surface bias as can be seen in Figure 6.1. The feature has little impact on the overall running time.

*Parameter 8:* `final_gathering`: `bool [false]`. When final gathering is enabled, the calculation of the radiance estimate is extended as described in Section 3.2. Depending on the `final_gathering_samples` parameter, the irradiance estimate is much slower to calculate. We can make an image

with smoothly varying irradiance using very few photons (e.g. 1,000) in the photon map [Jen01, 89f]. We will see more examples rendered using final gathering later. Although we shall not discuss them, it is worth noticing that several other parameters can be used to control the precision of the final gathering results. We did not make a thorough test suite with final gathering since it is very slow. Instead, we just use a high precision to get the most out of the technique.

### 6.1.3 Photon tracing parameters

*Parameter 9:* `photons_from_light:int > 0 [200]`. In all our test scenes we have restricted the number of light sources to one. This parameter describes the amount of photons that are emitted from that light source for each frame. With this parameter we can partially control how many photons will be stored in the photon map. Recall that the size of the photon map was calculated by the algorithm in Figure 4.3. This parameter will increase rendering time since it affects the number of intersection tests as well as the nearest neighbor search time.

*Parameter 10:* `accumulation_frames:int > 0 [10]`. Determines how many frames to accumulate photons over. A high value will make the lighting react slowly to dynamic changes, but allow us to store more photons. A low value will give a faster update of lighting at the cost of reducing the size of the photon map. A too small photon map will give poor visual results. A value of ten will probably be quite alright if the frame-rate is 30 or 60 FPS. In Section 6.3 we try to fine-tune this parameter.

*Parameter 11:* `store_direct_light:bool [true]`. This will trigger storage of the photons that arrive directly from the light source. By turning it on we might be able to create a shadow effect since occluded parts of the scene will receive less energy. By turning it off we see only the indirect light in the scene. It might be possible to make shadows even without storing the direct light if so-called shadow photons are used [Jen01, 148]. If OpenGL lighting is enabled, the direct light should not be stored. We expect that it will be possible to augment our calculation of diffuse indirect light with normal hardware accelerated specular lighting.

*Parameter 12:* `recursion_depth:int >= 0 [1]`. A recursion depth of $n$ means that we allow up to $n$ diffuse reflections for a photon. The actual photon path might be shorter if the photon disappear into open space. A high value should in theory give more plausible results, but it will also waste many intersections on photons with a very low impact on the final result. One reflection is enough to simulate most color bleeding effects, so we rarely use more.

*Parameter 13:* `diffuse_reflections:int > 0 [36]`. This number specifies how many new photons a single photon is split into during a diffuse

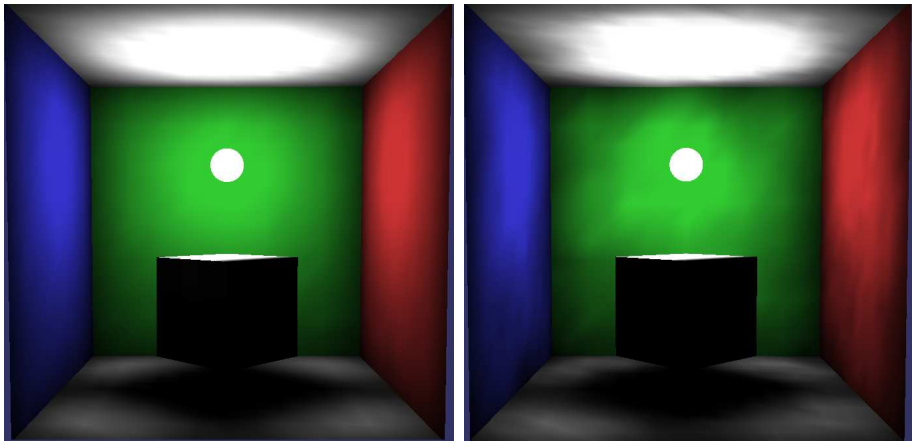


Figure 6.2: Left: when the Saff-Kuijlaars method is used to generate diffuse light directions. Right: naive rejection sampling.

reflection. A high number will require more intersections, but give a better dispersion of photons throughout the scene. If the number is too low the effect of the diffuse reflections becomes unpredictable.

*Parameter 14:* `snowball_size`: `int > 0 [1]`. By filling more photons into the map for a given intersection, we can hope to get a better distribution of the photons without increasing the number of intersections. Once an intersection have been found, we generate some random points in a small circle in the intersection plane. This way we hope to get the same effect as if that circle had been hit by several photons. The circle must be relatively small to avoid storing photons situated in open air and to avoid that shadow areas are hit by photons. Currently this feature make the photon distribution worse than the Saff-Kuijlaars method (see Section 3.6). Obviously, we need to use stratification or quasi-random numbers when the points are generated.

*Parameter 15:* `light_distribution_type`: `int [Saff-Kuijlaars]`. Unfortunately we have not implemented all the distributions described in Section 3.4. Therefore we can only compare the Saff-Kuijlaars method to naive random sampling which can be seen in Figure 6.2. The method is much better than naive rejection sampling. It is unfortunate that we cannot show how quasi-random sequences perform.

*Parameter 16:* `diffuse_light_distribution`:`int [stratified]`. Instead of generating the diffuse scattering randomly, we have tested the benefit of using stratification with one random direction in each strata. The results are as expected: the stratified sampling is much better than naive random sampling.

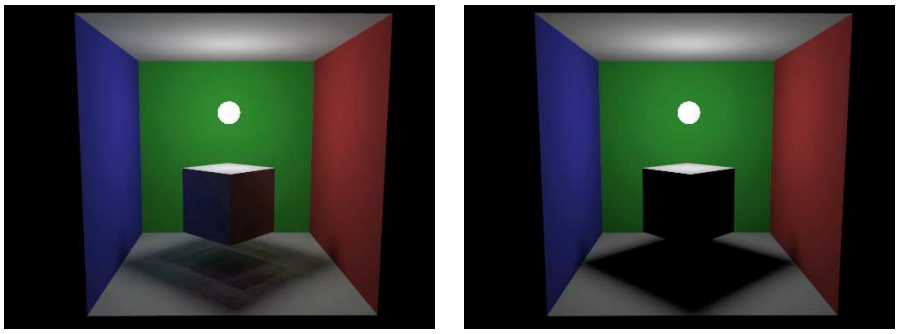


Figure 6.3: Two pictures rendered with photon mapping in RenderPark. The map contained 250,000 photons and 80 neighbor photons were used in the irradiance estimate. Left: A rendering with a maximum path length of 8. Notice the color bleeding on the sides of the little box. Right: A rendering with direct light only.

## 6.2 Visual quality

In this section we compare the best results we can achieve with a photo-realistic rendering from RenderPark. This will give an impression of how good our implementation is currently. We also investigate how detailed the scene must be in order to give decent results.

### 6.2.1 Color bleeding and shadows

Figure 6.3 shows two pictures rendered with RenderPark. We can see that the illumination is very smooth on both pictures. There is a little difference in these two scenes compared to those we use in real-time rendering: the wall we look through is not culled, but simply missing. This means that we will get a brighter color bleeding in the real-time generated pictures. Nevertheless, it should be possible to get a good idea of the overall quality of the illumination.

In Figure 6.4 the same two pictures are rendered with final gathering. There are some problems in the corners, but it is not because of boundary bias—it is an unresolved implementation problem when making a final gather in a corner.

If we compare the left-hand side of Figure 6.3 with the left-hand side of Figure 6.4 we can see that both images have round circles on the walls due to the nearby light source. We can also see that the color bleeding on the box is in both images although it looks rather different; this is because of the white culled wall in the real-time picture. If we compare with Figure 6.4 on the right we see the color bleeding is more visible on the right. The picture only contains indirect illumination which is why the color bleeding

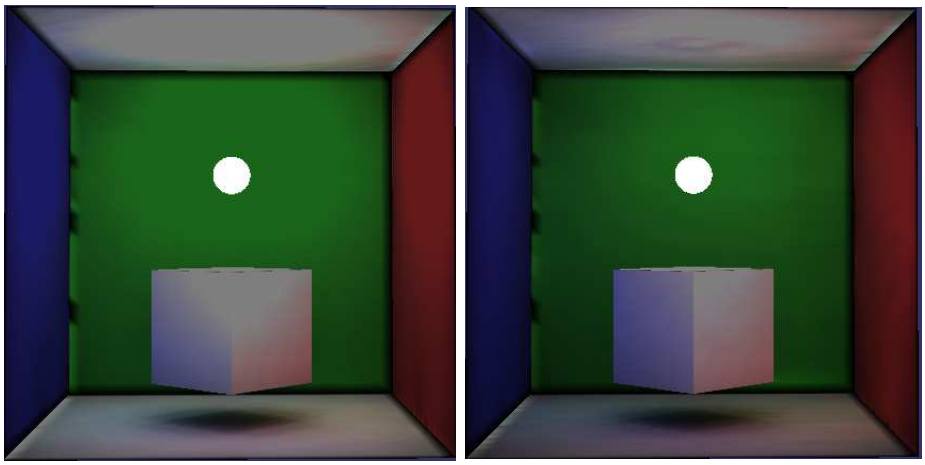


Figure 6.4: Two pictures rendered in "real-time" with final gathering. Both pictures are rendered using $10{,}000$ photons and $625$ final gather rays per estimate. Left: when the direct light is included in the photon map. Right: when no direct light is included in the photon map.

on the floor and in the roof is much more dominant. It is worth noticing that we cannot produce shadows with final gathering.

To continue our discussion of color bleeding, consider Figure 6.5. This shows a real-time version with one diffuse reflection. A very interesting result is that a frame-rate of $20$ FPS can produce similar results to a frame-rate of $1$ FPS. The picture on the right reveals some problems with the diffuse reflections. The diffuse reflections can in certain cases ruin the photon density so that the image becomes speckled.

Figure 6.6 is a real-time rendering without indirect light. If we compare the shadows on this figure with Figure 6.3 on the right, we can see the real-time shadow is not that bad although its slightly softer. If we compare the shadows in Figure 6.5 with Figure 6.3 on the left, then both shadows are more bright due to the indirect illumination. The sharpness on the real-time shadow could better, though. Filtering might be used to sharpen the shadow boundaries (see Section 3.5).

### 6.2.2 Tessellation

Let us now discuss the test of the necessary level of refinement. Differences in refinement level affect both fluctuation and visual quality of the illumination. These results can be used as a rule of thumb for how many triangles we must have in a scene. Figures 6.7 and 6.8 show the scene with a spot light pointing from the center of the scene towards the culled roof. The pictures are made with a recursion level of one. Therefore all the surfaces we see are indirectly illuminated—with OpenGL lighting we would have

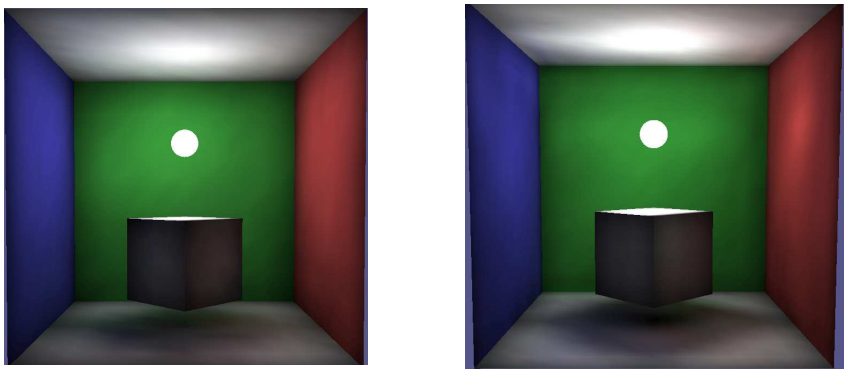


Figure 6.5: Two screen shoots of color bleeding from the $\pi$-engine. The picture on the left is rendered with 20 FPS whereas the picture on the right is rendered with 1 FPS.

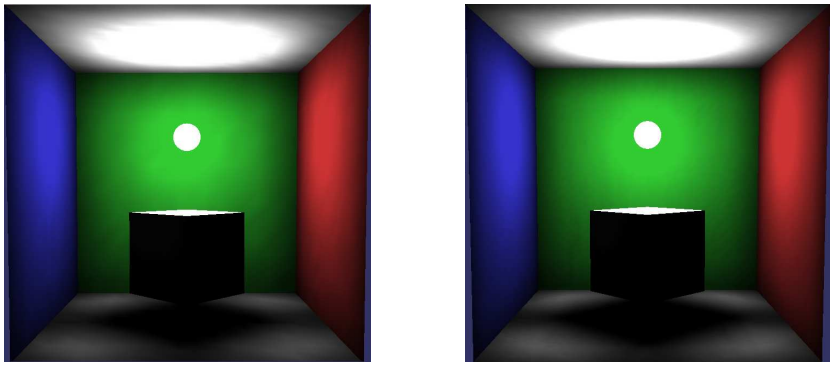


Figure 6.6: Soft shadows from the $\pi$-engine. Only the direct illumination is stored in the photon map. The picture on the left is rendered with about 20 FPS whereas the picture on the right is rendered with 1 FPS.

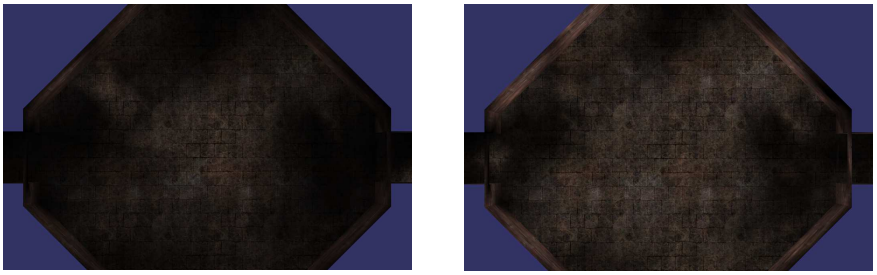


Figure 6.7: Indirect illumination in the small Quake scene. Left: 1,600 triangles. Right: 3,200 triangles—here the illumination becomes reasonable.

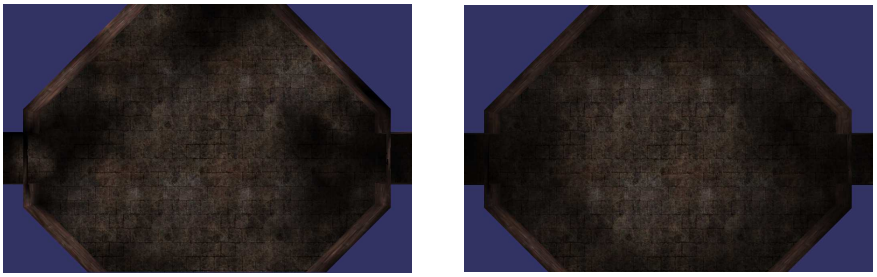


Figure 6.8: Indirect illumination in the small Quake scene: Left: 6,400 triangles. Right: 12,800 triangles—going from 6,400 to 12,800 triangles seems to have a large impact on quality.

seen only black surfaces. In OpenGL we can add an ambient factor, but that would look unrealistic as all surfaces that do not receive direct illumination would be equally lit. This would also happen to the small adjoining rooms which are otherwise mostly dark.

All four pictures have been rendered with a total of 3,200 photons and with a naive rejection sampling from the light source and naive random diffuse reflection. If Figure 6.7 we see that the illumination is very speckled, but also that doubling the number of triangles does not have a large impact on the quality. In Figure 6.8 the quality is still not too good on the left. It is very beneficial to make a higher tessellation as it is can be seen on the right.

## 6.3 Performance

In this section we first describe how performance depends on the size of the photon map and the refinement level of the scene. A test of the intersection testing follows (no pun intended), and we discuss tradeoffs regarding the `accumulation_frames` parameter. Then we make a little case study of the small Quake scene where we keep the frame-rate high and see how good we can make the illumination.

The first test investigates how the size of the photon map and varying

| triangles \ photons | 400 | 800 | 1600 | 3200 | 6400 | 12800 | 25600 |
|---|---|---|---|---|---|---|---|
| 400 | 460 | 298 | 179 | 101 | 52 | 26 | 12 |
| 800 | 344 | 243 | 156 | 90 | 49 | 25 | 12 |
| 1600 | 225 | 172 | 124 | 78 | 45 | 24 | 11 |
| 3200 | 124 | 105 | 83 | 58 | 37 | 21 | 10 |
| 6400 | 63 | 56 | 49 | 38 | 27 | 17 | 9 |
| 12800 | 30 | 28 | 26 | 22 | 17 | 12 | 8 |
| 25600 | 15 | 14 | 13 | 11 | 10 | 8 | 5 |

Table 6.1: Frame-rate as a function of the total number of photons and the total number of triangles. Recall that the number of triangles is just a measure of how refined the scene is.

refinements levels affect the frame-rate of the application. In Table 6.1 we see (by looking down the diagonal) that doubling the number of photons and triangles approximately halves the frame-rate. Hence, the frame-rate seems to be inverse proportional to the sum of the number triangles and the number of photons being traced.

Let us turn our attention to intersection testing. The two Quake scenes contains 369 and 5,355 triangles, respectively. When the intersection testing is done in isolation, we can trace 47,500 and 6,200 primary rays, respectively. So by using circa 14-15 times as many triangles, we only get a factor of 7-8 fewer intersections. This means that the BSP tree does enhance intersection testing, but also that it could be much better.

On a single 800 MHz Pentium-III it should be possible to trace from about 200,000 to almost 1.5 million primary rays per second [WBWS01, 6]. The algorithm makes better use of computational resources such as caches and SIMD instructions. It shows that our implementation is somewhat slow compared to what should be possible. With our hardware it should be possible to trace from 600,000 to 4.5 million primary rays (!). We can also verify that intersection tests is a key performance bottleneck by other means. In the large Quake scene with approximately 55,000 triangles (only 5,355 triangles for intersection testing) we stored 1,500 photons per frame over ten frames. If intersection testing was enabled, the frame-rate was 2-3 FPS whereas it was around 40 FPS if no intersection was done.

We can increase the frame-rate by distributing intersection testing over several frames. This will, however, also increase the time it takes to fully renew all photons in the photon map (we call this time for the **photon renewal time**). Table 6.2 shows how the frame-rate and photon renewal time evolve when the number of accumulation frames is increased and the total number of photons is kept constant. The table shows a very interesting result: we can cache illumination over several frames without increasing the photon renewal time noticeable.

As the last test we shall try to keep the frame-rate high and constant

| Accumulation frames | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|---|
| Frame-rate | 0.63 | 1.24 | 2.41 | 4.52 | 8.25 | 12.51 | 18.01 |
| Photon renewal time | 1.59 | 1.61 | 1.66 | 1.77 | 1.94 | 2.56 | 3.56 |

Table 6.2: Relation between accumulation frames, frame-rate, and photon renewal time. The total amount of photons is kept constant at 10,000. Notice how going from 1 to 16 accumulation frames increases the frame-rate considerable while it only increases the photon renewal time slightly.

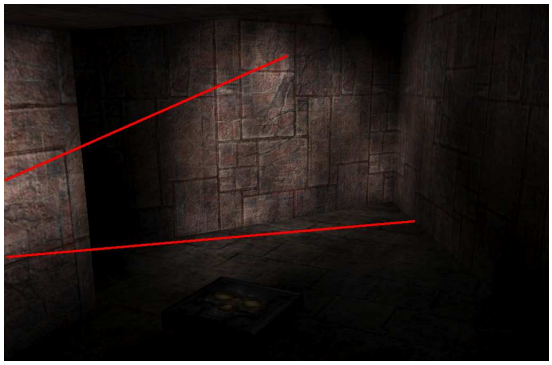


Figure 6.9: The flashlight demo. The demo runs with 30 FPS. The cone of the spot light is indicated (approximately) with red lines.

while tuning the visual quality. This test has been done in the small Quake scene, where we walk around with a spot light pointing forward from the view. In Figure 6.9 we can see a screen shoot from the scene. The scene has been refined until it contains 10,369 triangles. At a frame-rate of 30 FPS we can trace 100 photons from the light source and make 9 diffuse reflections one time for each photon. As we can see in the figure, we do get some indirect illumination, but the quality could be much better. As a last result, we tried running the demo without use of the estimate cache—in that case the frame-rate dropped to 21 FPS.

## 6.4 Summary

In Section 6.1 we described the most important parameters that can tweak the $\pi$-engine. Using the BSP tree is an advantage and the advantage increases with the size of the scene. `max_search_count` and `max_search_radius` should be as small as possible since they have a large impact on performance. The radius heuristic seems to work, although it hard to say if it can be fine tuned independently of the scene. The sphere slice filtering can effectively remove most surface bias. The snowball feature is not very usable in its current status. Photon scattering was best done with the

Saff-Kuijlaars method from the light source and with stratified diffuse sampling in the reflection step; unfortunately we have not implemented the use of quasi-random sequence yet.

Section 6.2 gave an overview of the visual quality we can achieve. It is possible to get real-time color bleeding; occasionally the diffuse reflections can interfere with the surfaces that are directly lit and give speckled images. With some fine-tuning it is possible to get as good images with 20 FPS as with 1 FPS. The scene tessellation needs to be fairly high; for the small Quake scene this means that the scene should contain between 6,400 and 12,800 triangles.

A small performance test revealed that in our implementation the frame-rate is inverse proportional to the sum of the photon map size and the size of the scene (see Section 6.3). When we compare the speed of our intersection testing with the work of others, it is clear that intersection testing is a major performance bottleneck. The frame-rate could be increased by using more accumulation frames with little impact on the photon renewal time. If the frame-rate must be 30 FPS in the small Quake scene, it clear that we cannot yet trace enough photons to get satisfactory results.

# 7 Conclusion

*At any rate it seems that I am wiser than he is to this small extent, that I do not think that I know what I do not know.*
—Socrates

Photon mapping was originally proposed as a technique for photo-realistic rendering. The technique enhances both the quality and the speed of a photo-realistic rendering. This project has integrated photon mapping with a traditional real-time 3D graphics engine. It is desirable to add photon mapping for two reasons:

1. Many of the special cases used to handle different lighting effects in 3D graphics engines can be avoided.

2. Global illumination including indirect dynamic light can be simulated. This will enable lighting effects such as color bleeding, caustics, and shadows in real-time rendering.

Our first report served as a feasibility study for this report [OK02]. The conclusion was that photon mapping could potentially add color bleeding and soft shadows to a real-time context. From that conclusion we derived the goals stated in Chapter 1.

The long term goal of the project is to maintain a frame-rate of at least 30 FPS while simulating dynamic light. The main areas we have focused on to fulfill this goal are related to either visual quality or performance. The requirement of both high performance and high visual quality constitutes opposing demands. We have therefore explored ways to improve one requirement without degrading the other. For example, stratified sampling can improve the visual quality without decreasing the performance. On the other hand, we can also improve the performance by using more accumulation frames without affecting the visual quality noticeable.

We will first describe the status of the implementation followed by the main contributions of this work. Then we compare the status of the project with the goals from the introduction. At the end of the chapter we describe directions for future work.

## 7.1 Implementation status

In this project we have shown that it is possible to use photon mapping to add dynamic light to a real-time 3D graphics engine. Our engine can simulate color bleeding and shadows.

The traditional 3D graphics engine must be altered in the update and the draw stage. Photon tracing is added in the end of the update stage. The draw stage must make an irradiance estimate for all visible vertices. To improve the visual quality it is important to refine the geometry. The intersection testing performance is kept independent of this refinement by using a separate unrefined copy of the geometry. The implementation of the real-time photon map performs well despite its extra functionality and flexibility.

A number of important observations have been made throughout this project:

- The tessellation level needs to be relatively high. Our medium sized test scene needed a factor of $40$ more triangles than in the original scene to get good visual results.
- The photons must be dispersed with as low a discrepancy as possible to make optimal use of the limited number of photons. To generate directions one should use stratification or quasi-random numbers.
- Intersection testing is the key performance bottleneck. One should have this in mind when designing the scene tree.
- A good compromise between the number of accumulation frames and the photon renewal time can be found. Without decreasing the photon renewal time significantly, a much larger frame-rate can be achieved.
- It is important to only make irradiance estimates for the unculled vertices. Therefore it pays off to spend more time in the cull stage in order cull as many vertices as possible.

Several spin-offs of the project will be made available. This includes an `Exact_photon_map` class that can be used to verify photon map implementations against, an easy to understand implementation of the real-time photon map, and a Quake 3 BSP loader for use with Open Scene Graph.

## 7.2 Contributions

We will now discuss the main contributions of this project. In Chapter 3 we describe how diffuse reflections can be improved for use in a real-time context. Instead of decreasing the number of diffusely reflected photons with Russian Roulette, we *increase* the number of diffusely reflected photons. Thereby fewer reflections are needed to cover the scene with photons and less time is spend on rather unimportant photons.

We have proposed a terminology for describing different types of leaking: **surface leaking** is defined as leaking from surfaces with a different normal or translated surfaces with a similar normal, and **distance leaking** is defined as leaking from a differently lit area on the same flat surface.

We also describe a new method for effectively removing surface bias using a sphere slice. This method is faster than commonly advised approaches such as cylinders an ellipsoids.

Chapter 5 described how frame-coherent random numbers can be used to decrease fluctuation. We showed how one could also improve performance by pre-calculating all needed directions.

The tests in Chapter 6 revealed an interesting property of the number of accumulation frames. It is possible to increase the frame-rate considerably while only increasing the photon renewal time slightly.

## 7.3 Comparison with goals

We will compare the obtained results with the goals as they were laid out in section 1.3. This is done by discussing the achievements related to each goal.

**Goal 1:** *Improve the image quality and reduce fluctuation to an acceptable level.*

**Achievement 1:** We are able to produce images with acceptable quality and low fluctuation. The most important improvement of image quality comes from a better distribution of photons. Fluctuations are decreased considerable by using frame-coherent photon paths. We expect the image quality to improve as soon as we can trace more photons or when support for quasi-random sequences are added. To further reduce fluctuation, a high tessellation level should be used. The current status indicates that using a high tessellation level is not a performance problem.

**Goal 2:** *Build a dynamic scene that can work as a test scenario.*

**Achievement 2:** We have a simple test scenario with dynamic objects and a dynamic light source. To test larger scenes we can load Quake 3 BSP scenes. This gives access to scenes of a realistic size.

**Goal 3:** *Assemble the 3D engine so important modules are implemented and functioning in a realistic manner.*

**Achievement 3:** Most required modules in a 3D engine are present in states adequate for identification of real performance bottlenecks. In particular, we have intersection testing based on spatially sorted geometry and a decent culling performance.

**Goal 4:** *The integration of photon mapping and the real-time 3D graphics engine should be as easy as possible.*

**Achievement 4:** Adding photon mapping requires integration in three places. (1) The photon mapping module must have access to the scene geometry to perform intersection testing. (2) The scene module must install a single callback that can update the irradiance estimates of each vertex in the draw stage. (3) It must be possible to make a refined model of the scene. We believe that (1) and (2) are straightforward extensions to normal real-time 3D graphics engines whereas we are uncertain about (3).

**Goal 5:** *Test different methods to scatter photons throughout the scene and find out which methods that should be preferred in the engine.*

**Achievement 5:** Only a few different ways of distributing photons in a scene has been tested. Currently the best strategy is to use the Saff-Kuijlaars method for light source directions combined with stratified sampling for diffuse reflections. However, we are convinced that using quasi-random sequences will give even better results.

**Goal 6:** *Identify areas that clearly seems to be performance bottlenecks.*

**Achievement 6:** Many performance bottlenecks have been identified and optimized during development. In the current application intersection testing is the key bottleneck. Currently we have optimized the intersection testing module by using bounding volume hierarchies, but nevertheless the performance is still rather low compared to state-of-art methods.

**Goal 7:** *Keep the focus on dynamic indirect lighting with color bleeding in diffuse environments.*

**Achievement 7:** We are able to render our test scenes with dynamic indirect illumination including color bleeding at acceptable frame-rates. Furthermore, soft shadows are present although blurry. We hope that this can be improved with use of filtering.

We can conclude that most of our goals have been fulfilled, but that some issues still persist. In the next section we summarize the areas that are subject to future work.

## 7.4 Future work

Our implementation can be still improved. Performance wise our intersection testing is slow (Section 6.3). The photon map implementation can be improved by using an iterative nearest neighbor search and by using several photon maps (Section 5.3). Quality wise the most important enhancement will probably be to use quasi-random sequences (Section 3.4).

In this report we have continuously improved both performance and visual quality. As future work we recommend this prioritized list for both categories:

- Performance:
  1. Improve intersection testing.
  2. Remove the need for an estimate cache or use a hash map to make the implementation as efficient as possible.
  3. Culling should be made more fine-grained to minimize the amount of unculled vertices outside the view frustum.
  4. Iterative nearest neighbor search and the use of several photon maps.
- Visual quality:
  1. Quasi-random sequences for use during photon scattering.
  2. Use filtering to sharpen shadows.
  3. Improve the area estimate to remove boundary bias.

As indicated by industry leaders, global illumination will become the norm. There are several competing approaches that tries to achieve this goal. We can categorize the methods as either pure ray-tracing-based rendering, pure hardware-based rendering, and the hybrid approaches like the $\pi$-engine.

Most hybrid methods have started out with a very high requirement to the visual quality which means that they are very slow. Our approach has been the opposite. We have started with an expectation of interactive frame-rates and then tried to improve the visual quality.

Recent advances in ray-tracing suggests that real-time ray-tracing will be possible in the near future. Before that happens we believe hybrid approaches will the best alternative. Modern graphics hardware are excellent at rendering dynamic direct light and this capability should be used together with something similar to the $\pi$-engine. In a way the two methods complement each other perfectly: the specular effects and the direct light can be simulated by hardware whereas special effects like color bleeding and caustics can be rendered by the real-time photon mapping algorithm. Therefore we believe that real-time photon mapping can be used in modern real-time 3D graphics engine in the foreseeable future.

# Bibliography


[Ale02a] Andrei Alexandrescu. Efficient generic sorting and searching in c++ (i): In search of a better search. *C/C++ Users Journal*, October 2002.

[Ale02b] Andrei Alexandrescu. Efficient generic sorting and searching in c++ (ii): Sorting through sorts of sort algorithms (well, sort of). *C/C++ Users Journal*, December 2002.

[BB82] Dana H. Ballard and Christopher M. Brown. *Computer Vision*. Prentice Hall, 1982.

[BdLPM] Philippe Bekaert, Frank Suykens de Laet, Pieter Peers, and Vincent Masselus. Renderpark: A test-bed system for global illumination. http://www.renderpark.be/.

[Cha01] Allen Y. Chang. A survey of geometric data structures for ray tracing. Technical report, Department of Computer and Information Science Polytechnic Universityn Brooklyn, New York, 11201, 2001.

[DBMS02] K. Dmitriev, S. Brabec, K. Myszkowski, and H. Seidel. Interactive global illumination using selective photon tracing, 2002.

[DDM03] Cyrille Damez, Kirill Dmitriev, and Karol Myszkowski. State of the art in global illumination for interactive applications and high-quality animations. *COMPUTER GRAPHICS forum*, 21, 2003.

[Ebe01] David H. Eberly. *3D Game Engine Design - A Practical Approach to Real-Time Computer Graphics*. Morgan Kaufmann Publishers, 2001.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[HP01] Heinrich Hay and Werner Purgathofer. Global Illumination with Photon Map Compensation. available at ftp.cg.tuwien.ac.at/pub/TR/01/TR-186-2-01-04Paper.pdf, January 2001.

[HPP00] Vlastimil Havran, Jan Přikryl, and Werner Purgathofer. Statistical Comparison of Ray-Shooting Efficiency Schemes. Technical report, Department of Computer Science, Czech Technical University; VrVis Center for Virtual Reality and Visualization; Institute of Computer Graphics, Vienna University of Technology, 2000.

[II01] Ru Igarashi and Joss Ives. Sampling in non-cartesian coordinate systems. http://nucleus.usask.ca/Geant/Exercises/sampling2.html, 2001.

[ISO98] ISO/IEC. *International Standard, Programming languages — C++*, 1st edition, 1998.

[JCS01] Henrik Wann Jensen, Per H. Christensen, and Frank Suykens, editors. *A Practical Guide to Global Illumination Using Photon Mapping*, Los Angeles, USA, August 2001. ACM SIGGRAPH.

[Jen01] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001.

[Joh03] Marlon John. *Focus On Photon Mapping*. The Premier Press Game Development Series. Premier Press, 2003.

[Joz02] Timothy R. Jozwowski. Real time photon mapping. Master's thesis, Michigan Technological University, May 2002.

[Kel97] Alexander Keller. Instant radiosity. http://www.uni-kl.de/AG-Heinrich/Alex.html, 1997.

[KK02] Thomas Kollig and Alexander Keller. Efficient multidimensional sampling. EUROGRAPHICS Volume 21, Number 3, 2002.

[LC03] B. D. Larsen and N. J. Christensen. Optimizing photon mapping using multiple photon maps for irradiance estimates, feb 2003.

[Len02] Eric Lengyel. *Mathematics for 3D Game Programming & Computer Grahics*. Charles River Media, Inc., 2002.

[Obj03] Object Management Group, Inc. *OMG Unified Modeling Language Specification*, March 2003.

[OK02] Thorsten Ottosen and Dennis Kristensen. The $\pi$-engine—part 1—illumination for 3d game engines using photon mapping. Master's thesis, Department of Computer Science, Aalborg University, Denmark, December 2002.

[Oud99] Juri A. Oudshoorn. Ray tracing as the future of computer games. Technical report, Department of Computer Science, University of Utrecht, November 1999.

[PP98] Ingmar Peter and Georg Pietrek. Importance driven construction of photon maps. http://ls7-www.cs.uni-dortmund.de/research/projekte/visibility-problems/, 1998.

[Pro00] Kekoa Proudfoot. Unofficial quake 3 map specifications. http://graphics.stanford.edu/~kekoa/q3/, 2000.

[PVTF02] William H. Press, William T. Vetterling, Saul A. Teukolsky, and Brian P. Flannery. *Numerical Recipes in C++*. Cambridge University Press, 2002.

[Rab02] Steve Rabin, editor. *AI Game Programming Wisdom*. Charles River Media, Inc., 2002.

[SBJWJ00] Raymond A. Serway, Robert J. Beichner, and Jr. John W. Jewett. *Physics For Scientist and Engineers*. Sounders College Publishing, 2000.

[Sed98] Robert Sedgewick. *Algorithms in C++, Third Edition*. The Premier Press Game Development Series. Addison Wesley, 1998.

[Shi91] Peter Shirley. Discrepancy as a quality measure for sample distributions. In *Proceedings of Eurographics 91*, pages 183–193, June 1991.

[SK97] E.B Saff and A.B.J Kuijlaars. Distributing many points on a sphere. *Mathematical Intelligencer 19.1*, pages 5–11, 1997.

[SKP98] László Szirmay-Kalos and Werner Purgathofer. Analysis of the quasi-monte carlo integration of the rendering equation. Technical report, Department of Control Engineering and Information Technology, Technical University of Budapest, 1998.

[Sta03] Nick Stam. The future of 3d graphics. http://www.extremetech.com/article2/0,3973,1091416,00.asp, May 2003.

[Suy02] Frank Suykens. *On robust Monte Carlo algorithms for multi-pass global illumination*. PhD thesis, Department of Computer Science, Katholicke Universiteit Leuven, 2002.

[SW00] Frank Suykens and Yves D. Willems. Density control for photon maps. http://www.cs.kuleuven.ac.be/cwis/research/graphics/CGRG.PUBLICATIONS/, 2000. Department of Computer Science, K.U. Leuven, Belgium.

[VMKK00] Valdimir Volevich, Karol Myszkowski, Andrei Khodulev, and Edward A. Kopylov. Using the visual differences predictor to improve performance of progressive global illumination computation. In *ACM Transactions on Graphics*, volume 19 (2), pages 122–161, 2000.

[Wat00] Alan Watt. *3D Computer Graphics, Third Edition*. Addison-Wesleu Publishing Ltd., 2000.

[WBWS01] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek. Interactive rendering with coherent ray tracing. In Alan Chalmers and Theresa-Marie Rhyne, editors, *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001*, volume 20, pages 153–164. Blackwell Publishers, Oxford, 2001. available at http://graphics.cs.uni-sb.de/~wald/Publications.

[WKB+02] Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. Interactive Global Illumination using Fast Ray Tracing. In *Proceedings of the 13th EUROGRAPHICS Workshop on Rendering*. Saarland University, Kaiserslautern University, 2002. avail.at http://www.openrt.de.

[WP01] Alan Watt and Fabio Policarpo. *3D Games — Real-time Rendering and Software Technology*. Addison-Wesley, 2001.

[WRC88] Gregory J. Ward, Francus M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. In *Computer Graphics*, pages 85–92, August 1988.

[Wu03] Ying Wu. Radiometry, brdf and photometric stereo. http://www.ece.northwestern.edu/~yingwu/teaching/ECE510/Notes/lighting.pdf, 2003.