Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Master's Thesis

Data Structures for Interpolation of Illumination with Radiance and Irradiance Caching

Bc. Ondřej Karlík

Supervisor: Ing. Vlastimil Havran, Ph.D.

Study Programme: Open Informatics

Field of Study: Computer Graphics and Interaction

May 13, 2011

Aknowledgements

I would like to thank Vlastimil Havran for supervising and correcting this thesis, and for his helpful ideas, cjx3711 for the chess pieces model, Jiří "Biolit" Friml for providing the Diffuse interior model, and Ludvík "Rawalanche" Koutný for making the Glossy interior model and testing the irradiance caching algorithm. Finally, I would like to thank my family and friends for their support.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

Prague, 13. 5. 2011 .....

Abstract

Irradiance and radiance caching are important algorithms for solving the light transport problem in realistic image synthesis. They both require geometric search data structures for efficient rendering. Our goal was to improve the caching algorithms by improving these data structures.

We have implemented 6 different data structures for irradiance caching, 2 previously used and 4 newly adapted to the problem. Our testing showed that multiple-reference data structures offer the best traversal performance at the cost of higher memory consumption.

For interpolation on glossy surfaces we have implemented the *spatial directional radiance caching*. Instead of modifying its data structures we have created a novel radiance caching algorithm by merging its spatial and directional interpolation phases, creating the *unified radiance cache*.

Abstrakt

Irradiance a radiance caching jsou důležité algoritmy používané pro řešení problému přenosu světla v realistické syntéze obrazu. Oba algoritmy potřebují k efektivní činnosti datové struktury pro geometrické vyhledávání. Našim cílem bylo vylepšit tyto algoritmy vylepšením jejich datových struktur.

Implementovali jsme celkem 6 různých datových struktur pro irradiance caching (2 standardně používané a 4 které jsme adaptovali pro použití pro tento problém). Naše měření ukázalo, že struktury založené na *duplikování záznamů* umožňují nejrychlejší vyhledávání za cenu vysoké spotřeby paměti.

Pro interpolaci osvětlení lesklých povrchů jsme naimplementovali algoritmus *spatial directional radiance caching*. Místo modifikace jeho datových struktur jsme pak vytvořili nový algoritmus typu radiance caching spojující prostorovou a směrovou fázi interpolace – *unified radiance cache*.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Realistic image synthesis a very important field of computer graphics. It gained prominence in the last two decades, as the exponential growth in computer power allowed artists to create artificial images (renders) indistinguishable from real world photographs at first sight. What was once only a research interest of a small group of scientists is now a multi-billion dollar industry producing countless pictures and animations every day, which are used in entertainment, advertising, architecture, design, education, science and many other areas.

Main task of a rendering software is to produce realistically looking image from scene description created by the artist. This is achieved by simulating the light energy transport through the scene from light sources into the observer's eye as it would happen in real life. To perform this simulation, image synthesis combines several disciplines: physics, used for describing the light behaviour in real world, mathematics, used for solving obtained light transport equations and finally computer science, which provides robust and efficient implementation of solver algorithms. The physics of the light transport is solved problem, as it is described by the *rendering equation*. The research is instead focused on algorithms used for solving the light transport problem, because even after the tremendous advancements we saw in last years these algorithms still cannot produce satisfactory results in real time[1].

Algorithms solving the rendering equation are usually called *global illumination* (GI) algorithms. This name stems from the fact that when rendering an image with the global illumination, single surface is lit by the entire scene via the *diffuse inter-reflections*, not just by a few localized direct light sources. Traditional shading algorithms are sometimes referred to as local illumination in contrast to global illumination. Some of the most well-known global illumination algorithms are *Monte Carlo path tracing*, *photon mapping*, *radiosity*, and *irradiance caching*.

It is the last algorithm, irradiance caching, that is the main focus of this thesis. It works by precomputing the illumination only in few sparse points in the scene, storing them in an acceleration data structure and then searching and interpolating between them to get the illumination in final image. Because this algorithm is simple, robust, produces high quality results fast, and can be combined well with other global illumination methods, it is used by many commercial and free renderers. Its main disadvantage is that it cannot handle other materials than perfectly diffuse. Extensions to non-diffuse materials are however possible; they are collectively called *radiance caching*.

[1] Real time in this context means frame rate of 25 frames per second or higher.

Because the irradiance caching algorithm received much attention over the years and many efficient and highly optimized implementations exist, we find it surprising that a simple *multiple reference octree* is still considered standard data structure for accelerating irradiance cache queries. Because we feel that the algorithm has reserves in this area we implement various geometric searching acceleration data structures and integrate them into the same irradiance caching algorithm so that their performance can be directly compared. We also explore different possibilities of data storage in radiance caching, which result in new radiance caching algorithm formulation.

Thesis structure

The rest of this thesis is organized as follows: in *Chapter 2* we introduce basic radiometric quantities, the *bidirectional reflectance distribution function* (BRDF) and the rendering equation as a way to describe the light transport, and standard algorithms for solving it. In *Chapter 3* we give detailed description of the irradiance caching algorithm for accelerating GI computation on diffuse surfaces. We also introduce radiance caching as a way to extend the original algorithm to glossy surfaces. Next, in *Chapter 4* we describe the problem of geometric searching. After brief theoretical introduction we discuss some commonly used acceleration data structures for search in low dimensional space, such as *octrees* and *kd-trees*. Then we describe possible modifications to allow queries over non-point data.

In *Chapter 5* we postulate the search problem in irradiance cache interpolation and then analyse our examined data structures. Similarly in *Chapter 6* we analyse the search problem for radiance cache interpolation and derive new radiance caching algorithm. In *Chapter 7* we describe the implementation of data structures from Chapters 5 and 6 in our ray tracing system *Corona*. Finally, in *Chapter 8* we present experimental results of all implemented data structures in our set of 10 test scenes and discuss them. We conclude in *Chapter 9* by summing up our findings and suggesting possible directions of future research.

Chapter 2

Basics of light transport and global illumination

To produce realistically looking images we need precise mathematical description of the light transport. We present a way to describe the light interaction in a scene with single integral equation, called *rendering equation*. This is the standard solution presented by many textbooks [SM03, DBB06, PH10]. To obtain this reasonably simple model, for which efficient solutions exist, we first have to make few simplifications.

In real world, light consists of subatomic particles, *photons*, which behave both as waves and particles (this phenomenon is called the *wave-particle duality*). We choose to ignore the wave behaviour, which creates such effects as interference, diffraction, and polarization. We also do not take into account participating environments which can absorb or scatter photons on the way, instead we assume that photons travel through the environment unchanged in a straight line and that they interact only upon hitting a scene surface. It is worth mentioning that it is possible to simulate even these advanced light phenomena [CPP+05, JMLH01, WTP01].

The number of photons in real life situations is so high[1] that the discrete nature of light cannot be observed in vast majority of situations. Because of that we can describe light by *continuous quantities*. This description is simpler and more elegant. It is however used only conceptually for deriving needed formulas. To solve them using a computer we need to discretize the light representation again. Because we cannot afford to simulate each photon individually, the discretization is much coarser. Instead of photons we use *rays*, idealized narrow beams of light with energy much higher energy than in physical reality.

2.1 Radiometric quantities

Before we can start describing the light transport, we need to define physical quantities we will use. There are two ways to describe the problem – with *radiometric* and *photometric* quantities. The difference is that while radiometric quantities are based on physical measurements of light, photometric ones are based on its subjective perception by human visual system. We use the former system, since photometric qualities can be easily derived from corresponding radiometric terms.

[1] For example, single 100 watt light bulb produces approximately 10^{20} photons per second [SM03].

Basic radiometric quantity is the *radiant energy* (Q). Electromagnetic radiation consists of photons, each of which has associated energy proportional to its frequency. Radiant energy is the sum of all photon energies E_{p_i} in the area of interest:

$$Q = \sum E_{p_i}.$$

We are typically more interested in the *radiant flux* (Φ). It corresponds to power and is expressed in Watts (W). It measures how much energy flows from or to a surface per unit of time:

$$\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t}.$$

If we express the radiant flux per unit area, we get either *irradiance* (E) or *radiosity* (B), also called radiant exitance. Difference between the two is that irradiance is the *incident* radiant flux per unit area and radiosity is the *exitant* radiant flux per unit area. Irradiance at a surface point $\mathbf{x}$ is denoted $E(\mathbf{x})$, radiosity at $\mathbf{x}$ is $B(\mathbf{x})$. They are both expressed as W/m^2 and computed as:

$$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A}, \quad B = \frac{\mathrm{d}\Phi}{\mathrm{d}A}.$$

Final and the most important quantity is the *radiance* (L). It is the radiant flux per unit projected area per unit solid angle:

$$L = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\vec{\omega}\,\mathrm{d}A\,\cos\theta} = \frac{\mathrm{d}E}{\mathrm{d}\vec{\omega}\,\cos\theta}. \tag{2.1}$$

The cosine term arises from perpendicular projection of the differential area where we measure the radiance (as we are almost exclusively interested in radiance with respect to a surface), as shown in Figure 2.1. Radiance at surface point $\mathbf{x}$ in direction $\vec{\omega}$ is denoted $L(\mathbf{x},\ \vec{\omega})$. We can also use $L(\mathbf{x} \leftarrow \vec{\omega})$ and $L(\mathbf{x} \rightarrow \vec{\omega})$ to distinguish between incident and exitant radiance, respectively.

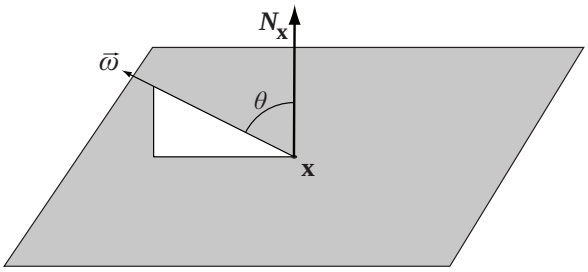


Figure 2.1: *Geometry for the definition of radiance. Radiance $L(\mathbf{x},\ \vec{\omega})$ is defined with respect to the differential surface $\mathrm{d}\mathbf{x}$ and differential solid angle $\mathrm{d}\vec{\omega}$.*

Radiance is the most important quantity in global illumination computation, because it is the one quantity that sensors like human eye or camera chip register. Moreover, radiance values are constant along straight lines in non-participating media. This law is the key to formulating the rendering equation.

We can express all other quantities as integrals of radiance. Irradiance at surface $\mathbf{x}$ is an integral of incident radiance multiplied by the cosine over the hemisphere of directions Ω (Equation 2.2). Radiosity is computed the same way, only with incident radiance replaced by exitant (Equation 2.3). Radiant flux

is an integral of irradiance or radiosity over a surface (Equation 2.4) and radiant energy is an integral of radiant flux over time (Equation 2.5):

$$E(\mathbf{x}) = \int_{\Omega} L(\mathbf{x} \leftarrow \vec{\omega}) \cos\theta \, \mathrm{d}\vec{\omega}, \tag{2.2}$$

$$B(\mathbf{x}) = \int_{\Omega} L(\mathbf{x} \rightarrow \vec{\omega}) \cos\theta \, \mathrm{d}\vec{\omega}, \tag{2.3}$$

$$\Phi = \int_{A} E(\mathbf{x}) \, \mathrm{d}A = \int_{A} \int_{\Omega} L(\mathbf{x}, \, \vec{\omega}) \cos\theta \, \mathrm{d}\vec{\omega} \, \mathrm{d}A, \tag{2.4}$$

$$Q = \int_{T} \Phi(t) \, \mathrm{d}t = \int_{T} \int_{A} \int_{\Omega} L(\mathbf{x}, \, \vec{\omega}) \cos\theta \, \mathrm{d}\vec{\omega} \, \mathrm{d}A \, \mathrm{d}t. \tag{2.5}$$

2.2 Bidirectional reflectance distribution function

Next we need to describe the light interaction upon hitting a surface. More precisely, we want to know reflected radiance distribution given particular incident radiance distribution. This can be described by the four-dimensional *bidirectional reflectance distribution function* (BRDF). This function takes as argument an incident ($\vec{\omega}_i$) and exitant ($\vec{\omega}_o$) direction on hemisphere around surface point $\mathbf{x}$ (as illustrated in Figure 2.2) and returns the fraction of irradiance due to the radiance incoming from the incident direction reflected to the exitant direction:

$$f_r(\mathbf{x}, \, \vec{\omega}_i, \, \vec{\omega}_o) = \frac{\mathrm{d}L(\mathbf{x} \rightarrow \vec{\omega}_o)}{\mathrm{d}E(\mathbf{x}, \, \vec{\omega}_i)} = \frac{\mathrm{d}L(\mathbf{x} \rightarrow \vec{\omega}_o)}{L_i(\mathbf{x} \leftarrow \vec{\omega}_i) \, \cos\theta_i \, d\vec{\omega}_i}$$

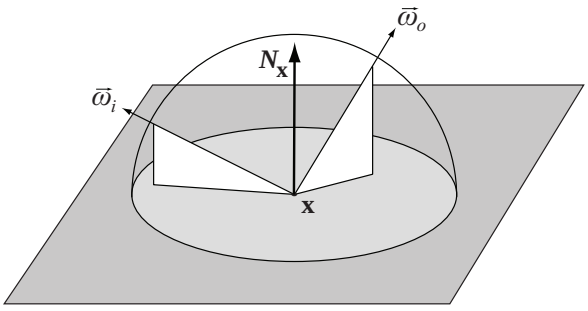


Figure 2.2: *Geometry for the BRDF function – visualization of the hemisphere around point* $\mathbf{x}$*, and incident and exitant directions* $\vec{\omega}_i$, $\vec{\omega}_o$.

More informally, BRDF can be viewed as function describing probability density that a photon hitting the surface from direction $\vec{\omega}_i$ is reflected to the direction $\vec{\omega}_o$. BRDF is however not a probability density function (PDF) because it may not integrate to one (the probability that a photon is reflected is lower or equal to 1 as it can also be absorbed).

2.2.1 BRDF properties

Thinking of BRDF as probability density function gives us its several key properties. BRDF, similarly to PDF, can take any positive value, even approach infinity (as for example *Dirac delta function* [Dir82]). PDF always have to integrate to 1, BRDF multiplied by the cosine for any fixed incident direction have to integrate to value between 0 and 1. This constraint can be expressed with equation (equation 2.6); It corresponds to the fact that we permit only realistic materials that obey the law of energy conservation:

$$\forall \vec{\omega}_i : \; 0 \; \leq \; \int_{\Omega} f_r(\mathbf{x}, \; \vec{\omega}_i, \; \vec{\omega}_o) \cos\theta_o \, \mathrm{d}\vec{\omega}_o \; \leq \; 1. \tag{2.6}$$

BRDF is purely material description and as such is not dependent on incident radiance distribution. Its final important property is the *Helmholtz reciprocity*, which states that swapping the input and output direction arguments of BRDF do not change its value:

$$f_r(\mathbf{x}, \; \vec{\omega}_i, \; \vec{\omega}_o) \; = \; f_r(\mathbf{x}, \; \vec{\omega}_o, \; \vec{\omega}_i). \tag{2.7}$$

2.2.2 BRDF examples

Simplest example of a BRDF is that of ideal diffuse surface (Figure 2.3a). Such surface, also called *Lambertian*, reflects any incident light uniformly, independently of incident or exitant direction. The BRDF value is constant:

$$f_D(\mathbf{x}, \; \vec{\omega}_i, \; \vec{\omega}_o) \; = \; \frac{\rho_d}{\pi}, \tag{2.8}$$

where ρ_d is the surface *diffuse albedo*, fraction of incident radiance that is diffusely reflected off the surface. Clearly $0 \leq \rho_d \leq 1$. This BRDF is frequently encountered in the real world.

Opposite of diffuse surface is the ideal mirror, that is a surface with BRDF producing only ideal specular reflection (Figure 2.3c). Such BRDF is an instance of the *Dirac delta function* mentioned in the previous section: each incident direction has associated only single exitant direction with non-zero BRDF value (the direction of ideal specular reflection, computed using the well-known *law of reflection*). Its values are:

$$f_S(\mathbf{x}, \; \vec{\omega}_i, \; \vec{\omega}_o) = \begin{cases} +\infty & \text{if } \vec{\omega}_o \text{ is the ideal reflection of } \vec{\omega}_i, \\ 0 & \text{otherwise.} \end{cases}$$

Integral of this BRDF multiplied by the cosine is ρ_s – *specular albedo* of the surface, i.e. fraction of incident radiance that is specularly reflected.

We can create a new BRDF as mixture (convex combination) of others, just like we can create a mixture density from several probability density functions. We can for example create a diffuse BRDF with ideal specular component as:

$$f_{mix} = \alpha f_D + (1-\alpha) f_S; \qquad 0 < \alpha < 1.$$

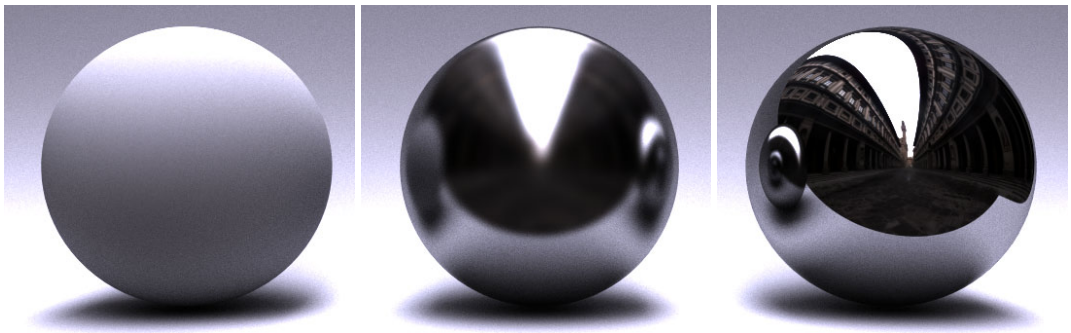
Another example of such mixture is the *extended Phong BRDF* (Figure 2.3b) [LW94]. It is a mixture of diffuse BRDF and imperfect specular one. It is based on traditional Phong reflection model of local

illumination [Pho75], which is made into a BRDF and modified to be physically correct (based on the rules from previous section):

$$f_r(\mathbf{x},\, \vec{\omega}_i,\, \vec{\omega}_o) \;=\; \frac{\rho_d}{\pi} + \frac{(n+2)}{2\pi}\,\rho_s\,\cos(\vec{\omega}_o,\, \vec{\omega}_s)^n; \quad \rho_d + \rho_s \leq 1, \tag{2.9}$$

$\vec{\omega}_s$ is the ideal reflection of $\vec{\omega}_i$.

The specular component is the strongest in the direction of ideal reflection and gradually decreases with increasing deviation from it. Single parameter, *specular exponent n* describes how rapid this change is and as a consequence how blurred is the reflection. Value of 1 produces almost diffuse reflection and values approaching infinity produce almost ideal mirror. Notice the multiplicative term of specular component which causes the amount of energy specularly reflected to be dependent only on ρ_s and not on n.



(a) *Diffuse BRDF.* **(b)** *Glossy Phong BRDF.* **(c)** *Ideal mirror BRDF.*

Figure 2.3: *This scene demonstrates how different BRDFs can alter material appearance. The left sphere has an ideal diffuse material, middle has highly glossy Phong BRDF and the last one has an ideal mirror material.*

Phong BRDF is an example of *empirical model* – simple model which works well in practice. Other examples of models falling into this category are Ward [War92b] and Lafortune [LFTG97] BRDFs. There is another category of *physically-based models*, that are based on simulating the light interaction on *microfacets* of the surface. Examples include Cook-Torrance [CT81] and Torrance-Sparrow [TS67]. Last possibility is to physically measure the BRDF and use these values directly [DR05]. Although there are several difficulties with this method, for example with data acquisition and compression, it seems promising in the future.

2.2.3 BRDF generalizations

A BRDF can only capture limited range of light interactions with the surface. So far we have only considered directions on the hemisphere over a surface. In order to describe the refraction of light, we need to extend the domain to full sphere. Such extended function is called *bidirectional scattering distribution function* (BSDF); its refractive component description is called the *bidirectional transmittance function*. It employs the *Snell law* which describes the relation between angle of incidence θ_1 and angle of refraction θ_2 using indices of refraction of the two optical media n_1 and n_2:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1}.$$

A BRDF can vary in space, for example on textured objects. This is extremely common in the real world. Functions describing it have two additional arguments for space parametrization, making them 6-dimensional in total (7-dimensional when also considering the light wavelength as another argument). These functions are called *bidirectional texture functions* (BTF) [FH09] or *spatially-varying bidirectional reflectance distribution functions* (SVBRDF) [NRH+77].

We have naturally assumed that once a ray hits surface, it is reflected from the same point. This is however not always true as there exist materials exhibiting *subsurface scattering* (SSS) phenomenon. When a ray hits such material in one point, it is absorbed at first, travels randomly inside, and then exits the surface at another point. This gives the material distinctive "soft" look. This behaviour is mathematically described by the *bidirectional surface scattering reflectance distribution function* (BSSRDF) [JMLH01].

2.3 Rendering equation

We can compute exitant radiance from single surface point $\mathbf{x}$ in any direction $\vec{\omega}_o$ if we know its BRDF $f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o)$, distribution of incident radiance $L(\mathbf{x} \leftarrow \vec{\omega}_i)$ and distribution of surface own radiance emission $L_e(\mathbf{x} \rightarrow \vec{\omega}_o)$ by integrating the product of incident radiance and BRDF and adding the surface emission. This is called the *reflection equation*:

$$L(\mathbf{x} \rightarrow \vec{\omega}_o) = L_e(\mathbf{x} \rightarrow \vec{\omega}_o) + \int_\Omega L(\mathbf{x} \leftarrow \vec{\omega}_i)\, f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) \cos\theta_i \, \mathrm{d}\vec{\omega}_i. \tag{2.10}$$

We can however solve this equation only if we know the incident radiance distribution. It is generally unknown, but because, as we have postulated before, radiance stays constant along straight lines we can replace the incident radiance with exitant radiance from another surface. This gives us the *rendering equation*, as first formulated by Kajiya in 1986 [Kaj86]:

$$L(\mathbf{x} \rightarrow \vec{\omega}_o) = L_e(\mathbf{x} \rightarrow \vec{\omega}_o) + \int_\Omega L(r(\mathbf{x}, \vec{\omega}_i) \rightarrow -\vec{\omega}_i)\, f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) \cos\theta_i \, \mathrm{d}\vec{\omega}_i. \tag{2.11}$$

$r(\mathbf{x}, \vec{\omega}_i)$ is the *ray-casting function* – a function that returns the closest surface from $\mathbf{x}$ in direction $\vec{\omega}_i$. The unknown quantity, $L(\mathbf{x}, \vec{\omega}_o)$ is present on both sides of the equation, once inside the integral. This makes the equation very hard to solve.

This formulation is called the *hemispherical formulation*, because we integrate over all directions on hemisphere around $\mathbf{x}$. It is also possible to integrate over the set A of all surfaces in the scene. The corresponding *area formulation* of the rendering equation is:

$$L(\mathbf{x} \rightarrow \vec{\omega}_o) = L_e(\mathbf{x} \rightarrow \vec{\omega}_o) + \int_A L(\mathbf{y}, \mathbf{y} \rightarrow \mathbf{x}) f_r(\mathbf{x}, \mathbf{x} \rightarrow \mathbf{y}, \vec{\omega}_o)\, V(\mathbf{x}, \mathbf{y})\, \frac{\cos\theta_x \cos\theta_y}{\|\mathbf{x} - \mathbf{y}\|^2} \, \mathrm{d}A_\mathbf{y}. \tag{2.12}$$

The $V(\mathbf{x}, \mathbf{y})$ term is called *visibility term* and is defined as follows:

$$V(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 & \text{if } \mathbf{y} \text{ is visible from } \mathbf{x}, \\ 0 & \text{otherwise.} \end{cases}$$

2.4 Solving the rendering equation

Computing global illumination in the scene means solving the rendering equation. Because the equation has no analytical solution in general scene, we must settle for approximate solutions. Such solutions produce errors which are perceived in the image as *artefacts* – various splotches, blurred-out details, or noise (Figure 2.4). Wide variety of algorithms was developed for solving the light transport problem. They differ in principle, speed, quality of result, artefacts produced, and other aspects. But before we can discuss examples of these algorithms, we need to introduce ray casting and Monte Carlo methods as common elements used by these solvers.

(a) *Monte Carlo path tracing.* **(b)** *Photon mapping.* **(c)** *Irradiance caching.*

Figure 2.4: *Examples of artefacts created by different GI algorithms. Left image (Monte Carlo path tracing) shows high-frequency noise. Middle image (photon mapping) exhibits typical low-frequency noise. The last image (irradiance caching) shows splotches from imperfect interpolation together with overall blurriness of the GI.*

2.4.1 Ray casting

From the realistic image synthesis point of view ray casting is an efficient way to calculate the visibility terms in the rendering equation. The algorithm was first introduced by Appel in 1968 [App68]. In ray casting, we take a *ray* (in this context a geometric half-line) and search for its closest intersection with the scene. Naïve algorithm does this by individually testing the ray for intersection with each object (primitive) in the scene and returning the closest intersection.

Intersection of the ray with a parametric or implicit surface can be computed by solving a system of equations. A fairly simple examples for sphere and triangle are given in [SM03]. An example of more complicated but faster algorithm for triangle intersection is the *Möller-Trumbore intersection test* [MT97]. One of the advantages of ray casting is that we can use it to visualize any primitive for which we are able to find its intersection with a ray.

In practice it would be too slow to test each ray for intersection with each primitive, as scenes can contain millions of polygons. Because of this specialized ray casting *acceleration data structures* are used. They store additional data about the spatial distribution of objects to achieved sub-linear expected query times. An example of such data structure is the *kd-tree* [MB90, Hav00], which recursively partitions the space by axis-aligned planes to create analogue to binary search tree in higher dimensions. *Bounding volume hierarchy* (BVH) [GS87] is another data structure that works similarly, but partitions primitives instead of space, creating a hierarchy of their axis-aligned bounding boxes.

Both BVH and kd-tree have $O(\log n)$ expected query time, but there are large variable factors hidden behind the big-O notation. Overall ray casting performance is dependent on quality of the tree, which is determined by its build algorithm. Simple approach of creating a balanced tree in terms of number of primitives or amount of space in each sub-tree works, but much better results are obtained with the *surface area heuristic* (SAH) [GS87]. It is simple recursive cost function predicting computational cost of each sub-tree traversal. It is derived from observation that probability of a random uniformly distributed ray hitting a convex object is proportional to its surface area. This leads to the SAH formula:

$$SAH(X) = \begin{cases} N \cdot C^i & \text{if X is leaf,} \\ C^t + SAH(L)\frac{S(L)}{S(X)} + SAH(R)\frac{S(R)}{S(X)} & \text{if X is inner node,} \end{cases}$$

where C^i and C^t are costs of intersecting a primitive and traversing an inner node, L, R are left and right children of the node, N is number of primitives and S is the surface area of node bounding volume. The formula estimates cost of traversing leaf node as cost of intersecting all its primitives and cost of traversing inner node as sum of its children costs multiplied by the probability a ray hits them. The SAH function is evaluated for multiple possible split positions during each build step and best split is then selected. Recursion in the SAH function is eliminated by approximating *SAH(L)* and *SAH(R)* by leaf node formula. Using the SAH greatly increases ray casting performance (by orders of magnitude) at cost of more complicated and slower build [Hav00].

Different example of acceleration data structure is the *uniform grid* [ISP07]. It is non-recursive non-adaptive partition of space into set of cells with $O(\sqrt[3]{n})$ expected query time. Its biggest advantage is the $O(n)$ build time (compared to $O(n \log n)$ time for tree data structures), but its inability to deal with non-uniform primitives distribution makes it impractical for many scenes [HKH11].

We need two ray casting operations for solving the rendering equation. First is finding the closest surface visible from some point $\mathbf{x}$ in direction $\vec{\omega}$. This operation appears in the hemispherical formulation of rendering equation (Equation 2.11) as function r. Evaluation of $r(\mathbf{x}, \vec{\omega}_i)$ means casting a ray with origin in $\mathbf{x}$ and direction of $\vec{\omega}$ and returning the closest intersection found. The area formulation of rendering equation (Equation 2.12) features the visibility function V instead of r. Evaluation of $V(\mathbf{x}, \mathbf{y})$ involves casting a *shadow ray* which originates at $\mathbf{x}$ and has direction of $\mathbf{y} - \mathbf{x}$. Shadow ray are traced differently – we are not interested in the closest intersection, but instead in the mere fact if there exists an intersection on the line segment between $\mathbf{x}$ and $\mathbf{y}$. This modification permits more efficient traversal of some acceleration data structures.

2.4.2 Monte Carlo integration

One apparent problem in evaluation of the rendering equation is the presence of an integral. It is clear that it cannot be solved analytically for arbitrary scene. We have to resort to an approximation by numerical integration. There is wide variety of deterministic methods, from simple quadrature rule to more sophisticated ones [DB08], but instead stochastic Monte Carlo methods [Met87] are used almost exclusively. It is because they are easy to implement, their convergence rate is independent of the integration domain dimension, and errors produced by them manifest in image as noise, which is not as disturbing as regular patterns produced by deterministic methods.

Monte Carlo integration works by evaluating the integrated function using random samples. Imagine we would like to compute the integral of function f:

$$\int_a^b f(x)\,\mathrm{d}x.$$

We need to select N random samples x_i over the integration domain. We select them with probability described by probability density function $p(x)$. *Monte Carlo estimator* $\langle I \rangle$ estimating value of the integral is then simply:

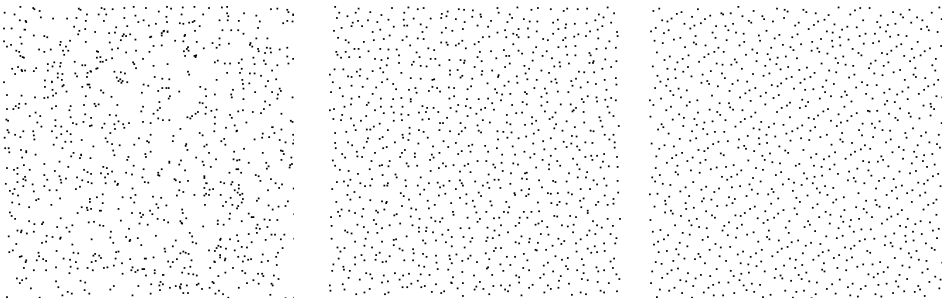
$$\langle I \rangle \;=\; \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)}.$$

Variance of the estimator decreases linearly with N. Standard deviation σ decreases at rate $\sqrt{N}$. This means that to reduce the error in image to half we need to quadruple the number of samples. This is somewhat unfortunate, as it means that we need a lot of samples to get image without visible errors. Each sample evaluation typically involves casting a ray, which is generally an expensive operation. There are several strategies to increase the rate of convergence. They all exploit the way we generate the random samples.

Stratified sampling breaks a single integral over whole domain into sum of integrals over a set of sub-domains:

$$\int_{\alpha_1}^{\alpha_n} f(x)\,\mathrm{d}x \;=\; \sum_{i=1}^{n-1}\int_{\alpha_i}^{\alpha_{i+1}} f(x)\,\mathrm{d}x; \qquad \alpha_i \;\leq\; \alpha_{i+1}.$$

Each subdomain is computed with a single sample. All strata should have equal size (with respect to the underlying PDF). Stratified sampling is implemented implicitly by using stratified random sampling (first dividing the sampled domain to several sub-domains and then generating single sample for each one). Expected variance of stratified estimator is guaranteed to be lower or in worst case equal to that of standard one [DBB06, p. 70].



(a) *Random points.* **(b)** *Stratified points.* **(c)** *Halton points.*

Figure 2.5: *A visualization of 900 random samples generated in 2D domain. It is clearly visible that stratified and low discrepancy sampling is superior to simple random sampling in terms of uniformity and clumping prevention.*

The stratified sampling is better because it prevents "clumping" of random samples (Figure 2.5). Unfortunately, it has also disadvantages: its optimal number of samples grows exponentially with dimension

and the number of samples needs to be known in advance, which may cause implementation troubles. There are other ways to prevent samples clumping, such as *low discrepancy sampling*. This method does not use any random numbers, instead it uses deterministic *low discrepancy sequence* (sequence that fills the space with samples with minimal clumping). Because no random numbers are used, this method is called *quasi-Monte Carlo*. There are several low discrepancy sequences: Halton (Figure 2.5c) [Hal64], Hammersley [WLH97], Sobol [BF88] and others.

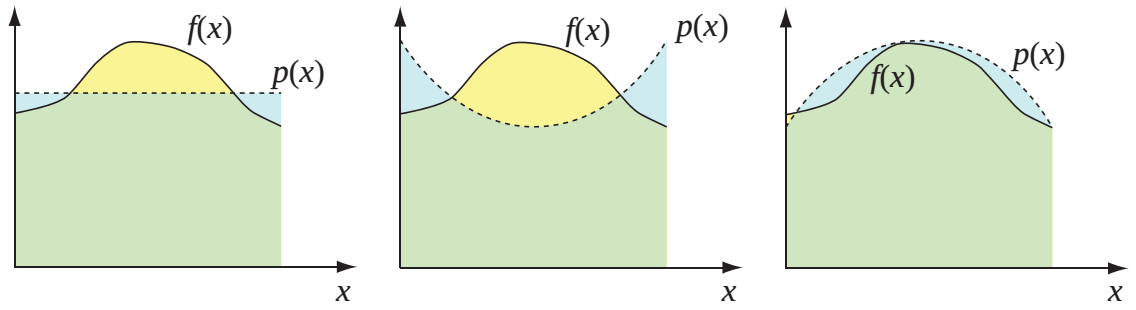
Another effective way to reduce the variance is to use better PDF for generating samples. This is called *importance sampling*. When integrating non-negative function $f(x)$ we would intuitively like to place more samples where the value of $f(x)$ is high, as these regions influence the integral value the most. Optimal PDF in this sense is:

$$p_{opt}(x) = \frac{f(x)}{\int_a^b f(x)\,\mathrm{d}x}.$$

It is easy to show that estimator using this PDF has zero error for any number of samples:

$$\langle I \rangle = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p_{opt}(x_i)} = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{\frac{f(x_i)}{\int_a^b f(x)\,\mathrm{d}x}} = \frac{1}{N}\sum_{i=1}^{N}\int_a^b f(x)\,\mathrm{d}x = \int_a^b f(x)\,\mathrm{d}x.$$

The problem is that to obtain this PDF we would need to know the value of $\int_a^b f(x)\,\mathrm{d}x$, which is what we are trying to compute in the first place. We can however still greatly lower the estimator error by using importance sampling according to terms of integrand which we do know in advance. Our goal is for the PDF to match the shape of integrand as closely as possible (Figure 2.6). However if we do not have any knowledge about the integrand shape, it is best to sample with uniform (constant) PDF, as wrong PDF can actually increase the estimator variance.



(a) *No importance sampling (uniform PDF).*

(b) *Bad importance sampling.*

(c) *Goood importance sampling.*

Figure 2.6: *Three examples of different probability density functions $p(x)$ that can be used for sampling $f(x)$. Left image shows uniform PDF (no importance sampling). Middle image shows an example of bad PDF that does not match the shape of $f(x)$ and will increase the variance. The last image shows an example of good PDF that matches the shape of $f(x)$ very closely and will significantly reduce the variance.*

2.4.3 Monte Carlo path tracing

Monte Carlo path tracing is straightforward application of Monte Carlo sampling for solving the rendering equation introduced by Kajiya [Kaj86]. It computes GI only for surfaces visible from the camera. Colour of single sample in image[2] corresponds to single radiance value $L(\mathbf{o},\ \vec{\omega}_{a,b})$, where $\mathbf{o}$ is the origin of camera and $\vec{\omega}_{a,b}$ is the direction of ray from $\mathbf{o}$ through point in image plane with coordinates (a,b). As we have shown previously, this is equal to $L(r(\mathbf{o},\ -\vec{\omega}_{a,b}),\ -\vec{\omega}_{a,b})$. Use of the ray casting function r means that we first cast a *primary ray* from the camera.

Let us assume that the ray hits a surface point $\mathbf{x}$ with associated BRDF f_r. Rendering equation states that the exitant radiance in direction $-\vec{\omega}_{a,b}$ is:

$$L(\mathbf{x} \to -\vec{\omega}_{a,b}) = L_e(\mathbf{x} \to -\vec{\omega}_{a,b}) + \int_\Omega L(r(\mathbf{x},\ \vec{\omega}_i) \to -\vec{\omega}_i)\, f_r(\mathbf{x},\ \vec{\omega}_i,\ -\vec{\omega}_{a,b}) \cos\theta_i \,\mathrm{d}\vec{\omega}_i.$$

$L_e(\mathbf{x} \to -\vec{\omega}_{a,b})$ is known from the scene description. Monte Carlo estimator for the integral using general PDF p is:

$$\langle L(\mathbf{x} \to -\vec{\omega}_{a,b})\rangle \;=\; \frac{1}{N}\sum_{n=1}^{N} \frac{L(r(\mathbf{x},\ \vec{\omega}_{i_n}) \to -\vec{\omega}_{i_n})\, f_r(\mathbf{x},\ \vec{\omega}_{i_n},\ -\vec{\omega}_{a,b}) \cos\theta_{i_n}}{p(\vec{\omega}_{i_n})}. \tag{2.13}$$

We can use the importance sampling according to BRDF and cosine terms (assuming reasonable BRDF), but not according to the radiance term as it is unknown. This gives us estimator:

$$\langle L(\mathbf{x} \to -\vec{\omega}_{a,b})\rangle' \;=\; \rho\frac{1}{N}\sum_{n=1}^{N} L(r(\mathbf{x},\ \vec{\omega}_{i_n}) \to -\vec{\omega}_{i_n}) \tag{2.14}$$

using probability density function

$$p(\vec{\omega}_{i_n}) \;=\; \frac{f_r(\mathbf{x},\ \vec{\omega}_{i_n},\ -\vec{\omega}_{a,b}) \cos\theta_{i_n}}{\rho}. \tag{2.15}$$

For each sample we take we need to cast another ray evaluating $r(\mathbf{x},\ \vec{\omega}_i)$ and recursively evaluate the rendering equation. Obvious problem in this approach is how to terminate the recursion as it would never end in closed environment on its own. Limiting the depth of recursion to any fixed number would cause bias. Better approach is to use *Russian roulette*. It is stochastic algorithm for terminating path tracing paths. Suppose we want to compute value of function $f(x)$. Russian roulette estimator generates single random number α uniformly distributed in interval $[0,1]$ and returns:

$$\langle I_{RR}\rangle = \begin{cases} \frac{1}{P} f(x) & \alpha \le P \\ 0 & \alpha > P \end{cases}$$

P is a constant between 0 and 1. This estimator is unbiased for any valid P. Suitable choice of P is the total surface albedo ρ. Chance that ray terminates in each bounce is $1-P$, chance that the ray survives n rounds of Russian roulette is P^n. Rays in lower level of recursion are therefore efficiently terminated without introducing any bias.

[2] We intentionally avoid the term pixel, because pixel has non-zero area and to get its colour, we have to integrate over it.

Another problem is the ray branching – if our estimator uses N samples on each recursion level there can be as many as N^x rays traced at x-th level of recursion, each of which with very small contribution to the final image. We prevent this by using only single random sample in the estimator during the recursion. This eliminates branching of rays; single continuous path (also called *random walk*) is created instead as shown in Figure 2.7. Because of this the method is called *path tracing*.

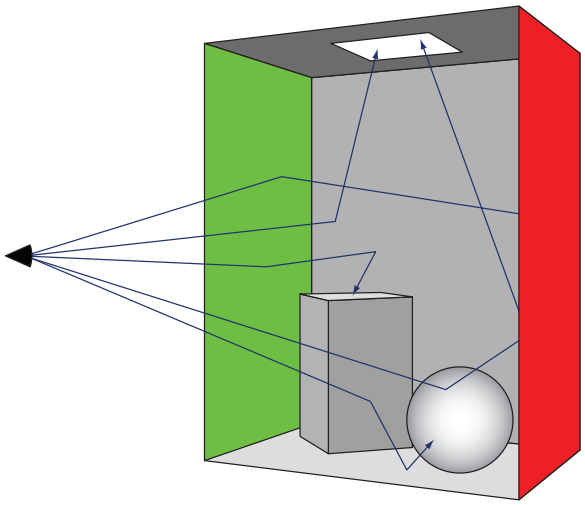


Figure 2.7: *Random walks generated by the Monte Carlo path tracing algorithm. Several rays are shot from the camera into the scene. These rays bounce from surface to surface until they are absorbed or leave the scene. Only a very small fraction of rays hits the light.*

Variance of the path tracing estimator creates large amounts of noise in final image. To decrease it, multiple paths are traced through single pixel. Hundreds to tens of thousands different paths may be used. Notice that the path needs to hit a light to return non-black colour. This is a problem in scenes with small bright light sources, because the probability of a path hitting them is very small. *Explicit light sampling* is used to deal with this. It splits the illumination integral into a direct and indirect part. Direct illumination is due to emitted radiance L_e incoming directly from other surfaces (lights), indirect is due to other (reflected) incoming radiance. The indirect part is solved by the recursive Monte Carlo sampling of hemisphere, we just no longer add the material emission L_e to the result. For computing the direct component we modify the area formulation of rendering equation (Equation 2.12) – we integrate emitted radiance L_e over the area of all light emitting surfaces in the scene:

$$L_{direct}(\mathbf{x} \to \vec{\omega}_o) = \int_{A_{\text{lights}}} L_e(\mathbf{y}, \mathbf{y} \to \mathbf{x})\, f_r(\mathbf{x}, \mathbf{x} \to \mathbf{y}, \vec{\omega}_o)\, V(\mathbf{x}, \mathbf{y})\, \frac{\cos\theta_x \cos\theta_y}{\|\mathbf{x} - \mathbf{y}\|^2}\, \mathrm{d}A_{\mathbf{y}}. \tag{2.16}$$

Monte Carlo estimator for this integral using general PDF $p(\mathbf{y}_i)$ is:

$$\langle L_{direct}(\mathbf{x} \to \vec{\omega}_o) \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{L_e(\mathbf{y}_i, \mathbf{y}_i \to \mathbf{x})\, f_r(\mathbf{x}, \mathbf{x} \to \mathbf{y}_i, \vec{\omega}_o)\, V(\mathbf{x}, \mathbf{y}_i)\, \frac{\cos\theta_x \cos\theta_{y_i}}{\|\mathbf{x} - \mathbf{y}_i\|^2}}{p(\mathbf{y}_i)}.$$

Evaluation of this integral is non-recursive. Because use the visibility function $V(\mathbf{x}, \mathbf{y})$ instead of ray casting function, direct light calculation involves casting shadow rays. By explicitly sampling lights we get much better convergence. In addition *point lights* cannot be used without explicit light sampling as

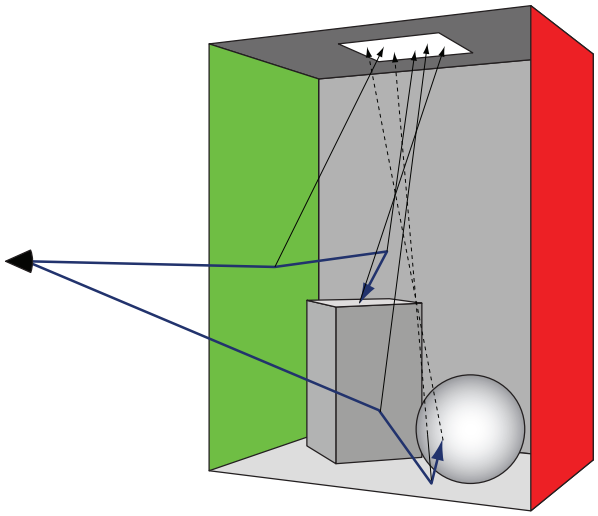


Figure 2.8: *The path tracing algorithm with explicit direct lighting. Any time a ray hits a surface in the scene we shoot a shadow ray towards the light. Because of that paths no longer have to hit the light to result in non-zero illumination.*

they have no area and therefore cannot be hit during the hemisphere sampling. Path tracing with explicit light sampling is depicted in Figure 2.8.

Path tracing is simple, unbiased algorithm derived by direct application of the Monte Carlo integration and Russian roulette to the rendering equation. It does not need any precomputation phase and the only error it produces is noise in the rendered image, which can be lowered by adding more samples. Usually, several thousand paths need to be traced for a single pixel to get rid of the noise. The random walk which the algorithm produces mimics how photons bounce and get absorbed through the scene, with the difference that path tracing random walk starts at camera and may hit a light, but photons are emitted at lights and may hit the camera.

2.4.4 Photon mapping

Another approach to solving the rendering equation is the *photon mapping* algorithm introduced by Jensen [Jen96]. It is a biased two-pass method which first emits a set of *photons* from light sources, traces them through the scene, stores them in a data structure called *photon map* and then computes the global illumination by querying this structure. The term "photons" in this algorithm refers to conceptual particles carrying flux information, not to photons in their physical sense as light radiation particles carrying energy.

The algorithm operates as follows: we are given scene description including emission function $L_e(\mathbf{x}, \vec{\omega})$. We discretize the radiant flux resulting from L_e into N samples, which means we emit N photons. Each photon has random origin $\mathbf{o}_n$ and random direction $\vec{\omega}_n$. We choose them with probability density function:

$$p(\mathbf{o}_n, \vec{\omega}_n) = \frac{L_e(\mathbf{o}_n, \vec{\omega}_n)}{\Phi_{\text{total}}}.$$

As a result, each photon carries the same portion of the radiant flux $\frac{\Phi_{\text{total}}}{N}$. Because the flux is discretized, we can easily propagate it through the scene by recursively reflecting it off its surfaces. We use the surface BRDF multiplied by cosine (similarly to path tracing – Equation 2.15) to create the probability density function for selecting the direction in which the photon reflects. Before the reflection the algorithm randomly decides if the photon should be absorbed (with absorption probability of $1-\rho$). The propagation is therefore terminated using the Russian roulette. Because of the way we choose the next interaction reflecting the photon does not change its associated flux. We store each photon position and incoming direction every time it interacts with a surface.

After the emission phase we build a kd-tree over all photon hits recorded. This ends the precomputation phase. To visualise the result we need to perform *density estimation*. We start with the reflection equation:

$$L(\mathbf{x} \to \vec{\omega}_o) \;=\; L_e(\mathbf{x} \to \vec{\omega}_o) + \int_\Omega L(\mathbf{x} \leftarrow \vec{\omega}_i)\, f_r(\mathbf{x},\ \vec{\omega}_i,\ \vec{\omega}_o) \cos\theta_i \,\mathrm{d}\vec{\omega}_i.$$

This equation uses incoming radiance, but we have only radiant flux information available. We therefore substitute radiant flux according to Equation 2.1:

$$L(\mathbf{x} \to \vec{\omega}_o) \;=\; L_e(\mathbf{x} \to \vec{\omega}_o) + \int_\Omega \frac{\mathrm{d}^2\,\Phi_i(\mathbf{x},\ \vec{\omega}_i)}{\mathrm{d}A}\, f_r(\mathbf{x},\ \vec{\omega}_i,\ \vec{\omega}_o).$$

We approximate $\mathrm{d}A$ by surface area of sphere centred at $\mathbf{x}$ with radius r. This allows us to estimate the integral by summing fluxes Φ_p of all photons inside the sphere:

$$L(\mathbf{x} \to \vec{\omega}_o) \;\approx\; L_e(\mathbf{x} \to \vec{\omega}_o) + \sum_{p=1}^{n} \frac{\Phi_p}{\pi r^2} f_r(\mathbf{x},\ \vec{\omega}_p,\ \vec{\omega}_o).$$

To get all photons inside a sphere, we perform a *k-nearest neighbour* (k-NN) query in the kd-tree. This sets the value of r to distance to k-th nearest neighbour. Adjusting the parameter k allows us to trade estimator variance for bias – higher value will decrease the noise, but they will also blur the result. Commonly used values are 50–250. The two phases of photon mapping are illustrated in Figure 2.9.

The photon map stores the flux information decoupled from scene geometry. This means that photon mapping can be used with arbitrary scene geometry. The method is also easy to implement, precomputation is short and non-diffuse BRDFs are supported. Unfortunately, it has one major drawback – the density estimation produces artefacts which are very hard to get rid of (Figure 2.10a). We could try increasing the number of emitted photons and the number of nearest neighbours used, but this would result in too high memory usage even for simple scenes.

Better solution is to use *final gathering*. We can view the photon mapping as a *bidirectional method* because it first creates paths from lights (photons), then paths from camera (primary rays) and connects them through the density estimation. Length of photon paths is arbitrary, but camera paths have always length of 1. Final gathering extends camera paths by tracing an extra bounce of the ray from primary hit point. To eliminate the noise we trace several hundreds or thousands of rays from the primary hit point, using importance sampling according to the product of BRDF and cosine, as in path tracing (Equations 2.14 and 2.15). This method eliminates the low frequency noise of photon map (Figure 2.10b), but the number of rays traced and photon map lookups is orders of magnitude higher, which makes final gathering very slow. Irradiance cache can be used to speed up the process, as we show in the next chapter.

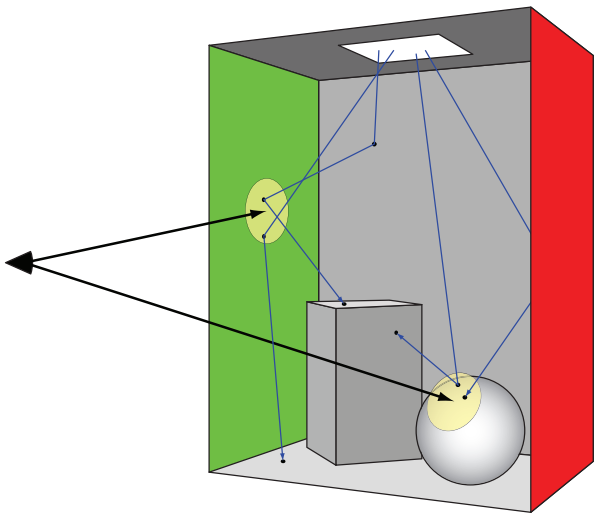
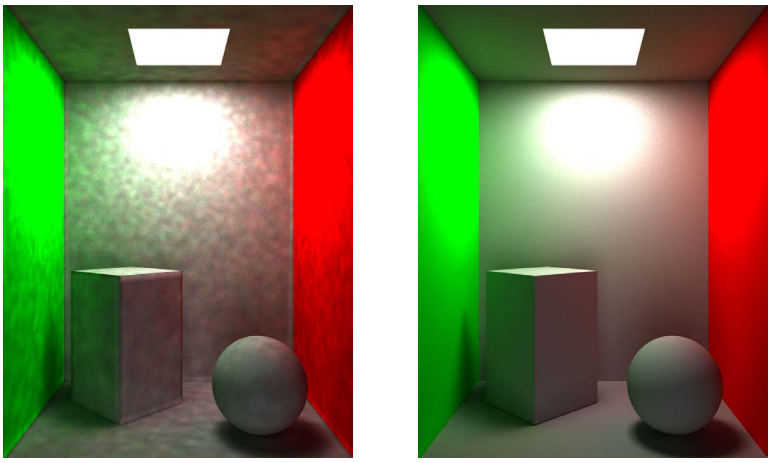


Figure 2.9: *An illustration of the photon mapping in a simple scene. First photons are traced from the light and stored (blue rays). Then primary rays are shot and in each of their hit points nearby photons are located to perform the density estimation.*



(a) *Direct photon map visualization.* **(b)** *Using final gathering.*

Figure 2.10: *Differences between direct photon map visualization and final gathering. In the left image we visualize the photon map directly by performing the density estimation in each primary ray hit point, in the right image we first shoot several (1000) final gather rays and perform the density estimation at their hit points. The second image does not suffer from artefacts but it took significantly longer to compute.*

Chapter 3

Irradiance and radiance caching

Irradiance caching is another global illumination algorithm. It was invented more than 20 years ago by Ward et. al. [WRC88]. It exploits that indirect irradiance in scene changes very slowly over most surfaces and as a consequence it can be easily interpolated from a sparse set of samples. Because of its simplicity and efficiency irradiance caching is nowadays the preferred algorithm for GI computation in the majority of production renderers [KG09].

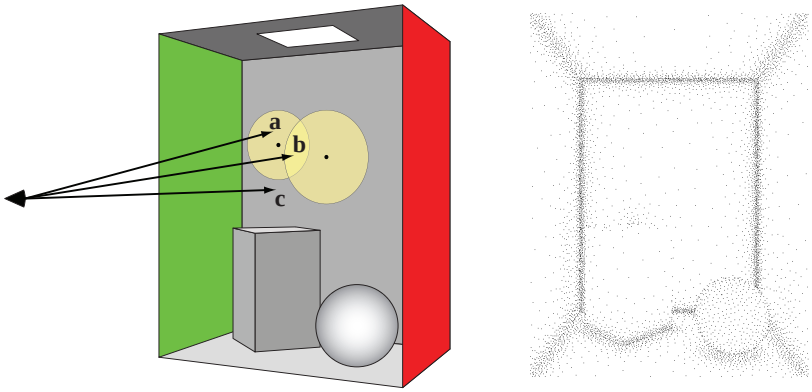
Because the algorithm caches only the irradiance it works only with diffuse surfaces. This major weakness of irradiance caching led to creation of *radiance caching* by Křivánek et. al. [KGPB05]. This algorithm allows interpolation of illumination on glossy surfaces by storing directional information about incoming radiance in form of *spherical harmonics* vectors. Although it performs well for low-frequency BRDFs, it fails for high-frequency ones. In addition this method requires converting all materials BRDFs to spherical harmonics representation.

Because of these shortcomings, an alternative algorithm was introduced by Gassenbauer et. al. [GKB09]. The *spatial directional radiance caching* also stores directional information, but directly in form of discrete radiance samples. This removes the need for special BRDF representation and also allows use of BRDF importance sampling, which is essential for efficient rendering of highly glossy materials.

3.1 Irradiance caching

The irradiance caching algorithm is based on *lazy evaluation* of irradiance: for each irradiance query we first try to interpolate the result from existing records, and if that fails we create new record at the point of query. This process is illustrated in Figure 3.1a. The cache stores incident irradiance values for few selected surface points (records) in the scene (Figure 3.1b). The interpolation takes place in the world space, but all created records are directly visible from the camera. Because of that no unnecessary work for surfaces not visible in image is needed, unlike in radiosity or photon mapping.

The records store illumination information only in form of *incident irradiance* E; no directional information is stored. This implies the need for diffuse BRDF (independent of directions). We can compute the



(a) *Irradiance caching visualization.* **(b)** *Record positions.*

Figure 3.1: *Irradiance caching example in a simple scene. Two records with areas where they can be reused (yellow) are shown in the left picture. Three primary rays trigger three interpolations in the cache: point* **a** *is interpolated using single record, point* **b** *is interpolated using both records, and point* **c** *cannot be interpolated; a new record has to be created in it. Because of the caching the irradiance needs only to be computed in few selected points, shown in the right picture.*

exitant radiance $L(\mathbf{x} \to \vec{\omega})$ in any direction $\vec{\omega}$ due to incident irradiance $E(\mathbf{x})$ for surface $\mathbf{x}$ with diffuse albedo $\rho_d(\mathbf{x})$ as:

$$L(\mathbf{x} \to \vec{\omega}) = \frac{\rho_d(\mathbf{x})}{\pi} E(\mathbf{x}). \tag{3.1}$$

3.1.1 Record creation

If the interpolation at surface point $\mathbf{x}$ is unsuccessful (because there are no usable records nearby) we need to compute a new record. Recall that incident irradiance is an integral of incident radiance weighted by cosine:

$$E(\mathbf{x}) = \int_\Omega L(\mathbf{x} \leftarrow \vec{\omega}) \cos\theta \, \mathrm{d}\vec{\omega}. \tag{3.2}$$

We solve this integral by the Monte Carlo method, using stratified sampling and importance sampling according to the cosine term. Stratification breaks the hemisphere into a set of cells; single ray through a random point in each cell is traced. The probability density function for this importance sampling is:

$$p(\vec{\omega}) = \frac{\cos\theta}{\pi}. \tag{3.3}$$

Estimator for integral 3.2 using this PDF is:

$$\langle I \rangle = \frac{1}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} \frac{L(\mathbf{x} \leftarrow \vec{\omega}_{ij}) \cos\theta}{p(\vec{\omega}_{ij})} = \frac{\pi}{MN} \sum_{i=1}^{M} \sum_{j=1}^{N} L(\mathbf{x} \leftarrow \vec{\omega}_{ij}).$$

We use $M \cdot N$ samples due to the stratification which divides θ to M and ϕ to N strata. We usually needs several hundred or thousand samples to get a good estimate. M to N ratio can be arbitrary, although

Křivánek and Gautron [KG09] suggests $N \approx \pi M$. Note that we need to compute $L(\mathbf{x} \leftarrow \vec{\omega}_{ij})$. We will discuss possible ways to do so in separate chapter. During the hemisphere sampling we also calculate *distance to other visible surfaces* R and *irradiance gradients*. Both of these will be discussed later. After the irradiance cache record is created it is inserted in the cache.

3.1.2 Interpolation

To interpolate the irradiance at given point $\mathbf{x}$ we need to select a set $S(\mathbf{x})$ of records which can possibly influence the value $E(\mathbf{x})$ and determine their weights $w_i(\mathbf{x})$. Interpolated irradiance is then simply:

$$E_{\text{interp.}}(\mathbf{x}) = \frac{\sum_{i \in S(\mathbf{x})} E_i \, w_i(\mathbf{x})}{\sum_{i \in S(\mathbf{x})} w_i(\mathbf{x})}. \tag{3.4}$$

To get formula for interpolation weights we use the *split sphere model*. This model describes one particular example of scene with rapidly changing irradiance: single flat surface enclosed in half white, half black sphere (Figure 3.2). It was introduced by Ward et. al. in the original paper on irradiance caching [WRC88]. The argued that under certain restricting conditions[1] this model provides an upper bound on interpolation error made by reusing an irradiance value at point with different position and normal. Interpolation weight is just inverse of this error bound minus user-adjustable term controlling the accuracy of interpolation (based on user selectable maximal permitted interpolation error a). Formula for the weight is:

$$w_i(\mathbf{x}) = \frac{1}{\frac{\|\mathbf{x}-\mathbf{p}_i\|}{R_i} + \sqrt{1-\mathbf{n}\cdot\mathbf{n}_i}} - \frac{1}{a}, \tag{3.5}$$

where $\mathbf{x}$ is position of the point where we interpolate the irradiance and $\mathbf{n}$ is its normal. Position and normal of i-th record in the cache is denoted $\mathbf{p}_i$, $\mathbf{n}_i$. R_i is the distance to visible surfaces from record i. Its value is also derived from the split sphere model as harmonic mean of lengths r_{ij} of the rays traced from $\mathbf{p}_i$ during the hemisphere sampling:

$$R_i = \frac{MN}{\sum_{i=1}^{N}\sum_{j=1}^{M}\frac{1}{r_{ij}}} \tag{3.6}$$

Tabellion and Lamorlette [TL04] suggest alternative interpolation scheme. Instead of a single error term which combines errors due to position and normals difference they use two separate terms – one for positions and one for normals – and interpolate using *maximum* of the two:

$$w_i(\mathbf{x}) \;=\; 1 \;-\; \alpha \max\left(\frac{\|\mathbf{x}-\mathbf{p}_i\|}{0.5R_i}, \frac{\sqrt{1-\mathbf{n}\cdot\mathbf{n}_i}}{1-\cos 10^\circ} \right), \tag{3.7}$$

where 10° and $0.5R_i$ are arbitrarily chosen constants specifying sensitivity to position and normal differences. This scheme has also user-adjustable parameter for interpolation accuracy – α. Note that, unlike the original scheme, maximum weight possible is always 1, even when a record has the same position and normal as the point being interpolated. Eliminating the singularity allows smoother interpolation

[1] We assume that there are no concentrated sources of indirect illumination, which is true in most scenes.

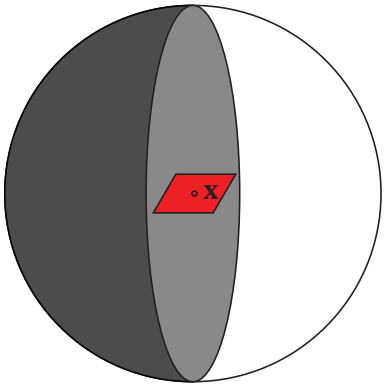


Figure 3.2: *The split sphere model scene. The scene features single surface containing the point to interpolate (* **x***). The environment is divided into white and black halfs, with the point lying exactly in between them.*

without some artefacts at cost of increased bias. This scheme also defines an alternative way to compute the R_i as *minimum* of ray lengths:

$$R_i = \min_{j,k} r_{ij}$$

Rays close to horizon should not be included in this version of R_i calculation, because that would cause trouble on slightly concavely curved surfaces. Using minimum instead of harmonic mean causes the algorithm to put more samples around small details, increasing overall accuracy.

3.1.3 Irradiance gradients

Irradiance gradients are an extension of the original algorithm, proposed by Ward and Heckbert [WH92]. They improved the algorithm by performing first-order (linear) approximation of the irradiance function during the interpolation. This means that during the interpolation we do not simply interpolate using the irradiance values stored in records, but we first use the rate of change to compute the estimate of each record irradiance projected to the query point. To be able to perform this approximation we need to calculate and store additional information for each record. Big advantage of the algorithm is that this computation can be done during the hemisphere sampling and is essentially free.

Irradiance of a surface depends both on its position and normal. We are able to compute irradiance anywhere in scene using the hemisphere sampling. It can be therefore described by a five-dimensional scalar field[2] – three degrees of freedom are due to position and two due to rotation. To get the linear approximation of the rate of change in a single point, we need the first derivative (*gradient*). This gradient is single five-dimensional vector, but for simplicity and ease of computation we use two independent three-dimensional vectors instead – rotational and translational gradients. Additionally, both vectors have only two degrees of freedom – we constrain them to lie in the tangent plane of the surface. As a consequence we lose one degree of freedom of translational gradient. This is however not a problem, but because as we want to interpolate using only samples lying approximately in the plane of interpolated point, we do not need information about how the irradiance changes with translation in the direction of surface normal.

[2]We do not consider colour now for the sake of clarity.

The *rotational gradient* expresses how the irradiance changes with rotation of the surface normal. Its direction is the axis around which the rotation causes greatest increase in irradiance. This increase can be caused by new bright surfaces appearing on the horizon, or by already visible bright surfaces being projected higher over the horizon, where they will cause higher amount of irradiance due to the cosine weighting. Size of the rotational gradient vector is the rate of this change. The gradient is calculated during the hemisphere sampling using this formula:

$$\nabla_r(E) \approx \frac{\pi}{MN} \sum_{i=1}^{N} \left(\mathbf{v}_i \sum_{j=1}^{M} -L_{i,j} \tan \theta_j \right),$$

where

- (i, j) are indices of a hemisphere stratification cell,
- $L_{i,j}$ is incoming radiance from cell (i, j) and
- $\mathbf{v}_i$ is unit vector in the tangent plane in direction $\phi_i + \frac{\pi}{2}$.

The *translational gradient* indicates how the irradiance changes with translation in the surface tangent plane (as we have established before we do not consider translation perpendicular to this plane). This change can be caused by bright surfaces getting closer, moving higher on the horizon, or by their occlusion and dis-occlusion. Direction of this gradient is the direction of translation that yields highest increase in the irradiance, its magnitude is the rate of this change. Formula for the gradient is:

$$\nabla_t E \approx \sum_{i=1}^{N} \left(\mathbf{u}_i \frac{2\pi}{N} \sum_{j=2}^{M} \frac{\cos^2 \theta_{j_-} \sin \theta_{j_-}}{\min(r_{(i,j)},\, r_{(i,j-1)})} (L_{i,j} - L_{i,j-1}) + \right.$$
$$\left. \mathbf{v}_{i_-} \sum_{j=1}^{M} \frac{\cos \theta_j (\cos \theta_{j_-} - \cos \theta_{j_+})}{\sin \theta_{i,j} \min(r_{(i,j)},\, r_{(i-1,j)})} (L_{i,j} - L_{i-1,j}) \right), \tag{3.8}$$

where:

- i, j are indices of a hemisphere stratification cell,
- $L_{i,j}$ is incoming radiance from cell i, j,
- $r_{(i,j)}$ is length of the ray traced through cell i, j,
- θ_{j_-} and θ_{j_+} are the lower and upper bound of elevation angle of any cell x, j, respectively,
- ϕ_{i_-}, ϕ_{i_+}, and ϕ_i are the lower bound, upper bound, and centre value of azimuthal angle in any cell i, x, respectively,
- $\mathbf{u}_i$ is unit vector in tangent plane in direction ϕ_i and
- $\mathbf{v}_{i_-}$ is unit vector in tangent plane in direction $\phi_{k_-} + \frac{\pi}{2}$.

Interpolation with gradients is done by substituting the E_i in Equation 3.4 by $E_i^{\text{grad}}(\mathbf{x})$ which is calculated by performing the first-order approximation:

$$E_i^{\text{grad}}(\mathbf{x}) = E_i + \nabla_r E_i \cdot (\mathbf{n_i} \times \mathbf{n}) + \nabla_t E_i \cdot (\mathbf{x} - \mathbf{p_i}).$$

3.1.4 Two-pass rendering

Although the irradiance caching in its basic form does not need a precomputation pass because of the lazy evaluation scheme, it has one fatal flaw: results of the algorithm are dependent on the order in which are pixels rendered. For example when visiting pixels in the *scanline order*[3] we may create a record for first pixel, and then reuse it for several next pixels, until finally for next pixel it has zero weight. Then we create another record and use it for next few pixels. But even though each of these new records can also be used for interpolation of several previous pixels, we have not used it. This creates an error called *missed contribution* (Figure 3.3a).

Natural solution to this problem is to use *hierarchical refinement* – instead of the scanline order we first render only few sparse pixels to get low-resolution version of image and then progressively refine it to get the full resolution render. This pixel ordering not only significantly decreases the number of samples with missed contributions, but also lowers the number of cache records needed.

Even though the hierarchical refinement is great improvement, it does not eliminate the problem of missed contributions entirely. Because of that we use the *two-pass rendering*. This means that before the final image rendering we perform another rendering with the sole purpose of populating the cache. The populating is performed in multiple passes using the hierarchical refinement for maximum efficiency (Figure 3.3b). Results of this precomputation render are not shown[4], the interpolation is done only to see if it is possible or if a new record should be created.

Another advantage of this approach is that the user-adjustable maximum interpolation error constant a can be set to different values during each phase. By setting it low for precomputation and slightly higher for visualization we can get both enough details in the cache due to denser records created during the precomputation and smooth result of interpolation due to the higher permitted error during visualization (Figure 3.3c). Křivánek and Gautron suggest to multiply a by a constant from 1.4 to 1.5 [KG09].

(a) *Simple lazy evaluation.* **(b)** *Using precomputation pass.* **(c)** *Using precomputation pass and cache smoothing.*

Figure 3.3: *We render the same simple scene with different irradiance cache precomputation methods using low quality overall settings to amplify differences. First picture was created without precomputation; missed contribution artefacts are clearly visible. Second picture uses precomputation with hierarchical refinement. Interpolation has no discontinuities, but soft shadows are not smooth. Increasing maximum permitted error after precomputation in the third picture gives the best result.*

[3] That is for example from top to bottom, row by row, single row from left to right at a time.

[4] Although renderers often display it to user to show the progress and allow him to estimate how the final render will look.

3.1.5 Algorithm improvements

Many improvements were proposed to improve the quality of irradiance caching results. One of the most trivial is to reject records "in front of" the interpolated point. When we interpolate irradiance in $\mathbf{x}$ we calculate quantity $d_i(\mathbf{x})$ for each record i:

$$d_i(\mathbf{x}) = (\mathbf{x} - \mathbf{p}_i) \cdot \frac{(\mathbf{n} + \mathbf{n}_i)}{2},$$

where $\mathbf{n}$ is the normal of interpolated point and $\mathbf{p}_i$, $\mathbf{n}_i$ is the position and normal of i-th record. If $d_i(\mathbf{x})$ is smaller than a small negative constant we set weight of the record i to zero, because it lies in front of $\mathbf{x}$ (with respect to its normal). This prevents unoccluded records from being used for interpolation in various trenches and pits, as shown in Figure 3.4. One could argue that we need to do the same in the other direction, preventing records in grooves and pits from being reused on surrounding open surfaces. This is however taken care of automatically by the R_i, which is very small for any occluded sample.

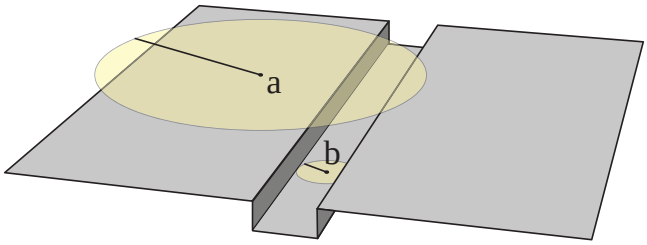


Figure 3.4: *A geometry detail with two irradiance cache records displayed with their distances to surfaces. We have to make the additional "in front of" test to make sure that the unoccluded record a is not reused in the groove. We do not however need any inverse test for record b as distances to surfaces of any records in the groove are very small.*

Another problem encountered is the clumping of records in corners. Record density is by default determined by their distances to surfaces. This is sufficient in most cases, but sometimes it creates unnecessary high record density near corners. The density there can be higher than 1 sample per pixel, causing the algorithm to degenerate into a brute force solution. In the same time missing close surfaces during the hemisphere sampling can cause some record R_i to be too big. Such record is then reused over too large area, which may result in artefacts. We solve this by imposing limits on the record spacing by limiting R_i to be between R^{min} and R^{max}. Because perspective camera projection changes apparent sizes of different objects in the scene, no single *world-space* limit would produce satisfactory results. We instead use *camera space* limits: each record has its own R_i^{min} and R_i^{max} that are used to clamp the value of R_i. User sets minimum and maximum record spacing in pixels and R_i^{min} and R_i^{max} are computed individually for each record by multiplying user-submitted values by the projected pixel size at each record location. Recommended values are 1-3 projected pixel size for R^{min} are no more than 20 for R^{max} [TL04].

Another way to improve the record spacing is by limiting R_i by the translational gradient magnitude. Because the split sphere model makes assumptions which are not necessarily valid for all scenes, it is possible to create a record with too high R_i in an environment where the indirect illumination changes rapidly. Because we have estimate of this rate of change (translational gradient), we can use it to clamp the R_i to lower the interpolation error using this formula:

$$R_i := \min\left(R_i, \frac{E_i}{\|\nabla_t E_i\|}\right).$$

The translational gradient itself can also be a source of errors. It is computed using ray lengths (Equation 3.8) that could be very short for records in corners and in areas with lots of geometric details. During the gradient computation we divide by these rays near-zero lengths. This may result in excessively high value of $\nabla_t E_i$, which consequently causes artefacts during the interpolation. Solution to this problem is to clamp the gradient magnitude using this formula:

$$\nabla_t E_i \;:=\; \nabla_t E_i \cdot \min\left(1,\; \frac{R_i}{R^{min}}\right),$$

where R^{min} is the minimal distance to surfaces described previously and R_i is the true geometric distance to surfaces before clamping.

Artifacts can appear even in scenes for which split sphere model holds, because the hemisphere sampling is stochastic and can randomly hit or miss important close objects which therefore may or may not cause significant reduction of computed distance to surfaces. In some situations, such as in Figure 3.5, even one record which missed the thin geometry would cover the area with its influence radius and no other records would be created, resulting in complete lack of detail.

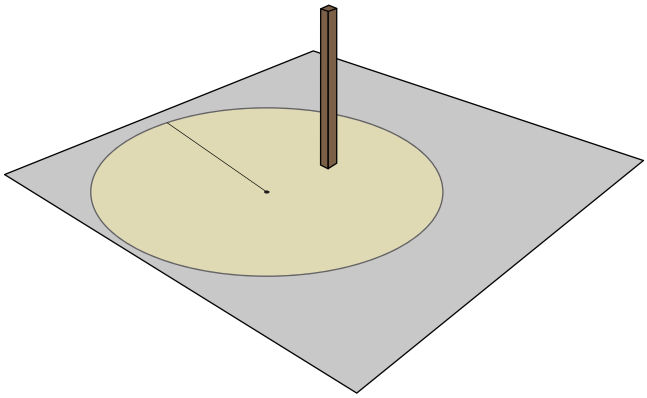


Figure 3.5: *An illustration how single record which missed small geometry detail can ruin the interpolation without neighbour clamping. The single record shown has erroneously high value of R_i because no ray during its hemisphere sampling hit the thin object. No new records will be created in the yellow area, because the irradiance can be interpolated even from single record. This means that the final image will completely lack details around the object.*

This problem is solved by another very important improvement of the original algorithm called *neighbour clamping* [KBPv08]. It solves the problem using *triangle inequality*. When inserting a new record i into the cache, we test all nearby records if their stored R_i plus distance to the new record is smaller than the new record R_i. If this inequality does not hold, it indicates missed features during the hemisphere sampling and we clamp the new record R_i accordingly. Then we also clamp distances to surfaces of nearby records in the same way. After clamping these two inequalities hold for all neighbouring records j:

$$\forall j : R_i \;\leq\; R_j \;+\; \|\mathbf{p}_i - \mathbf{p}_j\|,$$
$$\forall j : R_j \;\leq\; R_i \;+\; \|\mathbf{p}_i - \mathbf{p}_j\|.$$

Because of this even single small value of R_i from a record where small geometric feature was hit during the hemisphere sampling can propagate to all relevant nearby records, as shown in Figure 3.6. Because of that this method can sometimes provide dramatic improvement in final image quality.

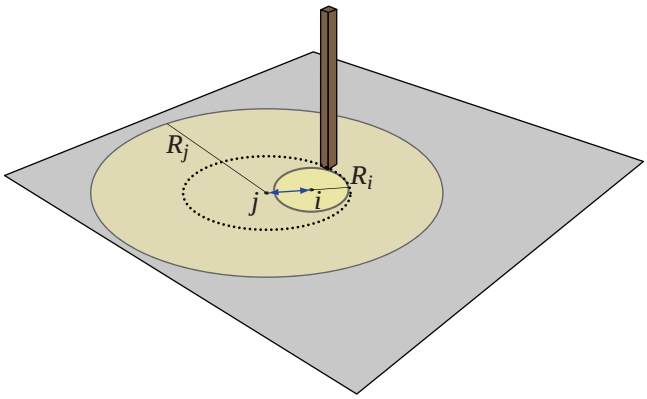


Figure 3.6: *The missed geometry problem with neighbour clamping. The scene is similar to that in Figure 3.5, the difference is that we now use the neighbour clamping. Because of that the record that did not miss the small geometry (i) causes reduction of R_j to $R_i + \|\mathbf{p}_i - \mathbf{p}_j\|$ (dotted circle).*

3.1.6 Integrating irradiance caching into complex global illumination solution

When describing the hemisphere sampling during irradiance computation we have completely skipped the incoming radiance calculation. We will now discuss several possibilities how to calculate the illumination at these *secondary hit points*, as it is an important part of the algorithm. The simplest possibility is to consider direct lighting only. This creates so-called *single-bounce indirect illumination*. While it creates visually plausible smooth gradients of varying illumination, it is not physically correct and ignores significant portion of the light transfer. Its main advantage is that it is non-recursive, simple and fast to compute.

If we want physically correct full GI solution, we can either use a secondary GI algorithm, or apply the irradiance caching recursively. *Recursive irradiance caching* was first described in the original paper on irradiance caching [WRC88]. To get the illumination at a secondary hit point, we simply try to interpolate it from the cache and if that fails we create new cache record. This means that first record is always computed extremely slowly, as the cache is initially empty and multiple secondary records need to be recursively created, but successive records are computed faster and faster, because the cache is gradually populated and an interpolation can be performed more often than new record creation. The recursive creation of new records must be stopped at some depth. This is done by substituting some constant, possibly zero, *ambient irradiance* value.

Because the secondary records do not affect the final result as much as primary ones, we can get away with increasing the maximum interpolation error a and making fewer samples during the hemisphere sampling. Ward et. al.[WRC88] suggest halving the number of rays cast and increasing the maximum error a by 40% in each recursive step for scene with mean surface reflectivity of 50%. Because of resulting decreased cache accuracy, it is advisable to create a separate irradiance cache for each level of recursion, and avoid interpolation using records from cache belonging to deeper recursion level.

Using suitable *secondary GI algorithm* has the advantage of combining strong points of irradiance caching with those of another method. Many algorithms integrate seamlessly with irradiance caching. For example to use path tracing we just bounce each ray cast during the hemisphere sampling multiple times, as described in Chapter 2.4.3. When using the photon mapping algorithm with irradiance caching we

simply perform the density estimation at each secondary hit point. The whole process of hemisphere sampling, where multiple rays are cast and their colours are averaged, is analogous to final gathering. Irradiance caching can be therefore regarded as an acceleration method for photon mapping with final gathering. Many other GI algorithms can be accelerated the same way.

During the hemisphere sampling we have ignored direct lighting. Because of that we have to sample it explicitly during the visualization phase with shadow rays. Finally, we need to deal with the fact that irradiance caching supports only diffuse BRDFs. We split each BRDF to diffuse and non-diffuse component during visualization. The diffuse component is used for modulating the result of irradiance cache interpolation (see Equation 3.1); illumination reflected by the non-diffuse component have to be calculated by another method. Simplest approach is to use "brute force" and calculate the result with Monte Carlo path tracing. This however generates noise and many samples may be needed to get clean result. Other possibility is to use one of the extensions of irradiance caching to non-diffuse BRDFs. They are described in the next section.

3.2 Radiance caching algorithms

We cannot handle handle non-diffuse surfaces if we store only irradiance, we need the directional information represented by radiance. We cannot however simply modify the irradiance caching algorithm to store radiance samples obtained during the hemisphere sampling instead of irradiance. Main problem of this modification is that there can be thousands of samples in each record which would have to be stored and iterated through during each interpolation. This would cause significant memory consumption and computational overhead to the point where there would be no advantage over the "brute-force" path tracing.

Efficient radiance storage and interpolation is therefore a crucial part of any radiance caching algorithm. We present two different approaches: Radiance caching presented by Křivánek et. al. [KGPB05] and *spatial directional radiance caching* (SDRC) presented later by Gassenbauer et. al.[GKB09]. We will call the former method *spherical harmonics radiance caching* (SHRC) to avoid confusion with radiance caching as general idea of extending the irradiance caching to glossy surfaces.

3.2.1 Spherical harmonics radiance caching

This method approximates the incident radiance on a hemisphere using spherical or hemispherical harmonics [Gre03, MS67]. Spherical harmonics are generalization of Fourier series to a sphere. They can be used to approximate a function $F(\vec{\omega})$ defined on a sphere as:

$$F(\vec{\omega}) \approx \sum_{l=0}^{n-1} \sum_{m=-l}^{l} \lambda_l^m Y_l^m(\vec{\omega}).$$

Approximation of $F(\vec{\omega})$ by spherical harmonics of order n consists of the vector Λ of n^2 coefficients $(\lambda_0^0, \lambda_1^{-1}, \lambda_1^0, \lambda_1^1, ..., \lambda_n^{-n}, ..., \lambda_n^n)$. Functions Y_l^m are called spherical harmonics basis functions.

All coefficients λ_l^m of the vector can be computed by projecting the function $F(\vec{\omega})$ onto the corresponding basis:

$$\lambda_l^m = \int_\Omega F(\vec{\omega})\, Y_l^m(\vec{\omega})\, \mathrm{d}\vec{\omega}.$$

This integral can be easily solved using Monte Carlo sampling. Very important property of spherical harmonics is that an integral of product of two functions $F_1(\vec{\omega})$, $F_2(\vec{\omega})$ represented by spherical harmonics is simply dot product of their coefficients vectors Λ_1, Λ_2:

$$\int_\Omega F_1(\vec{\omega}) \cdot F_2(\vec{\omega})\, \mathrm{d}\vec{\omega} \;\approx\; \Lambda_1 \cdot \Lambda_2. \tag{3.9}$$

Hemispherical harmonics are very similar to spherical ones, only they are defined over a hemisphere. We continue to describe the algorithm using spherical harmonics. The hemispherical version is largely identical. For detailed discussion of differences see the original proposal [KGPB05].

The interpolation scheme for SHRC is identical to the original irradiance caching. We precompute the indirect illumination at sparse locations in the scene, based on the same split sphere heuristic and using the same hemisphere sampling, but instead of integrating the samples and storing resulting irradiance $E(\mathbf{x})$ we use the samples to approximate the incoming radiance function $L(\mathbf{x} \leftarrow \vec{\omega})$ by spherical harmonics and store the resulting vector of coefficients. We must also convert scene BRDFs (pre-multiplied by the cosine) to spherical harmonics representation. To do so we discretize all exitant directions $\vec{\omega}_o$ into finite samples and calculate spherical harmonics vectors $\mathbf{C}_{\vec{\omega}_o}$ for each such direction.

To interpolate the illumination at single point, we first calculate weighted average of spherical harmonics vectors of all records influencing the point. The weights are computed from surface normal and position differences identically to the original algorithm (Equation 3.5). Each cached spherical harmonics vector must be first rotated to the coordinate frame of the interpolated surface point by rotation matrix $\mathbf{R}_i$. Since we now have the interpolated incoming radiance distribution and BRDF pre-multiplied by the cosine for the outgoing direction we want, we can compute resulting radiance simply as dot product of these two vectors, using Equation 3.9.

The rotation of spherical harmonics coefficients performed during interpolation can be very costly operation. Křivánek however developed fast rotation approximation by truncated Taylor expansion of the rotation matrix [KGPB05], which performs well for small rotation angles. Many other improvements were proposed, mostly taken over from the irradiance caching, for example neighbour clamping, gradient clamping and use of gradients. Rotational gradients are no longer needed as they are replaced by the rotation, but translational gradients can still substantially enhance the interpolation.

Because of the spherical harmonics radiance representation, the algorithm offers good approximation for low-frequency BRDFs, it is highly memory-efficient and the needed rotations are very fast. It however fails to capture high-frequency BRDFs and does not allow importance sampling because the radiance representation does not allow directional localization.

3.2.2 Spatial directional radiance caching

Because the SHRC algorithm has several shortcomings, Gassenbauer and Křivánek proposed the *spatial directional radiance caching* (SDRC) [GKB09]. They have decided to keep the original radiance samples

in each record, because none of the potential representation methods (spherical harmonics, wavelets, ...) meet all demands: directional localization, ability to capture high-frequency functions, and fast rotation. The defining characteristics of this method is the use of BRDF *importance sampling* both during precomputation and interpolation phases, which provides good quality and performance even for high-frequency BRDFs with sharp specular lobes.

SDRC creates a set of sparse samples in the scene again in the same way as the original irradiance caching. The hemisphere sampling is however different: only small part of the samples is distributed uniformly according to the original PDF (Equation 3.3) to compute the harmonic mean of distances to other surfaces (Equation 3.6). Most samples are created using importance sampling of the local surface BRDF. This is the key of the algorithm efficiency: only directions close to specular lobe are sampled densely. After the sampling we project the samples to 2D domain $\mathbf{D}$ using *paraboloid mapping* [HS98]. The mapping from unit direction vector on y-up hemisphere (x,y,z) to point (a,b) in $\mathbf{D}$ is:

$$\begin{aligned} a &= \frac{x}{y+1}, \\ b &= \frac{z}{y+1}. \end{aligned} \tag{3.10}$$

This mapping is fast to compute, has no singularities or discontinuities on the hemisphere and has low distortion[5]. A kd-tree (called *L-tree*) is built over the entries after mapping to allow fast range queries later. To save memory, each sample position in $\mathbf{D}$ is discretized into 1 byte per axis (2 bytes total) and its colour is saved using Ward RGBE format [War92a] (4 bytes). This allows to pack each sample into 8 bytes.

Interpolation in SDRC hast two stages – *spatial* and *directional*. Spatial interpolation is similar to irradiance caching and SHRC – we identify the set $S(\mathbf{x})$ of records influencing the interpolated point $\mathbf{x}$ and assign them *spatial weights* $w_i^s(\mathbf{x})$ calculated from position and normal differences (Equation 3.5).

It is the directional interpolation that sets this algorithm apart. Instead of using all directions stored in records of $S(\mathbf{x})$ we first generate set of samples (directions on hemisphere) using importance sampling of BRDF at $\mathbf{x}$. For each such direction $\vec{\omega}_j$ we locate all nearby cached directions in records from $S(\mathbf{x})$ and use them to interpolate radiance in that direction, as shown in Figure 3.7. If no such sample is found, we simply create a new sample with the direction $\vec{\omega}_j$ in random record. In order to adapt to varying sample density we limit the search radius in L-trees for each BRDF sample $\vec{\omega}_j$ to:

$$r_j = \min\left(r_{max},\ \frac{1}{2\pi}\frac{1}{M\sqrt{p(\vec{\omega}_j)}}\right). \tag{3.11}$$

M is the number of BRDF samples generated, $p(\vec{\omega}_j)$ is the probability density corresponding to generating a sample in direction $\vec{\omega}_j$ and r_{max} is the search radius upper bound, used to reduce artefacts in areas with low probability density, set to 0.15 in the original paper. *Directional weight* of kth sample (with direction $\vec{\omega}_k$) reported from i-th L-tree is:

$$w_{ik}^d(\vec{\omega}_j) = \max\left(0,\ 1 - \frac{\|\vec{\omega}_j - \vec{\omega}_{ik}\|_{\mathbf{D}}}{r_j}\right).$$

[5]Density of uniformly distributed points on hemisphere varies after the transformation no more than by factor of 4.

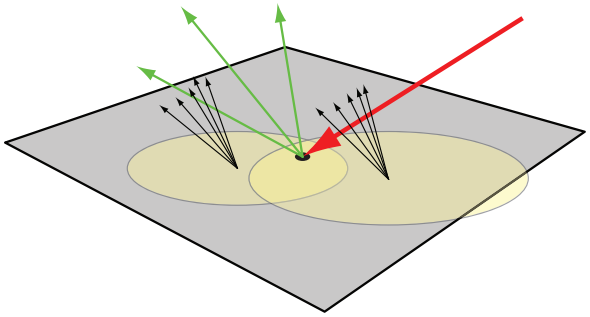


Figure 3.7: *An illustration of the spatial directional radiance caching. Single primary ray (red) hits the surface. Multiple BRDF samples are created (green rays). Instead of ray tracing these samples we perform the interpolation by locating the two nearby radiance cache records and searching their stored directions.*

Total weight of single sample is simply product of the spatial weight of its parent record w_i^s and its directional weight w_{ik}^d. Total radiance estimate for BRDF sample $\vec{\omega}_j$ is:

$$L(\mathbf{x} \leftarrow \vec{\omega}_j) \approx \frac{\sum_i \sum_k w_i^s(\mathbf{x}) w_{ik}^d(\vec{\omega}_j) L_{ik}}{\sum_i \sum_k w_i^s(\mathbf{x}) w_{ik}^d(\vec{\omega}_j)}. \tag{3.12}$$

If set $S(\mathbf{x})$ contains n records and we use M BRDF samples, then total of nM lookups are performed. Using these interpolated incident radiance values we can easily compute exitant radiance using the estimator in Equation 2.13.

This variant of radiance caching is more memory-consuming and requires many searches in the directional domain for single interpolation, but these disadvantages are more than compensated by elimination of the need for special BRDF representation and the ability to use importance sampling according to a surface BRDF. Although the algorithm can handle all materials, it is advisable to combine it with simple irradiance caching for diffuse surfaces, as the radiance caching is always slower than irradiance caching.

Chapter 4

Geometric searching

The problem of efficient searching in a multidimensional space is frequently encountered in computational geometry [Mat94, AE97], realistic image synthesis [Hav00, Jen96, WRC88], databases [GG98], computer vision, pattern recognition, geographical information systems, etc. Algorithms and data structures for multidimensional searching can be substantially more complicated than in one-dimensional domain, where efficient search using sorted arrays or balanced search trees is possible [Knu98].

The searching problem comes in many flavours – for example database applications (which are called *spatial databases*) usually deal with low-dimensional spaces, but they must be able to process big datasets, which do not fit in the main memory. As a result external memory accesses are the performance limiting factor and the performance of used data structures is measured solely in the number of disk input/output operations [MNPT]. Because of that *R-trees* (multidimensional generalizations of B-trees) are the most commonly used spatial database data structures, even though they are not the most efficient ones in terms of computational complexity [Arg01].

On the other hand, applications in computer vision and pattern recognition usually deal with smaller datasets, but they work in extremely high-dimensional spaces. Because of the *curse of dimensionality* [Cla94], most data structures become quickly inefficient with increasing dimensionality to the point where linear time algorithms become once again competitive [CS02].

4.1 Problem formulation

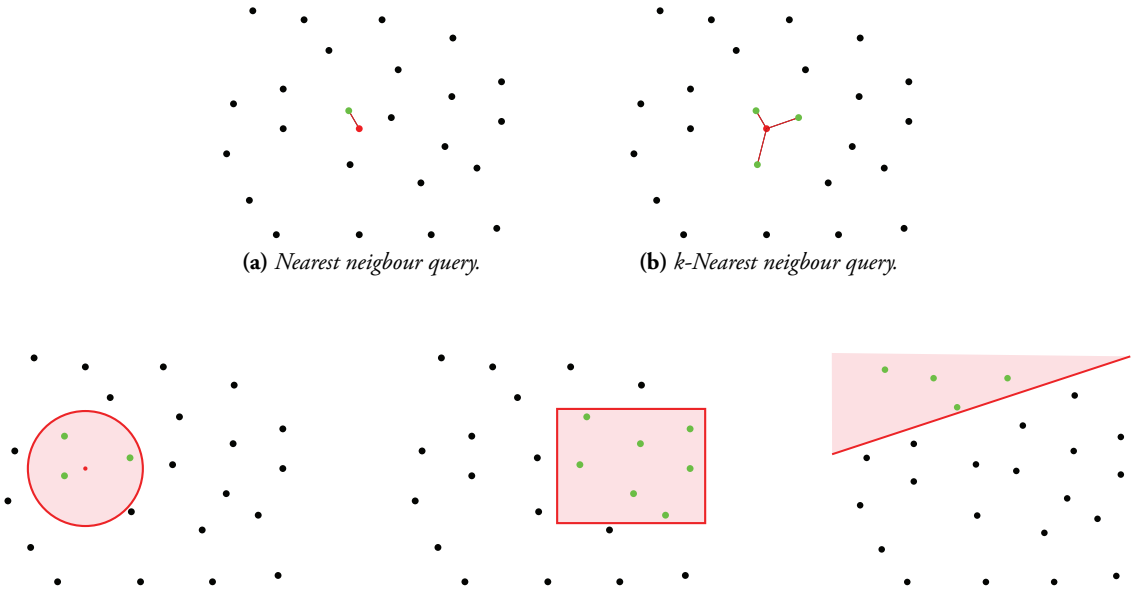
In the typical point search problem we are given a set S of n points in $\Re^d$ and a distance metric. We want to preprocess this set by creating an auxiliary data structure, using which we could answer queries efficiently, that is in asymptotically lower time than by using simple sweep through the data.

There are various types of queries. *Nearest neighbour* (NN) query (Figure 4.1a) returns single point $p \in S$ which is closest to the query point q according to the metric used. A generalization of the nearest neighbour query is the *k-nearest neighbour* (k-NN) query (Figure 4.1b), which returns the set $\{p_1, p_2, \ldots, p_k\}$ of k closest points to the query point. NN and k-NN queries are examples of a *similarity search*.

Another type of query is the *range search* query. In range searching we have the set of possible *search ranges* **R**, and we want to efficiently report or count all points inside a particular range $R \in \mathbf{R}$. Examples of ranges commonly used are rectangles, balls (Figure 4.1c), rectangles (Figure 4.1d), halfspaces (Figure 4.1e), and simplices. There are various flavours of range queries: a *range-reporting query* reports all points inside the query range, a *range-counting query* only reports the number of points in the query and a *range-emptiness query* only reports whether there are any points in the query at all.

Specific subclass of range queries are the *optimization queries*. Answer to an optimization query is a point lying inside the query range that maximizes certain optimization criterion (for example has highest x-coordinate). Ray casting can be viewed as a type of optimization query, where the ranges are half-lines. Another example of an optimization query is the linear programming problem.

Another variant of geometric range searching problem is the search in non-point data. In this case the set S does not contain points, but objects with spatial extent (for example lines, rectangles, balls, polytopes, etc.). A *point* or *range query* returns all such objects which overlap the query point or region. *Containment query* in contrast returns all objects entirely contained in the query range. Similarly to point search we can define large variety of additional query types. Point data structure cannot be directly used for non-point data; special data structures are needed that can be very complicated.



(a) *Nearest neigbour query.* **(b)** *k-Nearest neigbour query.*

(c) *Disk range query.* **(d)** *Rectangular range query.* **(e)** *Half-space range query.*

Figure 4.1: *Examples of queries in 2D: top row shows similarity queries (nearest neighbour and 3-nearest neighbour). Query points are red and query answers are green. Bottom row shows range queries with 3 different ranges: a disk, a rectangle, and a half-space.*

4.2 Space subdivisions

A common approach to constructing a point data structure is to introduce some type of *spatial subdivision*. Such scheme partitions the space $\mathfrak{R}^d$ into finite set of disjoint cells, possibly with hierarchy built over them. Each point from S is stored in the cell it overlaps. The data structure thus provides additional spatial information about our dataset which we may use to speed up the queries. For example in range-searching queries, only cells which overlap the query region need to be inspected, as only they can contain points inside the query. Examples of such data structures are in Figure 4.2.

4.2.1 Uniform grid

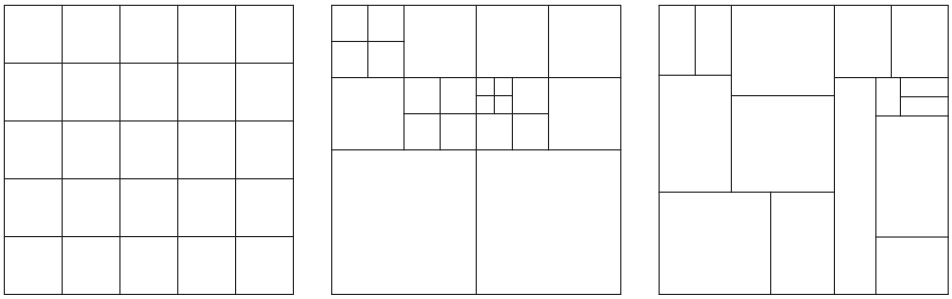
Simple example of a spatial subdivision data structure is the *uniform grid*. This data structure divides the volume of dataset axis-aligned bounding box uniformly in each axis, creating a grid, and places each data point in its corresponding cell. The data structure is built by simply assigning all points to their cells in $O(n)$ time. Searching is simple – for range queries we enumerate cells intersecting the query and test points they store, for similarity searches we iterate cells in the order of increasing distance from the query point until we have the result.

Because the cell containing a point can be computed as well as addressed directly, the expected query time for similarity search in uniformly distributed data is $O(1)$, making uniform grid the optimal data structure [BWY80]. However because it is non-adaptive, its performance degrades quickly for non-uniform data. In a highly non-uniform dataset most points are stored in single or very few cells and have to be all processed if the search procedure encounters such cell. Recursive or non-uniform grids can be used to overcome this problem [Wei78]. Its other disadvantage is that it scales very poorly with increasing dimensionality of the search space, as the total number of cells needed to maintain constant subdivision of all axes rises exponentially with dimension.

4.2.2 Octree

Octree improves upon the uniform grid in that it can adapt to the data distribution – it induces recursive space subdivision that is finer in areas with higher data density. This data structure is an adaptation of the two-dimensional *quadtree* (Figure 4.2b) [FB74] to 3D space. Adaptations to arbitrary-dimensional spaces are possible. During the data structure build we start with the same axis-aligned box as in the uniform grid, but we divide it only once in each axis. This creates 2^d child cells, where d is the search space dimension. This means that in 2D there are 4 and in 3D there are 8 children, hence the names quadtree and octree. After the split we move all points to children that contain them, and we recursively repeat the split procedure for any children that contain too many points.

Expected NN and k-NN query times are logarithmic, because the tree has logarithmic depth on average. The general search procedure for any query is similar: we recursively traverse the tree from root to leaves; in each leaf node we process all stored points and in each inner node we decide in which children we will continue the search.



(a) *Regular grid.* **(b)** *Quadtree.* **(c)** *Kd-tree.*

Figure 4.2: *Examples of different space subdivisions in 2D: the regular grid, quadtree, and the kd-tree.*

4.2.3 Kd-tree

A k-dimensional (kd) tree, introduced in 1975 by Bentley [Ben75], is the generalization of binary search tree into higher-dimensional space. Each inner node splits the search space in one dimension, creating hierarchical spatial subdivision, as shown in Figure 4.2c. Nodes containing at most some small constant number of points are no longer divided, but instead become leaves.

Inner nodes of the tree can be either implicit or explicit. When using implicit nodes, data points themselves create the tree - each inner node is just data point with the split axis and child pointers saved. Explicit inner nodes are stored separately from data points and the resulting tree is independent on the data it stores. Implicit nodes conserve memory, but they also restrict the number of possible split positions, make the search slower because points are not stored only in leaves and make deletion of a point very difficult. Inner nodes of the tree can be either implicit or explicit. When using implicit nodes, data points themselves create the tree - each inner node is just data point with the split axis and child pointers saved. Explicit inner nodes are stored separately from data points and the resulting tree is independent on the data it stores. Implicit nodes conserve memory, but they also restrict the number of possible split positions, make the search slower because points are not stored only in leaves and make deletion of a point very difficult.

Important factor determining the kd-tree performance is the split plane axis and position selection. In the original paper the build algorithm cycles through all axes in round-robin fashion. Friedman et. al. [FBF77] suggest splitting in the dimension with the *largest spread* of points. Another simple and yet efficient technique is to always select the axis in which is the node bounding box largest [DDG00].

There are several methods for selecting the position of splitting hyperplane. In the original paper the split position was simply position of a randomly chosen point. Better alternative is to use the median of stored points with respect to the split dimension [FBF77], or spatial median (value of node bounding box centre) [Moo90]. Note that the last splitting rule can result in nodes that are empty. This can negatively affect similarity search query performance. Solution to this is the *sliding midpoint* rule [MM99]. If one of the children after the split should be empty we move the split hyperplane so that single point resides in the previously empty node as shown in Figure 4.3. This prevents creation of empty nodes, which are undesirable in NN search.

Both search and build procedures for kd-tree are very similar to those of the octree. The biggest difference

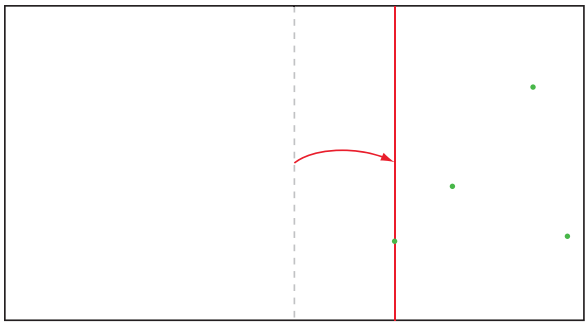


Figure 4.3: *Picture of a single kd-tree node. Splitting this node using the spatial median rule would produce an empty left child. Because of that we slide the split plane to the right to move a single point to the left child.*

is that the kd-tree is always binary. Because it is highly adaptive to data distribution, it does not suffer from high branching factors in higher dimensions and is generally less complicated than octree, the kd-tree is widely used in point search applications.

4.3 Handling non-point data

Point-location data structures cannot be used with non-point data directly, but there are several techniques we can apply in order to handle this type of data. Simple data objects can be transformed to higher-dimensional points and then stored in a conventional data structure [SK88]. This approach, called *object mapping*, is common in computational geometry. For example simple axis-aligned rectangle in two dimensions can be transformed to four-dimensional point (c_1, c_2, e_1, e_2), where (c_1, c_2) are coordinates of its centre and (e_1, e_2) are its half width and half height. An alternative transformation maps the rectangle to (l_1, l_2, u_1, u_2), where (l_1, l_2) and (u_1, u_2) are coordinates of its lower left and upper right corner, respectively. Choice of the mapping obviously affects the search algorithm and search performance. The higher-dimensional point space is called *dual space*. More complex objects, for which we cannot construct the transformation, can be also used with this method, when approximated by their bounding boxes or bounding spheres.

Another quite straightforward way to store non-point data in space partitioning data structures is by *object duplication*. Normally we would not be able to store arbitrary objects in these data structures as it may be impossible to construct a non-trivial space partitioning in which each object belongs to only single cell. We solve this by cutting objects straddling any partition hyperplane and inserting references to them into cells on both sides of the hyperplane. Object duplication increases the memory consumption and splits on lower level of the data structure may become very inefficient (because most objects have to be inserted in both children nodes), but resulting data structure has good query performance – to obtain all objects overlapping a query point, only single path from the root to a leaf has to be traversed. A prominent example of such data structure is the R^+-tree [SRF87].

A different approach is to abandon the space partitioning schemes completely and permit *overlapping regions*, as illustrated in Figure 4.4. This allows us to construct data structures where each object is referenced only once, leading to low memory consumption and simple updates. Downside of this method is the worse query performance, because more paths from the root to leaves have to be traversed due to

the spatial overlap of nodes. Example of such object-partitioning data structures are R-trees [Gut84] and R*-trees [BKSS90], both used in spatial databases, and bounding volume hierarchies used frequently in collision detection [Eri05] and ray tracing [GS87].

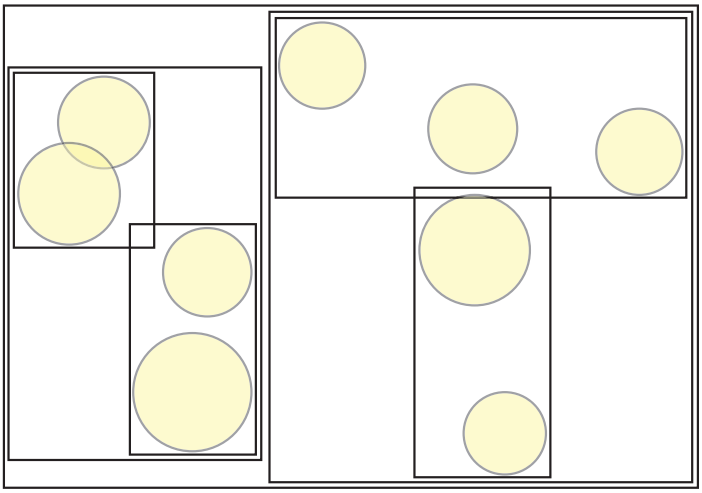


Figure 4.4: *An example of the bounding volume hierarchy build over simple scene with non-point data (balls). The bounding volumes are intentionally not drawn tightly enclosing their interiors for better clarity.*

Chapter 5

Data structures for irradiance caching

During the irradiance interpolation using irradiance caching at a surface $\mathbf{x}$ (described in Section 3.1.2) we need to determine the set $S(\mathbf{x})$ of all records that can influence the interpolation result. For each of these candidates we evaluate the weight function $w_i(\mathbf{x})$ (Equation 3.5 or 3.7) and if it is non-zero, we use the candidate for the interpolation. For the irradiance caching to be efficient we need a way to determine $S(\mathbf{x})$ efficiently. We also want $S(\mathbf{x})$ to contain as little *false positives* (records with zero weight) as possible, because weight function evaluation is costly.

Solution to this search problem is to create a ball from each record with centre at the record position and radius equal to the maximum distance from record where it can have non-zero weight and perform a *point query* which reports all balls intersecting $\mathbf{x}$. The radius of each ball (which we will call *validity radius*) is not identical to record distance to surfaces R_i. Relation between the two can be derived from the weight equation by letting $\mathbf{n} = \mathbf{n}_i$ and creating an inequality by requiring the weight to be greater than zero. For the Ward weight function we get:

$$\begin{aligned}
\frac{1}{\frac{\|\mathbf{x}-\mathbf{p}_i\|}{R_i} + \sqrt{1-\mathbf{n}\cdot\mathbf{n}_i}} - \frac{1}{a} &> 0, \\
\frac{1}{\frac{\|\mathbf{x}-\mathbf{p}_i\|}{R_i}} - \frac{1}{a} &> 0, \\
\frac{R_i}{\|\mathbf{x}-\mathbf{p}_i\|} &> \frac{1}{a}, \\
aR_i &> \|\mathbf{x}-\mathbf{p}_i\|.
\end{aligned}$$

We are able to derive similar bounds for the weight function of Tabellion and Lamorlette:

$$\begin{aligned}
1 - \alpha \max\left(\frac{\|\mathbf{x}-\mathbf{p}_i\|}{0.5R_i}, \frac{\sqrt{1-\mathbf{n}\cdot\mathbf{n}_i}}{1-\cos 10^\circ} \right) &> 0, \\
1 &> \alpha \frac{\|\mathbf{x}-\mathbf{p}_i\|}{0.5R_i}, \\
\frac{0.5R_i}{\alpha} &> \|\mathbf{x}-\mathbf{p}_i\|.
\end{aligned}$$

As we can see, in both cases ball radii are dependent only on original records validity radii and maximal error settings.

Irradiance caching have specific requirements on the search data structure. It needs to support only single object type (ball) and single search method (point query). Number of objects stored is low (typically in orders of tens of thousands), which means that the entire data structure will fit in main memory and we cannot measure data structure performance by the asymptotic time complexity alone. Because of the lazy irradiance evaluation scheme the data structure needs to support insertions on the fly. This can be a problem for many data structures. We solve it with a compromise – we require that the data structures support insertions, but we rebuild them few times during and after the precomputation phase. Because of the relatively low number of records rebuild times are generally negligible and it allows us to use "semi-incremental" data structures, which slightly deteriorate with multiple insertions and are periodically repaired. This approach is further supported by the fact that only very few records are inserted after the precomputation phase. Additionally because of the neighbour clamping (Section 3.1.5) some record radii can get significantly reduced after they are inserted into the cache. This means that any data structure would slightly deteriorate without complicated re-insertion procedure or rebuilding.

Now that we have formulated the problem we try applying different algorithms and data structures for solving it. First we review already known and used data structures: Ward's original octree and the multiple-reference octree. Then we try using multiple-reference kd-tree as a multiple-reference octree alternative, reducing the problem to simple point k-NN search with a kd-tree, employing a bounding volume hierarchy and finally transforming the problem to higher dimension range query over point data.

5.1 Ward's octree

Octree is the data structure originally proposed for irradiance caching by Ward et. al. [WRC88]. It is built by recursively subdividing bounding cube of the scene to 8 smaller cubes. Each record is stored in a single node which contains its position. Records can be stored at any level of the tree, not only in leaves. Where are they stored is determined by their validity radii: each record is placed in a node with side length 2-4 times larger than its validity radius.

To find all records influencing an area, we have to traverse multiple paths from the root to leaves. In each node we report its stored records and recursively continue to all its children that are closer to query point than half of their size (cube side length). This in combination with the way records are inserted into the tree guarantees that search will find all records possibly influencing the query point. Expected query time is $O(log(n))$, as at most 8 nodes have to be traversed at any level of the tree.

Biggest advantage of the octree is its memory efficiency, as it avoids object duplication and its relative simplicity. Because it adapts only very little to input data, it is easy to implement with full insertion support (with no deterioration after any sequence of inserts). On the other hand its traversal performance is very poor, because the data structure fails to sufficiently separate overlapping and non-overlapping balls and many false positives are reported.

5.2 Multiple-reference octree

Multiple-reference octree is based on the *object duplication* technique (Section 4.3) – single object (ball) can be referenced in multiple nodes[1]. This results in increased memory consumption, but better query performance. The key invariant of the multiple-reference octree is that each object is referenced exactly once on each path from the root to any leaf it intersects. Objects can be again referenced both in leaves and inner nodes. This allows us to find all records of $S(\mathbf{x})$ by simply traversing single path from the root to the leaf containing the query point and returning all objects referenced by visited nodes.

Unlike the simple and efficient search procedure the insertion is more complicated. We have to recursively propagate each inserted record ball from the root node to all children it intersects. The recursion is stopped when a leaf is reached or when the node size becomes approximately the same as the ball radius. When the number of records referenced in single node exceeds a pre-set maximum value (the node "overflows") we split the node creating 8 children, keep large balls in the node and propagate all other to children they intersect. The same procedure is also used for rebuilding from the scratch.

In our implementation we change the somewhat vague original insertion recursion stopping criterion. The original criterion does not reflect mutual position of the node and record. It may place a record with big radius of influence unnecessarily high in a sub-tree it barely intersects, as illustrated in Figure 5.1. We change the stopping criterion to:

$$\|\mathbf{C}^{\text{furthest}} - \mathbf{P}_i\| \leq k r_i, \tag{5.1}$$

where $\mathbf{C}^{\text{furthest}}$ is the corner of node bounding box furthest from the record position $\mathbf{P}_i$, r_i is the record validity radius and k is a constant determining how aggressively we will propagate the records to lower levels of the tree. In our implementation we use value of 1.4. This criterion takes into account both size of record ball and its position with respect to the node.

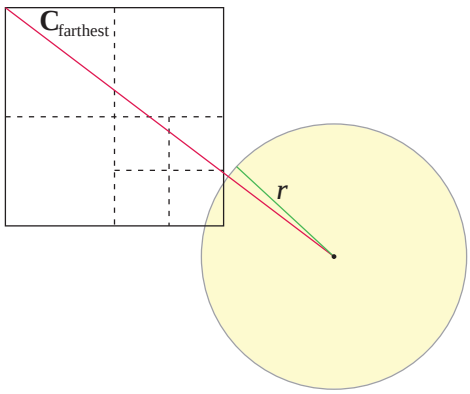


Figure 5.1: *An example (in 2D) of a ball position that is particularly bad for the original insertion recursion stopping criterion. Because the ball itself is large, it would be placed in the root node, although it barely intersects it. Our stopping criterion compares ball radius r and the distance from ball centre to the furthest point* $\mathbf{C}_{furthest}$ *and propagates the record deeper.*

Multiple-reference octree is a simple, yet very efficient data structure. Because of that it is used in most irradiance cache implementations [KG09]. Its main problem is the memory consumption – cache with

[1] In order to save memory the records are not actually duplicated, but stored separately from the tree and only their numerical indices are duplicated in tree nodes.

only tens of thousands of records can generate an octree with millions of references. The data structure thus requires careful fine-tuning of maximum node depth parameter and insertion recursion stopping criterion to generate trees with reasonable memory requirements without causing performance degradation due to insufficient space subdivision.

5.3 Multiple-reference kd-tree

Multiple-reference kd-tree is our new irradiance cache data structure based on the multiple-reference octree. It uses the same concept of object duplication, search is again done by traversing single path from the root to a leaf containing the query point and the insertion of a record is a recursive procedure analogical the octree. The main difference between the two is in their branching factor – each octree node has zero or 8 children because it splits the space in all dimensions at once; each kd-tree node has zero or only 2 children as it splits the space only in single dimension.

We hope that the different branching factor will help in several ways. First is improving the adaptivity of the data structure to scene records distribution. Every node split in octree results in 8 new nodes. Equivalent space partitioning in kd-tree is obtained by 3 consecutive splits in alternating axes. We have however the possibility to make only 1 or 2 splits, which allows for finer control of the level of space subdivision. It also allows to keep the node aspect ratios close to 1 in non-cube scenes while keeping the root node tightly enclosing the scene. Finally, because the split procedure is greatly simplified by restricting the split to single axis, we can derive a heuristic for determining best split position.

5.3.1 Split position heuristic

Our heuristic is similar to the surface area heuristic known from ray casting (Section 2.4.1). The idea of improving a geometric search data structures outside of ray tracing using a SAH analogy is not new, it was previously explored for example in the context of k-nearest neighbour search in photon mapping by Wald et. al. [WGS04]. We cannot however use their results directly as they deal with slightly different scenario, we have to manually derive our own cost function.

During the multiple-reference kd-tree traversal we follow single path from the root to a leaf. In each inner node we continue to exactly one child containing the query point. Probability that a node contains the query point is proportional to its *volume*[2], not to its surface area as in ray casting. Another difference is that any node can hold records, not only a leaf node. These facts lead to following cost function for a (sub)tree:

$$f(X) = \begin{cases} N \cdot C^o & \text{if X is a leaf,} \\ N \cdot C^o + C^t + f(L)\frac{V(L)}{V(X)} + f(R)\frac{V(R)}{V(X)} & \text{if X is an inner node,} \end{cases}$$

where N is the number of records stored on the actual level, C^o and C^t are costs of reporting a record and descending to next node, respectively, and $V(X)$ is the volume of node X.

[2]Assuming uniformly distributed queries.

5.3.2 Tree updates

Insertion procedure is similar to multiple-reference octree. We use the same criterion for stopping record propagation as in the octree (Equation 5.1). We use the heuristic for splitting nodes that became too big due to insertions and in recursive top-down rebuild-from-scratch algorithm to select both the axis and position of the split by evaluating multiple possibilities and selecting the one minimizing $f(X)$. We also use the heuristic to determine whether to further split a node or create a leaf during the rebuild by comparing the cost of creating a leaf node to the best split cost found.

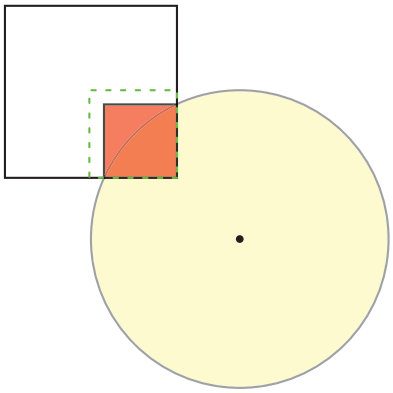


Figure 5.2: *An example (in 2D) of a tight bounding box of sphere-node intersection. Intersecting bounding boxes of the node and sphere would give us correct but not optimal result (green dashed box). Computing optimal (tight) result is more complicated (red box).*

To efficiently evaluate the cost function for multiple splits we cannot work with balls directly, we have to use their axis-aligned bounding boxes instead. Obtaining a bounding box from a ball is trivial, but we want to keep the record bounding box tight with respect to the ball intersection with a node (Figure 5.2). This is analogous to *perfect splits* known from kd-trees for ray tracing [SSK08]. Each time a ball is copied to a child during the splitting we calculate its tight bounding box: we start with the intersection of node and ball bounding boxes. Then we iterate over all three axes and shrink this bounding box by applying the Pythagorean theorem as illustrated in Figure 5.3.

In each level we can make the split in any of the 3 axes. There are infinitely many possible split positions, but similarly to SAH in kd-trees for ray tracing the cost function is piecewise linear [Hav00] with discontinuities at the beginning and end of each tight bounding box. We therefore evaluate the cost at these locations only for each axis. Similarly to SAH builders we approximate *f(L)* and *f(R)* by leaf function values ($N_L \cdot C^o$ and $N_R \cdot C^o$). Note that because of the object duplication, $N_L + N_R$ varies with different splitting plane positions. The cost function implicitly favours not only balanced splits, but also splits with minimal object duplication.

Multiple-reference kd-tree is very similar to the multiple-reference octree and same advantages and disadvantages apply. Multiple-reference kd-tree may deteriorate after several insertions when using the heuristic because we do not search for new optimal split after each insertion. The data structure however always reports correct results and can be easily rebuilt. We hope that the heuristic will lower overall size of the tree and improve performance in comparison to the multiple-reference octree.

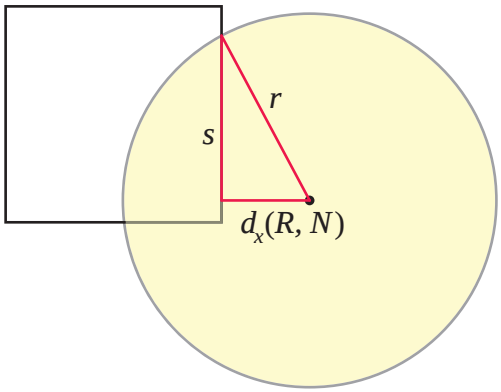


Figure 5.3: *To compute the tight bounding box of ball-node intersection we iterate over all axes and calculate s as $\sqrt{r^2 - d_i(R,N)}$, where r is the ball radius and $d_i(R,N)$ is the minimal distance between node N and record ball R in axis i. Because of the Pythagorean theorem the half-length of the tight bounding box in the other two dimensions cannot be larger than s, so we trim it accordingly.*

5.4 Point kd-tree

Obviously no point data data structure without any modification can give correct answers to our ball searching problem. We however argue that the k-nearest neighbour (k-NN) search can give sufficient approximate solution in most cases. We exploit the observation that usually only a small constant number of records is used for the interpolation. This is the result of irradiance caching lazy evaluation scheme – the cache can be always saturated using finite number of records to provide irradiance estimate for all visible surfaces. Once it is possible to interpolate the irradiance at a surface no new records have to be created there from that moment on. The *neighbour clamping* (Section 3.1.5) also reduces number of usable records at a point by shrinking their validity radii. Finally, if we report k nearest neighbours we are guaranteed that for any unreported record with non-zero weight there are at least k closest ones. These will most probably have greater weights since they are closer than the unreported one. Therefore neglected records will have relatively small weights.

We test our hypothesis of k-NN search sufficiency by implementing a point kd-tree and using it in the irradiance caching algorithm with k-NN search for fixed k. We compare its visual results, statistics (number of records really used for the interpolation) and performance to a reference method giving exact solution.

We use the *adaptive kd-tree* [FBF77] with explicit inner nodes, which decouples the tree data structure from data, permits splitting at any position and axis, and stores data only in leaves. Each inner node is split in half in its longest side. We use the *sliding midpoint* rule to get optimal performance for k-NN queries (see Section 4.2.3). The tree uses many additional improvements – *tracking nodes*, *depth first branch and bound*, *path ordering* and *bounds-overlap-ball (BOB) test*, all described by Sample et. al. [SHAP01]. We also use *node packing* – using bit manipulation we store each node in only 8 bytes. Because of that we cannot afford to store both child pointers for an inner node. We instead store numerical index of only the left child. Right child is *implicit* – stored immediately after the left one.

Record insertion into the kd-tree is simple, we just locate the leaf containing the record position and insert it there. For each leaf we store tight bounding box of all its records for the BOB test, so we need to

update it as well. If the leaf exceeds maximum size, we split it in two using the longest side rule. Rebuild from the scratch is done by recursively applying the splitting rule, until no leaves exceed the maximum size.

The *k*-NN search uses two priority queues - one for results and one for open nodes. They both order elements by their distance from the query point. We traverse multiple paths in the tree, starting with the root. For each node encountered during the traversal we compute its distance from the query point (this is accelerated by the tracking nodes) and if it is smaller than current distance to *k*-th nearest neighbour we enqueue it in the open nodes queue. If we encounter an inner node we only try to enqueue both its children. If we encounter a leaf we compute distances of all its records from the query and enqueue those closer than previous best results. Before that we compute minimal distance from the query to the leaf tight bounding box to determine if it is even possible to find closer points than we already have (BOB test). The open nodes queue allows us to first process nodes closer to the query (path ordering) and to stop the search when there are no unprocessed nodes closer than *k*-th nearest neighbour.

Biggest advantage of this search method is that it uses a well-known, extensively researched data structure with many existing implementations. It is easy to visualize and debug, and it has very good search performance in point search. Kd-tree quality may degrade after multiple insertions into a region that was previously empty and triggered the sliding midpoint, this is however taken care of during rebuilds.

5.5 Bounding volume hierarchy

Bounding volume hierarchy (BVH) is able to store non-point data without object duplication by allowing node regions to overlap. Nodes do not partition space, each one is instead associated with an explicit axis-aligned bounding box. Each node is either an inner node with two children or a leaf node with a list of stored balls. Records are stored only in leaves. Search procedure is very simple: we traverse the tree from the root to all leaves containing the query point. In each inner node we test both children bounding boxes if they contain the query and descend into those which do. Insertion procedure simply finds a suitable leaf to insert the ball, enlarging bounding boxes of nodes on the path.

Similarly to the ray tracing variant there are many possibilities how to build the tree and the data structure quality dramatically affects search performance. We would like to minimize overlapping of nodes as well as surface area of node bounding boxes, because we expect these parameters to affect query performance the most [BKSS90]. For incremental updates we use simple heuristic: in each level we insert the ball into the child whose bounding box have to be enlarged less. If inserting a record causes a leaf record count to exceed a pre-set maximum value (“overflow”), we split the leaf according to a heuristic similar to that of multiple-reference kd-tree (Section 5.3).

In an inner BVH node we can descend into both children, but the condition for traversing a child (it containing the query point) remains the same, therefore the probability is the same as in kd-tree. Only difference reflected by the function is that inner nodes can no longer contain records:

$$f(X) = \begin{cases} N \cdot C^o & \text{if X is a leaf,} \\ C^t + f(L)\frac{V(L)}{V(X)} + f(R)\frac{V(R)}{V(X)} & \text{if X is an inner node.} \end{cases} \tag{5.2}$$

The heuristic is again used for splitting nodes both during insertion and rebuild. We find the best split by iterating over all axes, sorting the list of n records in each axis, computing the costs of $n-1$ possible divisions of the list in two and selecting the one with lowest cost. Values of $f(L)$ and $f(R)$ are again estimated by $N_L \cdot C^o$ and $N_R \cdot C^o$. We also use the heuristic to determine if a node with less than maximum amount of records should be further divided or if a leaf should be created. This is done by comparing the cost of creating a leaf from Equation 5.2 with the lowest cost of split found and selecting the option with lower function value.

BVH is simple and robust data structure, easy to implement and with low memory consumption, because it avoids any object duplication. Its traversal performance is however lower, because multiple paths from the root need to be traversed to answer a query and also because point-bounding box tests are more expensive than simpler point-half-space tests of space-partitioning data structures.

5.6 Dual space kd-tree

Another method for dealing with non-point data we want to try is the *object mapping*, described in Section 4.3. Because a ball in 3D space have 4 degrees of freedom (3 for its origin position and 1 for radius), we transform it to a single point in 4D euclidean space. Exact formula for the transformation is closely linked to the search procedure. We can derive both from the point-ball overlap test:

$$(p_x - c_x)^2 + (p_y - c_y)^2 + (p_z - c_z)^2 \leq r^2,$$

where $\mathbf{p} = (p_x, p_y, p_z)$ is the query point position and $\mathbf{c} = (c_x, c_y, c_z)$ is the ball centre position:

$$\begin{aligned} p_x^2 - 2p_xc_x + c_x^2 + p_y^2 - 2p_yc_y + c_y^2 + p_z^2 - 2p_zc_z + c_z^2 &\leq r^2, \\ -2p_xc_x - 2p_yc_y - 2p_zc_z + c_x^2 + c_y^2 + c_z^2 - r^2 &\leq -p_x^2 - p_y^2 - p_z^2, \\ (-2p_x,\ -2p_y,\ -2p_z,\ 1) \cdot (c_x,\ c_y,\ c_z,\ c_x^2 + c_y^2 + c_z^2 - r^2)^T &\leq -p_x^2 - p_y^2 - p_z^2. \end{aligned}$$

We can easily recognize the 4D point in half-space test in the last inequality. We therefore transform the problem of reporting all balls overlapping a point in 3D to the problem of reporting all points inside a half-space in 4D (Figure 4.1e). Mapping of each record ball with centre (c_x, c_y, c_z) and radius r to 4D point (x', y', z', w') is:

$$(c_x,\ c_y,\ c_z),\ r \rightarrow (c_x,\ c_y,\ c_z,\ c_x^2 + c_y^2 + c_z^2 - r^2).$$

Query point position (q_x, q_y, q_z) is transformed to query half-space defined by this inequality:

$$(-2q_x,\ -2q_y,\ -2q_z,\ 1) \cdot \mathbf{p}^T \leq -q_x^2 - q_y^2 - q_z^2.$$

We organize the transformed points in a kd-tree to accelerate the half-space queries. The tree is identical to our point kd-tree (Section 5.4), except it is built in 4 dimensions and the sliding midpoint rule is not used because we do not perform k-NN queries. The search procedure is, as all range queries, very simple. We just traverse the tree from the root, visiting all nodes which intersect the query range (half-space) and reporting records in all leaves encountered.

One potential problem of this object mapping approach is that the non-linearity of the fourth coordinate transformation may induce a non-uniformity in resulting point distribution, which could negatively affect kd-tree performance. Because of this we translate the scene to have its bounding box centre in the coordinate system origin to lower the magnitude of some of the mapping terms. We also scale the fourth dimension to set its size (difference between minimum and maximum values of stored data) equal to the mean size of first three dimensions. This is to ensure that the tree is built correctly, with roughly ¼ of splits occurring in each dimension.

Transforming the problem to a higher dimension may seem overcomplicated and unintuitive, but it allows us to solve the problem exactly using well-known point kd-tree and half-space range searching algorithms. Biggest problem is the dimensionality of transformed problem – 4D tree is hard to visualize and debug and the increase of dimensionality from 3D to 4D negatively affects its performance. The fact that we do not work directly with original data, but with transformed points, also further complicates debugging.

Chapter 6

Unified radiance caching

Optimizing the search in radiance caching is not as straightforward as in the irradiance caching. We do not implement the *spherical harmonics radiance caching* (SHRC), because its search problem is identical to that of irradiance caching. We instead investigate the *spatial directional radiance cache* (SDRC), which is simpler and promises more potential for improvement.

The search problem in SDRC has two stages: spatial and directional. The spatial phase identifies *spatially close* records. The search is again identical to both irradiance caching and spherical harmonics radiance caching. It is performed once per interpolation and its computational cost is insignificant to the rest of the interpolation. After close records are located we generate random samples on the hemisphere around interpolated point using importance sampling according to the surface BRDF. For each one of them we search all spatially close records for any directionally close radiance samples stored in them. This means that the second search is performed $M \cdot |S(\mathbf{x})|$ times, with M being the number of BRDF samples and $|S(\mathbf{x})|$ the number of spatially close records.

The second search is a k-NN point search in the two-dimensional domain $\mathbf{D}$ in and it is accelerated using a standard kd-tree. It is unlikely that any other data structure would consistently perform better, as the data can be both uniform and highly non-uniform, but it is possible to reduce the number of searches needed to perform the interpolation. This however cannot be done only by changing the search data structure; we need to change the algorithm as well. Because of this we derive our own variant of radiance caching combining spatial and directional interpolation into a single process, called *unified radiance caching*.

6.1 Unified radiance caching overview

Both SHRC and SDRC algorithms are heavily based on the original irradiance caching algorithm. We instead use the *phase space rendering* approach presented by Hinkenjann [HR07]. We no longer create any records in sense of radiance samples clusters and we abandon the split sphere model spatial interpolation phase.

We still reconstruct the exitant radiance using cached incident radiance similarly to SDRC, but we store individual incident radiance samples in a single data structure, the *unified radiance cache*. Each radiance

sample has an origin and a direction, that means that the *phase space* of all possible radiance sample configurations has 5 degrees of freedom and the unified radiance cache have to operate in at least 5D space.

To compute the reflected radiance in a point we solve the reflection equation (Equation 2.10) using the Monte Carlo method. We create M samples on the hemisphere using importance sampling according to BRDF and cosine terms. The estimator is simply:

$$\langle I \rangle = \rho \frac{1}{M} \sum_{n=1}^{M} L(\mathbf{x} \leftarrow \vec{\omega}_{i_n}), \tag{6.1}$$

with the PDF:

$$p(\vec{\omega}_{i_n}) = \frac{f_r(\mathbf{x}, \vec{\omega}_{i_n}, -\vec{\omega}_o) \cos\theta_{i_n}}{\rho}. \tag{6.2}$$

We are able to solve the integral non-recursively by using the cached incidence radiance values. For each direction $\vec{\omega}_{i_n}$ generated by the sampling we need to search the cache for values close to $(\mathbf{x}, \vec{\omega}_{i_n})$. If such values exist, we interpolate them to get the estimate of $L(\mathbf{x} \leftarrow \vec{\omega}_{i_n})$. If the interpolation is not possible, we create new radiance sample by casting a ray from $\mathbf{x}$ in the direction of $-\vec{\omega}_{i_n}$, insert it into the cache and use its value in the estimator. Stored radiance samples and generated BRDF samples are shown in Figure 6.1.

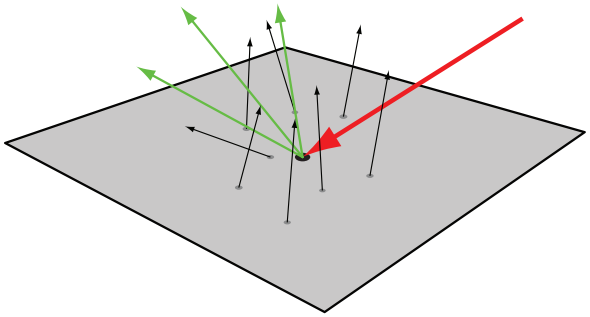


Figure 6.1: *An illustration of the unified radiance caching. Single primary ray (red) hits the surface. Multiple BRDF samples are created (green rays). These samples are not ray traced, but instead we perform the interpolation for each of them, using cached radiance samples (black).*

6.2 Radiance samples interpolation

To calculate the exitant radiance at a point using Equation 6.1 we need to interpolate incoming radiance for several directions. This interpolation is an example of the *scattered data interpolation* problem. Standard approach is to interpolate between stored samples using weights computed as inverse of squared distances between stored samples and the query point (this process is called *inverse distance weighting* [She68]). We use this method, but for efficiency reasons we do not use all stored samples for each interpolation. We instead limit the interpolation to k nearest neighbours that are not farther from the interpolated point than a maximum interpolation distance.

The problem is how to define the distance both for finding nearest neighbours and for the weight computation. To be able to use standard data structures for efficient searching we need the metric to be Euclidean. Because of that we simply unite spatial and directional dimensions (by directional dimensions we mean dimensions of the Euclidean space to which we map directions, as described later) to create single 5D space. Distance between two arbitrary radiance samples is then:

$$d(L(\mathbf{x}_1,\ \vec{\omega}_1),\ L(\mathbf{x}_2,\ \vec{\omega}_2)) = \sqrt{\alpha\|\mathbf{x}_1-\mathbf{x}_2\|^2+\beta\|\vec{\omega}_1'-\vec{\omega}_2'\|^2}, \tag{6.3}$$

where $\vec{\omega}'$ is the mapped radiance direction vector and α and β are spatial and directional sensitivity coefficients, respectively. They determine how sensitive is the interpolation to spatial and directional deviations of cached values from the interpolated point.

The interpolation is now very simple. If we wish to interpolate the value of $L(\mathbf{x},\ \vec{\omega})$, we simply locate k closest cached samples according to the metric from Equation 6.3 and assign them weights by inverse distance weighting:

$$w(L(\mathbf{x}_i,\ \vec{\omega}_i)) \ =\ \max\left(\frac{1}{d(L(\mathbf{x},\ \vec{\omega}),\ L(\mathbf{x}_i,\ \vec{\omega}_i))^2}\ -\ 1,\ \ 0\right). \tag{6.4}$$

If all weights are zero, we simply compute a new radiance sample, store it in the cache and use its value as the interpolation result. If there is at least one sample with non-zero weight, we perform the interpolation:

$$L^{\text{interp.}}(\mathbf{x},\ \vec{\omega}) \ =\ \frac{\sum_{i=1}^{k} w(L(\mathbf{x}_i,\ \vec{\omega}_i))\ L(\mathbf{x}_i,\ \vec{\omega}_i)}{\sum_{i=1}^{k} w(L(\mathbf{x}_i,\ \vec{\omega}_i))}.$$

We subtract 1 from the weight in Equation 6.4 to limit the maximum distance at which any sample can be reused. This is an arbitrarily chosen constant; the actual domain over which each sample can be reused is determined by our choice of coefficients α and β in the distance computation (Equation 6.3).

Biggest problem is how to determine the coefficients α and β. Setting them too high would cause caching inefficiency because the samples could not be properly reused, but setting them too low would cause artefacts and bias as a result from reusing samples over too large area. For the spatial coefficient α we draw inspiration from record spacing limits in irradiance caching (Section 3.1.5). We choose the coefficient so that, assuming the maximum distance for interpolation of 1, a radiance sample can be reused no farther than few (3–10, 5 in our implementation) projected pixels. For the directional coefficient β we use the same approach as in SDRC and base it on the number of BRDF samples N and the probability of generating a particular sample $p(\vec{\omega}_i)$ (Equation 6.2). We also limit it by using a global minimum directional sensitivity β_{min} to avoid errors from reusing samples over too large area in directions where $p(\vec{\omega}_i)$ is low. The resulting formula for β is:

$$\beta \ =\ \max\left(\beta_{min},\ \ k\sqrt{Np(\vec{\omega})}\right). \tag{6.5}$$

The formula is based on similar formula from the SDRC (Equation 3.11) that operates with maximum interpolation distance instead of sensitivity. These two quantities are reciprocal. We use a constant multiplier of directional difference sensitivity k. Its value is determined empirically. In our implementation we let $k = 15$. The β value adapts both to the number of BRDF samples N and to the BRDF, causing

more accurate directional interpolation at sharp BRDF peaks and when using more samples in the exitant radiance estimator (Equation 6.1).

Note that we use $\sqrt{N}$ instead of directly N in the last formula, unlike the original SDRC formula (Equation 3.11). If we would have used N directly then doubling the number of BRDF samples would double the sensitivity and reduce the maximum search radius by half. This would reduce the area of the disk region where each sample can be reused to one quarter. Probability that we can reuse a sample is proportional to the area of this disk. This means that we would need quadratically more cached samples when linearly increasing the BRDF samples count. By using $\sqrt{N}$ we keep the relation linear.

6.3 The directional mapping problem

Biggest problem in the unified radiance caching is how to integrate the directional dimensions into the Euclidean space, to allow efficient searching. Ideal mapping would be area-preserving and shape-preserving (so it does not warp the radiance proximity metric) and continuous (because discontinuities would cause close samples to be neglected during the interpolation). Unfortunately, no single mapping can satisfy these requirements.

There are several existing mappings we could possibly use, with some being better than other. Building the tree over the standard θ, ϕ parametrization of a sphere is out of the question, as this parametrization extremely deforms the space, has two singularities on poles (segments that map to single points) and a discontinuity along one edge. As a result the directional sensitivity of the cache would be greatly influenced by the direction of query and many valid samples would be missed while searching near the discontinuity. More straightforward way is to simply build the tree over endpoints of direction vectors on the unit sphere. This would work, as the sphere surface is locally sufficiently flat, but it would add an additional unnecessary dimension to the search problem. Because of the curse of dimensionality 6D search is considerably more difficult than the basic 5D variant. Another option is the paraboloid mapping [HS98] used in the SDRC, but it is not area-preserving and it is discontinuous along the equator.

We solve the problem by dividing the sphere of dimensions into a north and south hemisphere and keeping two separate trees for each one. This obviously creates a discontinuity along the equator. We remedy this by enlarging both hemispheres to create a central overlapping belt, as shown in Figure 6.2. Each record that falls into the belt is duplicated – stored in both north and south hemisphere. Size of the belt is determined by the maximum directional distance at which a sample can be reused (which is reciprocal to the minimal directional sensitivity β_{min}).

To map a direction (unit vector end point) on the north hemisphere to a plane we use the area-preserving bicontinuous mapping from unit hemisphere (x, y, z) to unit disk (r_d, ϕ_d) [Dut03, p. 19]:

$$\begin{aligned} r_d &= \sqrt{1-y}, \\ \phi_d &= \operatorname{atan2}(x, z), \end{aligned}$$

combined with the mapping from a disk to square by Shirley [SC97], which is also area-preserving, bicontinuous and with low distortion. The first mapping is sufficient for our task on its own, the second

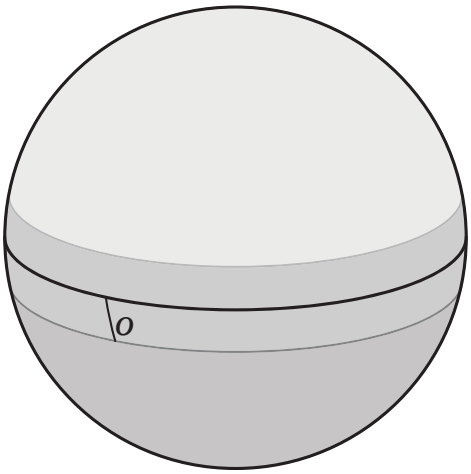
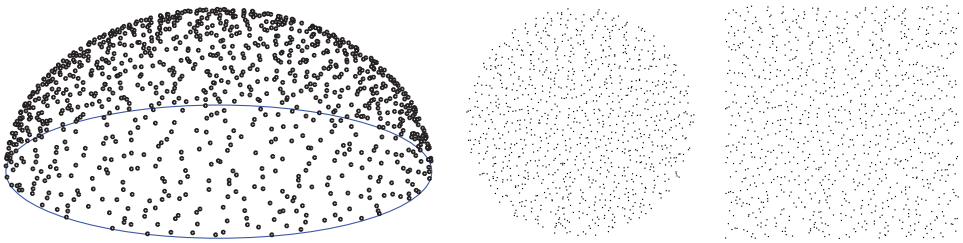


Figure 6.2: *This illustration shows how we divide the entire sphere to two hemispheres and then enlarge them to get the central overlapping belt. Any point on that belt belongs to both north and south domains. Half-size of the belt o is equal to* $1/\beta_{min}$.

mapping is not necessary for the algorithm to work. We however use it because it is simple, fast to compute when we already have polar coordinates of the point on disk computed and it makes the domain rectangular, which benefits the kd-tree we use for searching. The mapping results are shown in Figure 6.3.



(a) *Uniformly distributed points on a hemisphere.*

(b) *Result of mapping from the hemisphere to a unit circle.*

(c) *Result of mapping from the hemisphere to unit circle to unit square.*

Figure 6.3: *Mapping of directions in 3D to a plane in 2D: first picture shows original data (uniformly distributed directions on a unit hemisphere). Second picture shows the same directions mapped to a unit disk. Third picture shows the same directions additionally mapped to a unit square using Shirley's mapping.*

Mapping for the south hemisphere is analogical, we only change the sign of y. Both mappings have no problem with the overlap belt at the equator – they map points from the other hemisphere continuously outside the original unit rectangle. We could in fact use single tree for the whole sphere with this mapping, but the mapped domain would have a singularity at one of the poles and nearby areas would be greatly distorted by the mapping.

6.4 Samples storage

Because the samples are simple points in 5D space, we can use the standard kd-tree, virtually identical to the one we use in irradiance caching (Sections 5.4 and 5.6) except for dimensionality. Because of the mapping we have to actually keep two kd-trees in parallel. Before storing each sample we first determine whether it should go in the north, south or both trees. After that the mapping is performed, the tree is traversed and the sample is inserted into the node it intersects.

During searching we map the query to a 5D point and perform the standard k-NN query using the distance metric from Equation 6.3 with coefficients α and β computed separately for each query. The kd-tree needs to be build carefully, with keeping in mind that spatial dimensions can have dramatically different sizes than directional ones. We need to build the tree with approximately ⅔ splits occurring in directional dimensions.

6.5 Ray length heuristic

One big problem of the described algorithm is that by abandoning aggregate records we have given up the information about R_i, the distance to other surfaces. We cannot use it to make the algorithm adaptive to scene geometry, which is the key to producing high quality results. We achieve the adaptivity another way – for each radiance sample we store its ray length (distance to the surface from which the radiance was emitted) and during the incident radiance interpolation we calculate weighted variance of all used sample ray lengths. If the variance exceeds a pre-set threshold we compute a new radiance sample instead of interpolation, use it and store it in the cache. This gives us some degree of geometry adaptivity, because high variance indicates that reflected rays hit very different objects in the scene, as shown in Figure 6.4.

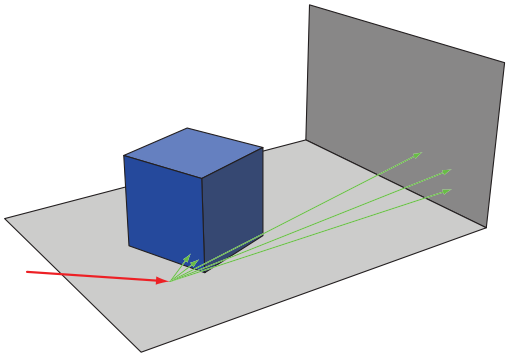


Figure 6.4: *An example of scene where varying ray lengths of radiance samples used for interpolation indicate fast-changing indirect illumination: because a part of samples hit the cube and the rest hits the background, the variance of ray lengths is there very high.*

Chapter 7

Implementation

We have implemented the data structures and algorithms described in Chapters 5 and 6 in our rendering system, *Corona*. It was created as a part of bachelor thesis [Kar09] and since then it was being continuously expanded and improved. In its current version it implements wide variety of algorithms for realistic image synthesis in static scenes via ray tracing.

7.1 Corona framework architecture overview

Corona is a rendering framework written in C++ language. It consists of multiple applications. Central component is *CoronaCore* - the rendering core, which takes already loaded scene, camera description, and render settings, and renders it. It does not handle scene loading nor output display. It is realized as a dynamic-link library (.dll) loaded by a front-end application handling input and output. There are two front-ends currently available: *CoronaStandalone* – a standalone (.exe) application, which is capable of loading and parsing various file inputs and displaying the results using simple graphical window and *CoronaMax* – a plug-in for the Autodesk 3ds Max modelling software [Inc11]. There is also a set of utility plug-ins (materials, data visualization tools, etc.) for 3ds Max called *CoronaMaxUtils*.

7.1.1 Rendering core interface (*CoronaCore*)

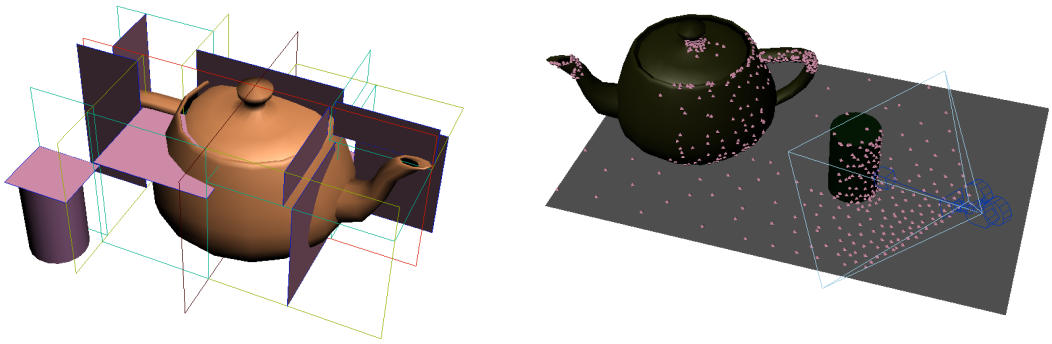
Only task of CoronaCore is to create the final render from an already prepared scene. Its interface is very simple. Caller simply creates an instance of class `CoronaCore` and then calls its `render` method, which renders a single image. All input data are passed to this method in an instance of the `Context` class. This aggregate class holds pointer to the scene being rendered (an instance of the `Scene` class) and pointers to several objects realizing the input/output interface. These objects are implemented by the caller; callee only specifies the needed interface in form of pure virtual (abstract) methods. They are used to retrieve render settings (`Abstract::Settings`), store resulting image (`Abstract::FrameBuffer`), log events, progress and statistics during rendering (`Abstract::Logger`) and visualize various data structures for debugging and tuning (`Abstract::Visualizator`).

7.1.2 3ds Max plug-in integration (*CoronaMax* and *CoronaMaxUtils*)

CoronaMax is implemented as a plug-in for 3ds Max. There are two ways to create a 3ds Max plug-in: either via *MAXScript*, a built-in high-level scripting language, or by creating a C++ dynamic-link library using 3ds Max software development kit (SDK) [Lan07]. Due to performance considerations and limitations of MAXScript, CoronaMax uses the latter option.

Main class of the plug-in, `CoronaMax`, is derived from the `Renderer` class provided by 3ds Max API. Any such derived class is automatically listed as an additional renderer in the system. The plug-in acts as an adaptor between the rendering core and the 3ds Max API. It provides the rendering core with implementations of all needed classes, each of which encapsulating different functionality of 3ds Max. This allows the core to be application-independent.

Settings are handled by the `MaxSettings` class. It uses 3ds Max integrated *Parameter blocks* system [Inc09] for setting and retrieving the render settings, as well as saving and loading it in the 3ds Max native scene format. An instance of `MaxFrameBuffer` handles the image output – it holds pointer to a `Bitmap` class instance provided by 3ds Max API, which is mapped to the rendered frame window inside the 3ds Max. Logging is handled by `MaxLogger`, which logs events to a file and updates the progress bar in 3ds Max render dialogue provided by the API. The visualizator can be of two types: a `DummyVisualizator`, which does nothing, and `DataViz`, which is realized as a scene object (a part of the scene hierarchy in the modeller) derived from the `HelperObject` API base class. This visualizator can display various data structures it receives from CoronaCore as geometry objects in 3ds Max viewports. Examples of these visualizations are shown in Figure 7.1.



(a) *Kd-tree visualization.* **(b)** *Irradiance cache records visualization.*

Figure 7.1: *Two examples of data structure visualizations in the viewport of 3ds Max – a ray casting kd-tree (left) and an irradiance cache (right). The irradiance cache in the second image was created by rendering the image using the depicted camera.*

The renderer also implements its own material class, `CoronaMtl`, with custom adjustable parameters, which directly translate to Corona native material properties. It is implemented by deriving the class from API base class `Mtl`. Its parameters are once again stored using the *Parameter blocks* system.

During the start-up of 3ds Max an instance of `MaxEnvironment` is created. It parses the default settings default configuration files (*.conf* files, described in detail in Appendix C), holds it together with other data

and remains in memory until the plug-in in unloaded. It can be accessed by anyone using the Singleton pattern [GHJV95].

3ds Max starts the render by calling the `Render` method of `CoronaMax`. First settings from configuration dialogue are saved, then `MaxSceneParser` is instantiated. It iterates through scene objects and extracts all geometry, lights and materials. `MaxSceneBuilder` accumulates them and builds the scene. The scene is saved to a `Context` instance and passed to `MaxEnvironment`, which instantiates `CoronaCore` and invokes its render method, starting the actual render. In order to provide maximum responsiveness of user interface during rendering, `MaxEnvironment` calls the method in another thread. The main thread only periodically updates the rendered frame window and handles any eventual user input.

A renderer in 3ds Max have to render not only the final image, but also small material previews in the *material editor*. Special care have to be taken to handle this. Fortunately 3ds Max API provides a way to differentiate standard render from material render. For a material render we disable logging and render progress bar updating (because there is no progress bar), add default lights to the scene and most importantly, use different set of settings. This is to ensure consistent behaviour of material editor regardless of render settings for final image and because of optimization, as material editor does not require such high quality rendering as final image.

7.1.3 Standalone renderer (*CoronaStandalone*)

The "standalone" version of Corona is much simpler. It loads scene description from file (or multiple files), uses CoronaCore to render the result and displays it on the screen using the Simple DirectMedia Layer (SDL) multimedia library [Gam11].

The application is capable of parsing multiple input formats. Its primary input is a scene (.scn) file, which contains list of all files to parse for the scene. It supports Wavefront OBJ format for geometry [Ali] (triangles and quads) and MTL files for corresponding materials definition [RRT95]. Settings are loaded from the same *.conf* format as in CoronaMax. Textures are supported in BMP format using the EasyBMP library [Mac06] to load them.

Because both OBJ and MTL formats are very limited, we have developed simple extensions to the original format – syntax for specification of camera, lights and advanced material properties. These additions are in form of special comments. Because of that standard parser ignores them and they do not cause syntax error. List of these extensions in Appendix C.

CoronaStandalone can also load one binary scene format – *Corona binary file* (.cbf). This format is as close as possible to a simple binary dump of memory where the scene is stored. This causes the saved scene to be small and saving/loading to be fast, but scenes saved in older versions of Corona may be incompatible with newer builds. Single .cbf file contains entire scene including materials and textures. Only settings are stored separately in an independent .conf file. CBF export is built into the core. When the scene is to be exported, no rendering takes place and CBF and OBJ exports are created instead. The OBJ file created uses the extensions described earlier.

Single run of CoronaStandalone renders a single image. First, all input files are parsed, scene is created and settings are loaded. Settings are stored using an instance of `StandaloneSettings`. Frame buffer for the standalone renderer is implemented in the class `SdlFrameBuffer`. It holds pointer to a `SDL_Surface`

provided by the SDL library, using which results are displayed on screen. Second implementation of a frame buffer is `SimpleFrameBuffer`, a simple class with no GUI used for testing. Both frame buffers also save the final image after rendering as a BMP image using the EasyBMP library. Logging is handled by `StandaloneLogger`, which writes log events to standard output as well as to a log file. CoronaStandalone always uses `DummyVisualizator` as visualizator, as it does not support visualization of any data structure. The rendering is once again invoked in another thread in order to provide responsive user interface.

7.2 Corona rendering core

The renderer has modular architecture. Many key components are realized as exchangeable components by specifying required interface in an abstract base class and using virtual methods. Centre of the renderer is the `InternalInterface` object, which holds the scene, input/output objects passed by the front-end (frame buffer, settings, logger, visualizator), camera, ray tracing acceleration data structure and the integrator (object which computes colour for given ray). Many other objects hold pointer to the `InternalInterface`, as it provides other parts of renderer with key functionality, such as the ability to cast a ray, create a ray from image sample, and turn a ray into colour via shading.

7.2.1 Scene and settings

Scene description in Corona is a single instance of the `Scene` class, which holds pointers to all primitives, lights and materials. All primitives in the scene are stored in a single array; no other information about their hierarchy in the scene is preserved. Corona supports multiple primitive types via polymorphism: all primitives are stored up-casted to the base class `Abstract::Primitive`. This class provides interface for ray intersection and stores the material ID (index to the materials array) of the primitive. Currently implemented primitives are triangle, quad (two triangles sharing common edge) and sphere. Both triangles and quads use the *Möller-Trumbore intersection test* [MT97]. Other primitives can be added easily, but virtually all available scenes use exclusively triangles. All materials are derived from the `Abstract::Material` base class, which defines interface for shading and holds pointer to its BSDF. Each material in Corona has an associated BSDF object. It is an instance of class derived from `Abstract::Bsdf` and it has two key methods: `transported`, which calculates the BSDF value pre-multiplied by cosine $(f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_o) \cdot \cos\theta_i)$ for given pairs of incident and exitant directions, and `sample`, which performs importance sampling of exitant directions for given fixed incident direction according to the BSDF value multiplied by cosine (Equation 2.15).

Difference between a material and a BSDF is that material is more complex object. It stores all colours and textures required for shading the surface, and it is capable of performing this shading on its own. BSDF is much simpler, as it only represents the reflectivity distribution function. It cannot perform shading independently, but instead is used by GI algorithms. It obtains all informations about the surface (e.g. diffuse and specular colour, specular exponent, ...) from its parent material.

Corona currently supports two materials: `NativeMtl` – the standard Corona material, used for realistic rendering and `UnsupportedMtl` – dummy material with constant colour, used when unknown material is encountered at input. Implemented BSDFs are `NoBsdf`, a dummy BSDF whose value is always

zero (ideal black body), and `PhongBsdf`, the modified Phong BRDF with possible refractive component (Equation 2.9).

All lights of the scene are stored in an array, up-casted to their common base class `Abstract::Light`. This class, however, does not provide any interface for illumination computation, because there are two principally different types of lights: *delta lights* and *area lights*. Illumination from a delta light can be easily determined by a single shadow ray, because this type of light is an idealized model which does not emit energy from a surface, but only from single point (point light, spot light, ...) or only in single direction with no origin (directional light). Area lights in contrast have non-zero light-emitting area and have to be sampled using the Monte Carlo method. They represent the L_e factor in rendering equation (Equation 2.11).

Because each light type requires different interface for the illumination computation method, there two additional abstract base classes derived from the `Abstract::Light` class: `Abstract::DeltaLight` and `Abstract::AreaLight`. Each one has the same method `getIllumination` but with different parameters; area light variant takes number of shadow rays to use and an array of random samples as additional parameters. Delta lights does not need them as they are deterministic. The illumination method in both cases takes the shaded surface BSDF as a parameter and returns radiance reflected in viewing direction by the BSDF due to the light, not radiance incoming to the surface from the light[1]. There is one additional light type: `Abstract::EnvironmentLight`. It represents illumination coming from non-black scene environment. Although it could be represented by an area light type instance, we choose to separate it, as it is often advantageous to handle direct environment lighting differently from ordinary lights. This separation allows us for example to selectively enable or disable explicit environment sampling for different types of rays.

Corona currently supports single delta and single area light type. The delta light supported is `PointLight` – a standard omnidirectional point light with optional quadratic attenuation. It illuminates surfaces deterministically by casting a single shadow ray between the light origin and surface point. The area light implemented is `TriangleLight`, a light-emitting triangle. We use triangles instead of rectangles, because any light-emitting surface, including rectangles, can be triangulated. A `TriangleLight` instance is automatically created for any triangle in scene with light-emitting material. The emission is required to be diffuse and constant across the triangle ($L_e(\mathbf{x} \rightarrow \vec{\omega}_o)$ must be constant with respect to both $\vec{\omega}_o$ and $\mathbf{x}$). Illumination due to this light is an integral over the triangle surface, which is estimated by Monte Carlo method (using Equation 2.16 and the estimator in Equation 2.4.3).

There is currently only one implementation of environment light - class `EnvironmentLight`. It allows explicit sampling of the environment using shadow rays. Directions to be sampled are chosen with PDF proportional to the environment colour intensity.

The environment is stored discretized and rendered to a bitmap. Cumulative intensities are computed for pixels in each row and for all rows as wholes. Sampling is then done by generating a random sample and locating corresponding cumulative intensity value using binary search. Importance sampling according to the lighting intensity gives bad results in scenes with uniform environment colour. It is usually better not to use explicit environment lighting and sample it implicitly (through rays that missed the scene). Exceptions are scenes with highly non-uniform environment, for example scenes lit by an HDRI map.

[1] This is because of area lights, which would otherwise have to return some sort of incoming radiance distribution description, which would then have to be integrated together with the BSDF.

7.2.2 Lighting solvers

The lighting computation in Corona is split into multiple terms for efficiency reasons; each one is handled by different algorithm. These algorithms are once again made interchangeable by wrapping them in solver objects which make use of virtual methods. The quantity computed by all solvers is the radiance reflected off the surface towards the viewer. Some solvers may need to perform precomputation. Because of this the base classes have the `precompute` method, which has to be called before the solver is used. Another shared methods are `dumpToFile` and `loadFromFile`, which are used for saving and subsequent loading of the GI solution from a file. The `loadFromFile` method is called instead of `precompute`.

Direct lighting (light arriving at surface directly from light, with no bounces on its path) is computed by a *direct light solver* (derived from `Abstract::DirectLightSolver` class), which determines which lights in scene to sample. First implemented solver, `SimpleLightSolver`, offers simple implementation which iterates over all lights and accumulates their contributions. `SmartLightSolver` is more sophisticated solver, which samples all lights as whole. It works by iteratively randomly selecting single light with probability proportional to its power output and sampling it. This allows to calculate lighting in scenes with many lights using only constant number of shadow rays (as described in [SWZ96]).

Global illumination (light arriving at surface that was reflected at least once in the scene) is computed by `Abstract::GiSolver`. To further increase modularity, there are two types of GI solvers: *primary* and *secondary*. Primary solver (`Abstract::PrimaryGiSolver`) computes the GI for primary hit points (points directly visible from camera). A secondary solver (`Abstract::SecondaryGiSolver`) is used only by some types of primary solvers which need to compute GI at additional points in the scene. All these solvers are held by the `LightingEngine` class, which encapsulates the entire lighting process in single method, `getIllumination`.

Simplest GI solver is the `NoGiSolver`, which always return black colour. It is a dummy solver used for disabling GI computation. `PathtracingGiSolver` computes GI using the Monte Carlo path tracing described in Section 2.4.3. Photon mapping (Section 2.4.4) in Corona is implemented by the `PhotonMapGiSolver`. It stores all photons in the `PhotonMap` object, with additional kd-tree built over them (realized by the universal `PointKdTree` class). The photon emission and kd-tree build takes place during the precomputation. Irradiance and radiance caching implementations are encapsulated in the `IcGiSolver` class, which holds irradiance and radiance cache solvers. Because of performance reasons the solvers are supplied as template parameters. Our irradiance caching implementation is described in greater detail in Section 7.3.

The relations between solvers are quite complicated. Because the primary GI solver depends on the secondary and they both depend on the direct light solver, direct light solver is initialized and runs precomputation first. Then the secondary GI solver is initialized, if primary solver needs it. Primary GI solver is initialized last. Another intricacy is that some solvers can compute some direct lighting components virtually for free. For example the photon mapping can easily compute direct lighting from scene lights (by storing photons at their first bounce) and irradiance caching can easily compute diffusely reflected direct lighting from environment. Because of this each GI solver has a `componentsComputed` method, which returns direct lighting components it computes, and direct lighting solvers and environment maps can be configured to compute only those components not covered by the GI solver.

`NoGiSolver`, `PathtracingGiSolver` and `PhotonMapGiSolver` can be used both as primary and secondary

solvers. `IcGiSolver` can be only used as the primary solver. By combining different solvers we can easily recreate common algorithms traditionally used for computing global illumination: by using photon mapping as primary solver we get *direct photon map visualization*, but by using it as secondary solver and selecting path tracing (with high first bounce branching) as primary solver we obtain the *final gathering*. The final gathering can be accelerated by selecting irradiance caching as primary solver, irradiance caching on its own with no secondary GI (`NoGiSolver`) produces single bounce GI approximation, and so on.

7.2.3 Rendering workflow

The render is started when a front-end instantiates the `CoronaCore` class, calls its `render` method and passes it a `Context` class instance holding the prepared scene. `CoronaCore` instantiates the `InternalInterface` and initializes its various components (GI solvers, camera, ...). First the acceleration data structure is built using user-selected data structure builder. Then, if physically correct lighting is needed, direct and indirect lighting solvers run their precomputation[2]. After this initialization phase a *renderer* is instanced. Renderer in this context is an instance of class inheriting from `Abstract::Renderer` that controls the rest of the rendering process.

There are various renderers implemented. `LightTracerRenderer` is simple unbiased, progressive renderer that uses *light tracing* – the dual algorithm to path tracing [DBB06, p. 143]. It propagates photons through the scene the same way as photon mapping does, but does not save them. Instead, a *contribution ray* from each surface hit by a photon is sent to the camera and if it is unoccluded, we add its contribution to the pixel it projects to. This process is depicted in Figure 7.2. Because the photons are not stored and the method is unbiased, the renderer can run in an infinite loop, constantly refining the image, until it is interrupted by user.

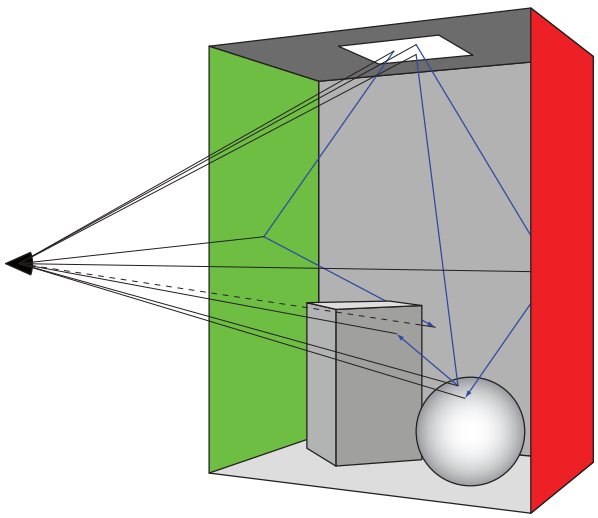


Figure 7.2: *Visualization of the light tracing rendering algorithm. Rays are shot from the light and traced through the scene. Any time a ray hits any surface a contribution ray is created and traced towards the camera. If there is no obstacle in the way then contribution of the ray is computed and added to the image.*

[2]Precomputation phase for various algorithms differs greatly: photon mapping solver emits photons and builds a kd-tree from them, irradiance cache solver populates the cache, path tracing solver does nothing.

The standard renderer implementation is the `MultithreadedRenderer`. It allows multithreaded rendering by splitting the rendered region into multiple square tiles, called *buckets*. It then uses a *scheduler* (`Abstract::Scheduler` subclass) to spawn worker threads, each of which renders single bucket at a time. Two scheduler implementations are available: `SingleThreadedScheduler` uses only single thread to render the image sequentially while the `WinThreadsScheduler` spawns multiple threads using Microsoft Windows API functions [Mic10].

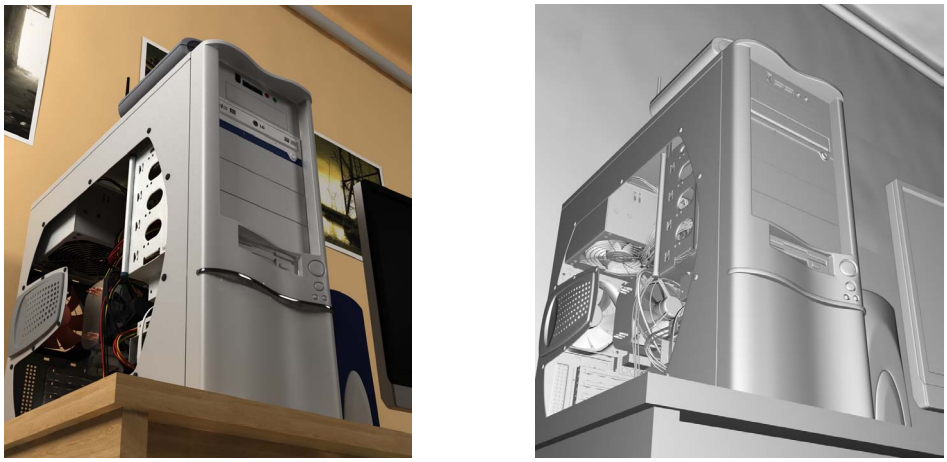
`MultithreadedRenderer` instantiates an *image sampler* that determines which pixels and how to sample, creates primary rays for these samples using the *camera*, turns these samples into colour using an *integrator*, and accumulates results in the frame buffer.

The image sampler interface is defined in the `Abstract::ImageSampler` class. It handles the integration of radiance over each pixel area to create its final colour. It determines both the number and positions of samples for each pixel as well as order in which the pixels are evaluated. Currently the only implementation of image sampler in Corona is the `AdaptiveSampler`. It supports adaptive anti-aliasing by iterating multiple times over the image. All pixels are sampled in the first pass, then in subsequent passes it computes perceptual difference for all neighbouring pixels (based on the *Weber law* [FPSG96]), and places additional samples at pixels which differ from their neighbours by more than a specified threshold.

The camera does not only create rays from samples, it can also perform the inverse operation (called *unprojection* in Corona) and compute estimated projected pixel area in scene. The standard implemented camera is the `PinholeCamera`, an idealized model of projective camera (camera obscura), producing rays with a single common origin and different directions. `ThinLensCamera` is another camera implementation. It simulates a real world camera with non-negligible aperture size, which causes blurring of objects that do not lie in the focal plane (the *depth of field* effect). It is simulated by selecting the ray origin for each ray randomly on the surface of the aperture (which is simplified to a disk). Other than perspective projection is possible, as demonstrated by the `OrthogonalCamera`. It implements the orthographic projection with arbitrary direction vector. It generates rays sharing the same direction, but differing in origin.

The final step in the rendering process is obtaining the colour from the ray. This is done by a subclass of `Abstract::Integrator`. Default integrator is `SimpleIntegrator`. It works by tracing the ray it is given, and then calling the `shade` method of material it hits, or returning background colour for any rays missing the scene (Figure 7.3a). This effectively delegates the shading process to materials, which allows combining various shading methods. Other integrators can be however used to override the materials. One implemented example is the `DotProductIntegrator`, which after tracing the ray simply returns shade of grey computed as cosine of the angle between ray direction and surface normal (Figure 7.3b). Because this shading is simple and predictable, it is useful for debugging.

There are another specialized renderers – `SaveHitpointsRenderer` and `LoadHitpointsRenderer`. They are used for saving and then loading hit points of primary rays to disk. Because this eliminates randomness of image sampling and the overhead of ray shooting, it can be useful for debugging and shading algorithms performance measurements.



(a) *SimpleIntegrator.* **(b)** *DotProductIntegrator.*

Figure 7.3: *Two images of the same scene made with different integrators. The first image uses the standard integrator – SimpleIntegrator. Second image uses specialized intergrator (DotProductIntegrator) that creates very different shading.*

7.2.4 Ray casting engine

The `InternalInterface` class provides three different methods to cast a ray: `intersect` returns the closest intersection of a ray with the scene. It implements the *ray-casting function* $r(\mathbf{x}, \vec{\omega})$ (Section 2.3). The *visibility function* $V(\mathbf{x}, \mathbf{y})$ is realized by the `castsShadow` method. It casts a *shadow ray*, which searches for any intersection closer than specified distance, and returns only true or false value indicating whether any such surface exists. The final ray casting method, *intersectAll*, returns all intersections of a ray with the scene, stored in an array.

All ray casting methods use an acceleration data structure to achieve sub-linear query time. Corona implements naïve list, uniform grid, kd tree and bounding volume hierarchy (described in Section 2.4.1). They are all derived from a single abstract base class (`Abstract::AccelerationStructure`). The virtual methods mechanism allows to change the data structure during run time. Type of the data structure is determined by *structure builder*, which is selected by the user. There can be multiple builders for single data structure, each one using different method to build it.

Uniform grid is built by single sweep over the data. Both kd-tree and bounding volume hierarchy are built using the surface area heuristic (SAH) (Section 2.4.1). Standard top-down build algorithms recursively split the list of primitives to create the tree. SAH determines how to make the splits by evaluating many possible candidates and picking the one that minimizes the SAH cost function.

For BVH we have implemented "full SAH" builder (class `SahBvhBuilder`), which sorts primitives in each axis and evaluates $3(n-1)$ possible splits [WBS07] (with presorting optimization that makes it run in $O(N \log(N))$ time [WH06]), and "binning" builder (class `Binning1BvhBuilder`), which evaluates the SAH only for a fixed number of bins [Wal07], trading tree quality for the build speed. Similarly for kd-tree we have "full SAH" (`SahKdBuilder`) [Hav00] and "binning" (`BinningKdBuilder`) builders. Standard data structure ordinarily used for rendering in Corona is the BVH built using full SAH builder,

because it gives the best results in our implementation.

7.3 Irradiance and radiance caching in Corona

In Corona we use irradiance and radiance caching together – irradiance caching is used for the diffuse part of shaded surface BRDF and radiance caching is used for the remaining non-diffuse part. This BRDF decomposition is allowed by the linearity of BRDF (Section 2.2.2). We implement irradiance and radiance caching as separate independent classes that are joined together in a single GI solver – `IcGiSolver`. This solver encapsulates both caches, distributes work between them (it calls only irradiance cache interpolation, only radiance cache interpolation or both, based on surface properties of interpolated point) and combines their results.

Particular irradiance and radiance caches can be interchanged via the C++ templating mechanism. `IcGiSolver` has two template parameters – class names of both solvers – and is instantiated with right template parameters based on configuration from the user. After that it is up-casted to its abstract predecessor like other GI solvers, and each different template configuration is treated by C++ as a different derived class. This creates a sort of "compile-time" polymorphism that allows us to select solvers at run time without duplicating source code and without additional overhead incurred by the virtual calls mechanism.

Both used caches have to implement similar interface. They have the `init` method for initialization and configuration, `rebuildStructure` method for rebuilding the acceleration data structure used, method `precomputeInPoint` that precomputes the cache in a single given point, and finally the `interpolate` method for interpolating the cache at given point. The solvers implement two methods for serialized to disk: `save` saves the cache contents into given stream and `load` loads previously saved cache from a stream. The latter has a parameter determining if the cache should be loaded incrementally (merged with current cache content). Finally, each cache have a `makeStats` method for computing statistics.

The solver initialization is simple – it obtains settings from `InternalInterface` and stores them in an `IcConfig` class instance, then it initializes irradiance and radiance caches used. Because we use two-pass rendering (Section 3.1.4), the `precompute` method needs to populate both caches. This is done by an `IcSampler` instance, which acts as the renderer and renders the image multiple times in progressively increasing resolution (*hierarchical refinement* technique). First pass image resolution is only a fraction of that of final image. Each successive pass doubles the resolution of previous pass in each dimension. For efficiency reason we do not perform full shading during the precomputation phase, instead we only compute primary ray intersections and try to interpolate GI in these points.

7.3.1 Irradiance caching implementation

There is only single irradiance cache implemented (class `IrradianceCache`), it has however a template parameter determining the type of data structure it uses. Different data structures have to implement the same interface so that the irradiance cache can work with them without any code redundancy. Any its initialization is done in its `init` method. This method is called before the data structure is queried and before any records are stored. It initializes and configures the data structure (using the `IcConfig` object that is passed as an argument). The data structure is rebuilt by its `build` method. This method takes the

scene bounding box and the list of all irradiance records as arguments. Individual records are added using the `addRecord` method. Last method is `makeStats`. It is used for compiling various statistics.

The search is not implemented as a simple method, because it is more complicated to unite all search procedures under single interface. We cannot simply make a method that returns an array of results, because there is no guaranteed maximum number of results, and dynamic allocation is out of the question for performance reasons. Because of that we use the *iterator* design pattern – we use a special object for iterating through the result set. Each data structure is required to have an inner class `Iterator`. Search is performed by instancing the iterator with the search data structure and query point as parameters. Result set is iterated through by the `getNext` method, which moves the iterator to next record and returns its index. Second method (`hasNext`) indicates whether there are any more records in the result set.

We implement all data structures described in Chapter 5 – Ward's original octree (Section 5.1, class `IcOctree`), multiple-reference octree (Section 5.2, class `IcMultirefOctree`), multiple-reference kd-tree (Section 5.3, class `IcMultirefKdTree`), point kd-tree (Section 5.4, class `IcKdTree`), bounding volume hierarchy (Section 5.5, class `IcBvh`) and dual space point kd-tree (Section 5.6, class `IcDualKdTree`). We also implement single dummy data structure that uses brute force search for validation and debugging (class `IcLinearList`).

The irradiance cache itself is straightforward implementation of the algorithm outlined in Section 3.1. We implement both interpolation schemes (Ward, Tabellion and Lamorlette), both rotational and translational gradients, neighbour clamping, record density control and other improvements described in Section 3.1.5. The irradiance interpolation is straightforward – we perform the query and go through its results. For each record reported we calculate its weight, perform the interpolation using gradients and add it the final result. If no record has non-zero weight the interpolation fails and a new record is created.

Creation of a record is handled by the `IcRecordFactory`. It handles the hemisphere sampling and computing of the radiance and all other stored data: distance to the surfaces R_i and rotational and translational gradients. We store single rotational and translational gradient for each record. We also have to store R^{min} and R^{max} values for each record because of the neighbour clamping. Indirect illumination during the hemisphere sampling is computed by another GI solver (secondary solver). After the record is finished it is inserted into the central array of records. Then the neighbour clamping is performed by locating nearby records and testing if the triangle inequality holds. As a last step the record is inserted into the search data structure, using its `addRecord` method. Because the rendering can run in multiple threads and there could be other threads accessing the cache during this modification we synchronize it using the reader-writer lock which allows multiple threads to read, but only single thread to write at a time. We use the Microsoft Windows API implementation (`SRWLOCK`) of the lock.

The incremental mode of cache loading can be used for example to render a walk-through animation, because the irradiance cache is view-independent. The full cache is rendered only for the first frame of the sequence, all other frames use the previously computed cache to save computing time and only create records in areas previously not computed. One problem that has to be addressed is that due to movement of the camera origin we may render a previously rendered surface from much closer distance. The cache from previous render allows interpolation of the entire surface, but it may have insufficient detail because its record spacing limits were computed for much more distant camera. Because of this we recompute the limits for all loaded records during the incremental load and clamp validity radii that are too large. We could do the same in the other way and enlarge validity radii of records that got farther away from the

camera, but this would cause only performance degradation because without any form of record pruning there would be unnecessarily large amounts of records influencing each point.

7.3.2 Radiance caching implementation

Unlike the irradiance caching, there are multiple different implementations of radiance caching. We implement the spatial directional radiance caching (Section 3.2.2) in class `Sdrc`, our unified radiance caching (Chapter 6) in class `UnifiedRc` and also a dummy radiance cache, that uses brute force path tracing instead of any caching and which encapsulates the standard path tracing GI solver, in class `PathTracingRcWrapper`. This last "pseudo-cache" allows us to disable radiance caching and use path tracing for glossy BRDF components instead, without having to treat it as a special case.

The SDRC implementation is done according to the original paper [GKB09]. The spatial search and interpolation implementation is identical to that of irradiance caching. The `Sdrc` class has single template parameter – spatial search data structure – identically to the `IrradianceCache`. Again, if no usable record is found during the spatial interpolation phase we create new one (using the class `SdrcRecordFactory` and its `createRecord` method), perform the neighbour clamping and insert it into the cache.

During record creation we first create small number of rays (30% of final amount) distributed according to the cosine PDF from irradiance caching (Equation 3.3) to compute the distance to surfaces R_i, then we sample the product of surface BRDF and cosine to get directions of other rays. We trace each ray and use the secondary GI solver to compute its radiance. After the sampling we store all colours obtained as radiance samples. We map their associated directions using the paraboloid mapping (Equation 3.10) and build a static kd-tree over them.

When we have non-zero number of usable records, we can perform the interpolation. We sample the BRDF to get directions for which we need to perform the interpolation. We map each one to the paraboloid domain $\mathbf{D}$, calculate the associated directional search radius r_j (Equation 3.11) and search for nearby samples in all records with non-zero spatial weight. If we find at least one sample, we perform the interpolation (Equation 3.12), otherwise we create a new sample by tracing a ray in the direction being interpolated. We store this sample in random nearby record with non-zero weight. This is because creating new spatial record would lead to many small records in the spatial cache, which would cripple the performance of the algorithm. Using closest record would present a risk of samples clumping in single record. Because local trees in records are static (to conserve memory) and do not support insertion we keep a buffer of inserted samples in each record, and when it is large enough, we merge them with the main storage and rebuild the tree. The algorithm is calibrated for sample insertions to happen only rarely, so this is not a problem.

Our implementation has several differences from the original paper. We do not pack radiance samples using the RGBE colour format and position discretization and we do not implement the *cache record density control heuristic* (a heuristic for automatic clamping the R_i based on a radiance rate of change estimate). We also change the maximum directional distance formula for interpolation by using a square root of number of samples to keep the number of samples needed linearly dependent on number of BRDF samples M, as described in Section 6.2:

$$r_j = \min\left(r_{max},\ \frac{1}{2\pi}\frac{1}{\sqrt{Mp(\vec{\omega}_j)}}\right).$$

Implementation of the *unified radiance cache* is much more straightforward. It simply keeps two arrays of radiance samples and two acceleration data structures build over them. Its *interpolate* method simply computes the spatial and directional sensitivity factors α and β, samples the surface BRDF and for each sample performs the interpolation. First a k-NN search in either north or south hemisphere is performed, then the samples found are interpolated. Each sample holds only its colour, position, direction and ray length. During the interpolation we calculate the weighted variance of sample ray lengths, then divide it by the mean value and if a user-set threshold is exceeded we stop the interpolation, create new sample via ray tracing, store it and add it to north, south, or both trees, based on the direction. We do the same if no close radiance samples were found.

The search data structure in unified radiance cache can be exchanged by the same template mechanism as in the irradiance cache, but we have implemented only one – `UnifiedRcKdTree`, a 5D dynamic point kd-tree. Its interface is very similar to that of irradiance caching and SDRC data structures – it is initialized by its `init` method, rebuilt with `build` method and new samples are added using the `add` method. Search is in this case done by a single method, `getClosest`, instead of using an iterator. Because we perform k-NN queries, we are able to safely preallocate space for results and thus we are able to simplify the interface to the single method that writes results into given array.

Precomputation is done simply by attempting to interpolate the illumination at points chosen by the hierarchical refinement technique and creating new samples in the cache where needed.

Chapter 8

Results

To objectively assess the quality of our implemented data structures and algorithms we have to perform extensive testing during which we will compare the performance of different data structures in wide variety of scenes from everyday 3D graphics production. It is important that we use correct testing methodology to be sure that the results are not biased or influenced by any random factors. We have the total of 10 test scenes: 5 scenes with mostly diffuse materials for testing irradiance caching data structures, with one being animated, and 5 scenes with heavy use of specular materials for testing radiance caching.

8.1 Testing methodology

We perform all tests by rendering our test scenes with different settings and measuring certain important statistics. We use the standalone version of Corona (*CoronaStandalone*) (Section 7.1.3) for all tests. It is because the 3ds Max-integrated version depends on closed-source unpredictable API that could potentially skew the results (for example by excessive time spent inside API calls). The standalone version can also be run in a scripted sequence using the Windows batch files. This is essential, given the amount of tests we perform.

Computer	Lenovo Thinkpad X201i	
CPU	Model	Intel Core i3 370M
	Frequency	2,4 GHz
	Core	Arrandale
	L1 cache	2×64 kB
	L2 cache	2×256 kB
	L3 cache	3 MB
RAM	4GB, 533 MHz	

Table 8.1: *Configuration of the PC used for experiments.*

All testing is done on the computer with configuration listed in Table 8.1. All tests are scripted to run sequentially using a batch file. Corona is the only computationally intensive application running during testing and all operating system power saving features are disabled. This is done to minimize any external factors that could influence the rendering performance. No user input is performed during testing and we run all tests multiple times and average the measured data to further make sure that we have correct results.

Operating system	Microsoft Windows 7 Professional 64bit	
C++ compiler	Microsoft Visual C++ 2010 x64	
Important compiler switches	`/Ox`	Full optimization
	`/Ob2`	Inline any suitable function
	`/Ot`	Favour fast code over small
	`/MD`	Multi-threaded dynamic runtime library
	`/fp:fast`	Fast floating point model

Table 8.2: *Compiler configuration for testing.*

The Corona executable used for testing is built using the 64-bit version of Microsoft Visual C++ compiler. Its configuration is shown in Table 8.2. Even though Corona allows it we do not use the multi-threaded rendering for our tests. This has several reasons: it allows us to easily gather exact statistics such as number of data structures iteration steps without the need for synchronization and it removes the inherent randomness of threads. This decreases the variance in performance across multiple runs and allows us to get completely identical pictures from multiple runs by setting the random number generator to the same state before each rendering.

We perform all tests with the BVH built using the full surface area heuristic (Section 7.2.4) as ray casting acceleration data structure. Secondary GI algorithm is the photon mapping where possible with 10^6 photons emitted and density estimation using 50 nearest neighbours. When photon mapping cannot be used (scenes with environment lighting) we use Monte Carlo path tracing instead. GUI frame buffer is disabled for the testing, `SimpleFrameBuffer` is used instead.

8.2 Irradiance caching tests

We test all irradiance caching data structures by rendering the same set of scenes with them and comparing results. Because we are interested only in irradiance cache data structure search performance we configure the rendering so that it takes the majority of time. This means that we reduce both the number of shadow rays and the number of secondary rays for glossy surfaces to the minimal possible value (1). We also use pre-computed irradiance caches for all tests. For each scene we have one precomputed cache (list of records) that is loaded before rendering. This is for two reasons: to save time on redundant computations and to improve comparability of results by performing the tests with exactly the same cache, only with different data structure built over it. We do not allow addition of records during rendering to further lower the amount of time spent outside of the irradiance cache searching and to simplify statistics processing. This restriction is not a problem in most tests. All caches are precomputed with sufficient detail and because of that only few hit points cannot be interpolated.

Irradiance caches for all scenes are precomputed using the hierarchical refinement with highest resolution of 16 samples per pixel. We use Ward's original interpolation schema (Equation 3.5) with $a = 0.3$ for precomputation and $a = 0.6$ for final image rendering. We use the screen-space record density limits with (Section 3.1.5) with $R^{min} = 1$ and $R^{max} = 20$ pixels. All images are rendered in 800×600 pixels resolution using 4 samples per pixel. Non-diffuse portions of BRDFs are sampled using the Monte Carlo path tracing with single path generated.

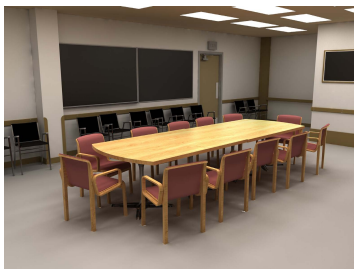
All measured interpolated performance data are calculated only from the time spent inside the rendering loop measured by Corona, not from the run time of the application as whole. Because of that scene

loading, initialization, precomputation and clean-up times are not included in measured times. We prefer to express the rendering performance in samples per second instead of render time. This allows us to easily compare results for images with different resolution or anti-aliasing settings.

8.2.1 Testing scenes

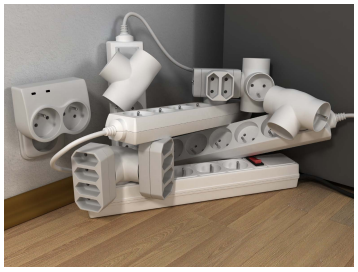
We test the irradiance caching data structures on 5 scenes shown in Figure 8.1. They are all geometrically complex to trigger creation of large number of records. All scenes are interiors with the exception of the Power sockets (Figure 8.1d), which is a still life. They are all lit by area lights and use photon mapping as the secondary GI method (with the exception of Power sockets that uses diffuse environment lighting together with area lights and therefore must use path tracing as the secondary GI method). The vast majority of surfaces have only diffuse materials to make irradiance caching the key factor in rendering performance. Important statistics for all scenes are shown in Table 8.3.

(a) *Diffuse interior.*



(b) *Conference.*

(c) *Sibenik cathedral.*



(d) *Power sockets.*

(e) *Sponza atrium.*

Figure 8.1: *All scenes used for irradiance caching tests. The Diffuse interior model is courtesy of Jiří "Biolit" Friml. Conference is a standard free model, Sibenik cathedral and Sponza atrium scenes are courtesy of Marko Dabrovic. The Power sockets model is our own. The images in higher resolution are provided in Appendix B. Only a single frame from the walk-through animation in Sponza atrium is shown.*

The last scene, "Sponza atrium" (Figure 8.1e) features an animated walk-through. We animate the camera to move in the arcades in a straight line looking at a stationary target. The animation has 30 frames and uses single precomputed irradiance map created by rendering the sequence while incrementally saving the irradiance cache. Because the standalone Corona has no support for animations, we use separate configuration files for each frame (generated with a script) containing interpolated camera data.

	 Diffuse interior	 Conference	 Sibenik cathedral	 Power sockets	 Sponza atrium
Triangle count	165 403	190 841	78 414	1 369 636	66 598
Primary rays	1 920 000	1 920 000	1 920 000	1 920 000	1 920 000
Shadow rays	1 554 584	1 556 926	1 330 249	1 796 358	991 782
Glossy (path tracing) rays	681 207	403 019	0	7 845	0
Irradiance cache interpolations	1 920 000	1 701 636	1 913 597	1 912 908	1 920 000
Irradiance cache records	36 095	17 595	24 970	38 111	71 949
Avg. IC records used	7.1	9.0	4.5	8.2	8.5

Table 8.3: *Various statistics for our irradiance caching testing scenes. Primary rays are the rays shot from the camera. Each primary ray that hits the scene can trigger shadow ray casting if the surface hit is facing a light, an irradiance cache interpolation if the surface is at least partially diffuse, and casting a path tracing GI ray if the surface is partially specular. Avg. IC records used is the average number of records with non-zero weight in single interpolation (when using acceleration data structure that returns all record balls overlapping the query). Values for the Sponza atrium scene are averaged over the 30 frames we render.*

8.2.2 Kd-trees leaf sizes

First we test how the search performance depends on the size of data structure leaves. We have to do this test only for point kd-tree and dual space kd-tree, because leaf sizes for octree and multiple-reference octree are determined by record distribution and stopping criteria. Similarly in multiple-reference kd-tree and BVH they are determined individually for each node by the heuristics.

For both kd-tree and dual space kd-tree we measure the performance in the Conference scene for different leaf sizes (15, 20, 25, 30, 35, and 40 for the kd-tree and 20, 30, 40, 50, 60, and 70 for the dual space kd-tree). We run each test 5-times and average the results. The point kd-tree is set to report 15 nearest neighbours.

Leaf size	15	20	25	30	35	40
Tree nodes	4 823	3 713	3 023	2 639	2 281	2 005
Data structure build time [ms]	12.8	13.0	9.8	10.0	10.2	10.0
Nodes visited per query	34.2	30.9	29.0	27.8	26.5	25.4
Leaves visited per query	8.6	7.4	6.8	6.4	6.0	5.7
Records visited per query	44.1	52.1	57.9	62.3	70.1	78.5
Performance [samples/s]	108 407	109 650	109 842	109 827	109 387	108 654

Table 8.4: *Key point kd-tree statistics for varying leaf size. Visited node is any node into which the search algorithm descends (leaf or inner). Visited record is any record for which the search algorithm computes distance from the query point.*

Results for the point kd-tree are in Table 8.4 and for dual space kd-tree in Table 8.5. We can see that increasing the leaf size decreases the build time and the number of visited nodes and leaves, but increases the number of visited records. Differences in performance are for both data structures only small in the selected ranges of node sizes. We conclude that the leaf size does not have particularly significant effect and choose near-middle values of 25 for point kd-tree and 40 for dual space kd-tree for next tests.

Leaf size	20	30	40	50	60	70
Tree nodes	5 505	3 783	2 717	2 117	1 735	1 491
Data structure build time [ms]	17.2	18.2	14.2	14.8	15	11.6
Nodes visited per query	698.6	561.9	459.4	385.3	325.1	290.8
Leaves visited per query	288.4	232.9	191.5	160.7	135.4	121.0
Records visited per query	1 624.9	1 888.2	2 181.9	2 435.0	2 692.1	2 863.7
Performance [samples/s]	31 879.7	33 437.6	34 270.6	34 450.9	34 336	33 998.2

Table 8.5: *The key statistics for dual space kd-trees with varying leaf sizes. Definitions of visited node and record are analogical to the kd-tree – visited node is any node into which the search algorithm descends, visited record is any record for which the half-space test is performed.*

8.2.3 Multiple-reference kd-tree and BVH build heuristics

Next we test the effect of tree build heuristics we have developed for multiple-reference kd-tree and BVH by directly comparing data structure quality and search performance of trees built with and without them[1]. We again use only the Conference scene with each test repeated 5-times. If we disable the heuristic, we have to manually specify leaf size for the BVH. We use value of 5 as it gives best results in on our preliminary testing.

	Heuristic	
	on	off
Data structure build time [ms]	788.2	503.6
Tree nodes	152 229	166 445
Total references	515 656	650 473
Tree size [MB]	5.7	6.6
Nodes visited per query	18.5	19.7
Records visited per query	16.7	19.9
Performance [samples/s]	135 210	133 578

(a) *Effect of heuristic on multiple-reference kd-tree performance.*

	Heuristic	
	on	off
Data structure build time [ms]	125.8	41.2
Tree nodes	11 063	10 613
Leaves visited per query	8.9	10.6
Nodes visited per query	38.0	54.9
Records visited per query	29.1	33.3
Performance [samples/s]	122 642	115 611

(b) *Effect of heuristic on BVH performance.*

Table 8.6: *This table shows results for both multiple-reference kd-tree and BVH heuristic compared to the basic version of each data structure. Total references for the kd-tree is the number of all record references duplicately stored in the tree. The tree size is an estimate (lower bound) of kd-tree size in the memory computed by summing sizes of all nodes and references.*

Results for the tests are in Table 8.6. In both cases the heuristic slightly improves all monitored statistics: number of nodes and records visited per query, and in the case of multiple-reference kd-tree the number of tree nodes and duplicate references. Because of that the heuristic reduces the kd-tree memory footprint. It is however surprising how little is the traversal performance influenced by these improvements. The benefits are also not free – data structure build times are greatly increased when using heuristics.

[1]Multiple-reference kd-tree without heuristic is built by recursively splitting each node in its largest axis; BVH without heuristic is built by recursively sorting records with respect to node largest axis and splitting the list in the middle.

8.2.4 Nearest neighbour count for the point kd-tree

The point kd-tree is unique in that it does not return all balls intersecting the query, but only a fixed number of balls with closest centres (k-nearest neighbour). Lowering the k increases the rendering performance (not only because the search is faster, but also fewer records have to be subsequently processed in the interpolation), but it may lead to artefacts from missed contributions of records not included in the query result.

We search for the optimal k by first rendering the Diffuse interior scene with varying k to measure how is the performance dependent on it. We choose this scene because it is illuminated mostly by the indirect lighting and so any differences in irradiance caching quality in it are more prominent than in other scenes. The scene is again rendered 5-times for each configuration and results are averaged. The kd-tree is built using maximum leaf size of 25 (Section 8.2.2).

k-NN	1	5	10	15	20	30	40	50
Nodes visited per query	16.8	23.3	28.3	32.4	35.8	41.6	46.9	51.7
Leaves visited per query	2.1	4.3	6.2	7.8	9.2	11.5	13.7	15.7
Records visited per query	19.1	35.7	50.4	63.1	74.6	95.1	113.7	131.3
Performance [samples/s]	107 912	100 421	92 273	86 093	80 746	72 416	65 217	59 655
Avg. IC records used	1.0	4.1	5.7	6.1	6.4	6.7	6.8	6.9

Table 8.7: *Illustration of point kd-tree search performance in relation to the number of nearest neighbours we search for. "Avg. IC records used" is the average number of records reported by the tree that are really used for the interpolation. Any data structure that reports all records influencing a point would have this statistic equal to 7.1 for this scene.*

Results of our tests are shown in Table 8.7. It is not surprising that increasing the k significantly increases the number of nodes and records we have to visit to get the result. The traversal performance decreases accordingly. The number of records used for interpolation would be equal to 7.1 for any data structure that gives exact results; it is lower for the point kd-tree because it gives only approximate results. It is interesting that we are unable to reach 7.1 even when reporting 50 nearest neighbours.

(a) *1-NN* **(b)** *5-NN* **(c)** *10-NN* **(d)** *15-NN* **(e)** *20-NN*

Figure 8.2: *Enlarged detail of the left wall in the Diffuse interior scene rendered using point kd-tree with different k (number of nearest neighbours reported). Artifacts are visible when rendering with low k. In the extreme case of k = 1 (interpolation using nearest neighbour) we can easily see the structure of Voronoi diagram.*

We evaluate the impact of k by rendering the scene in higher resolution and overall quality. When rendering with low k there are several areas where artefacts become visible, most prominently the left and back walls. We show higher quality images of the left wall with different k in Figure 8.2. In case of 1-NN and 5-NN artefacts are clearly visible. The 10-NN interpolation is nearly identical to reference, but because of safety margin we decide to use 15-NN as default setting.

8.2.5 Overall data structures comparison

Now that we have fine-tuned all irradiance caching data structures we perform the most important test – we render all scenes using all data structures multiple times for direct comparison of their performances. First 4 scenes we render once again 5-times per structure. The fifth scene is the animation (Sponza atrium). We render the whole animation (30 frames) only once and average results from all its frames.

The results are shown in table 8.8. As expected, the single-reference (Ward's) octree does not perform well. Although it is built fast and it is very memory inexpensive, its render performance is bad compared to its multiple-reference variant, especially in scenes with many cache entries (Power sockets, Sponza atrium). This is because both number of nodes and records visited is very high – because of the way the data structure works (Section 5.1) many paths from root to leaves have to be traversed for each query.

As expected, both multiple-reference data structures (octree and kd-tree) offer better query performance than any other data structure. Because of the object duplication only single path from the root to a leaf have to be traversed. As a result the number of visited nodes per query is much lower than for other data structures, especially in the case of multiple-reference octree because of its higher branching factor. We need 3 kd-tree nodes to split the space in the same way as 1 octree node does. Because of that the number of kd-tree nodes visited is indeed approximately 3-times larger. This however does not mean that multiple-reference kd-tree is slower, because its each traversal step is faster.

Biggest disadvantages of both data structures are their high build times and memory consumption. They can store millions of references even for our relatively small caches with only tens of thousands of records. Because of that they take up much more memory that the cache itself. For example in the Sponza atrium the cache size is approximately 5,5 MB (with 72 000 records and 76 bytes per record), but the multiple-reference octree presents another 21,7 MB and multiple-reference kd-tree 38 MB of memory allocated. Size of these data structures can be decreased by limiting their depth or changing the record insertion propagation criterion, but this negatively affects the performance.

There is one interesting anomaly in multiple-reference octree statistics in the Power sockets scene. The data structure creates unusually small number of references and because of that visits many records unnecessarily, which causes its traversal performance to drop. We assume it is because of the overall scene design. Only relatively small portion of it is shown in the image (Figure 8.3). Because of that if we build the octree over entire scene we end up with many nodes in higher levels empty and we reach the maximum depth before the records are sufficiently separated into different nodes. There is of course the simple solution of building the octree over the bounding box of records, not the scene. The problem is however that this bounding box is not known in advance.

If we directly compare the multiple-reference octree and multiple-reference kd-tree it is hard to determine which is better. The kd-tree is more adaptive and does not suffer the problem of octree in the Power sockets

		 Diffuse interior	 Conference	 Sibenik cathedral	 Power sockets	 Sponza atrium
Total references	Multi-ref octree	1 285 800	883 010	894 726	205 816	3 317 090
Tree size [MB]		8.3	6.3	6.2	1.0	21.8
Leaves visited/query	Kd-tree	7.8	6.8	7.7	8.3	7.3
Avg. IC records used		6.1	7.3	4.1	7.2	7.2
Ref. IC records used		7.1	9.0	4.5	8.2	8.5
Total references	Multi-ref kd-tree	1 253 350	515 656	819 745	3 379 320	3 768 360
Tree size [MB]		13.6	5.7	8.9	23.0	38.0
Data structure nodes	Octree	10 217	5 425	7 913	8 985	20 713
	Multi-ref octree	112 401	99 129	92 097	7 721	303 617
	Kd-tree	5 987	3 023	4 363	6 525	12 025
	Multi-ref kd-tree	356 377	152 229	233 591	394 699	954 485
	BVH	19 827	11 063	16 407	21 797	41 423
	Dual kd-tree	6 043	2 717	4 069	6 119	11 959
Nodes visited per query	Octree	75.3	77.7	81.1	90.2	104.8
	Multi-ref octree	7.5	7.4	7.3	7.9	8.5
	Kd-tree	32.5	29.0	31.5	35.4	35.8
	Multi-ref kd-tree	19.7	18.5	19.9	19.6	22.8
	BVH	33.1	38.0	35.1	40.0	46.5
	Dual kd-tree	715.4	459.4	640.3	3 138.2	1 672.3
Records visited per query	Octree	256.8	251.8	182.0	341.6	333.7
	Multi-ref octree	22.4	20.7	21.1	87.9	29.0
	Kd-tree	63.2	58.0	56.0	60.1	64.3
	Multi-ref kd-tree	17.2	16.7	16.3	24.9	20.5
	BVH	28.5	29.1	25.3	39.2	36.4
	Dual kd-tree	2 977.9	2 182.0	2 559.0	14 727.9	6 287.8
Data structure build time [ms]	Octree	31.2	13.0	22.4	40.4	74.8
	Multi-ref octree	312.6	229.2	244.0	65.4	894.7
	Kd-tree	21.4	9.4	15.8	38.8	64.4
	Multi-ref kd-tree	1 834.6	780.2	1 238.4	2 873.4	5 520.9
	BVH	295.4	126.6	207.6	334.2	675.9
	Dual kd-tree	27.4	13.8	21.2	46.0	86.8
Performance [samples/s]	Octree	40 903	48 512	51 164	41 763	35 769
	Multi-ref octree	107 519	134 863	179 370	150 376	181 147
	Kd-tree	86 331	110 373	130 863	144 956	135 342
	Multi-ref kd-tree	108 005	136 096	181 225	202 767	186 613
	BVH	99 243	121 842	159 794	167 343	153 661
	Dual kd-tree	21 235	34 509	26 588	4 800	10 314

Table 8.8: *Comparison of the best variants of all implemented irradiance caching data structures. Various data structure statistics are given at the beginning of the table. "Total references" in context of multiple-reference data structures means the total number of duplicately stored record references. The tree size is a lower bound estimate of data structure size in RAM computed by adding number of nodes multiplied by the size of a node and number of references multiplied by the size of a reference (integer index of a node). The "Avg. IC records used" statistic of the kd-tree is explained in Section 8.2.4. Reference values for data structures performing exact searches are given for comparison ("Ref. IC records used").*

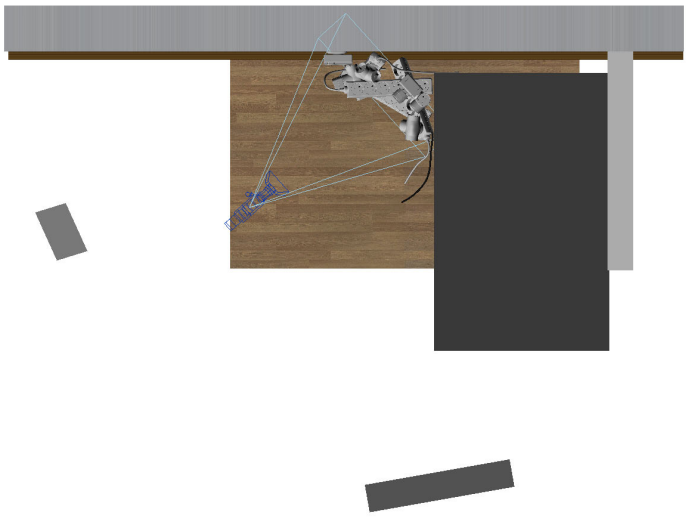


Figure 8.3: *A top-down view of the Power sockets scene, with camera position and angle of view visible. The two isolated rectangles on left and bottom are area ligts.*

scene. It stores generally lower or the same number of references as the octree, but because of its higher node count it takes more memory. Its traversal performance is only slightly better than that of octree, but its build times are significantly higher because of the heuristic (although they could be easily lowered using the binning technique [Wal07]).

When considering results for the point kd-tree we have to keep in mind that it is the only data structure that does not return exact solution, even though this is not a problem in praxis. It has lowest build times from all data structures, but its traversal performance is worse than that of BVH and multiple-reference data structures. This is because the k-NN search procedure is rather complicated.

Results of BVH are a pleasant surprise for us. It has the best performance except for the memory-intensive multiple-reference data structures. Its build times are relatively high because of the heuristic, but they could be easily lowered by presorting [WH06].

The dual space kd-tree performs consistently worst of all tested data structures. The half-space range search in higher dimension is extremely computationally demanding. Hundreds of nodes and thousands of records have to be visited to get the result.

8.3 Radiance caching tests

The radiance caching tests are very different from irradiance caching tests because we have implemented two very different algorithms that cannot be compared directly in terms of rendering time. Because they are both biased and produce specific low-frequency artefacts we also cannot compare them using root mean square error (RMSE) of rendered images. We instead configure them to run approximately the same amount of time and then visually compare the results. We also include the “brute force” approach (path tracing glossy parts of BRDFs) as a third method into these tests.

We use the irradiance caching for diffuse parts of all BRDFs. We again precompute the cache for each scene and then load it for each rendering. To eliminate the randomness and ensure reproducibility of our results we again disable multi-threaded rendering. Other irradiance caching settings are identical to previous tests (Section 8.2). Both irradiance caching and SDRC use multiple-reference octree as the spatial search data structure in this set of tests. We employ a precomputation phase for both unified directional caching and SDRC using the hierarchical refinement that pre-renders the image[2] four times with increasing resolution, identically to irradiance caching. Resolution of the last precomputation pass is identical to that of the final image. We use 5 shadow rays to explicitly sample direct lights for each primary ray in scenes with area lights.

SDRC caching uses the same configuration for its spatial phase as irradiance caching (Section 8.2). Its number of radiance samples per record is set to ten times of the number of interpolation samples. The unified radiance caching is configured to use the distance metric described in Section 6.2 with maximum permitted spatial distance of 5 projected pixels and directional sensitivity coefficient $k = 15$. The interpolation is performed using 10 nearest neighbours. We set the ray lengths variance threshold that triggers new sample insertion (Section 6.5) to 100. This forces the algorithm to generate additional samples in areas where the illumination changes rapidly without too many unnecessary samples generated, as shown in Figure 8.4.

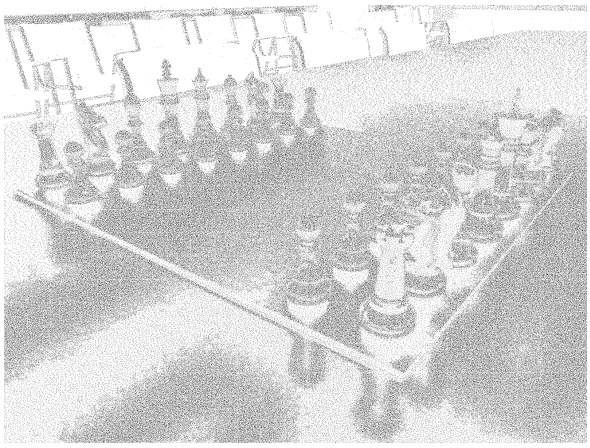


Figure 8.4: *Visualization of unified radiance caching samples density showing the effect of ray lengths heuristic. Each dot represents place where a radiance sample was created; dark areas have greater concentration of samples. Spikes in record concentration near the mirror reflections of chess pieces are clearly visible.*

8.3.1 Testing scenes

We test using 5 different scenes with mostly specular materials, shown in Figure 8.5. Scenes Computer case and Glossy interior are lit completely with area lights, scene Components plate is lit by a combination of area lights and diffuse environment. Scenes Stanford dragon and Glossy interior are lit entirely by environment lighting. The last two scenes are very similar; they feature complex models with specular materials sitting atop the table inside the Conference scene. Because the appearance of these objects was

[2]More specifically it only casts primary rays and tries to interpolate the indirect illumination.

too dependent on the direct lighting from lights at the ceiling we have disabled these lights and instead removed ceiling and geometry in windows to light the scene using environment lighting.

The scenes used vary greatly in geometry complexity (Glossy interior and Stanford dragon scenes and simpler; Computer case and Components plate are very complicated) and percentage of specular surfaces (with Components plate scene being only partially specular, and Chessboard and Stanford dragon scenes being almost entirely specular).

(a) *Components plate.*

(b) *Glossy interior.*

(e) *Computer case*

(c) *Chessboard.*

(d) *Stanford dragon.*

Figure 8.5: *All scenes used for irradiance caching tests. Computer case and Components plate are our own scenes. Stanford dragon model is a free model retrieved from the Stanford 3D Scanning Repository [Sta10]. The Glossy interior scene was kindly provided by Ludvík "Rawalanche" Koutný. Chessboard features a free chess pieces model by cjx3711 [cjx10]. Models in Stanford dragon and Chessboard scenes are place atop a table in the Conference standard scene. The images rendered in in higher resolution are provided in Appendix B.*

8.3.2 Radiance caching results

Important scene statistics and results of SDRC, unified radiance caching, and brute force path tracing are shown in Table 8.9. We have tried to configure each algorithm to run approximately the same time (in terms of the sum of rendering time and irradiance/radiance caching precomputation time). Note that, even though the path tracing does not need any precomputation phase, we still run it (without precomputing anything) to keep the testing unbiased. This phase would have to be performed in any real-world scenario because of the irradiance caching.

We can see that, given the same same, the brute force path tracing is able to make the highest number of BRDF samples in most scenes. An exception is the Computer case scene, where the SDRC is faster. We can assume that it is because the scene features very complicated geometry that makes the ray casting very expensive. Second exception is the Glossy interior. This scene glossy surfaces are very simple, which makes both radiance caching algorithms very efficient.

It is difficult to directly compare rendering performance of SDRC and unified radiance caching. Unified radiance caching greatly outperforms SDRC[3] in Stanford dragon and Glossy interior scenes, but is outperformed in the other ones. There is no clear link between distinct scene features and relative performance of any algorithm. It is surprising that SDRC performs well in scenes where it is forced to make many spatial records (Computer case, Chessboard).

Statistics for unified radiance caching show that the overall number of searches is significantly lower than in SDRC when both algorithms use approximately the same number of BRDF samples (scene Components plate). This however does not mean that the unified radiance caching is faster, because it searches in the 5D space. Because of that the number of tree nodes and samples visited by the search algorithm is very high.

By comparing "Total samples in the cache" and "Total searches" we can see how the caching reduces the number of rays that have to be traced. Without the caching one ray would have to be cast for each search. Ratio of this reduction greatly varies, with extremes being scenes Glossy interior and Stanford dragon. The last important unified radiance caching statistic is the number of samples duplicated in the equator belt (Figure 6.2). As expected it is only a small fraction of total number of samples.

8.3.3 Visual comparison of radiance caching algorithms

Although comparison of numerical statistics in the previous section showed some interesting facts, key in this case is comparing the images radiance caching algorithms produce. We do this by comparing renders of all scenes with direct lighting disabled for primary lights. We do not show entire images here because they are for the most part very similar, we instead focus on interesting regions. Figure 8.6 shows details of Computer case and Chessboard scenes. They demonstrate how different algorithms deal with the interpolation of illumination on flat glossy surfaces in complex environments. Path tracing creates only noise. Unified radiance caching generates considerable low-frequency noise created by variance in cached radiance values. SDRC has the best results in these scenes. It performs smooth interpolation with less noise than both unified radiance caching and path tracing.

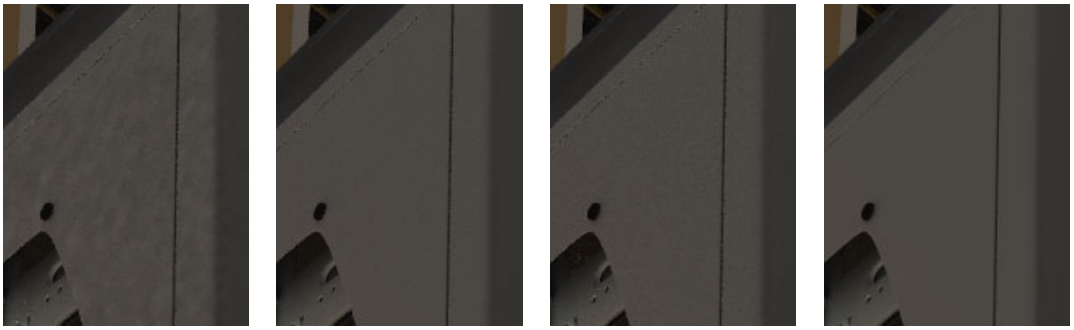
Figure 8.7 shows details of Stanford dragon and Components plate scenes. These feature more complicated geometry. Unified radiance caching again creates the low-frequency noise, but in this case it is not as distracting because it is masked by the geometry (for example on the dragon). SDRC creates again smooth interpolation with less noise than path tracing, but in these scenes it exhibits some problems: in the Components plate scene it creates black artefacts on edges of the copper heat sink leaves. This is because SDRC uses the same spatial interpolation as irradiance caching, that has problems with the same type of geometry (thin layered objects). In the Stanford dragon scene we can clearly see how the SDRC alters material perception by excessive blurring of reflections. This is because the algorithm is not suitable for high-frequency BRDFs, as was already established in the original paper [GKB09].

In most scenes each structure show the same behaviour. Although brute force path tracing is able to use larger amount of BRDF samples than both radiance caching algorithms, it creates largest amount of noise. SDRC generates least amount of noise, but it produces very biased results. Unified radiance caching offers much more accurate interpolation than SDRC (for mirror-like materials and near complicated geometry), but exhibits large amounts of low-frequency noise.

[3] In terms of number of BRDF samples it is able to interpolate in the same time.

		 Computer case	 Components plate	 Stanford dragon	 Glossy interior	 Chessboard
Secondary GI algorithm		PM	PT	PT	PM	PT
Triangle count		1 079 448	1 431 468	288 339	464 970	427 483
Primary rays		2 880 000	1 440 000	1 920 000	1 440 000	1 920 000
Glossy surfaces hits		1 897 702	400 667	1 498 921	737 416	1 697 293
Specular surfaces [%]		65.9	27.8	78.1	51.2	88.4
Irradiance cache interpolations		2 859 931	1 439 216	1 225 083	1 436 449	1 725 829
Irradiance cache records		108 923	23 192	13 479	57 971	41 741
Path tracing	BRDF samples/interpolation	7	32	25	8	17
	Rendering time [s]	239.7	157.8	254.8	104.5	197.0
	Precomputation time [s]	4.9	2.8	1.1	12.1	3.4
SDRC	BRDF samples/interpolation	13	22	12	8	10
	Rendering time [s]	201.5	133.9	217.0	68.6	144.4
	Precomputation time [s]	63.5	44.3	31.4	30.9	36.8
	Radiance samples/record	130	220	120	80	100
	Spatial records	20 491	8 733	10 485	14 527	18 322
	Directional searches	158 381 184	55 805 240	136 646 416	32 493 024	124 909 248
Unified RC	BRDF samples/interpolation	5	20	20	20	7
	Rendering time [s]	242.9	144.3	255.2	74.7	156.9
	Precomputation time [s]	16.5	10.7	24.8	19.1	12.9
	Total samples in the cache	1 293 035	1 842 321	13 139 824	451 875	3 063 202
	Equator samples	79 367	79 898	1 230 189	76 382	312 666
	Total searches	9 487 970	8 008 917	29 979 688	14 748 842	11 878 849
	Tree nodes visited/search	251.8	138.8	72.5	60.5	133.9
	Samples visited/search	553.3	275.9	107.6	114.5	270.9
	Avg. samples reported	5.4	5.8	2.7	3.4	6.5

Table 8.9: *Results of the radiance caching testing in all scenes. "PM" and "PT" in the secondary GI algorithm fields mean photon mapping and path tracing, respectively. "Glossy surfaces hits" is the number of primary rays that triggered radiance cache interpolation because they hit glossy surfaces. "Specular surfaces" is the ratio of primary rays that hit glossy surfaces to all primary rays. "Radiance samples/record" in SDRC is the number of radiance samples we create in each SDRC record. "Directional searches" is the total number of searches in L-trees we make during the directional phase of interpolation. "Equator samples" in the unified radiance caching is the number of records that are stored duplicately in both north and south hemispheres. "Avg. samples reported" is the average number of results for each k-NN query.*

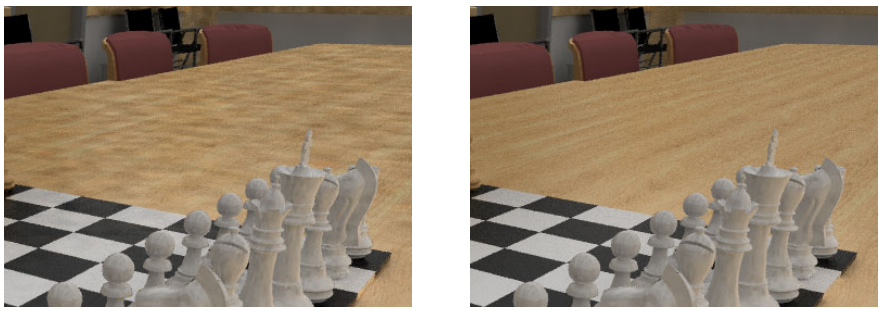


(a) *Computer case detail - unified radiance caching*

(b) *Computer case detail - SDRC*

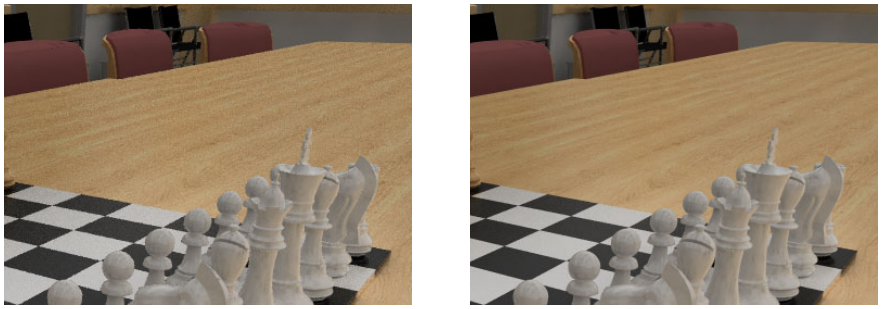
(c) *Computer case detail - path tracing*

(d) *Computer case detail - reference*



(e) *Stanford dragon detail - unified radiance caching*

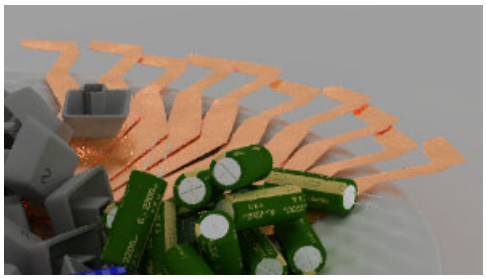
(f) *Stanford dragon detail - SDRC*



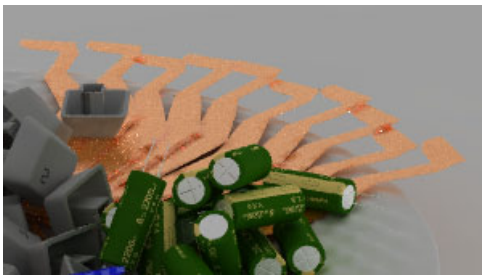
(g) *Stanford dragon detail - path tracing*

(h) *Stanford dragon detail - reference*

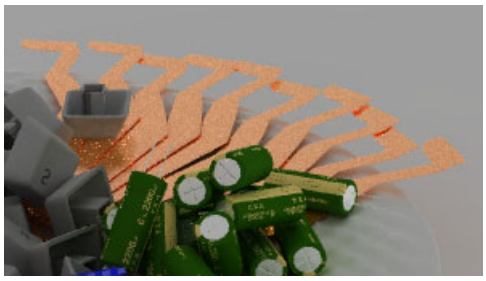
Figure 8.6: *Radiance caching details in Chessboard and Computer case scenes*



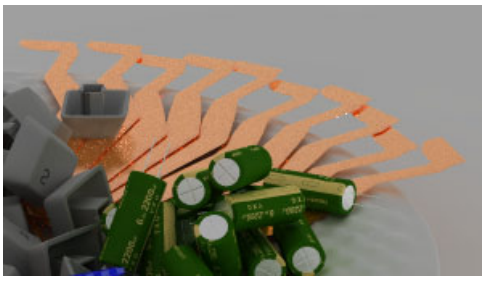
(a) *Components plate detail - unified radiance caching*



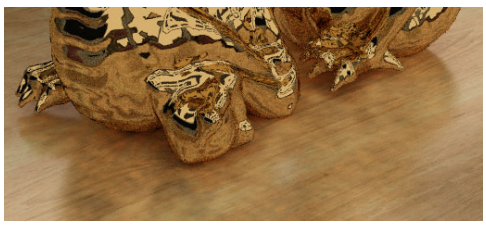
(b) *Components plate detail - SDRC*



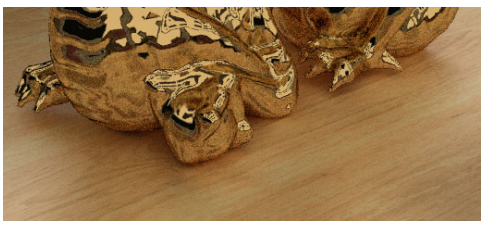
(c) *Components plate detail - path tracing*



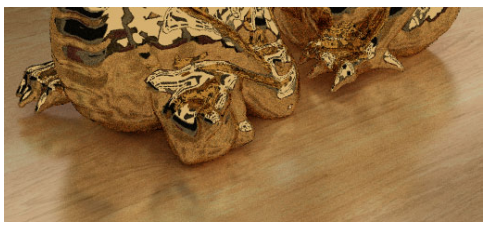
(d) *Components plate detail - reference*



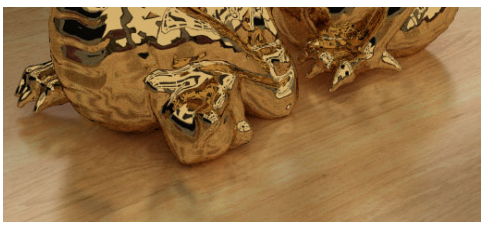
(e) *Stanford dragon detail - unified radiance caching*



(f) *Stanford dragon detail - SDRC*



(g) *Stanford dragon detail - path tracing*



(h) *Stanford dragon detail - reference*

Figure 8.7: *Radiance caching detail in Stanford dragon and Components plate scenes*

Chapter 9

Conclusion

We have analysed the problem of realistic image synthesis and its solution by irradiance and radiance caching. Because the irradiance caching performance is highly dependent on the quality of its record data structure, we have tried to improve it by implementing a total of 6 different data structures, 2 previously used in this context (Ward's original *octree* and the *multiple-reference octree*) and 4 new ones created by adapting existing techniques to the problem of irradiance caching – *point kd-tree*, *multiple-reference kd-tree*, *bounding volume hierarchy*, and kd-tree in 4D space (dual space) created by *object transformations*. We have also developed a heuristics for building multiple-reference kd-trees and bounding volume hierarchies, based on the *surface area heuristic* known from ray tracing.

We have tested all implemented data structures using our set of 5 geometrically complex scenes. Results have shown that, although they are very memory-intensive, the multiple-reference data structures give best traversal performance. Multiple-reference kd-tree have shown slightly better traversal performance, but also much higher build times and memory consumption. Best data structure without object duplication in terms of traversal performance is the bounding volume hierarchy. Point kd-tree is slightly worse and in addition it gives only approximate results, although this have turned out to be no problem. The last two data structures, Ward's octree and dual space kd-tree have significantly worse performance than the rest, with the dual space kd-tree being consistently the worst data structure.

For illumination interpolation on glossy surfaces we have implemented the *spatial directional radiance caching* (SDRC) algorithm. During analysis of this algorithm we have concluded that there is no room for improvements by a change of its data structures alone. Because of that we have created an entirely new radiance caching algorithm – *unified radiance caching*. Then we have compared this algorithm with the SDRC and brute force (path tracing without caching).

Results of these three algorithms on our set of another 5 scenes with varying glossy materials show that when running the same amount of time SDRC gives smooth illumination interpolation with less noise than path tracing, but it is unsuitable for near mirror-like surfaces because it is unable to reproduce sharp reflections. Unified radiance caching also creates images with less high-frequency noise than path tracing, and it is able to reproduce even mirror-like reflections, but it generates very distinctive low-frequency noise.

Future work

There are many things left that could be improved. A heuristic similar to that of multiple-reference kd-tree and BVH could be implemented for the multiple-reference octree. Algorithms for data structures building with heuristics could be definitely made faster by using presorting [WH06] or binning [Wal07] techniques.

The unified radiance caching can also be improved in many ways, partially by removing the low-frequency noise it generates by reducing the variance of stored radiance samples. We could also improve the adaptivity of the algorithm to scene geometry, incident radiance distribution, and surface materials.

Another field of interest are GPU implementations of the data structures and algorithms presented, for example using the CUDA technology [NVI07].

Bibliography

[AE97] Pankaj K. Agarwal and Jeff Erickson. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry*, pages 1–56. American Mathematical Society, 1997.

[Ali] Alias|Wavefront Inc. Object Files (.obj). `http://paulbourke.net/dataformats/obj/`.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[Arg01] Lars Arge. External memory data structures. In *Proceedings of the 9th Annual European Symposium on Algorithms*, ESA '01, pages 1–29, London, UK, 2001. Springer-Verlag.

[Ben75] Jon L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, September 1975.

[BF88] Paul Bratley and Bennett L. Fox. Algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Trans. Math. Softw.*, 14:88–100, March 1988.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19:322–331, May 1990.

[BWY80] Jon L. Bentley, Bruce W. Weide, and Andrew C. Yao. Optimal expected-time algorithms for closest point problems. *ACM Trans. Math. Softw.*, 6:563–580, December 1980.

[cjx10] cjx3711. Glass Chess Set. `http://www.turbosquid.com/FullPreview/Index.cfm/ID/544320`, July 2010.

[Cla94] Kenneth L. Clarkson. An algorithm for approximate closest-point queries. In *Proceedings of the tenth annual symposium on Computational geometry*, SCG '94, pages 160–164, New York, NY, USA, 1994. ACM.

[CPP+05] Eva Cerezo, Frederic Pérez, Xavier Pueyo, Francisco J. Seron, and François X. Sillion. A survey on participating media rendering techniques. *The Visual Computer*, 21(5):303–328, June 2005.

[CS02] Sung-Hyuk Cha and Sargur N. Srihari. A fast nearest neighbor search algorithm by filtration. *Pattern Recognition*, 35(2):515 – 525, 2002.

[CT81] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. In *Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '81, pages 307–316, New York, NY, USA, 1981. ACM.

[DB08] Germund Dahlquist and Åke Björck. *Numerical Methods in Scientific Computing: Volume 1*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2008.

[DBB06] Phil Dutre, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination*. A K Peters, Natick, MA, 2nd edition, 2006.

[DDG00] Matthew Dickerson, Christian A. Duncan, and Michael T. Goodrich. K-D Trees Are Better when Cut on the Longest Side. In *Proceedings of the 8th Annual European Symposium on Algorithms*, ESA '00, pages 179–190, London, UK, 2000. Springer-Verlag.

[Dir82] Paul Adrien Maurice Dirac. *The Principles of Quantum Mechanics (International Series of Monographs on Physics)*. Oxford University Press, USA, 1982.

[DR05] Julie Dorsey and Holly Rushmeier. Digital modeling of the appearance of materials. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[Dut03] Philip Dutré. Global Illumination Compendium. `http://www.cs.kuleuven.ac.be/ phil/GI/`, September 2003.

[Eri05] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology)*. Morgan Kaufmann, January 2005.

[FB74] Raphael A. Finkel and Jon L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, March 1974.

[FBF77] Jerome H. Friedman, Jon L. Bentley, and Raphael A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, September 1977.

[FH09] Jiří Filip and Michal Haindl. Bidirectional Texture Function Modeling: A State of the Art Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31:1921–1940, 2009.

[FPSG96] James A. Ferwerda, Sumanta N. Pattanaik, Peter Shirley, and Donald P. Greenberg. A model of visual adaptation for realistic image synthesis. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 249–258, New York, NY, USA, 1996. ACM.

[Gam11] Galaxy Gameworks. Simple DirectMedia Layer. `http://www.libsdl.org/`, 1998–2011.

[GG98] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30:170–231, June 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[GKB09] Václav Gassenbauer, Jaroslav Křivánek, and Kadi Bouatouch. Spatial Directional Radiance Caching. *Computer Graphics Forum*, 28(4):1189–1198, 2009.

[Gre03] Robin Green. Spherical Harmonic Lighting: The Gritty Details. *Archives of the Game Developers Conference*, March 2003.

[GS87] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl.*, 7:14–20, May 1987.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57, June 1984.

[Hal64] John H. Halton. Algorithm 247: Radical-inverse quasi-random point sequence. *Commun. ACM*, 7:701–702, December 1964.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HKH11] Michal Hapala, Ondřej Karlík, and Vlastimil Havran. When It Makes Sense to Use Uniform Grids for Ray Tracing. In *Proceedings of WSCG'2011, communication papers*, pages 193–200, Feb 2011.

[HR07] André Hinkenjann and Thorsten Roth. Phase space rendering. In *Advances in Visual Computing*, volume 4842 of *Lecture Notes in Computer Science*, pages 691–700. Springer Berlin / Heidelberg, 2007.

[HS98] Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '98, pages 39–ff., New York, NY, USA, 1998. ACM.

[Inc09] Autodesk Inc. *3ds Max SDK Programmer's Guide*. 111 Mclnnis Parkway, San Rafael, CA 94903, USA, 2009.

[Inc11] Autodesk Inc. 3ds Max. `http://www.autodesk.com/3dsmax`, 2011.

[ISP07] Thiago Ize, Peter Shirley, and Steven Parker. Grid creation strategies for efficient ray tracing. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 27–32, Washington, DC, USA, 2007. IEEE Computer Society.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[JMLH01] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 511–518, New York, NY, USA, 2001. ACM.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[Kar09] Ondřej Karlík. Utilization of generic programming in ray tracing, 2009. Bachelor thesis, Czech Technical University in Prague, Faculty of electrical Engineering, Supervisor: Tomáš Davidovič.

[KBPv08] Jaroslav Křivánek, Kadi Bouatouch, Sumanta Pattanaik, and Jiří Žára. Making radiance and irradiance caching practical: adaptive caching and neighbor clamping. In *ACM SIGGRAPH 2008 classes*, SIGGRAPH '08, pages 77:1–77:12, New York, NY, USA, 2008. ACM.

[KG09] Jaroslav Křivánek and Pascal Gautron. Practical Global Illumination with Irradiance Caching. *Synthesis Lectures on Computer Graphics and Animation*, 4(1):1–148, 2009.

[KGPB05] Jaroslav Křivánek, Pascal Gautron, Sumanta Pattanaik, and Kadi Bouatouch. Radiance caching for efficient global illumination computation. *IEEE Transactions on Visualization and Computer Graphics*, 11:550–561, September 2005.

[Knu98] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[Lan07] David Lanier. *3D Studio Max SDK*, 2007.

[LFTG97] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. Non-linear approximation of reflectance functions. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 117–126, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[LW94] Eric P. F. Lafortune and Yves D. Willems. Using the Modified Phong brdf for Physically Based Rendering. Technical Report CW197, Department of Computer Science, K.U.Leuven, 1994.

[Mac06] Paul Macklin. EasyBMP. `http://easybmp.sourceforge.net/`, 2005–2006.

[Mat94] Jiří Matoušek. Geometric range searching. *ACM Comput. Surv.*, 26:422–461, December 1994.

[MB90] David J. MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6:153–166, May 1990.

[Met87] Nicholas Metropolis. The beginning of the Monte Carlo method. *Los Alamos Sci.*, Special Issue, 1987.

[Mic10] Microsoft Corporation. *MSDN: Processes and Threads (Windows)*, December 2010. `http://msdn.microsoft.com/en-us/library/ms684841.aspx`.

[MM99] Songrit Maneewongvatana and David M. Mount. Analysis of approximate nearest neighbor searching with clustered point sets. *Proc. Workshop Algorithm Eng. and Experiments (ALENEX '99)*, January 1999.

[MNPT] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Yannis Theodoridis. R-Trees Have Grown Everywhere.

[Moo90] Andrew W. Moore. *Efficient Memory-based Learning for Robot Control*. PhD thesis, University of Cambridge, Cambridge, UK, November 1990.

[MS67] Thomas M. MacRobert and Ian N. Sneddon. *Spherical harmonics; an elementary treatise on harmonic functions, with applications*, volume 98 of *International series of monographs in pure and applied mathematics*. Pergamon Press, New York, NY, USA, third edition, 1967.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2:21–28, October 1997.

[NRH+77] Fred E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometrical considerations and nomenclature for reflectance. *Washington DC*, 160(October):1–52, 1977.

[NVI07] NVIDIA Corporation. The CUDA homepage. `http://www.nvidia.com/object/cuda_home.html`, 2007.

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann, 2 edition, July 2010.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18:311–317, June 1975.

[RRT95] Diane Ramey, Linda Rose, and Lisa Tyerman. MTL material format (Lightwave, OBJ). `http://paulbourke.net/dataformats/mtl/`, October 1995.

[SC97] Peter Shirley and Kenneth Chiu. A low distortion map between disk and square. *journal of graphics, gpu, and game tools*, 2(3):45–52, 1997.

[SHAP01] Neal Sample, Matthew Haines, Mark Arnold, and Timothy Purcell. Optimizing Search Strategies in k-d Trees. In *5th WSES/IEEE World Multiconference on Circuits, Systems, Communications & Computers (CSCC 2001)*, July 2001.

[She68] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference*, ACM '68, pages 517–524, New York, NY, USA, 1968. ACM.

[SK88] Bernhard Seeger and Hans-Peter Kriegel. Techniques for design and implementation of efficient spatial access methods. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, pages 360–371, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[SM03] Peter Shirley and R. Keith Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2 edition, 2003.

[SRF87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases*, VLDB '87, pages 507–518, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.

[SSK08] Alexei Soupikov, Maxim Shevtsov, and Alexander Kapustin. Improving Kd-tree Quality at a Reasonable Construction Cost. In *Symposium on Interactive Ray Tracing*. IEEE, 2008.

[Sta10] Stanford University Computer Graphics Laboratory. The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/`, 2010.

[SWZ96] Peter Shirley, Changyaw Wang, and Kurt Zimmerman. Monte Carlo techniques for direct lighting calculations. *ACM Trans. Graph.*, 15:1–36, January 1996.

[TL04] Eric Tabellion and Arnauld Lamorlette. An approximate global illumination system for computer generated films. *ACM Trans. Graph.*, 23:469–476, August 2004.

[TS67] Kenneth E. Torrance and Ephraim M. Sparrow. Theory for off-specular reflection from roughened surfaces. *J. Opt. Soc. Am.*, 57(9):1105–1112, Sep 1967.

[Wal07] Ingo Wald. On fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society.

[War92a] Greg Ward. Real Pixels. In James Arvo, editor, *Graphics Gems II*. Academic Press. Inc., 1250 Sixth Avenue, San Diego, CA 92101, 1992.

[War92b] Gregory J. Ward. Measuring and modeling anisotropic reflection. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, pages 265–272, New York, NY, USA, 1992. ACM.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26, January 2007.

[Wei78] Bruce W. Weide. *Statistical methods in algorithm design and analysis.* PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA. Dept. of Computer Science., Pittsburgh, PA, USA, 1978. AAI7909881.

[WGS04] Ingo Wald, Johannes Günther, and Philipp Slusallek. Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristi. *Computer Graphics Forum*, 22(3):595–603, 2004. (Proceedings of Eurographics).

[WH92] Gregory J. Ward and Paul S. Heckbert. Irradiance Gradients. In *Eurographics Workshop on Rendering*, pages 85–98, 1992.

[WH06] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 61–69, September 2006.

[WLH97] Tien-Tsin Wong, Wai-Shing Luk, and Pheng-Ann Heng. Sampling with Hammersley and Halton points. *J. Graph. Tools*, 2:9–24, November 1997.

[WRC88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. *SIGGRAPH Comput. Graph.*, 22:85–92, June 1988.

[WTP01] Alexander Wilkie, Robert F. Tobler, and Werner Purgathofer. Combined rendering of polarization and fluorescence effects. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, pages 197–204, London, UK, 2001. Springer-Verlag.

Appendix A

List of abbreviations

API Application programming interface

BOB Bounds overlap ball

BRDF Bidirectional reflectance distribution function

BSDF Bidirectional scattering distribution function

BSSRDF Bidirectional surface scattering reflectance distribution function

BTF Bidirectional texture function

BVH Bounding volume hierarchy

CBF Corona binary file

CUDA Compute Unified Device Architecture

DLL Dynamic-link library

DVD Digital Versatile Disc

GI Global illumination

HDRI High dynamic range image

IC Irradiance caching

***k*-NN** *k*-nearest neighbour

NN Nearest neighbour

PDF Probability density function

RAM Random access memory

RC Radiance caching

RGBE Red, green, blue, exponent

RMSE Root mean square error

SAH Surface area heuristic

SDK Software development kit

SDL Simple DirectMedia Layer

SDRC Spatial directional radiance caching

SHRC Spherical harmonics radiance caching

SSS Bidirectional subsurface scattering

SVBRDF Spatially-varying bidirectional reflectance distribution function

Appendix B

Test scenes gallery

Figure B.1: *Diffuse interior by Jiří "Biolit" Friml.*

Figure B.2: *Conference (standard free test scene).*

Figure B.3: *Sibenik cathedral, courtesy of Marko Dabrovic.*

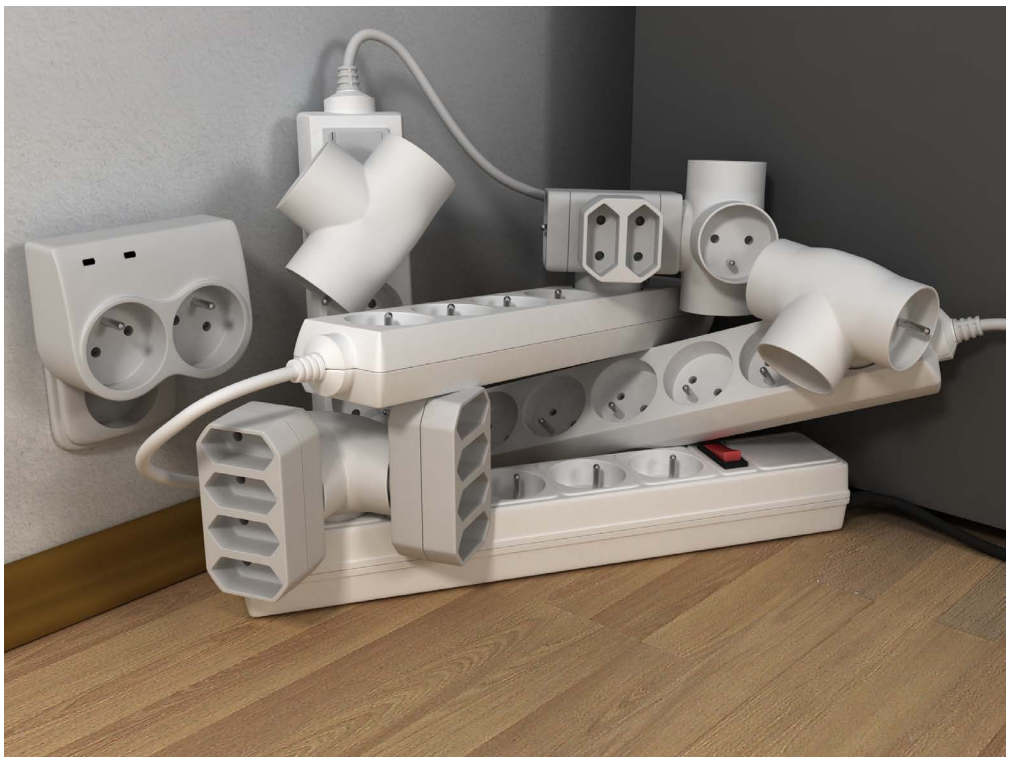


Figure B.4: *Power sockets.*

Figure B.5: *Sponza atrium, courtesy of Marko Dabrovic.*

Figure B.6: *Computer case.*

Figure B.7: *Components plate.*

Figure B.8: *Stanford dragon, courtesy of Stanford University Computer Graphics Laboratory [Sta10].*

Figure B.9: *Glossy interior, courtesy of Ludvík "Rawalanche" Koutný.*

Figure B.10: *Chessboard, courtesy of cjx3711 [cjx10].*

Appendix C

DVD Content

`bin`	Compiled corona executables together with needed libraries
`corona`	
`plugins`	Plugins needed by Corona
`src`	Source codes of Corona in a MSVS 2010 project
`demo`	Scripts for running demonstration renders
`doxygen`	Class documentation generated by Doxygen
`results`	Results of tests
`scenes`	Scenes used for testing in the thesis in the OBJ format
`tests`	Scripts that run the tests presented in the thesis
`thesis`	Both print-optimized and web-optimized versions of this thesis
`src`	XeLaTeX Source codes of this thesis
`figures`	All figures used in this thesis
`readme.txt`	File containing brief overview of the DVD content