Dynamic Indirect Illumination in Real-Time Ray Tracing using Lightcuts

Roeland Herman Reijerse
Dynamic Indirect Illumination in Real-Time Ray Tracing using Lightcuts

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Roeland Herman Reijerse
born in Alphen aan den Rijn, the Netherlands

TU Delft
Computer Graphics Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl
Cover picture: A rendering of the Stanford Dragon subjected to indirect illumination provided by our system.
Dynamic Indirect Illumination in
Real-Time Ray Tracing using
Lightcuts

Author: Roeland Herman Reijerse
Student id: 1249967
Email: roel.reijerse@xs4all.nl

Abstract

This thesis describes a system that adds plausible indirect illumination to the Arauna real-time ray tracer. Only the first diffuse reflection of light on static geometry is simulated. Light sources are allowed to be dynamic. Virtual point lights (VPLs) are used to simulate the reflection of light by surfaces. Their contribution to the final shading is stored in a sparse point set distributed randomly over the surfaces. During rendering the direct light is obtained through ray tracing and the indirect light from the VPLs is interpolated from the nearby points of the point set. When light sources are dynamically changed, the VPLs are updated and the indirect shading at the point set is updated. To avoid having to calculate the contribution of all VPLs (which may be thousands) for each point of the point set, the Lightcuts algorithm is used to approximate the illumination from the VPLs for each point. This sublinear scaling algorithm first creates clusters of lights which are adaptively selected during run-time while maintaining an upper bound on the approximation error. We describe bounding spheres as an alternative to the bounding boxes used in the original Lightcuts algorithm and derive new formulae for the error upper bounds. The system is tested with three different scenes. Our bounding sphere alternative is on average 1.3× faster than the box variant. In the tested scenes, plausible indirect illumination is obtained at interactive frame rates on commodity hardware.
Thesis Committee:

Chair: Prof. dr. ir. F.W. Jansen, Faculty EEMCS, TU Delft
University supervisor: Prof. dr. ir. F.W. Jansen, Faculty EEMCS, TU Delft
Company supervisor: J. Bikker, Faculty IGAD, NHTV Breda
Committee Member: Dr. ir. E.A. Hendriks, Faculty EEMCS, TU Delft
Committee Member: Ir. T. Kroes, Faculty EEMCS, TU Delft
Preface

This thesis is the result of my master project at the Delft University of Technology. Initially, my project was supposed to become a continuation of my literature survey, in which I studied methods to reconstruct 3D objects from photographs, including their material properties. However, that subject proved to be too extensive for a master project, and I was offered to do an alternative project: adding a form of indirect illumination to Jacco’s real-time ray tracer Arauna. It was not a difficult decision, since I already knew Jacco as the author of many tutorials on flipCode that I have read years ago in high school, when I started to explore computer graphics for fun. Back then I never envisioned ever working with him.

I would like to thank my supervisors: Erik Jansen, for providing me valuable insights and for his patience, and Jacco Bikker—his contagious enthusiasm helped me in times when I had less confidence or motivation. Furthermore I thank my family and friends for their support. I would like to mention one person in particular. Thanks, Malou, for being there for me.

Roeland Herman Reijerse
Sassenheim, the Netherlands
August 22, 2012
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
</tbody>
</table>

## 1 Introduction

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Arauna</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problem</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Thesis Outline</td>
<td>2</td>
</tr>
</tbody>
</table>

## 2 Rendering Global Illumination

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Ray Tracing and Rasterisation</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Whitted-Style and Distributed Ray Tracing</td>
<td>4</td>
</tr>
<tr>
<td>2.3 The Rendering Equation</td>
<td>5</td>
</tr>
<tr>
<td>2.4 Monte Carlo Ray Tracing</td>
<td>6</td>
</tr>
<tr>
<td>2.5 Non-Monte-Carlo Indirect Illumination</td>
<td>8</td>
</tr>
<tr>
<td>2.6 The Many-Light Problem</td>
<td>10</td>
</tr>
<tr>
<td>2.7 Real-Time Ray Tracing</td>
<td>11</td>
</tr>
</tbody>
</table>

## 3 System

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Overview and Rationale</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Architecture</td>
<td>14</td>
</tr>
<tr>
<td>3.3 The Point Set</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Virtual Point Lights</td>
<td>18</td>
</tr>
<tr>
<td>3.5 Visibility Precomputation</td>
<td>20</td>
</tr>
</tbody>
</table>

## 4 Lightcuts

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Original Lightcuts Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>4.2 Modifications to the Lightcuts Algorithm</td>
<td>33</td>
</tr>
</tbody>
</table>

## 5 Experiments

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Scenes Used for the Experiments</td>
<td>45</td>
</tr>
<tr>
<td>5.2 Time Performance</td>
<td>48</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Local illumination versus global illumination</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Ray tracing through a slightly opened door</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Three steps that are performed each frame</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>The architecture of our system</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>An example point set</td>
<td>18</td>
</tr>
<tr>
<td>3.4</td>
<td>Transferring energy from a light source to VPLs</td>
<td>20</td>
</tr>
<tr>
<td>3.5</td>
<td>Mappings of point distributions to a 2D domain</td>
<td>21</td>
</tr>
<tr>
<td>3.6</td>
<td>An example of bit-block compression</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Geometry of directional lights</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>An example light tree for a scene with four light sources</td>
<td>25</td>
</tr>
<tr>
<td>4.3</td>
<td>Three different example cuts</td>
<td>26</td>
</tr>
<tr>
<td>4.4</td>
<td>Computing upper bounds for clusters</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Finding the minimum distance to a bounding box</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>Bounding the direction of oriented lights with a bounding cone</td>
<td>31</td>
</tr>
<tr>
<td>4.7</td>
<td>Finding the minimum angle for oriented lights in a cluster</td>
<td>32</td>
</tr>
<tr>
<td>4.8</td>
<td>Transformation of a set of points bound by an AABB</td>
<td>33</td>
</tr>
<tr>
<td>4.9</td>
<td>An example light tree augmented with visibility information</td>
<td>35</td>
</tr>
<tr>
<td>4.10</td>
<td>Minimizing $\theta$ for a cluster bounding sphere</td>
<td>36</td>
</tr>
<tr>
<td>4.11</td>
<td>Finding an upper bound for $cos(\phi)$ for the geometric term</td>
<td>40</td>
</tr>
<tr>
<td>5.1</td>
<td>An overview of the scenes used for the experiments</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>Renderings of the test scenes</td>
<td>47</td>
</tr>
<tr>
<td>5.3</td>
<td>Fluctuation of the time spent at the Lightcuts computation</td>
<td>49</td>
</tr>
<tr>
<td>5.4</td>
<td>Examples of light positions that require less computation</td>
<td>49</td>
</tr>
<tr>
<td>5.5</td>
<td>Examples of light positions that require more computation</td>
<td>50</td>
</tr>
<tr>
<td>5.6</td>
<td>Computation time as a function of $N_{VPL}$ for the Colourful scene</td>
<td>51</td>
</tr>
<tr>
<td>5.7</td>
<td>Computation time as a function of $N_{VPL}$ for the Buddha scene</td>
<td>52</td>
</tr>
<tr>
<td>5.8</td>
<td>Computation time as a function of $N_{VPL}$ for the Maze scene</td>
<td>53</td>
</tr>
<tr>
<td>5.9</td>
<td>Relative time performance of the Lightcuts algorithm</td>
<td>53</td>
</tr>
<tr>
<td>5.10</td>
<td>Computation time and cut size for the box and sphere variants</td>
<td>55</td>
</tr>
<tr>
<td>5.11</td>
<td>Average cut sizes for the Maze and Rotated Maze scenes</td>
<td>56</td>
</tr>
<tr>
<td>5.12</td>
<td>Interpolation versus nearest point for difference images</td>
<td>57</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>5.13</td>
<td>Renderings of the Buddha scene for four values of $\tau$</td>
<td>59</td>
</tr>
<tr>
<td>5.14</td>
<td>Renderings of the Maze and Colourful scenes for two values of $\tau$</td>
<td>59</td>
</tr>
<tr>
<td>5.15</td>
<td>Difference images for the Buddha scene with constant magnification</td>
<td>60</td>
</tr>
<tr>
<td>5.16</td>
<td>Difference images for the Buddha scene with variable magnification</td>
<td>60</td>
</tr>
<tr>
<td>5.17</td>
<td>Difference images for the Maze scene with constant magnification</td>
<td>61</td>
</tr>
<tr>
<td>5.18</td>
<td>Difference images for the Colourful scene with constant magnification</td>
<td>61</td>
</tr>
<tr>
<td>5.19</td>
<td>Varying the density of the point set for the 'Buddha' scene</td>
<td>63</td>
</tr>
<tr>
<td>5.20</td>
<td>Illumination quality as a function of the point set density</td>
<td>64</td>
</tr>
<tr>
<td>5.21</td>
<td>Varying the density of the VPL set for the 'Colourful' scene</td>
<td>65</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Since the early days of computing, people have recognised the potential of computers to synthesise graphics. Where most of the early efforts were aimed at scientific and engineering applications, later on the use of computer graphics for entertainment purposes emerged. As computer hardware became more affordable, computer games moved from university mainframe computers to video game consoles and computers in people’s homes. For real-time rendering of 3D computer graphics in games, rasterisation or scan conversion is the de facto standard. Dedicated graphics hardware at the consumer level that accelerates this task was introduced in 1996 with the 3Dfx Voodoo 1, which is considered the beginning of a new era. More recently, mainly due to advances in available computation power and faster algorithms, an exploration of ray tracing for real-time graphics was started, as an alternative to rasterisation. Initially real-time ray tracing required supercomputers, but in 2001 real-time ray tracing was performed with commodity hardware. This thesis project is performed in the scope of the Arauna real-time ray tracer developed by Jacco Bikker.

1.1 Arauna

Arauna is a real-time ray tracer that has been specifically built with games and performance in mind. It has been used at the NHTV University of Applied Sciences for several student games. Since it is a ray tracer, it works fundamentally different from rasterisation—today’s ubiquitous paradigm for real-time computer graphics. It runs entirely on the CPU of commodity hardware and does not use graphics hardware to perform its computations. A simplified shading model is used which supports ambient, diffuse, specular and emissive components. Only direct lighting is considered: when a light path from a light source hits a surface, it is terminated.

\footnote{An overview of games using Arauna can be found at \url{http://igad.nhtv.nl/~bikker/}}
1. **Introduction**

1.2 **Problem**

Since only direct lighting is considered, places that are not directly lit are entirely black. In reality this does not happen: light is also reflected by other surfaces so that such places will eventually receive some light. A traditional solution is to use ambient lighting, a constant colour term that is added to any surface point regardless its visibility. The value of this term is typically fine-tuned by an artist so that the resulting imagery simply 'looks right'. Although this approach remedies overly-dark scenes, it is not physically correct and does not have a realistic appearance.

To increase the realism of the imagery produced by Arauna, we extend its lighting model to include indirect lighting. On the one hand, as Arauna is a real-time ray tracer, it is infeasible to do a computation that fully simulates light according to our current understanding of the physics of light. On the other hand, as Arauna is designed for usage in games, a plausible or approximate rather than a physically correct simulation of indirect lighting is required.

Limited by the constraints of being real-time, but exploiting the freedom to approximate, we will investigate in this thesis a system where:

- Indirect illumination is cached at sparse sample points throughout the scene;
- Reflection of illumination by surfaces is simulated via virtual point lights;
- The illumination from these lights is approximated with the Lightcuts algorithm [40].

1.3 **Thesis Outline**

The organisation of this thesis is as follows. First, in Chapter 2 we give a brief overview of the concepts of light transport and discuss related work. Next, in Chapter 3 our proposed system and its subsystems are presented. As the Lightcuts algorithm is rather complex and is playing an instrumental role in our system, Chapter 4 is entirely dedicated to it. This chapter is split into two parts, of which the first part describes the original algorithm, and the second part describes our application-specific modifications to it. After that, in Chapter 5 we evaluate our system in terms of scalability and accuracy. In Chapter 6 we conclude this thesis with a discussion and directions for future work. Lastly, in Appendix A the notation used throughout this thesis is described.
Chapter 2

Rendering Global Illumination

In this chapter, we give an overview of the basics and other work related to rendering global illumination effects, with a strong emphasis on ray tracing. We begin in Section 2.1 with a short overview of the difference between the two major rendering paradigms: rasterisation and ray tracing. Next, we discuss briefly two early ray tracing algorithms in Section 2.2: Whitted-style and distributed ray tracing. This is followed by Section 2.3 in which we describe the rendering equation, a fundamental model of light transport that any realistic rendering algorithm tries to solve in one way or another. The distinction between local illumination and global illumination is described as well. This is followed by Section 2.4 in which we describe Monte Carlo ray tracing, the most general and robust approach to solve the rendering equation. In Section 2.5 we discuss some non-Monte-Carlo methods that are specialised in accelerating the computation of indirect illumination. That discussion brings us to the many-light problem, which is described in Section 2.6. Finally, we conclude this chapter in Section 2.7 with an overview of real-time ray tracing, the branch of ray tracing that aims at interactive frame rates rather than physical accuracy.

2.1 Ray Tracing and Rasterisation

In computer graphics, two major paradigms are used to produce 2D raster graphics images of 3D geometric data: rasterisation (or scan conversion) [1] and ray tracing [26]. When compared in performance and realism, the two paradigms are each other’s opposite. Rasterisation is fast, but provides limited realism and physical correctness. Ray tracing is slow, but simulates light in a realistic way. In short, these algorithms work as follows.

- With rasterisation, polygons are converted into a raster image (a 2D pixel array). In its most basic form, rasterisation only computes which pixels are covered by a 2D polygon defined by line segments between vertices. For 3D computer graphics, typically only triangles are used. These triangles are projected from space into 2D, and then converted into pixels. To find out

\footnote{A third method, that is less ubiquitous but not negligible, is Reyes rendering [8], which is used primarily in film production.}
what colour a pixel should be, texture and shading calculations are applied. For the texture calculations, an image that is ‘glued’ onto the triangles is used to colour the pixels. In the shading calculations, the outgoing radiance is computed. To do this, information about light sources is used, as well as extra data provided with the vertices that provide material properties like the surface normal.

A major difference with ray tracing is that the shading computations in plain vanilla rasterisation only have access to local data. That is, a triangle being rasterised has no knowledge of other geometry. This means for example that shadows cannot be calculated: a triangle that is shaded does not know if another triangle is occluding the light source. The same also applies to reflection and refraction. Global data can be provided by collecting it in advance, though. For example, to handle shadows, one can do a pass prior to rendering, in which the scene is rasterised into a depth buffer with the light as eye point. During rendering, this depth buffer is queried to find occluding geometry.

Because of its speed, rasterisation has been the de facto technique for real-time computer graphics since the early days of computer graphics. Enhancements of the algorithm are aimed at increasing realism, for example by adding shadows or reflections, since these are not present in the basic algorithm.

- With ray tracing, paths of light are traced through an image plane and interactions with objects in the scene are simulated. The behaviour of light is simulated based on physics, and the resulting images can be very realistic. However, this is at the cost of high computation times—up to hours, for a single image—even on modern hardware. Because of the computation time, ray tracing has been primarily used for non-real-time computer graphics. Enhancements of the algorithm are aimed at increasing its speed, for example by smarter sampling of the lighting (e.g. Metropolis light transport), sub sampling of low frequency lighting (e.g. irradiance caching), etcetera.

### 2.2 Whitted-Style and Distributed Ray Tracing

The first ray casting algorithm was introduced by Appel in 1968. It works by sending rays from the eye point through pixels in the image plane, and finding the nearest intersection. After a ray has hit the surface, it is not traced further. In 1980, Whitted extended the algorithm by handling shadows and recursive reflection and refraction. The resulting algorithm is commonly referred to as Whitted-style ray tracing.

_Distributed ray tracing (DRT)_ (also referred to as _distribution ray tracing_ to avoid confusion with distributed computing), was introduced in 1984 by Cook et al. It increased the realism of image synthesis by rendering glossy reflections, translucency (blurred transparency), soft shadows from area lights, depth of field and motion blur. It works by using multiple samples per ray, instead of
2.3. The Rendering Equation

Where the previously mentioned algorithms only used simple approximations for the shading, a first full understanding of the physically correct reflection behaviour of light was only achieved with the introduction of the rendering equation by Kajiya in 1986 [18]. It describes how light behaves in vacuum, based on geometric optic principles. Contrary to methods as for example radiosity (see Section 2.5.1), it does make only a few simplifying assumptions on the simulation. Any realistic rendering algorithm tries to find solutions for this equation in one way or another. Different forms are used, but we will use the following form:

\[
L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{\Omega} f_r(x, \omega_i, \omega_o) L_i(x, \omega_i) \cos \theta_i d\omega_i, \tag{2.1}
\]

with \(L_o(x, \omega_o)\) the outgoing radiance from surface location \(x\) in direction \(\omega_o\), \(L_e(x, \omega_o)\) the emitted radiance from surface location \(x\) in direction \(\omega_o\), \(\Omega\) the hemisphere of directions above \(x\), \(f_r(x, \omega_i, \omega_o)\) the bidirectional reflectance distribution function (BRDF) [23], \(L_i(x, \omega_i)\) the incoming radiance at \(x\) from direction \(\omega_i\) and \(\theta_i\) the angle between \(\omega_i\) and the surface normal \(n\). Spectral and time dependence can also be included in this equation, but are omitted for brevity. An intuitive paraphrase is: the radiance from a surface point \(x\) towards \(\omega_o\) (e.g. the eye) is the sum of the radiance emitted by the surface and all incoming radiance reflected by the surface. The BRDF takes account of weighting how much of the radiance is reflected given the incoming and outgoing directions. Analytical solutions of the rendering equation are only known for trivial cases; typically numeric methods are used to solve it. The most common method is the Monte Carlo method, which will be discussed in Section 2.4.

Instead of finding full solutions to the rendering equation, approximations of it are used to accelerate the computation. One reason that makes the rendering equation expensive to compute is its recursive nature. A common approach to reduce the computational costs is to not recurse at all, and perform a limited number of evaluations of the integrand. That is, when shading a surface point, only direct illumination from light sources is computed. This approach is referred to as local illumination; the contrary approach is global illumination. In the local illumination model, surfaces are only directly lit by light sources. Except for the light sources, only local information is used for shading, e.g. the position, normal and material properties of the point to be shaded. This means that even shadows are not rendered; it would require global information (the geometry of the scene) to test for occlusion. In the global illumination model, the interaction of light with the entire scene is simulated for shading a point. Surfaces are also illuminated with light reflected from other surfaces, instead of restricting to
2. Rendering Global Illumination

(a) Local illumination.

(b) Global illumination.

Figure 2.1: Local illumination versus global illumination. (a) Only local illumination is used to light the scene. That is, light travels only directly from the light source to surfaces in the scene. Although the ceiling is made from the same material as the floor, it remains black. The reason for this is that there are not any direct paths from the light source to points on the ceiling. (b) With global illumination, light reaches surfaces also via other surfaces. In contrast with (a), illumination now reaches the ceiling being reflected from the scene below it. The reddish and greenish shades on respectively the left and right side of the ceiling is due to the left and right walls. Also note the bright spot (caustic) on the floor and wall near the left sphere where it focuses light from the light source.

direct illumination only. Figure 2.1 shows a visual example of the differences between images rendered with local and global illumination.

As mentioned earlier in this chapter, plain vanilla rasterisation uses the local illumination model, but it can be augmented with global information. Appel’s ray casting algorithm used only local information; the extension of Whitted added a limited form of global illumination by considering shadows and reflection. Solutions to the rendering equation yield full global illumination; the most general and robust approach to solve it is Monte Carlo ray tracing, which is described in the next section.

2.4 Monte Carlo Ray Tracing

Monte Carlo (MC) methods \cite{17} are widely used to numerically estimate integrals in many fields as engineering, physics, and finance. In rendering and ray tracing in particular, MC methods are nowadays prevalent to solve the rendering equation. In a nutshell, MC integration works by evaluating the integrand at random points over its domain. The weighted summation over the evaluations is an estimate of the integral. As it is a stochastic process, the output has variance. This variance leads to noise in the output image, which is proportional to the variance. To halve the variance, one has to quadruple the number of samples \cite{30}. 
Shirley calls this the diminishing return \([31]\): the more samples one adds, the less variance reduction is returned per added sample. Much effort has been put in variance reduction using other methods than simply increasing the number of samples. One is importance sampling, where important regions of the integrand (i.e. there where its value is large) are sampled at a higher density. Another method is stratified sampling, where the domain of the integrand is divided into several smaller domains called strata. The integrand is then evaluated as the sum of the integrals over the strata, which are typically evaluated using a single sample per stratum. The variance of stratified sampling is never higher than unstratified sampling, and it is often even far superior to importance sampling [17].

2.4.1 Biased versus Unbiased Algorithms

In rendering, and in particular in stochastic methods, a distinction is made between biased and unbiased methods [9]. A method is unbiased if its expected error (i.e. bias) is zero. That is, an unbiased method computes the correct answer on average. The reverse also holds: a biased method computes the wrong answer on average. Sometimes the term ‘unbiased’ is confused with the term ‘consistent’. A method is consistent if it converges to the correct answer if it is given more samples. For realistic image synthesis, unbiased methods are preferred over biased methods for obvious reasons. Bias can be introduced by renderers by for example ignoring indirect illumination (which leads to images that are too dark) or by not testing for shadows (which makes images too bright). Usually, bias is accepted in (real-time) rendering applications in return for faster computations.

2.4.2 Path Tracing

Path tracing [18] is an unbiased technique to solve the rendering equation using Monte Carlo. A pixel value is computed by sending a ray from the camera lens (image plane), and finding the nearest point of intersection with a surface. Next is randomly decided if light emitted from the surface is computed, or light reflected by the surface. The probability is based on the surface material’s properties. When light is emitted, it is returned over the path to the pixel. When light is reflected, the ray is scattered randomly according to the surface material’s properties and the algorithm recurses. For each pixel, multiple paths can be traced and averaged to reduce variance.

Path tracing is usually considered as a reference method: new rendering algorithms are compared to it to test their correctness. An undesired property is that its output has a high variance for scenes with indirect illumination. For example, consider Figure 2.3(a). It shows a scene with two rooms that are connected with a door. One room is brightly lit, and the door is slightly opened. When computing the lighting in the darker room, a path tracer will have difficulties tracing paths through the opening. If the probability that a ray is shot through the opening for some neighbouring pixels is say 0.1%, and one is using 1024 samples (which is a fairly large amount), it is likely that the
neighbouring pixels will appear noticeably differently, as some will have light paths through the opening, while others have not.

2.4.3 Bidirectional Path Tracing and Metropolis

Bidirectional path tracing \cite{20,34} reduces the variance by creating subpaths starting both at light sources and the image plane. These subpaths are connected on all vertices, and light transport is considered over all resulting paths. See Figure 2.2(b). Metropolis Light Transport (MLT) \cite{33} is a variant of bidirectional path tracing which tries to reduce variance further. Once a path from a light source to the eye has been found, paths are mutated. Mutations are accepted or rejected randomly with a carefully chosen probability so that paths are sampled accordingly to their contribution to the image. Both bidirectional path tracing and MLT are unbiased algorithms.

2.4.4 Photon Mapping

An algorithm that speeds up MC ray tracing is photon mapping \cite{16}. Contrary to the previously mentioned MC methods, photon mapping is biased, but it is consistent though \cite{9}. This means that it does not produce the correct result on average, but increasing the number of photons makes it converge towards the correct solution. It works in two passes. In the first pass, particles (photons) are emitted from light sources, and scattered throughout the scene. When a photon hits a surface, its position and incoming direction are stored in the photon map. After a hit, a photon is absorbed, reflected or transmitted with a probability based on the surface’s material. For reflection, the direction is randomly chosen with a probability based on the material’s BRDF. In the second pass, the scene is rendered by sending rays from the camera into the scene, and finding the nearest intersection. At that point, the various contributions to its surface radiance are handled in separate ways. Direct lighting is computed in a similar fashion as distribution ray tracing. Specular reflections are handled using ray tracing. Indirect illumination is queried from the photon map, and caustics are queried from a separate higher density photon map that only contains photons that are the result of refraction.

2.5 Non-Monte-Carlo Indirect Illumination

The rendering equation can be decomposed into separate terms, after which the terms are handled separately by algorithms that are specialised for a specific term. One of these terms that is suited for special treatment is the indirect illumination term, which is related to the recursive nature of the rendering equation. In this section, other algorithms than those that are based on the Monte Carlo method are discussed. First, Radiosity is discussed in Section 2.5.1. Radiosity computes the equilibrium state of light in scene with purely Lambertian surfaces. In Section 2.5.2, instant radiosity is described, where the rendering equation is reformulated so that virtual light sources approximate indirect illumination. Also, a more recent method that uses the same concept is described. Because of
2.5. Non-Monte-Carlo Indirect Illumination

(a) Path Tracing.
(b) Bidirectional Path Tracing.

Figure 2.2: Ray tracing of a scene with two rooms connected with a slightly opened door. The left room is brightly lit. The camera points towards the wall on the right. [a] With path tracing, the probability of creating a path from the camera through the doorway to the light is low. A large number of paths is required to bring the variance down to an acceptable level. [b] With bidirectional path tracing, paths are also started at the light source. These paths are connected with paths from the camera (dashed lines).

the low frequency nature of indirect lighting, it is a candidate for interpolation. This is the purpose of the irradiance cache, which is described in Section 2.5.3

2.5.1 Radiosity

One well-known algorithm to compute global illumination for diffuse surfaces is radiosity [13]. It has its origin in methods for the calculation of radiative heat transfer. It works by first dividing the surfaces of the scene into patches (which makes it a finite element method). Next, view factors (also referred to as ‘form factors’) are computed for each pair of patches. These view factors are coefficients that describe how well the patches can see each other. With these factors, equations are created that describe how much illumination is transferred between the patches. The equations of all patches together form a system of equations. When this system is solved, the solution is the equilibrium state of the illumination in the scene. A drawback is that radiosity handles only Lambertian surfaces. Radiosity is not very suitable for real-time applications: although view factors can be reused in the case of static geometry, solving the system of equations scales poorly: the fastest finite element methods have a complexity of $O(N \log N)$ [17].

2.5.2 Instant Radiosity

In instant radiosity [19], (quasi-)random light paths are traced from a light source. At the vertices of these paths, virtual point lights (VPLs) are generated. VPLs are hemispherical lights with a cosine fall-off. During rendering, the VPLs are used to approximate indirect lighting. This only works for materials which have diffuse or near-diffuse BRDFs as the emission function of the VPLs has a cosine distribution. A GPU is used to accelerate rendering. Using the GPU,
VPLs are rendered with shadows one by one, and the resulting images are accumulated. Instant radiosity can be thought of as a Monte Carlo solution to the rendering equation, but with samples that are correlated. This result in less noise, but might introduce banding. With minimum render times of several seconds per frame, it was not aimed at interactive rendering.

Incremental instant radiosity \[21\] uses the basic idea of instant radiosity, but now in a real-time setting, and using the GPU. It allows dynamic movement of light sources with static geometry. Instead of generating the whole set of VPLs for each frame, only a few VPLs are regenerated, and the majority of the VPL set is kept. This has the advantage that shadow maps (for each VPL) only need to be calculated for the regenerated VPLs; the shadow maps of the unmodified VPLs remain valid as the geometry is static. VPLs that have become invisible from the light source are deleted, as well as VPLs in the direction where the main light source has zero emission. For each deleted VPL, a new one is generated. The position of the new VPL is based on the distribution of the other VPLs as seen from the main light source, so that their distribution follows its emission function.

\[2.5.3\] Irradiance Cache

Because of the computational costs and the low frequency nature (spatial coherence) of indirect lighting, it is lucrative to do an exact computation of it on a limited number of surface points, and interpolate for locations in between. This is the idea of Ward et al.'s irradiance cache \[43\], presented in 1988. Because the irradiance is stored in the cache (without directional information), only diffuse materials can be used. Entries in the cache contain a position and normal of the surface point where the irradiance was calculated, as well as the irradiance value itself. Entries are computed on an as-needed basis. By measuring the surface curvature and how nearby other geometry is, the expected irradiance constancy (i.e. gradient) can be calculated. With this information, the decision is made whether to interpolate (when the approximated indirect irradiance from the cache remains within a predicted error bound) or to actually compute the indirect irradiance.

\[2.6\] The Many-Light Problem

When indirect illumination is approximated with virtual light sources, typically large numbers (several thousands) of lights are required to make the approximation accurate. Because of the inherent computation cost, it is preferred not to resort to an \(O(n)\) algorithm. Algorithms that approximate the illumination from many lights aim to solve what is called the many-light problem.

Ward describes a method \[42\] in which light sources are sorted according to their potential contribution prior to testing visibility. Only lights whose contributions are above a specified threshold are actually tested for visibility. The visibility of the remaining lights is estimated using statistics that are obtained when visibility is tested: the hit-ratios between lights and surface points. When lights are actually tested for visibility, the hit-ratio is kept track
2.7. Real-Time Ray Tracing

The hit ratio for lights seen from a surface point is stored per point. Using these two ratios, the visibility of the remaining lights is estimated. The product of these two ratios is used as the expected visibility that a certain light is visible from a certain surface point.

In [24] a method is described in which a hierarchy of lights is created. Lights are stored in an octree. At each level of the octree, virtual lights are created. These virtual lights are positioned at the intensity-weighted average position of the lights they represent. Which level’s virtual light is used, depends on the point that is rendered. A major problem is that this method only works for scenes without occlusion.

In [15], the many-light problem is formulated as a large matrix. The columns contain the intensities provided to each sample (surface point) for a single light. The rows contain the intensities received from each light for a single sample. Computing the full matrix gives an exact result, but is too expensive. Preferably, a small subset of the columns (lights) is computed as an approximation. To find out which columns form a good set, first a small set of rows (samples illuminated by all lights) is computed. This can be thought of as a very low-resolution rendering of the scene. With this information, the set of columns and weights is computed by clustering the columns and selecting representatives. Finally, the columns are computed using the GPU and a weighted sum of the columns gives the final result.

The Lightcuts algorithm [40] works by creating a hierarchical clustering of lights. Illumination from a cluster is approximated using one representative light from the cluster. An error upper bound is computed for the approximation, which is used to select clusters. A more extensive description of the Lightcuts algorithm is provided in Chapter 4.

2.7 Real-Time Ray Tracing

Real-time ray tracing is not fundamentally different from regular ray tracing: both typically use sophisticated algorithms and data structures together with a careful implementation to maximise the speed. The goals are different however: most regular ray tracing algorithms aim to produce physically correct images, at the cost of longer run times (hours); real-time ray tracing aims to achieve several images per second, at the cost of lower quality approximations.

In [39] a technique was proposed in which earlier obtained results are cached and reused. During camera and object motion, previously rendered results are reprojected to the new situations and reused if possible. More attention of the ray tracer is given to parts that are not present in the cache. For example, a part of a wall that was occluded first, but appeared due to camera movement, is ray traced. Parts of the same wall that were already visible are in the cache, and re-projected at their new screen position.

One of the earlier real-time ray tracing systems is described in [25], which was published in 1999. It uses conventional Whitted-style ray tracing [44], but is optimised for a supercomputer with shared memory, by taking care of parallelism and cache performance. In 2001, [Wald et al.] described a ray tracer that ran
on commodity hardware [35]. They claim to have obtained a performance gain of more than an order of magnitude compared to ray tracers of that time, by making better use of computational resources like caches and SIMD instructions and exploiting coherence of rays.

A notable and long running project was OpenRT [36]. Its goal was to provide a software ray tracer that was accessible through an API very similar to OpenGL. Although it could run on a single consumer level system, it was designed to run on several machines connected through a network, which was required to obtain interactive frame rates. Although to our knowledge never officially abandoned, the last update of the project’s page[2] was in 2006.

Arauna [3] is considered to be a state-of-the art real-time ray tracer [22]. It is specifically built with performance and application for games in mind. It aims at attractive visual appearance rather than photo realistic rendering. Because of this, it can omit parts of the rendering equation (or use cruder approximations) to save computation costs. For example, a simplified shading model is used, and for lights the quadratic fall-off (which gives lights an infinite sphere of influence) is replaced with an approximation that limits the influence of lights. Furthermore, much care has been taken to implement modern acceleration structures with low-level optimisations. This results in Arauna being the fastest open source ray tracer currently available [12].

Chapter 3

System

This chapter describes the system we devised to approximate indirect lighting in Arauna. The organisation of this chapter is as follows. First, in Section 3.1 a high level overview of the system is given. In Section 3.2 the architecture of the system is described, and the relations between subsystems are discussed. The remaining chapters then describe the subsystems in more detail. First, Section 3.3 describes the point set that is used to store indirect lighting. Next, Section 3.4 details how virtual point lights (VPLs) provide the indirect lighting. Finally, Section 3.5 discusses the visibility precomputation between the point set and VPLs.

3.1 Overview and Rationale

The purpose of this project is to extend Arauna with indirect illumination, with a good trade-off between rendering quality and speed. In our approach the way Arauna computes direct lighting is left unmodified. Our system will handle the indirect lighting term. The sum of those two terms yields the full lighting.

Only indirect lighting from the first bounce of light is considered. That is, only those light paths that start at a light source and are reflected by a single surface. Doing so reduces the computation time and is typically sufficient to provide plausible indirect lighting [32]. However, the result is biased since all other light paths are ignored. Being biased is not necessarily a problem, as the envisioned application of our system requires convincing indirect illumination rather than physically correct indirect illumination.

To simulate the indirect light transport, we follow the approach of instant radiosity. Illumination from surfaces reflecting light is simulated by virtual point lights that are distributed across the surfaces in the scene. The colour of a VPL is the colour of the surface it belongs to (e.g. a VPL on a red wall emits red light) and is modulated with the illumination from the real light source.

Instead of sampling the virtual point lights during rendering, prior to rendering we calculate their contribution only for a sparse set of points over the surfaces, in a similar fashion as the irradiance cache technique (see Section 2.5.3), exploiting the fact that indirect lighting is typically low frequent and can be interpolated from nearby points. We refer to these points as the point set or
3. System

**Sample points.** For performance reasons, the positions of both the set of VPLs and point set are kept static. This allows the costly computation of visibility between sample points and VPLs to be precomputed. As a consequence of the static sets, the system only supports static geometry. The point set does not contain directional information of light arriving at the point. As a result, the indirect illumination can only be used for Lambertian (or perfectly diffuse) shading.

Despite the precomputed visibility of VPLs for each sample point, it remains expensive to iterate over each VPL in order to obtain the indirect lighting. Since the ray tracer runs on the CPU, we cannot profit from the sheer stream processing power of GPUs. To accelerate this computation, we use the Lightcuts algorithm to approximate the lighting from the VPLs. This algorithm is described in Chapter 4. To recapitulate, the key components of our system are:

- A static point set, similar to the irradiance cache, is used to store indirect lighting;
- A static set of VPLs that emit light as if surfaces were reflecting light;
- The Lightcuts algorithm that approximates the illumination from the VPLs, whose output is stored into the point set.

The features and constraints of our system are:

- Light sources can be dynamic (colour, position, orientation);
- Geometry must be static;
- Only the first bounce of indirect light is computed;
- Only Lambertian (i.e. perfectly diffuse) shading is possible for indirect light.

During run-time, the computation of the indirect lighting is done prior to rendering a frame with the Arauna ray tracer. This computation consists of two steps. In the first step, energy from the primary light source(s) is transferred to the VPLs. In the second step, the irradiance at each sample point is computed using the VPLs and stored in the points. Next, during rendering, this irradiance is queried from the point set and is used for shading. This is graphically summarised in Figure 3.1.

3.2 Architecture

In Figure 3.2 the architecture of our system is shown. It is split into two parts: a pre-processing stage that is performed off-line, and a run-time stage that is performed for each rendered frame. The total system has two inputs provided by the user: geometry in the form of a polygon mesh with triangular faces and ‘real’ light sources (with a stress on real to avoid confusion with virtual lights).
Figure 3.1: Three steps that are performed each frame. Black dots represent sample points; white dots VPLs. (a) First, the intensity of the VPLs is calculated given the primary light source(s). (b) Second, sample points are lit by the VPLs. (c) Third, during rendering the sample points provide indirect illumination.
The pre-processing stage (Figure 3.2(a)) uses only geometry as input. The geometry is first provided to the ‘dart thrower’, which generates Poisson distributed point sets over the surfaces of the scene (this will be discussed in more detail in Section 3.3). It generates two sets: the sample points and the set of virtual point lights. The latter set has typically a lower density than the former. Both the sample points and VPLs have a position and an orientation vector. Sample points have a colour vector that will contain the irradiance at run-time. VPLs contain the colour of the surface whose indirect lighting they represent. Additionally, at run-time this colour is modulated with light to obtain their final intensity. Using the virtual point lights, the ‘light tree builder’ creates a ‘light tree’. This tree structure is used by the Lightcuts algorithm and will be discussed in Chapter 4. It contains the VPLs in its leaves, and clusters of VPLs in its nodes. The ‘visibility calculator’ has three inputs: the geometry, and the two sets produced by the dart thrower. For each sample point, it determines the visibility of each VPL, by testing the geometry for intersections. This information is stored into the ‘visibility info’. 

During run-time (Figure 3.2(b)), the ‘light tree’ and ‘visibility info’ that were generated during the pre-process are used to compute the irradiance of the sample points. Additionally, ‘real’ light sources for which the indirect illumination is to be calculated are an input. There are three stages. First, the ‘VPL intensity updater’ computes what the intensity of the VPLs should be, given the actual real light sources. The results are stored into the light tree’s leaves, which are the VPLs. Second, the light tree updater propagates the potential changes in the leafs upwards to the entire light tree (explained in Chapter 4). Third, Lightcuts are used to compute the irradiance from the VPLs for each sample point, using the light tree and the visibility information. Its output is put into the colour component of the sample points. Finally, in the renderer (i.e. Arauna), the irradiance from the sample points is used to approximate the indirect lighting term.

3.3 The Point Set

In general, indirect illumination is expensive to compute. However, it is typically low frequent, which makes it a good subject for sub sampling and interpolation. This property is exploited in Ward’s irradiance cache [43], a structure to cache indirect diffuse irradiance. Instead of calculating the indirect irradiance at all positions, it is computed on an as-needed basis. The decision if it is needed to compute, is based on a weighting function. This function uses the harmonic mean distance to neighbouring surfaces and surface curvature—two properties that affect irradiance constancy.

There are two reasons why such an irradiance cache would not work very well for our system. The first and most important reason comes from how modern hardware is used in Arauna (and in real-time ray tracing in general). To take full advantage of the cores found in modern multi-core systems, the screen is divided in several blocks, and blocks are fed to the threads running on the cores. These threads execute their jobs as autonomously as possible, to
3.3. The Point Set

(a) Architecture of the pre-processing stage.

(b) Architecture of the run-time stage.

Figure 3.2: The architecture of our system. Blank boxes indicate objects (e.g. lights, geometry). Gray boxes indicate processes that perform an operation on the objects. The double lined boxes contain original user input that describes the scene: geometry and lights. Arrows indicate the flow of information.

avoid stalls due to synchronisation. If one of these threads would need to create an entry in the irradiance cache, to maintain continuity across the screen blocks, it either must be shared with other threads (which leads to stalls), or other threads must replicate the already expensive irradiance computation (which is wasteful). The second reason is that indirect irradiance computations are very expensive for non-trivial scenes. For real-time applications, it is convenient if precomputed information exists to assist and accelerate the computation. Therefore it is desirable to compute indirect irradiance only at locations that are known beforehand, so that such information can be provided at these locations computed in advance.

These observations led to our own variant of an irradiance cache, which we refer to in this text as the ‘point set’ or the ‘sample points’. It is described in [4], but is briefly repeated here for the sake of completeness. Our scheme consists of a pre-calculated sparse set of fixed points that store shading information. In our application, this shading information is indirect irradiance, but other applications are possible too. Prior to rendering a frame, the points are updated. During rendering, the information in the points is used for any position in the scene using interpolation.

For the construction of the point set, a Poisson-disc process is used. Such a distribution makes aliasing artefacts less noticeable to the human visual system [1]. This distribution is the result of a random sampling process with a minimum distance rejection criterion. The point set is created on the surfaces of the scene as follows. First, dart sources are created, whose positions are determined by shooting particles from light sources. The use of light sources as origin for
3. System

(a) Scene. (b) Point set.

Figure 3.3: An example rendering of the point set. (a) A scene with some geometric objects. (b) A corresponding point set. Note that the density increases near other geometry, for example in the corners of the room and below the dodecahedron (in the upper centre of the image).

particles has a practical motivation: it is a cue what the inside of the scene is. These particles make several bounces; at each vertex of their path a dart source is created. This scheme guarantees that points are never generated on the inside of (watertight) solid objects or on the outside of the scene. Next, after the dart sources are generated, darts are thrown. Where a dart hits geometry and no other point exists within the search radius a new point is created. We make the search radius a function of the ambient occlusion so that the density increases in occluded areas. The motivation for this is the observation that indirect lighting is more likely to change in the presence of occluding geometry than in more open areas. An example of a point set is shown in Figure 3.3.

During rendering, the point set is kept in a regular 3D array (grid). An octree structure proved to be an order of magnitude slower than the grid. Visualisation of the shading information can be done by simply searching for the nearest point and using its information. This results in a Voronoi-like pattern when neighbouring points differ in shading information. A visually more pleasing result is obtained by interpolation between several points using a part of the Shepard approximation [29].

3.4 Virtual Point Lights

Virtual point lights (VPLs) are used to simulate the reflection of light by surfaces in the scene. All VPLs in our system are oriented hemispherical lights and emit light following a cosine fall-off distribution. They are distributed across the scene in a similar fashion as the point set, but in general with a lower density. As for the point set, the density is user configurable and controls the desired quality of the indirect lighting versus the computational costs. Typically, the number of VPLs is set to about a tenth of the number of sample points. Each VPL has a fixed base colour, which is set to the diffuse colour of the surface it represents, e.g. a VPL on a red wall has a red base colour. The light’s direction of maximum emission is set to the normal vector of the surface it
Virtual Point Lights

represents. The intensity of a VPL equals the illumination received from real light sources modulated with the VPL’s base colour. Whenever a primary light source changes (in position, orientation or colour), the intensity of each VPL needs to be updated. This ensures that the simulated reflection corresponds to the direct illumination provided by the real light source. The amount of illumination (or energy) that is transferred from a real light source to a VPL, depends on the visibility between the VPL and the light source (invisible VPLs receive no energy), and the solid angle covered by the VPL, observed from the primary light source. In this section it is assumed that the VPL intensities are computed for a single primary light source. The concept is easily extended to multiple light sources: the final intensity of a VPL is simply the sum of the VPL intensities computed for each individual light.

When energy is transferred from a real light source to the VPLs, one must take care of the distribution (and therefore the solid angle they cover) of VPLs relative to the light source. For example, consider Figure 3.4. The light source is an omnidirectional point light with a uniform distribution. The VPLs are more or less uniformly distributed across the geometry in the scene. However, when one considers the distribution of the VPLs projected onto a sphere around the light, it is far from uniform. If one would divide the sphere into two hemispheres, half the amount of the total energy would have to be transferred in the direction covered by a single hemisphere, since the light source is isotropic. More generally speaking, the light source transfers the same amount of energy over any pair of solid angles that have the same size. For instance, when the sphere is divided into two (equal) hemispheres across the dashed grey line in the figure, VPLs 1–3 together should receive the same amount of energy as all the remaining VPLs on the opposite hemisphere together. Simply giving all \( n \) VPLs an equal amount of \( \frac{1}{n} \)th of the total energy of the light source would transfer energy in a non-uniform fashion. If one would do so, for example, VPLs 1–3 would receive less energy—and the VPLs on top of the pillars would receive more energy—than their amount of covered solid angle (relative to the light source) justifies. Hence, when updating the VPL intensities, we must take care of the density of VPLs relative to the light source and distribute energy accordingly.

To measure the distribution of VPLs as observed from a point light source, we follow the approach of \[21\]. The intensity of a VPL should equal the emission function of the primary light source integrated over the solid angle represented by the VPL. When one only considers point light sources as primary light source, the directional distribution of VPLs as observed from the primary light source can be mapped into a 2D domain. For a 180° cosine fall-off spot light, a unit disc is used as 2D domain. Projection of VPLs into this domain is done as follows. First, the z-axis of an orthonormal basis is defined so that it matches the direction vector of the spot light. Next, normalised direction vectors to the VPLs are projected onto the \( xy \)-plane by simply setting their \( z \)-component to zero. This can be thought of as flattening a hemisphere (with its base on the \( xy \)-plane) to a disc. From the Nusselt analogue \[1\] follows that the areas on the disc correspond to the cosine-weighted distribution on the hemisphere. This is graphically illustrated in Figure 3.5. Figure 3.5(a) shows the unit disc that corresponds to points (VPL directions) on the hemisphere in Figure 3.5(b).
According to the Nusselt analogue, the uniform distribution of points on the unit disc in Figure 3.5(a) corresponds to the emission of the spot light with a cosine fall-off shown in Figure 3.5(b). For an omnidirectional light source with a uniform distribution (Figure 3.5(c)), the mapping is performed by normalising the direction vectors to the VPLs. The surface of the sphere is used as 2D domain.

After the points in the 2D domain are obtained, the solid angle covered by each VPL is computed. This is done by creating a Voronoi diagram for the points. The ratio between the area of the Voronoi cell around a point and the area of the unit disc $\pi$ is the fraction of energy assigned to the VPL that belongs to that point.

### 3.5 Visibility Precomputation

In a pre-processing stage the (mutual) visibility between sample points and VPLs is calculated. The result is stored on disk and is used during run-time to accelerate visibility queries. Although the pre-processing stage is performed off-line, its speed is not totally unimportant. For example, it is inconvenient for a user to endlessly wait after he modelled a new scene, or changed the density of the point set. Although it is still computational expensive, some care has been taken to accelerate this stage by using packet traversal. This functionality was already available in Arauna, but is briefly described in Section 3.5.1 for completeness. A simple scheme is used to encode the visibility. A unique integer $0 \leq i < N_{vpl}$ is assigned to each VPL. Then for each sample point, element $i$ of a bit-vector with length $N_{vpl}$ encodes the visibility between that particular sample point and light $i$. Since the visibility information can consume more memory space than available, it might be necessary to compress it in order to fit it into memory. We devised a simple scheme which is described in Section 3.5.2.
3.5. Visibility Precomputation

3.5.1 Packet Traversal

A common technique to accelerate ray tracing is to group rays into packets to take advantage of coherence. These packets of rays are then traversed and intersected in parallel. The performance improvement provided by packet tracing depends on the coherence of the rays. Typically, primary rays (from the camera to the scene) and shadow rays have a high coherence. This lends packet traversal to accelerate the visibility computation between points and VPLs. Packet tracing improves cache usage, and allows to efficiently use SIMD extensions that are ubiquitous on modern processors. Using SIMD, multiple (typically two or four) operations can be performed at the same time. When traversing a data structure like a bounding volume hierarchy (BVH), or performing ray-triangle intersection tests, multiple rays can be tested against the same entity (e.g. four rays versus a triangle). The advantage reduces when the rays are not coherent: for example, if only one ray intersects a leaf of the BVH, the other rays need to be terminated, so SIMD does not speed up anything.

3.5.2 Compression

The space required to store the visibility can grow fast: it is proportional to the product between the number of samples and the number of VPLs. To let the system work in real-time, it should fit into memory. However, as one can imagine there is a lot of coherence in VPL-visibility between the samples. For example, two neighbouring samples with the same orientation typically share the majority of the VPLs they can “see”. Visual inspection of the visibility encoding bit-vectors of neighbouring samples confirms this. Because of this coherence, compression is expected to yield a good compression ratio. This can be used to reduce memory or disk space. But the decompression should be fast.
We briefly looked into compression for coherent bit-vectors, and devised the following simple scheme that consists of two parts.

First, bit-vectors are clustered. A cluster consists of a single base vector and several delta vectors. For each bit-vector, a delta vector is created, together with a reference to a base vector. The delta vectors encode at which positions the original bit-vector is different from the base vector; the delta vector is the XOR between the bit-vector and the base vector. For the clustering, we used a simple but inefficient approach. For each bit-vector there is searched for a base vector with a Hamming distance of at most $h$. If it does not exist, a new base vector is created that is equal to the bit-vector. Next, the delta vector is computed by taking the XOR between the bit-vector and base vector. Because of the constrained hamming distance, this vector has at most $h$ 1’s. When one chooses $h$ sufficiently small, the number of base vectors will increase, but the number of 1’s in the delta vector will decrease. A bit-vector with a small fraction of 1’s is called a sparse vector.

Second, delta-vectors are compressed. We use a method from the field of database systems named bit-block compression (BC). This method is designed to compress sparse vectors that occur in database indices. It works as follows. First, the bit-vector is split into blocks of $b$ consecutive bits. For each block a signature of variable length is created that consists of at most three parts. The first part is a single bit that is one when the block contains one or more 1’s, or zero when the block consists entirely of zeroes. When there are any 1’s, the remaining parts describe where the 1s are. The second part encodes the number of 1s as a string of $n - 1$ 1’s and a terminating zero, with $n$ the number of 1’s in the block. The third part consists of $n$ indices of 1’s in the block, each index is $\log_2 b$ bits wide.

In Figure 3.6 an example encoding of the bit-vector 0001 0000 0110 0000 for a block size $b = 4$ is shown. With a compression ratio of $13/16 = 0.8125$ this is not very spectacular; in practice larger bit-blocks and sparser vectors yield better ratios.
Chapter 4

Lightcuts

Lightcuts play an instrumental role in our application to approximate the total illumination from many lights. This chapter consists of two parts. The first section, Section 4.1, describes the original Lightcuts algorithm as presented by Walter et al. The second section, Section 4.2, describes our modifications to the Lightcuts algorithm that are specific for our usage of Lightcuts.

4.1 Original Lightcuts Algorithm

Lightcuts [40] is an algorithm to approximate the total illumination coming from many light sources at a strongly sub linear computational cost. It works by grouping lights into clusters as a pre-process. Clusters are kept in a binary tree, the light tree. For the approximation of the illumination, clusters are selected using the tree based on an error bound, until the cluster approximation is below a user-specified error threshold.

In this section, we describe the original Lightcuts algorithm as presented by Walter et al. First, in Section 4.1.1 the algorithm is described. Next, the calculation of error bounds is described in Section 4.1.2. Finally, in Section 4.1.3 the usage of bounding boxes for clusters is described.

4.1.1 Algorithm

In this section, the Lightcuts algorithm is described. First, a high level overview of the algorithm is given. Next is described how lights are clustered. After that, the creation of the light tree is discussed. The last section details how this light tree is used during rendering to approximate illumination from the light sources.

Overview

In a pre-process, lights are grouped into clusters. This is done prior to rendering an image. Illumination from a cluster is approximated using a single light in the cluster, which is the cluster’s representative light. The selection of which light will become the representative light is random, proportional to its intensity.
Figure 4.1: A directional light illuminating a surface point. The light has position \( y \) and its direction of maximum emission is \( m \). The surface point is located at \( x \) where the surface normal is \( n \). The angle between the vector \((y - x)\) and \( n \) is \( \theta \); the angle between the vector \((x - y)\) and \( m \) is \( \phi \).

This is done to make the cluster approximation unbiased in a Monte Carlo way. Properties of the light that becomes the cluster’s representative light (such as position and orientation) are left unchanged, except for the intensity: it becomes the total intensity of all lights in the cluster (i.e. the summation of all light’s intensities in the cluster). A hierarchy of clusters is kept in a binary tree, which is called the light tree.

During rendering, the clusters are used as follows. A heap is used to keep track of the clusters contributing to the illumination. At the start of the algorithm, this heap is initialised to contain only the root of the light tree. Next, the light tree is iteratively descended, and clusters are refined (nodes are descended) where necessary. The set of clusters that are used (are on the heap) is called the light cut. The sum of the approximations from all clusters in the light cut is the total approximation of lighting. An approximation of the illumination coming from a single cluster is calculated using the cluster’s representative light. This approximation has an unknown error, but an upper bound for this error can be computed. A cluster is refined when its error upper bound exceeds a user-configurable fraction of the current total approximated lighting. When a cluster is refined, its contribution is removed from the total approximation of lighting, and its two children are used instead. The algorithm terminates when all clusters are refined below a user-specified threshold, or when the number of used clusters exceeds a user-specified maximum.

Clustering Lights

Although the Lightcuts algorithm supports various kinds of light sources, we focus in our description of Lightcuts on oriented lights. This is because oriented lights are the only light sources used in our application of Lightcuts. Oriented lights have a position \( y \) and a normalised direction or orientation vector \( m \). The emission of light is weighted by a cosine of the angle \( \phi \) between \( m \) and the direction of emission. See Figure 4.1. Thus, the direction of maximum emission is \( m \), where \( \phi = 0 \) so that \( \cos \phi = 1 \). For angles greater than 90°, the emission is zero, i.e. negative values of the cosine are clamped to zero. For readability, we follow the notation of [1] and define the clamped-cosine \( \cos \phi = \max(\cos \phi, 0) \).

For a Lambertian surface location \( x \), the radiance \( L \) from a single oriented
4.1. Original Lightcuts Algorithm

Figure 4.2: An example light tree for a scene with four light sources. (a) A scene consisting of a room containing the ‘Suzanne’ model from Blender and four lights. (b) An example tree built for the light sources. The leaves of the tree are the actual lights. Nodes higher in the tree are clusters of lights; the number in the node designates its representative light. For example, the root node is a cluster that contains all underlying lights and uses light #4 as representative light. (c) A reference rendering of the scene using the direct illumination from the four lights. (Figure based on Fig. 3 in [40])

Light source at position $\mathbf{y}$ is given by:

$$L(\mathbf{x}) = \frac{1}{\pi} \cdot \frac{\cos \phi \cos \theta}{\|\mathbf{y} - \mathbf{x}\|^2} \cdot V(\mathbf{x}) \cdot I,$$

with $I$ its intensity and $V(\mathbf{x})$ a visibility function that is one when the light is visible from $\mathbf{x}$ (i.e. the line $\overline{\mathbf{x}\mathbf{y}}$ does not intersect scene geometry), and zero otherwise. Now for a cluster $\mathcal{C}$ that is a subset of the oriented lights, the illumination is approximated as:

$$L_\mathcal{C}(\mathbf{x}) = \sum_{i \in \mathcal{C}} \frac{1}{\pi} \cdot \frac{\cos \phi_i \cos \theta_i}{\|\mathbf{y}_i - \mathbf{x}\|^2} \cdot V_i(\mathbf{x}) \cdot I_i \quad (4.2a)$$

$$\approx \tilde{L}_\mathcal{C}(\mathbf{x}) = \frac{1}{\pi} \cdot \frac{\cos \phi_j \cos \theta_j}{\|\mathbf{y}_j - \mathbf{x}\|^2} \cdot V_j(\mathbf{x}) \cdot \sum_{i \in \mathcal{C}} I_i. \quad (4.2b)$$

Or less formally stated: the illumination from a cluster $\mathcal{C}$ is approximated by using only its representative light $j$’s properties, but with an intensity that is equal to the total cluster intensity. This intensity is precomputed and stored with the cluster.

Building the Light Tree

Clusters of lights are stored in a binary tree, called the light tree. The tree’s leaves are the individual lights (which can also be thought of as clusters of one light). Nodes of the tree are clusters containing all lights lower in their branch. A cluster’s representative light is always equal to one of the representative lights of its direct children. This increases the performance of the algorithm: results of calculations performed with the representative light can be reused for the child that has the same representative light. An example light tree for a scene with four light sources is shown in Figure 4.2.
For a cluster in the tree, the following information is stored: its two children, its representative light, a bounding volume, an orientation bounding cone (see Figure 4.6) and its total intensity $I_C$.

The light tree is built bottom-up by progressively creating pairs of clusters (individual lights are considered as clusters of one). A naive implementation of the tree building would have a $O(N^3)$ performance, but using an agglomerative clustering algorithm as in [41] results in a sub-quadratic performance. To maximise the quality of the clusters, lights with the greatest similarity in position and orientation are grouped. The following cluster size metric is used:

$$\|C\| = I_C (\alpha_C^2 + c^2 (1 - \cos \beta_C)^2),$$  \hspace{1cm} (4.3)$$

with $I_C$ the cluster intensity (the result of the summation in Equation 4.2b), $\alpha_C$ the diagonal length of the cluster bounding box, $\beta_C$ the half-angle of the cluster’s bounding cone and $c$ a constant to control the scaling between the spatial and directional components, set to the diagonal length of the scene bounding box. [Walter et al.] provide no explanation of this metric, but one can think of $c$ as a factor to normalise the spatial term. The cluster/light pair that minimises this metric is merged into a new cluster, and added to the tree.
Using the Light Tree

During shading, the light tree is used to select a configuration of clusters that are used to approximate the illumination. Clusters are selected in such a way that every tree path from the root to a leaf contains a single node. This selection is referred to as a light cut. An example with three different light cuts is shown in Figure 4.3. The selection of clusters is made for each point where illumination is required, and may vary per point. This is illustrated in the bottom row of Figure 4.3 for some regions one light cut (selection of clusters) yields a lower error than other cuts and vice versa.

The selection of a light cut is done as follows. An initial cut that contains only the root of the tree (a cluster representing all lights) is created. A heap is used to keep track of the nodes (clusters) that are part of the light cut. When a node is added to the cut, its approximate illumination is computed using Equation 4.2b, as well as an error upper bound for it (this is explained further in Section 4.1.2). Then the following is repeated iteratively. The node in the light cut with the largest error upper bound is sought. If its error is above a fraction $\tau$ (e.g. 2%) of the total approximated illumination of the current light cut, the cluster is refined: it is removed from the current light cut (heap), and is replaced by its two children. When the error of the node with the highest error from the cut is below the fraction of the current approximated illumination, the algorithm terminates. The algorithm also terminates when the number of nodes in the light cut exceeds a specified maximum.

The use of a relative error threshold is inspired by Weber’s Law [6], which states that the minimum perceptible change in a signal is roughly equal to a fixed fraction of the original signal. For optical signals this means that for bright regions a larger error is unnoticed than for dark regions. A typical value for error threshold $\tau$ is 2%. Other values can be used to control the trade-off between performance and accuracy.

A pseudocode version of the previous description is shown in Algorithm 1. For generality, individual lights are handled like clusters: for a single light, the difference is that computeEstimate does an exact computation, therefore computeErrorBound returns 0.

4.1.2 Error Bounds

For the error upper bound $\hat{L}_C$ on the approximated illumination $\tilde{L}_C$, the maximum possible illumination from a cluster is used. That is, the illumination from a light positioned and oriented within the constraints of a cluster in such a way that it maximises Equation 4.2b. The intuition behind this is as follows. Both the true and approximate illumination from a cluster are always smaller than the maximum possible illumination. This also means that their absolute difference, the cluster approximation error $\epsilon_C = \|L_C - \tilde{L}_C\|$, is also smaller than $\hat{L}_C$. This can be illustrated further with an example. Say that for some cluster $C$, given its position, size, orientation of its lights and total intensity, the maximum possible illumination $\hat{L}_C$ at some point $x$, equals 10. From this follows that both the (unknown) true illumination $L_C(x)$ and (known) approximate illumination
Algorithm 1: Lightcuts cluster selection

```
input : Surface position x and -normal n; the root of the light tree root;
        relative cluster error threshold \( \tau \in [0, 1] \)
output : An estimate of the illumination at x

heap ← emptyHeap()  
Lest ← computeEstimate(root, x, n)  
add(heap, root)

while sizeOf(heap) < maxcutsize do
    worstnode ← getMaxError(heap)  
    \( \hat{L}_C \) ← computeErrorBound(worstnode, x, n)  
    if \( \hat{L}_C < Lest \cdot \tau \) then  
        return Lest
    else  
        // remove the contribution from the worst node
        removeMax(heap)  
        Lest ← Lest - computeEstimate(worstnode, x, n)  
        // include its two children
        left ← leftChild(worstnode)  
        right ← rightChild(worstnode)  
        Lest ← Lest + computeEstimate(left, x, n)  
        Lest ← Lest + computeEstimate(right, x, n)  
        add(heap, left)  
        add(heap, right)
    end
end
return Lest
```
4.1. Original Lightcuts Algorithm

Visibility Term Upper Bound

Visibility term $V_i$ for light $i$ has a value of zero for an invisible light, and one when visible. For a cluster, visibility upper bound $\hat{V}_C$ should be zero when none of the lights in the cluster is potentially visible, or one otherwise. As this is a hard problem to conservatively compute for scenes in general—without actually testing for occlusion—a trivial upper bound of one is used. We will come back to this in Section 4.2 where we discuss our modifications to the Lightcuts algorithm.

Material Term Upper Bound

The material term models the behaviour of light at the receiver side of a light’s illumination. It contains the $\cos \theta_i$ (or $n \cdot l$) factor that is used to obtain irradiance measured at a surface, and the material’s BRDF. In our application we use only Lambertian surfaces, therefore the material term $M_i$ for a directional light $i$ is:

$$M_i(x) = \frac{\cos \theta_i}{\pi}.$$ (4.6)

To find the material upper bound $\hat{M}_C$ for a cluster, the maximum value for $\cos \theta$ is sought, with $\theta$ the angle between surface normal $n$ and a vector to any point in the cluster’s volume. [Walter et al.] use the following approach. To simplify the computation, first a transformation is applied so that the surface normal $n$ matches the z-axis, and that the surface point $x$ is at the origin. This is shown in Figure 4.4(a). From trigonometry it is known that for a right-angled triangle $\cos \theta = \text{adjacent} / \text{hypotenuse}$, or that—after the transformation—for any point:

$$\cos \theta = \frac{p_x}{\sqrt{p_x^2 + p_y^2 + p_z^2}}.$$ (4.7)

This is shown in Figure 4.4(b). Next, to maximise Equation 4.7, the numerator is replaced by the maximum value of $p_x$ that is possible within the bounding box. The remaining components $p_y^2$ and $p_z^2$ are chosen to minimise the denominator.
4. Lightcuts

for a positive \( \max(p_z) \), or to maximise it otherwise:

\[
\cos \theta \leq \begin{cases} 
\frac{\max(p_z)}{\sqrt{\min(p_x^2) + \min(p_y^2) + (\max(p_z))^2}}, & \text{if } \max(p_z) \geq 0, \\
\frac{\max(p_z)}{\sqrt{\max(p_x^2) + \max(p_y^2) + (\max(p_z))^2}}, & \text{otherwise}.
\end{cases}
\] (4.8)

So that:

\[
\cos \theta \leq \begin{cases} 
\max(p_z) \sqrt{\min(p_x^2) + \min(p_y^2) + (\max(p_z))^2}, & \text{if } \max(p_z) \geq 0, \\
0, & \text{otherwise}.
\end{cases}
\] (4.9)

Note that the position of the square (\(^2\)) is important for the unary min and max operators. For example if \( p_z \in [-3, 2] \), then \((\max(p_z))^2 = 4\) but \(\max(p_z^2) = 9\).

Geometric Term Upper Bound

In the geometric term the emission distribution of lights is modelled, as well as the distance fall-off. For an omnidirectional light it would simply be equal to the reciprocal of the squared distance to the light. In our application in which only directed lights are used, the geometric term \( G_i \) for light \( i \) is:

\[
G_i(x) = \cos \phi_i \| y_i - x \|^2.
\] (4.10)

This means that for an upper bound of the geometric term for a cluster \( \hat{G}_C \), the following two parts must be calculated: 1. the minimum distance between \( x \) and the cluster (the denominator of Equation 4.10), and 2. the minimum angle between a vector from \( x \) and any orientation vector of lights inside the cluster (the numerator).

The first part is the least complicated. The minimum distance from a point \( p \) to an axis-aligned bounding box (AABB) can be found by taking for each dimension separately the shortest distance from the point to the AABB’s interval (min, max), squaring and summing these per-dimension distances and finally taking the square root of the sum. This is illustrated with three examples in Figure 4.5. First, in Figure 4.5(a), the point is inside the box, so for each dimension the point is inside the interval. Second, in Figure 4.5(b) the point is outside the interval \([\min x, \max x] = [-2, 5]\), and \( p_x = -4 \). The squared distance from \( p_x = -4 \) to \( \min x = -2 \) is 4. Taking the square root yields a final distance of 2. Third, in Figure 4.5(c) the point has the same distance in the x-dimension as the previous example, but is now also outside the interval \([\min y, \max y] = [-1, 3]\) in the y-dimension. The squared distance from \( p_y = 6 \) to \([\max y = 3]\) is 9. Now the final distance is \(\sqrt{2^2 + 3^2} = \sqrt{13}\).

The second part, finding an upper bound for the numerator of Equation 4.10, is more complicated. For the orientations of the lights in the cluster, a bounding
4.1. Original Lightcuts Algorithm

Figure 4.5: Finding the minimum distance from a point (green) to a bounding box with \( \textbf{min}=(-2, -1) \) and \( \textbf{max}=(5, 3) \).

Figure 4.6: Bounding the direction of oriented lights with a bounding cone. (a) Three oriented lights with a similar orientation (direction of maximum emission). (b) A bounding cone is constructed in such a way that the cone contains all orientation vectors. In this example, a 2D version of a bounding cone for the three lights is shown. The cone is defined by its axis \( c \) and its half-angle \( \beta_C \); its boundaries are marked with the bold green lines. \( \rho=\beta_C+90^\circ \). Lights 1 and 2 are at the boundaries; light 3 is contained in the cone.

The cone is stored for each cluster. This is shown in Figure 4.6. Using the cone, an upper bound can be computed for the lights without considering the lights individually. For an angle \( \phi \) between a vector to a surface point and \( c \), the following can be concluded for \( \cos \phi \). When \( 0 \leq \phi \leq \beta_C \), i.e. the vector is in the cone, the upper bound is one. When \( \phi>\rho \), i.e. the vector is in the dashed area, the upper bound is zero. Otherwise, the upper bound is given by \( \cos(\phi-\beta_C) \).

Figure 4.7(a) shows an example cluster with oriented lights and its bounding cone (the cone is shown multiple times for illustration purposes). For point \( x \), the smallest angle \( \phi \) is sought between a vector from a point in the volume towards \( x \) and the cone’s axis \( c \). Walter et al. made the following observation. First, for all light positions \( y_i \in C \) a new bounding box is created that contains the points \( (x-y_i) \). This box has the same size as the original, but it is mirrored. Next, a transformation is applied so that the axis of the cluster’s bounding cone aligns with the z-axis. The result is shown in Figure 4.7(b), which can be solved in the same way as the material upper bound: the equivalent for angle \( \theta \) of \( M_i \) is now \( \phi \) (which are both to be minimised); \( n \) of \( M_i \) corresponds to \( c \).
4. Lightcuts

Figure 4.7: Finding the minimum angle for oriented lights in a cluster. (a) For a point \( x \), the minimum angle \( \phi \) is sought for a cluster (dashed gray) and its light orientation bounding cone (shown at the corners with bold green for illustration purposes; note that there exist actually only a single cone per cluster and that bounding cones do not have any property like position). (b) After applying the transformation (see text), the problem has become very similar to the bounding of the material term. (Image after Fig. 5 in [40])

4.1.3 Transformation of Axis-Aligned Bounding Boxes

For the error upper bound calculation of the material and geometric terms, cluster bounding boxes must be transformed (rotated and translated). Translation of a bounding box that is described by a minimum and maximum vector is trivial: one simply adds the amount of translation to both vectors. Rotation is more complicated. To our knowledge there are two possible ways. One is to transform the contents of the box one by one, and compute a new AABB over the result. The other is to transform the box, and compute a new AABB over it. Both methods are illustrated in Figure 4.8. From this figure, although exaggerated for illustration purposes, it can be seen that it can make a major difference in the tightness of the resulting bounding box in which way it is computed. Its tightness has a direct effect on the tightness of the geometric and material term upper bounds, which in their turn steer the refining of clusters. Oversized bounding boxes lead to unnecessarily refining of clusters and wasted computation time.

On the one hand, when a bounding box is computed for the transformed contents, as in Figure 4.8(b), the resulting box is as tight as possible. However, this comes at a cost: each point inside the initial box must be transformed using a matrix-vector multiplication. For clusters that contain hundreds of points, this most likely does not weigh up to the improved accuracy of the resulting box. For these clusters, the transformed AABB is likely to be decent.

On the other hand, a bounding box that is computed around another transformed bounding box, as in Figure 4.8(c), is always equal to or larger than the initial bounding box. This leads inevitably to overestimation of the error upper bounds. However, it has the advantage that it is reasonably cheap to compute—regardless of the contents of the box [10] page 86].

From the Lightcuts publication it is not clear how the transformation of cluster bounding AABBs is performed. Correspondence with the authors provided...
4.2 Modifications to the Lightcuts Algorithm

In this section, our modifications to the Lightcuts algorithm are described. The first one, described in Section 4.2.1, is very specific to our application: precomputation of visibility. This results in a visibility upper bound, which is originally left unbounded. The second one is described in Section 4.2.2, and is more fundamental: cluster bounding boxes are replaced with bounding spheres. This eliminates the problem of the transformation of cluster bounding boxes; spheres are invariant to rotation and are trivial to translate. In Section 4.2.3, our modifications to the cluster size metric are described.

4.2.1 Taking Advantage of Precomputed Visibility

Despite all the effort that has been put into optimising ray versus geometry intersection checks, it is still an expensive computation. For coherent rays, packet-traversal \[28, 35\] can result in a significantly faster computation. However, in our application their is little coherence between the rays. Also, due to the nature...
of the Lightcuts algorithm, ray queries happen on a ray-per-ray basis. Potential
generation of new rays is based on the results of previous rays, which makes
grouping them into packets difficult.

Our solution to reduce the cost of the visibility check is to compute it in a
pre-process stage. This is possible because the geometry, and therefore the lights
(VPLs on the geometry), are assumed to be static. For each sample point, the
visibility of all lights is computed and stored with the sample point. A unique
integer \( i \in [0, N_{vpl}] \) is assigned to all \( N_{vpl} \) lights. The visibility can be encoded
as a bit vector of \( N_{vpl} \) bits wide, with \( i \) the index for the visibility bit. This bit
vector is stored for each sample point.

A Visibility Upper Bound

Consider the tree in Figure 4.9. The lights of nodes 4, 5, 6 and 10 are invisible.
As a result, clusters 2 and 3 are not visible either. When considering only
the visibility of lights, the Lightcuts algorithm would process node (cluster) 2
in the following way. The total cluster illumination estimate would have the
(correct) value of zero: none of the lights (4, 5, 6) is visible, so the representative
light is not visible either. However, the error bound would significantly be
overestimated, since a visibility upper bound \( \hat{V}_C \) of one is used. Hence, although
the entire cluster is invisible, its error bound is calculated as if all lights were
fully visible.

To compute tighter error bounds, the visibility information is exploited
further. Visibility is not only stored for leafs (lights) in the tree, but also for
clusters. A value of zero indicates that none of the lights in the cluster is visible,
a value of one means that one or more lights are visible. In fact, this is the
visibility upper bound \( \hat{V}_C \). When nodes are numbered depth-first, the visibility
for node \( i \) is:

\[
V_i = \begin{cases} 
\bigcup_{j=i+1}^k V_j, & \text{(clusters)} \\
v[i], & \text{(lights)} \end{cases}
\]

(4.11)

with \( k \) the highest node number in the sub-tree below \( i \), and \( v[i] \) the \( i \)-th element
of light visibility vector \( v \).

Now with the visibility of entire clusters available at a very low cost (i.e. a
bit test operation), this information can be used to accelerate Lightcuts in two
ways. Again, consider the traversal of the example tree in Figure 4.9. When
node 1 is to be refined, the entire branch of node 2 and lower can be discarded;
none of the lights in the cluster of node 2 will contribute to the illumination. So
instead of adding node 1’s two children to the cut, only node 7 is added. We
call this branch rejection. Another situation occurs when node 7 is to be refined.
Normally, nodes 8 and 11 would have been added to the cut. However, node 8
has only one visible child. Because of this, it is lucrative to directly skip node 8,
and add node 9 instead. Proceeding directly to lower children when only one of
them is visible has two advantages. First, it saves computation time; when not
proceeding directly to a child the node might get refined later anyway. When
proceeding to a child directly, this computation is skipped entirely. Second,
the error bounds for children are always lower than the error bounds of their
parents.


4.2. Modifications to the Lightcuts Algorithm

4.2.2 Using Bounding Spheres instead of Boxes

As we have seen in Section 4.1.3, the use of rectangular bounding boxes is not optimal, as the transformation of the bounding boxes for the error bound computation either introduces a recalculation of the bounding volume that is less tight than the original bounding boxes, or otherwise it requires a transformation of each element in the box which is expensive for boxes that contain many elements. The bounding volume transformation is important for the time performance of the Lightcuts algorithm, because of two reasons. First, the accuracy of the error bound computations directly depends on the accuracy of the bounding volume. Overestimating the error might lead to unnecessarily extension of the cut, so the bounding volume should expand as little as possible. Second, the computation of the error bounds is performed inside a tight loop. For each expansion of the cut, error bounds are recalculated. Therefore, the constant-time performance of the bounding volume transformation has a significant impact on the total constant-time performance.

Because of the importance of the bounding volume transformation for the time performance and bounding error tightness, we considered bounding spheres as an alternative for AABBs. The main reason is that this eliminates the rotation part of the bounding volume transformation; a sphere’s radius is unchanged when it is rotated around its centre. Then only the translation part remains, which is as simple as a single vector addition/subtraction from the sphere’s centre.

To our knowledge, the idea of using bounding spheres for Lightcuts is not described in literature. We therefore derive the new upper bound formulae for the material and geometric terms in the following sections.

Material Term Upper Bound for Bounding Spheres

For the computation of the material term upper bound \( \hat{M}_C \) (described in Section 4.1.2) using bounding spheres, the following three different conditions are distinguished:

1. The sample point is inside the volume of the bounding sphere. In this
4. Lightcuts

Figure 4.10: Finding a point \( y \) in the bounding sphere of cluster \( C \) that minimises the angle \( \theta \) between \( n \) and \( xy \). From the image can be concluded that \( y \) is to be sought at the sphere’s surface: any point \( y \) that makes \( xy \) intersect the sphere—instead of being tangent to its surface—results in a non-minimal \( \theta \).

case, the minimum value of angle \( \theta \) is zero, and hence the upper bound is one:

2. The sphere is in front of the sample point. That is, a line with the sample point as origin and the point’s normal vector as direction intersects the sphere. In this case the minimum value of angle \( \theta \) is zero as well, so the upper bound is also one;

3. None of the previous two is true. In this case, the computation described in the remainder of this section is performed.

An illustration of the problem is shown in Figure 4.10. For convenience, the sample point \( x \) is translated so that it is on the origin. It has a normal vector \( n \). A sphere bounding the cluster of interest is defined by its centre \( c \) and radius \( r \). The unknown point \( y \) in the cluster’s volume that minimises the angle between \( n \) and a vector from the origin to \( y \) is sought at the surface of the sphere (Equation 4.12a). The line from the origin to \( y \) is a tangent line, therefore it is perpendicular to the normal of the sphere’s surface at \( y \) Equation 4.12b:

\[
\|y - c\| = r, \quad \text{(4.12a)}
\]
\[
(y - c) \cdot y = 0. \quad \text{(4.12b)}
\]

In space, the set of solutions for this system of equations is infinite. All lines from the origin to the points that satisfy these equations form a cone, with its axis passing through \( c \). To reduce the number of solutions to two, we restrict \( y \) to be on the plane spanned by \( n \) and \( c \):

\[
y = un + vc. \quad \text{(4.13)}
\]

Care has to be taken in the situation where \( n \) and \( c \) are collinear. If so, \( y \) is restricted to be on a line instead of a plane, and solutions to Equation 4.12 cannot be found. However, this situation does not occur. When \( n \) and \( c \) are collinear, the sphere is right in front or behind of the normal. The first case cannot occur, as it is already handled by the second condition described at the
4.2. Modifications to the Lightcuts Algorithm

beginning of this section (page 35). The second case can also not occur: when the sphere’s centre \( c \) is exactly behind the sample point, then all lights in the cluster are behind the sample point too, as the sample point is not inside the sphere because of the first condition at the beginning of this section. When all lights are behind the sample point they are invisible, so such a cluster is not processed anyway.

Equation [4.12b] can be rewritten as follows:

\[
(y - c) \cdot y = 0 \iff y \cdot y - c \cdot y = 0 \iff y \cdot y = c \cdot y, \tag{4.14}
\]

and used in the expansion of Equation [4.12a]

\[
\|y - c\| = r \\
\iff (y - c) \cdot (y - c) = r^2 \\
\iff y \cdot y - 2c \cdot y + c \cdot c = r^2 \tag{4.15}
\]

to obtain the following equations:

\[
c \cdot y = a, \tag{4.16a}
\]
\[
y \cdot y = a, \tag{4.16b}
\]

with constant \( a = c \cdot c - r^2 \) for compact notation. When the plane restriction of Equation [4.13] is applied by substituting it into Equation [4.16a] \( v \) can be expressed in terms of \( u \):

\[
c \cdot (un + vc) = a \\
\iff uc \cdot n + vc \cdot c = a \\
\iff v = \frac{a - uc \cdot n}{c \cdot c}. \tag{4.17}
\]

Note that expressing \( u \) in terms of \( v \) is possible too, however, this results in a fraction with \( n \cdot c \) as denominator, which can be zero—contrary to \( c \cdot c \).

\[^1c \cdot c \] can actually be zero, but only when the centre of the sphere is on the origin, which means that the sample point is inside the sphere. This condition is already handled.
Next, by substitution of Equation 4.13 this time into Equation 4.16b and then substituting all occurrences of \( v \) with Equation 4.17 makes:

\[
(u_n + vc) \cdot (u_n + vc) = a \\
\iff \\
u^2 n \cdot n + 2 u n \cdot c + v^2 c \cdot c = a \\
\iff \\
(1 - \frac{(n \cdot c)^2}{c \cdot c}) u^2 = a - \frac{a^2}{c \cdot c} \\
u^2 = \frac{a^2 - 2u n \cdot c + u^2(n \cdot c)^2}{(c \cdot c)^2} c \cdot c = a \\
\iff \\
(1 - \frac{(n \cdot c)^2}{c \cdot c}) u^2 = a - \frac{a^2}{c \cdot c} \\
u^2 = \frac{a - \frac{a^2}{c \cdot c}}{1 - \frac{(n \cdot c)^2}{c \cdot c}} = \frac{a c \cdot c - a^2}{c \cdot c - (n \cdot c)^2} = \frac{r^2(c \cdot c - r^2)}{c \cdot c - (n \cdot c)^2}, \quad (4.18)
\]

so the solutions for \( u \) are given by:

\[
u = \pm \sqrt{\frac{r^2(c \cdot c - r^2)}{c \cdot c - (n \cdot c)^2}}, \quad (4.19)
\]

The argument of the square root must be positive for (real) solutions for \( u \). This is always the case: the numerator of the fraction in Equation 4.19 is always positive by construction, as \( c \cdot c - r^2 \), the distance from the origin to the surface of the sphere, is positive. Now only problems can arise when the denominator is zero or negative. Using the geometric interpretation of the dot product shows that the denominator is always positive. Recall that the dot product between vectors \( a \) and \( b \) is equal to \( \|a\| \|b\| \cos \theta \), with \( \theta \) the angle between them. Rewriting the denominator makes \( \|c\|^2 - \|n\|^2 \|c\|^2 \cos^2 \theta = \|c\|^2 - \|c\|^2 \cos^2 \theta \). Therefore, the denominator cannot be negative (\( \cos^2 \theta \leq 1 \)) or zero (since \( c \) and \( n \) are not allowed to be parallel).

With the solutions for \( u \), we can obtain \( v \) from \( u \) using Equation 4.17. Then with \( u \) and \( v \) available, \( y \) is given by Equation 4.13, which gives:

\[
\hat{M}_C = \frac{n \cdot y}{\|y\|}, \quad (4.20)
\]

where the division by \( \|y\| \) takes care of the fact that \( y \) is not normalised.

Now one question remains: Equation 4.19 provides two solutions for \( u \), but only one leads to the correct value for \( M_C \). One option is to simply try both values of \( u \) to get two values of \( y \), and then take the largest value of \( \hat{M}_C \). Although this is a valid approach, it is not optimal. A closer inspection of the various equations shows that this can be done more efficiently.

To find out how \( u \) influences the value of Equation 4.20, we first observe that the length of \( y \), \( \|y\| \), is constant for both values of \( u \). This can be shown
by plugging Equation 4.17 into Equation 4.13, expanding the dot product and doing some rewriting:

\[
\|y\|^2 = y \cdot y \\
= (u_n + v_c) \cdot (u_n + v_c) \\
= u_n^2 \cdot n + v_c^2 \cdot c + 2u_n v_c \cdot n \\
= u_n^2 + \frac{(a - u_c \cdot n)^2}{c \cdot c} + 2u_n a - u_c \cdot n \cdot c \cdot n \\
= u_n^2 + \frac{a^2 - u^2 (c \cdot n)^2}{c \cdot c}.
\]

This shows that the length of \(y\) is only a function of squared values of \(u\), so both the positive and negative versions of \(u\) in Equation 4.19 give the same values for \(\|y\|\).

With the knowledge that the denominator of Equation 4.20 is constant in terms of the sign of \(u\), we only have to show how \(u\)'s sign maximises the numerator. This can be done by expressing \(v\) in terms of \(u\) using Equation 4.17 and using this in Equation 4.13:

\[
n \cdot y = u_n \cdot n + \frac{a - u_c \cdot n}{c \cdot c} \cdot c \cdot n \\
= u + \frac{a}{c \cdot c} \cdot c \cdot n - u \frac{c \cdot n}{c \cdot c} \cdot c \cdot n \\
= \frac{ac \cdot n}{c \cdot c} + (1 - \frac{(c \cdot n)^2}{c \cdot c})u.
\]

From this we can see that the largest value of \(n \cdot y\) can be obtained when a positive version of \(u\) is used when \((1 - \frac{(c \cdot n)^2}{c \cdot c}) > 0\) and vice versa. However, by using the geometric interpretation of the dot product we can show that the last term cannot be negative at all:

\[
c \cdot n = \|c\| \|n\| \cos \phi_{cn} \\
\Rightarrow \\
(c \cdot n)^2 = \|c\|^2 \cos^2 \phi_{cn} \\
\Rightarrow \\
1 - \frac{(c \cdot n)^2}{c \cdot c} = 1 - \frac{\|c\|^2 \cos^2 \phi_{cn}}{c \cdot c} = 1 - \cos^2 \phi_{cn} \geq 0,
\]

which means that multiplication with a positive \(u\) always leads to the largest dot product between \(n\) and \(y\), and therefore to the correct \(\hat{M}_C\). Although it might appear as a minor optimisation, note that directly selecting the correct (positive) \(u\) saves several arithmetic operations and comparison/branch instructions that otherwise would have been unnecessarily executed hundreds of times per sample point.

**Geometric Term Upper Bound for Bounding Spheres**

For the computation of the geometric term upper bound \(\hat{G}_C\) (described in Section 4.1.2) using bounding spheres, the same approach for the bounding box version is taken.
First, the minimum distance from sample point $x$ to the cluster ($G_C$’s denominator) is calculated. This is easy for a bounding sphere: one computes the distance to its centre $\|x - c\|$ and subtracts the sphere’s radius $r$. The result is the minimum distance.

Next, an upper bound for $\cos \phi$ has to be found. We can do this in the same fashion as for the bounding box version by transforming the problem in such a way that it can be solved using the mathematics that have been established for the material term.

Consider Figure 4.11. The original problem is shown in Figure 4.11(a). All points $x$ for which a line with direction $-m$ and origin $x$ intersects the sphere have an upper bound of one. This is because all points that are both on the line and inside the sphere have an angle $\phi$ of zero. The hatched area indicates this region. For points $x$ outside the region, the situation is a bit more complicated. Figure 4.11(b) shows the result of the following transformation: the sphere is moved so that its centre is at $x$. Now a point $y'$ can be found for which the angle $\phi$ between $cy'$ and $m$ is the same. Point $y'$ can be found using the same mathematics that are used for the material term, by using $m$ instead of surface normal $n$ and $c$ as sample point $x$.

Now $\phi$ is obtained, the same procedure as for the box version is followed. When $0 \leq \phi \leq \beta_C$, i.e. the vector is in the cone, the upper bound is one. This is because all directions of the lights in the cluster fall inside the cone, so a light’s direction of maximum emission can point directly at $x$ when $\phi$ is in the cone. Otherwise, the upper bound is given by $\cos(\phi - \beta_C)$.

In Algorithm 4 the algorithm for the computation of the product of the
4.2. Modifications to the Lightcuts Algorithm

geometric and material upper bound is shown. It roughly works as follows. On line 4 a trivial total upper bound of one is returned when the point \( \mathbf{x} \) is in the volume of the bounding sphere. Then the lines 5–24 perform the computation of the geometric upper bound. The check if the bounding sphere is in front of the light is performed on line 7. This same line also checks if \( \cos \beta_{C} \) is \(-1\): this happens when the light orientations in the cluster deviate so much that the ‘cone’ captures all possible directions. If any of these two conditions is true, the numerator of the geometric upper bound \( \hat{G}_{\text{num}} \) is set to one. Otherwise, the cosine of \( \phi \), \( c_{\phi} \), is computed. When it falls inside the cone (line 14), \( \hat{G}_{\text{num}} \) is set to one. Otherwise, another test is performed at line 17: when the cone is very narrow (this happens for smaller clusters of lights on flat surfaces) and \( c_{\phi} \) is smaller than zero, the entire cluster can be rejected. When this is not the case, \( \hat{G}_{\text{num}} \) is computed by subtracting the angles and taking the cosine. The final step for the geometric upper bound is to compute its denominator at line 24; the squared distance to the sphere, or \( d^2 \). Next, the material term upper bound is computed. At line 27 is tested if the cluster is in front of the surface point. If this is true, the material upper bound is one, and the geometric upper bound (times one) is returned as total bound. Otherwise, \( \cos \theta \) is computed on the remaining lines. Finally, the product of the two terms is returned: \( \hat{M} \) times \( \hat{G}_{\text{num}}/d^2 \) (the full geometric term). The correctness of the C++ equivalent of this algorithm has been verified numerically by computing the exact upper bounds of several thousands of clusters with brute force and comparing the results to the output of this algorithm.

4.2.3 Modifications to the Cluster Size Metric

Our application of Lightcuts and our usage of bounding spheres require two changes to the cluster size metric. We repeat the original metric here for convenience:

\[
||\mathbf{C}|| = I_{C}(\alpha_{C}^2 + c^2(1 - \cos \beta_{C})^2),
\]

with \( I_{C} \) the cluster intensity, \( \alpha_{C} \) the diagonal length of the cluster bounding box, \( \beta_{C} \) the half-angle of the cluster’s bounding cone and \( c \) a constant to control the scaling between the spatial and directional components, set to the diagonal length of the scene bounding box.

The first change is due to the static light tree. We keep the light tree static because of two reasons. The first one is that rebuilding the tree each frame is expensive in general, and prohibitive to do this for each frame. The second is because of the visibility upper bound discussed in Section 4.2.1 if the light tree changes, then for each sample point the visibility information for clusters needs to be propagated from the leaves upwards to the root. This static light tree is built in a pre-process. At that moment the intensity of the lights (which are VPLs) is not known: they are a function of primary (real) light sources, and can have any value. Because of this, \( I_{C} \) is not available, and we simply remove it from the original metric.

The second change is a result of the replacement of bounding boxes with bounding spheres. The radius of the bounding sphere is used instead of the
Algorithm 2: Total cluster upper bound for bounding spheres.

\[\text{input} : \text{Surface position } \mathbf{x} \text{ and normal } \mathbf{n}; \text{ sphere centre } \mathbf{c}_{\text{sphere}} \text{ and radius } r; \text{ cone axis } \mathbf{m} \text{ and the cosine of its half-angle } \cos \beta_C\]

\[\text{output} : \text{The product of the geometric and material upper bounds terms}\]

1. \( \mathbf{c} \leftarrow \mathbf{c}_{\text{sphere}} - \mathbf{x} \)
2. \( l_c \leftarrow \mathbf{c} \cdot \mathbf{n} \)
3. \( a \leftarrow l_c - r^2 \)
4. if \( a \leq 0 \) then return 1
   // Compute geometric upper bound
5. \( c_m \leftarrow \mathbf{c} \cdot \mathbf{m} \)
6. \( t_G \leftarrow l_c - c_m^2 \)
7. if \( (c_m \leq 0 \lor l_c \leq r^2) \land (t_G \leq r^2) \lor \cos \beta_C = -1 \) then
8. \( \hat{G}_{\text{num}} \leftarrow 1 \)
9. else
10. \( u \leftarrow \sqrt{a \cdot r^2 / t_G} \)
11. \( v \leftarrow (a + u \cdot c_m) / l_c \)
12. \( y_G \leftarrow u \mathbf{n} - v \mathbf{c} \)
13. \( c_\phi \leftarrow (u - v \cdot c_m) / \|y_G\| \)
14. if \( c_\phi \geq \cos \beta_C \) then
15. \( \hat{G}_{\text{num}} \leftarrow 1 \)
16. else
17. if \( \cos \beta_C > 1 - \epsilon \land c_\phi \leq 0 \) then
18. \( \text{return 0} \)
19. else
20. \( \hat{G}_{\text{num}} \leftarrow \cos(\arccos c_\phi - \beta_C) \)
21. end
22. end
23. end
24. \( d^2 \leftarrow (\sqrt{l_c} - r)^2 \)
   // Compute material upper bound
25. \( c_n \leftarrow \mathbf{c} \cdot \mathbf{n} \)
26. \( t_M \leftarrow l_c - c_n^2 \)
27. if \( (c_n \leq 0 \lor l_c \leq r^2) \land (t_M \leq r^2) \) then return \( \hat{G}_{\text{num}} / d^2 \)
28. \( u \leftarrow \sqrt{a \cdot r^2 / t_M} \)
29. \( v \leftarrow (a + u \cdot c_n) / l_c \)
30. \( y_M \leftarrow u \mathbf{n} - v \mathbf{c} \)
31. \( c_\theta \leftarrow u - v \cdot c_n \)
32. if \( c_\theta \leq 0 \) then return 0
33. \( \hat{M} \leftarrow c_\theta / \|y_M\| \)
34. return \( (\hat{G}_{\text{num}} \cdot \hat{M}) / d^2 \)
4.2. Modifications to the Lightcuts Algorithm

Bounding box diagonal. This leads to the new cluster size metric:

\[ ||C'|| = r_{C}^2 + r_{scn}^2 (1 - \cos \beta_{C})^2, \quad (4.25) \]

with \( r_{C} \) the radius of the cluster bounding sphere, \( \beta_{C} \) the half-angle of the cluster’s bounding cone and \( r_{scn} \) the radius of the scene bounding sphere.
Chapter 5

Experiments

In this chapter, the performance of our system is investigated. Three scenes are used during our experiments, these are described in Section 5.1. In Section 5.2 the time performance of our system is considered. The scaling as a function of the number of VPLs is evaluated, and the constant time performance of our sphere approach is compared with a box version. Next, in Section 5.3 the error with respect to the brute-force reference solution is shown using difference images, for different values of error threshold $\tau_e$. After that, the visual effects of changing the density of the sample point set and VPL set are described in Section 5.4 and Section 5.5 respectively.

5.1 Scenes Used for the Experiments

For the experiments, three different test scenes are used. These are shown in Figure 5.1. The scenes are created to have different properties:

![Buddha](image1)
![Maze](image2)
![Colourful](image3)

(a) Buddha. (b) Maze. (c) Colourful.

Figure 5.1: An overview of the scenes used for the experiments. (a) **Buddha**: a rectangular room with in the centre a Buddha statue; (b) **Maze**: a set of connected corridors that form a small maze; (c) **Colourful**: a coloured room with two alcoves, the upper one contains a dragon and the lower a dodecahedron. The main room is filled with cubes and a spiral.
5. Experiments

- The *Buddha* scene is designed to function as a simple test scene to showcase the effect of colour bleeding. The Buddha statue and walls are white, except for the two longest walls, which are red and blue. These two walls provide all colour in the scene through indirect lighting. The scene consists of 100k triangles.

- Contrary to the Buddha scene, the *Maze* scene is designed so that from any point in the scene much of the remaining of the scene is occluded. This is an interesting property as our visibility information can exploit this. Furthermore, it is practical for testing, as its low polygon count of 336 triangles reduces precomputation times.

- Where the previous two scenes are rather specific, the *Colourful* scene is created to approach the complexity of real world scenes. It has regions with low and high occlusion, objects with low and high curvature (the cubes and spiral respectively) and objects with low and high polygonal complexity (the dodecahedron and dragon). The total number of triangles in the scene is 27k.

The Buddha statue in the Buddha scene and dragon in the Colourful scene are from The Stanford 3D Scanning Repository [27].

In the experiments, illumination is provided by a single white spot light with a 180° fall-off. This light rotates around its vertical axis; its position is a sine of $t$ with such an offset, amplitude and phase so that it slowly moves back and forth between the extremes of the scene.

Six renderings of the test scenes are shown in Figure 5.2. Figures 5.2(a)–(c) are rendered with direct illumination only, as Arauna originally did. Shadowed areas are completely black, since only light paths between the light source and surfaces is simulated. Figures 5.2(d)–(f) are rendered including indirect illumination as provided by our system. The effect of indirect lighting is the clearest in the Buddha scene. The colours of the walls bleed onto the statue, giving it reddish and blueish shades. Light now also arrives at places that are not directly lit by the light source. This is not only aesthetically pleasing, it is also practical. Where in Figure 5.2(a) most of Buddha’s feet were invisible due to shadow, they appeared in Figure 5.2(d) due to indirect lighting. The same effect happens in the Maze scene in (Figure 5.2(b), Figure 5.2(e)) where the corridor becomes illuminated through indirect lighting, and in the Colourful scene (Figure 5.2(c), Figure 5.2(f)) where the sides of the cube in the centre become visible. Also note the colour bleeding in the Colourful scene on the dodecahedron. The wall facing away from the camera and towards the dodecahedron is red, accounting for the reddish and purplish (in combination with the blue wall on the right side) shades on it.
5.1. Scenes Used for the Experiments

(a) Buddha, direct lighting.

(b) Maze, direct lighting.

(c) Colourful, direct lighting.

(d) Buddha, direct + indirect lighting.

(e) Maze, direct + indirect lighting.

(f) Colourful, direct + indirect lighting.

Figure 5.2: Renderings of the test scenes. The top row is direct lighting only; in the bottom row our system added indirect illumination.
5. Experiments

5.2 Time Performance

In this section, the time performance of our system is evaluated. The computation time depends on the distribution of intensity over the VPLs, which is a function of the primary light source(s). This is shown in Section 5.2.1. Next, in Section 5.2.2 the time performance as a function of the number of VPLs is evaluated. In the original Lightcuts paper, boxes are used as bounding volumes. In Section 5.2.3 we compare the performance of spherical bounding volumes as used in our system with that of a box implementation.

5.2.1 Time Performance as a Function of the Light Position

Because of the relative error threshold used by the Lightcuts algorithm (see Section 4.1.1), there is a direct relation between the distribution of intensity over the VPLs and the size of the cut through the light tree that satisfies the error threshold. On its turn, the number of iterations in the Lightcuts algorithm is linked to the size of the cut, and therefore to the time required to compute a single approximation of illumination for a sample. The intensities of the VPLs are a function of the intensity, position and orientation of the real light source(s).

To get an impression of how much time is spent on the Lightcuts computation for different real light source configurations, we placed an oriented light that is translated and rotated over time in the Colourful scene. Its position above the ground is kept fixed about halfway between the floor and ceiling; its position over the x-axis (the line between the dragon and dodecahedron) is set to \( \sin(t) \). Its orientation is set so that it makes a full rotation over the axis orthogonal to the floor every twenty seconds. For each frame, we have collected the total time spent at the Lightcuts algorithm to compute all point set samples, for twenty seconds. We repeated this for two VPL sets: one of 617 VPLs and one of 6506 VPLs. For this scene, this can be considered as two extremes: 617 VPLs lead to noticeable artefacts, more than 6506 VPLs do not lead to perceptible improvement. In Figure 5.3 two graphs of the data are shown.

Three conclusions can be drawn from the graphs. The first is that both graphs follow approximately the same pattern. If one light configuration leads to a relatively high computation cost for some number of VPLs, one can safely assume that for another number of VPLs the configuration will also require a relatively high computation cost. The second is that the maximum required time is about three times as large as the minimum required time, for both number of VPLs. The third is that there is no linear relationship between the computation cost and the number of VPLs. The raw data shows that at the minimum at \( t = 11.5s \) the tenfold of VPLs requires \( \sim 3\times \) more time; for the maximum at \( t = 24.0s, \sim 3.4\times \) more time is needed. This relationship is examined in more detail in Section 5.2.2.

Figure 5.4 and Figure 5.5 illustrate which light configurations require respectively less or more effort. For Figure 5.4 the light configurations at \( t = 11.5s \) and \( t = 26.8s \) are shown, which correspond to the two local minima in the graphs. For Figure 5.5 the local maxima at \( t = 14.4s \) and \( t = 24.0s \) are shown. The images with the lower computation times have in common that the light
5.2. Time Performance

Figure 5.3: Fluctuation of the time spent at the Lightcuts computation (vertical axis) for VPLs controlled by a light source that moves over time (horizontal axis). The computation times for two numbers of VPLs are shown: a low \((n = 617)\) and a high \((n = 6506)\) amount.

![Figure 5.3: Fluctuation of the time spent at the Lightcuts computation (vertical axis) for VPLs controlled by a light source that moves over time (horizontal axis). The computation times for two numbers of VPLs are shown: a low \((n = 617)\) and a high \((n = 6506)\) amount.](image)

Figure 5.4: Two examples of light positions that require lower computation times in the ‘colourful’ scene. The light position and orientation is a function of the wall-clock time, which is shown in the captions. It corresponds to the time on the horizontal axis of Figure 5.3. Note that the local minima of Figure 5.3 are found at these two instants of time.

![Figure 5.4: Two examples of light positions that require lower computation times in the ‘colourful’ scene. The light position and orientation is a function of the wall-clock time, which is shown in the captions. It corresponds to the time on the horizontal axis of Figure 5.3. Note that the local minima of Figure 5.3 are found at these two instants of time.](image)

is oriented so that only a small part of the scene is directly lit, contrary to the images with the higher computation time. This is not surprising: for the configurations where a minority of the VPLs is lit, and where the majority is totally black, a larger number of clusters of totally black VPLs exist than for the opposite configuration. These black clusters are directly rejected by the Lightcuts algorithm, as they are below the error threshold.
5. Experiments

5.2.2 Time Performance as a Function of the Number of VPLs

To determine how the computational cost changes as a function of the number of VPLs $N_{VPL}$, four sets of VPLs were generated for the ‘colourful’ scene, with sizes of 617, 1900, 3546 and 6506 VPLs. From the data in Figure 5.3, we first computed the average Lightcuts time. For both graphs of 617 and 6506 VPLs, the light source configuration at $t = 13.2$s requires an average Lightcuts time. Next, we fixed the time at $t = 13.2$s, and measured the Lightcuts time for all four sets while varying the error threshold. Also, a reference solution was computed by simply brute force iterating over all VPLs. In Figure 5.6, the results are shown. From the graph, it is clear that the Lightcuts algorithm scales sublinear as a function of $N_{VPL}$. The reference computation is perfectly linear, which is expected from theory. For the lowest number of VPLs in this experiment ($N_{VPL} = 617$), the reference solution outperforms our Lightcuts implementation, even for the highest error threshold. However, as the number of VPLs increases, the Lightcuts perform better. At $N_{VPL} = 1900$, our Lightcuts implementation already outperforms the reference solution except for the lowest error threshold $\tau_e$ of 0.02. As $N_{VPL}$ is increased, the difference in time between the reference and Lightcuts approximation increases. For the highest number of VPLs in our experiment, $N_{VPL} = 6506$, our Lightcuts implementation is $1.93 \times$ faster for $\tau_e = 0.02$, and $3.84 \times$ faster for $\tau_e = 0.2$.

To see how the Lightcuts algorithm scales in the other two scenes, and to find out where the reference solution is outperformed, we followed the same procedure as we did for the Colourful scene in the previous experiment. For both scenes, five sets of VPLs were generated. The sizes of these sets were chosen to be close to the sizes of the sets used in the Colourful experiment. In that experiment, four sets were created. The additional fifth set is a smaller

---

1These numbers look rather arbitrary, but since the set of VPLs is the result of a random process steered by a minimum distance parameter, it is impractical to produce ‘prettier’ numbers.
5.2. Time Performance

Figure 5.6: Computation times for the reference solution and Lightcuts implementation as a function of the number of VPLs $N_{VPL}$, for error thresholds $\tau_\epsilon$ of 0.02, 0.05, 0.1 and 0.2, in the Colourful scene.

set that contains about 300 VPLs. This smaller set was useful to examine the break-even point of the reference and Lightcuts methods, as is shown later on. After the VPL sets were generated, a primary light source was moved and rotated through the scenes. For the Maze scene, it was moved through the first corridor on the right, seen from the bottom of Figure 5.1(b). For the Buddha scene, it was moved in a circle around the statue. Using the data, a primary light configuration was selected that results in average computation time. Next, the primary light was fixed at that ‘average’ position and orientation, and for each combination of $N_{VPL}$ and error threshold the computation times were gathered. The graphs for the Buddha scene are shown in Figure 5.7; the graphs for the Maze scene are shown in Figure 5.8. Note that the graphs for the three scenes show different absolute computation times, also for the reference solution. This is due to two factors: the number of sample points differs slightly for the scenes, and the average visibility between points and VPLs is different as well. However, the relative speed ups with respect to the reference solution and the intersection points of the graphs are more important than the absolute times.

The performance of the Buddha scene (Figure 5.7) has several different properties compared to the Colourful scene that was discussed before. In the Colourful scene, at $N_{VPL} \approx 600$, the performance of the Lightcuts algorithm was approximately the same regardless the error threshold $\tau_\epsilon$ (Figure 5.6), and the reference solution performed better. However, for the Buddha scene, the difference in performance for the various $\tau_\epsilon$ is larger (for $\tau_\epsilon = 0.2$ it is $2.3 \times$ faster than for $\tau_\epsilon = 0.02$ for the Buddha scene, for the Colourful scene this is $1.2 \times$), and the error thresholds of $\tau_\epsilon = 0.2$ performs even better than the reference solution. A fifth set of 300 VPLs was created to investigate where the reference solution outperforms the Lightcuts algorithm. For this set, the
5. Experiments

Figure 5.7: Computation times for the reference solution and Lightcuts implementation as a function of the number of VPLs $N_{VPL}$, for error thresholds $\tau_\epsilon$ of 0.02, 0.05, 0.1 and 0.2, in the Buddha scene.

The reference solution outperforms the Lightcuts algorithm even for $\tau_\epsilon = 0.2$. A similarity with the Colourful scene is that the reference solution and Lightcuts algorithm at $\tau_\epsilon = 0.02$ perform equally around $N_{VPL} \approx 2k$. For the largest set of VPLs of $N_{VPL} \approx 6.5k$, Lightcuts at $\tau_\epsilon = 0.02$ is $2.6 \times$ faster than the reference solution; for $\tau_\epsilon = 0.2$ it is $10.3 \times$ faster.

Finally, the performance of the Maze scene is shown in Figure 5.8. For the two smallest VPL sets of $N_{VPL} \approx 300$ and $N_{VPL} \approx 600$, the differences in execution times are small and barely visible from the graph. Manual inspection of the data reveals that for $N_{VPL} \approx 600$ the reference solution is outperformed by the Lightcuts algorithm for all values of $\tau_\epsilon$. At $N_{VPL} \approx 300$ the reference solution outperforms the Lightcuts. It is interesting to see that lines between $N_{VPL} \approx 3500$ and $N_{VPL} \approx 6500$ are almost flat for $\tau_\epsilon = 0.2$ and $\tau_\epsilon = 0.1$. This means that the cost of adding 3,000 VPLs is negligible.

In Figure 5.9 the improvement in the time performance due to the Lightcuts algorithm is illustrated for all scenes and for four different values of $\tau_\epsilon$. The bars show the execution time of the Lightcuts algorithm relative to the reference solution. The height of a bar indicates how many times the Lightcuts algorithm is faster than the reference solution. Thus, a bar with a height of one means an execution time that is equal to the reference solution; a bar with a height of e.g. eight means that Lightcuts algorithm performs eight times better; bars below one indicate that the Lightcuts algorithm performs worse than the reference solution. For the lowest error threshold of $\tau_\epsilon = 0.02$, the Maze scene has the best relative performance. As $\tau_\epsilon$ is increased, the Buddha scene gradually performs better, starting at $\tau_\epsilon = 0.1$ and higher.
5.2. Time Performance

Figure 5.8: Computation times for the reference solution and Lightcuts implementation as a function of the number of VPLs $N_{VPL}$, for error thresholds $\tau_\varepsilon$ of 0.02, 0.05, 0.1 and 0.2, in the Maze scene.

Figure 5.9: Relative time performance of the Lightcuts algorithm with respect to the reference solution for different values of $\tau_\varepsilon$. 
5. Experiments

5.2.3 Constant Time Performance of Bounding Volumes

To compare the constant time performance of axis-aligned bounding boxes (AABBs) as cluster bounding volumes as used by Walter et al. with our bounding sphere approach, we lit the Colourful scene with a moving light source in the same fashion as in Section 5.2.1. We chose to use the Colourful scene because of its versatile geometry; it contains both large flat surfaces (e.g. the walls) as well as complex curved surfaces like the dragon. On the surfaces of the scene, approximately 20k sample points and 1900 VPLs were generated. Two light trees were built: one was optimised using the original box metric and the other was optimised using our sphere metric. For both the box and sphere implementations of the Lightcuts algorithm, statistics of the average light cut size and computation time for the full point set were collected at \( \tau_e = 0.02 \). The average light cut size is the number of nodes in the cut through the light tree for which the Lightcuts algorithm terminates (i.e. the size of the set of clusters used for the illumination approximation), averaged over all sample points. This number is interesting, since it tells how efficient either variant of the algorithm works its way through the light tree to a configuration of clusters/nodes that approximate the total illumination.

We collected data over one period of light source movement (from the one extreme of the room to the other side and back), which was 50 frames. The average cut size and computation time for the box and sphere variants are shown in Figure 5.10. By computing the error upper bound using spheres, the average cut size increased by 5–7.5% of the average cut size obtained by the box variant. We believe that this is caused by an overestimation of the error upper bound, so that more clusters are unnecessarily refined. Overestimation can occur for example with clusters on walls, where boxes can capture the VPLs on it more tightly than spheres can. Despite the larger cut size, our sphere variant still performed 1.26–1.33 \( \times \) faster than the original box method, with an average of 1.3 \( \times \). After decreasing the quality to an error threshold of \( \tau_e = 0.2 \), the sphere variant performed 1.21–1.29 \( \times \) faster, with an average of 1.26 \( \times \) (not shown).

For further investigation of the behaviour of the box and sphere variants, we used the Maze scene. The original Maze scene is a perfect case for the bounding box version. It is entirely composed of planes that are aligned with the three axes of the coordinate system. The AABBs used in the box variant of Lightcuts efficiently capture the VPLs on such planes. For example, an AABB that tightly bounds a set of VPLs on the \( xy \)-plane has a zero length in the \( z \)-direction, and has a volume of zero. Furthermore, the transformation of the AABBs (described in Section 4.1.3) is also efficient for this scene: since all surfaces are aligned with the three axes, all surface normals (and therefore VPL orientations) are aligned with one of these axes. When rotating an AABB using such an axis-aligned normal (or orientation) during the computation of error upper bounds (see page 29 for the material term and page 31 for the geometric term), the rotation is always a multiple of 90° over any axis. This is the best case for AABB rotation, since this does not expand the resulting AABB. Therefore, the error upper bounds computed via the AABBs do not increase either.

To analyse the behaviour of the box and sphere variants in a situation that
is not optimal for boxes as well, we created a modified version of the Maze scene. In this version, its geometry is rotated so that all its planes (walls, floor) are more or less diagonal relative to the coordinate system’s axes. We refer to this scene as the Rotated Maze. This scene is a bad case for bounding boxes: to capture VPLs, nearly all of the boxes now need to grow in all three dimensions rather than in only two. Also, these boxes now undergo arbitrary rotations (instead of multiples of 90°) during runtime, which cause them to expand. For spheres, this rotated scene can be expected to yield results comparable to the original scene, as spheres are insensitive to rotation.

For both scenes, we generated a point set and VPLs, and measured the average light cut size over 160 frames while moving a light source through the scenes, using the box and sphere variants. For the Rotated Maze, we applied the same rotation to the light source’s translation and orientation so that its movement relative to the geometry is the same for both scenes. The results of the measurements are shown in Figure 5.11. For the original Maze scene, the box variant requires the smallest average cut size, with a size of 25.4 averaged over 160 frames. However, when the scene is rotated, the average increases by 25% to 31.2, making it the worst performing variant in terms of cut size. The sphere variant performs for both the Maze and Rotated Maze approximately the same: averaged over 160 frames the average cut size is 28.7 (13% higher than the box variant) for the original Maze and 29 (7% smaller than the box variant) for the Rotated Maze. We believe that this difference is caused by chance; the point set and VPLs are the result of a random process and were regenerated for both scenes. Although the box variant uses smaller cuts than the sphere variant in the (non-rotated) Maze scene, the sphere variant still executes faster: 1.33× for the Maze, and 1.59× for the Rotated Maze, on average.
5.2 Visual Analysis of Error

To visually analyse how error is distributed across the scene, difference images are used. These images are created by taking the absolute differences between the colours of a reference and an approximate rendering, and scaling it with a magnification constant. For the rendering of these images, direct lighting is disabled: it is a constant term for both images—so the difference is zero—but it introduces rounding errors. Furthermore, interpolation between the points is disabled in these renderings. Instead, surface points are illuminated with only the information from the nearest sample point. The purpose of the difference images is to illustrate where error occurs in the scene; the interpolation would only spread it over the screen and mix it with neighbours, obscuring the true position and magnitude of error. This is illustrated in Figure 5.12, where two renderings are shown, one using interpolation and the other using nearest point filtering.

In Figure 5.13 four renderings of the Buddha scene are shown for increasing values of $\tau_e$. Only indirect lighting is rendered, and nearest-point filtering is used for the sample points. A reference rendering is not shown: the rendering at $\tau_e = 0.02$ is visually indistinguishable from it. These images are used for the production of the difference images. The effect of increasing the error threshold $\tau_e$ is rather difficult to see when comparing the images side by side, as the resulting error mainly consists of larger low-frequent patches. Alternating the images on screen makes it easier to see, but the difference images will reveal it as well. For this scene, most of the error at higher thresholds can be spotted on the walls and floor, the statue itself seems to appear identical.

In Figure 5.15 the corresponding difference images for the Buddha scene are...

Figure 5.11: Average cut sizes for the box and sphere methods in the Maze and Rotated Maze scenes. Using spheres, the average cut sizes for both scenes are virtually identical (about 1%). Using boxes, rotating the Maze increases the average cut size by 25% averaged over 160 frames.
shown. In the top row, a constant magnification of six is used. This value is chosen as it is the largest possible value that does not cause colour clipping (i.e. none of the eight-bit colour channels exceeds 255) in any of the four images. From the images, it is clear that $\tau = 0.02$ is not much different from the reference solution: even magnifying the difference six times yields an almost entirely black difference image. Increasing the Lightcuts error threshold $\tau$ increases the brightness (or magnitude) of the differences.

The bottom row of Figure 5.16 requires a different interpretation. In these images, the maximum possible magnification that does not cause colour clipping is maximised for each image individually. This is different from the top row, where the magnification was maximised for the four images together. Since now the magnification is variable, the bottom row images cannot be compared with each other based on their intensity. However, the variable magnification allows us for example to see more easily what is happening in the dark areas of Figure 5.15(a). For $\tau = 0.02$ in Figure 5.16(a) it becomes visible that the differences are rather randomly distributed across the scene and appear noisy. When $\tau$ is gradually increased towards $\tau = 0.20$, more coherence starts to appear. We believe that this can be explained as follows. With a lower error threshold, the Lightcuts algorithm typically descends deeper into the light tree than for a higher error threshold. Deeper in the light tree, there are more possible configurations of clusters than higher in the tree. When there are more clusters to select from, neighbours will typically select different configurations. Vice versa, for higher error thresholds, fewer clusters are needed, so there is more chance that neighbours use the same clusters, resulting in more coherence.

In Figure 5.14 renderings for the Maze and Colourful scene are shown. For each scene two renderings are shown, one at $\tau = 0.02$ and one at $\tau = 0.20$. The corresponding difference images, including those for error thresholds in between, are shown in Figure 5.17 and Figure 5.18 for the Maze and Colourful scene respectively. In the renderings of the Maze scene, the effect of increasing $\tau$ is quite difficult to see. It is most noticeable at left wall, on the border between the green and white areas. Also on the white wall in the centre facing to the bottom left, the colour gradient is less smooth. In the Colourful scene, the error

Figure 5.12: Interpolation versus nearest point for difference images.
is easier to spot. The two green walls on the left show artefacts, for example, and the shading on the floor becomes less smooth.
5.3. Visual Analysis of Error

(a) $\tau_\epsilon = 0.02$.

(b) $\tau_\epsilon = 0.05$.

(c) $\tau_\epsilon = 0.10$.

(d) $\tau_\epsilon = 0.20$.

Figure 5.13: Renderings of the Buddha scene using nearest-point indirect illumination for four different values of $\tau_\epsilon$.

(a) Maze at $\tau_\epsilon = 0.02$.

(b) Maze at $\tau_\epsilon = 0.20$.

(c) Colourful at $\tau_\epsilon = 0.02$.

(d) Colourful at $\tau_\epsilon = 0.20$.

Figure 5.14: Renderings of the Maze and Colourful scenes using nearest-point indirect illumination for two different values of $\tau_\epsilon$ for each scene.
5. Experiments

(a) $6 \times$ difference; $\tau = 0.02$.
(b) $6 \times$ difference; $\tau = 0.05$.
(c) $6 \times$ difference; $\tau = 0.10$.
(d) $6 \times$ difference; $\tau = 0.20$.

Figure 5.15: Difference images for the Buddha scene at four different values of $\tau$, with a constant magnification of $6 \times$.

(a) $42 \times$ difference; $\tau = 0.02$.
(b) $19 \times$ difference; $\tau = 0.05$.
(c) $10 \times$ difference; $\tau = 0.10$.
(d) $6 \times$ difference; $\tau = 0.20$.

Figure 5.16: Difference images for the Buddha scene at four different values of $\tau$. Each image uses the largest possible magnification that does not cause colour clipping.
Figure 5.17: Difference images for the Maze scene at four different values of $\tau_e$ with a constant magnification of $36\times$.

(a) $36\times$ difference; $\tau_e = 0.02$.  
(b) $36\times$ difference; $\tau_e = 0.05$.  
(c) $36\times$ difference; $\tau_e = 0.10$.  
(d) $36\times$ difference; $\tau_e = 0.20$.  

Figure 5.18: Difference images for the Colourful scene at four different values of $\tau_e$ with a constant magnification of $23\times$.

(a) $23\times$ difference; $\tau_e = 0.02$.  
(b) $23\times$ difference; $\tau_e = 0.05$.  
(c) $23\times$ difference; $\tau_e = 0.10$.  
(d) $23\times$ difference; $\tau_e = 0.20$.  

5.3. Visual Analysis of Error
5.4 Visual Effects of Sample Density

The number of points in the point set, and therefore its density across the scene, is a trade-off between under-sampling of the indirect illumination on the one hand and performance on the other hand. Since the time performance of the algorithm scales linear with the number of sample points (since our system simply iterates over all sample points), it has a larger performance impact than the density of the VPLs, which scales sublinear. Performance wise, the lowest sample density is desired, but visual quality should be maintained. To show the impact of the sample point density, several images were created.

First, in Figure 5.19 the sample points are shown in the Buddha scene to give an impression of what the numbers actually mean visually. Two properties of the point set generation process are immediately visible. The first is the adaption of the density as a function of the ambient occlusion. This is visible in particular near the edges of the room, where the ambient occlusion is higher, and therefore the density as well. The second is the creation of points when the differences in surface normals of existing neighbouring samples are above a specified threshold, even though they are closer than the minimum allowed range. This explains why the folds in the clothing of the statue are visible as strokes of dense points.

Next, Figure 5.20 shows how the previously shown point sets actually capture the indirect illumination. From visual inspection of these images, two things can be concluded. The first is that light is smeared over larger surfaces at lower densities, due to interpolation. In this way, light can leak to places where it should not be. For most parts of the scene interpolation works well (e.g. on the walls), but for some parts this is less effective. In particular Figure 5.20(a) the statue seems to be floating above the ground rather than being actually connected to it. The second is that at higher densities, shadows due to indirect illumination—that are supposed to be smooth—start to appear jagged. This can be seen in Figure 5.20(d) on the floor. This effect is caused by the discreteness of the VPLs. When only a few \( n \) strong VPLs contribute to the shadow, the shadow is actually composed of \( n \) overlapping sharp shadows. With low point densities, the sharp shadows are combined and interpolated over the sample points. The point set effectively acts as a low-pass filter. At higher point densities, more of the true illumination from the VPLs is shown, and thus sharp shadows are no longer hidden. A solution would be to increase the VPL density as well, so that the shadow is composed of more and fainter sharp shadows.

To recapitulate: low point densities can lead to light leaking, high densities can lead to artefacts due to the discreteness of the VPLs. An obvious question is: what should the density of the point set be? We leave this question unanswered and let the user choose it, as it depends on numerous other variables. First, the scale of the scene. For example: is the Buddha statue a small decoration in a room, or is it a meters high statue? The latter would require a higher point density than the former. Second, the desired quality of the indirect illumination versus the available processor time. What is the target system and what is the processor time budget? For example, an architectural walk-through allows probably more time to be spent on lighting than a game that also has to perform
5.4. Visual Effects of Sample Density

Figure 5.19: Varying the density of the point set for the Buddha scene by altering the minimum allowed distance between two sample points $g_{dist}$. During the point set creation, potential new points that are closer to another point than $g_{dist}$ are rejected. The values shown for $g_{dist}$ are base values, it is scaled as a function of the ambient occlusion. When occluding geometry is nearby to a potential sample location, $g_{dist}$ is reduced to increase the density. This is visible in the corners of the room.

other jobs, like physics simulation, artificial intelligence, etcetera. Finally, lighting artefacts are less apparent on textured surfaces than on flat coloured surfaces. Game developers have been exploiting this for years, by heavily under sampling static precomputed light maps and applying the information from the light maps on textured surfaces like grass, without visible artefacts. So, for applications that use more textured surfaces than those in the images in this text (which are intentionally flat coloured), one can use less samples. What a ‘good’ number of VPLs is, is a similar problem as selecting the number of sample points, except for one advantageous difference: the computation time scales sublinear as a function of the number of VPLs. However, there is a relation with the density of the point set that affects the visual quality. As we have seen, a high density point set reveals the discrete nature of the VPLs, while a low density point set hides it—but increases light leakage. And increasing the number of VPLs can hide their discrete nature. Based on our experiments, we recommend
5. Experiments

(a) 2292 points; $g_{dist} = 0.6$.

(b) 4216 points; $g_{dist} = 0.4$.

(c) 8614 points; $g_{dist} = 0.25$.

(d) 12204 points; $g_{dist} = 0.2$.

Figure 5.20: Illumination quality as a function of the point set density. A light source is positioned behind the statue and oriented away from it. Indirect lighting from the room is represented by 2368 VPLs. For the lower density sets (especially (a) but also (b)), light leaks between Buddha’s feet, because a representative sample point is not nearby. Although this leaking is reduced for higher densities (c), (d), jags start to appear around shadows.

the following rough guideline. First, the number of sample points should be decided experimentally. Next, select a number of VPLs that is 10%–25% of the number of sample points. Overestimation of the number of VPLs is usually far less expensive than larger point sets.

5.5 Visual Effects of VPL Density

Like the density of the point set, the density of the VPL set controls the quality of the indirect illumination, at the cost of computation time and memory. However, there is a difference that makes the selection of the VPL density less critical than the sample point density: the computational costs of the VPLs scale sublinear, while the sample points scale linear.

In animation with a moving light source, the differences between lower and higher VPL densities are more noticeable than in still images. At lower densities,
fewer VPLs have to transfer the same amount of energy, so the intensity of individual VPLs is higher. When the visibility of a VPL (seen from the light source) changes, its contribution to the lighting of the scene also abruptly changes. The higher the intensity of the VPL, the more noticeable this is. When at higher densities more low intensity VPLs provide the lighting, the individual changes are smaller and the total illumination will change more gradually.

The effects of a lower and higher VPL density are shown in Figure 5.21. Although less noticeable in still images than in animation, several clues reveal the discrete nature of the VPLs at lower densities. At the top right of Figure 5.21(a), near the corner of the white walls, a vertical shadow boundary is visible. This is caused by a relatively bright VPL on the blue wall at the left of the dragon, illuminating the white wall. When more (but fainter) VPLs contribute to the illumination on the right white wall in Figure 5.21(b), the shadow boundary is softened. There are more regions in the scene where shadows are sharper because of being cast by a single or a few VPLs: around the dragon, under the spiral and near the cube in the bottom centre.
Chapter 6

Conclusion and Future Work

In this thesis we proposed a system that adds plausible indirect illumination to the real-time ray tracer Arauna. While the geometry is kept static, light sources are allowed to be dynamic. Only a single reflection of illumination by Lambertian surfaces is handled, which is considered to yield convincing results in most cases [32].

Virtual point lights (VPLs) are used to simulate the reflection of light by surfaces. These VPLs are distributed across the surfaces of the scene following a Poisson distribution, and have the same colour as the surfaces they represent. Their intensity is a function of one or more primary light sources. To approximate the illumination from the set of VPLs, the Lightcuts algorithm [40] is used. This algorithm uses a tree with clusters of lights as nodes. When the Lightcuts algorithm approximates illumination from the lights, it adaptively selects clusters while maintaining an upper bound on the approximation error. The original Lightcuts algorithm uses axis-aligned bounding boxes to compute the upper bound. We proposed to use spheres since they are invariant to rotation, and derived new formulae for the various error upper bound calculations.

Indirect illumination is not computed at all positions. A sparse point set of sample points is distributed across the scene in a similar fashion as the VPLs, but with a higher density. These sample points cache indirect illumination. In the shading computation of the ray tracer, indirect illumination is interpolated by querying nearby sample points. Since the geometry is static, the visibility between any pair of sample points and VPLs does not change. We exploit this by computing it in a pre-processing stage so that it is available during run-time at a negligible computational cost.

In our experiments, the system has proven to be capable to add convincing indirect illumination to several test scenes in real-time on commodity hardware. The constant time performance of our sphere approach was about 1.3× better than the original box method. However, the average number of used clusters increased with approximately 10%. Selection of the number of sample points is a trade-off between quality and performance, and depends on the scene. It is also linked to the density of the VPL set: on the one hand, lower point set densities can act as a low-pass filter and hide artefacts caused by low VPL densities. On the other hand, lower point set densities can cause the indirect illumination to
leak through geometry due to interpolation, creating the false impression that empty space exists at places where it actually does not exist.

**Future Work**

Our current design does only support static geometry. For architectural walkthroughs, for example, this would be sufficient, but games typically require dynamic geometric objects. Extending our design to allow this would be possible by handling dynamic objects separately. Games typically consist of largely static geometry (the game ‘level’) with a small fraction of dynamic objects (characters, items, etcetera). When testing the visibility of a VPL during run-time, a line between the sample point and VPL can be tested for intersection with dynamic geometry. Acceleration structures like a bounding volume hierarchy \(1\) can speed this up. These tests can be integrated into the existing infrastructure and exploit it in two ways. First, VPLs that are already known to be invisible due to occluding static geometry do not need an (expensive) additional test for occluding dynamic geometry. Second, the Lightcuts algorithm selects mostly relevant lights from the full VPL set, only these lights need to be tested. Furthermore, due to the low frequent nature of indirect illumination and its interpolated representation through the point set, a coarse version of the dynamic geometry might yield results that are as convincing as a—more expensive—test with the exact geometry.

In our current implementation, the full set of sample points is computed each frame, regardless of the visibility of the individual sample points. One obvious way to increase the performance of our system is to compute only the sample points that actually contribute to the final image. Computing only the subset of sample points that are inside the camera frustum is not sufficient: sample points outside the frustum can also contribute to the final image because of the interpolation. One option is to expand the camera frustum accordingly to the (constant, predefined) maximum sample point search radius used in the interpolation algorithm. This would make sure that even pixels on the boundary of the screen would never consult non-updated sample points. Although such a mechanism would already radically reduce the size of the set of sample points to be computed, it does not take account of occlusion. For example, an adjacent room filled with sample points can be completely invisible due to a wall, but all sample points can pass the camera frustum containment test. This problem is equivalent to the problem known in computer graphics as occlusion culling, where one tries to avoid drawing geometry that is occluded by other geometry. Existing approaches for occlusion culling might be reusable to discard occluded parts of the point set. However, it may vary from scene to scene whether the cost of occlusion culling outweighs the cost of computing unused sample points.

Another method that would require less effort to implement is to divide the set of sample points into two disjoint subsets: sample points that have been used in the previous frame, and those that have not been used. In advance of rendering, only the sample points that were used in the previous frame are updated. With this scheme, sample points that change from unused to used between two frames are not updated, while they should have been. This
potentially leads to artefacts, whose severity depends on the amount of camera movement and changes in lighting conditions. These artefacts can possibly be reduced by excluding any contribution of non-updated sample points, or by computing the value for a non-updated sample on-demand. However, the latter approach has practical difficulties because of synchronisation of such on-demand updates in multi-threading.

Spatial coherence is already exploited in our system by interpolating positions in between the sample points. It might be possible to exploit it further, as for neighbouring samples it is likely to have a similar configuration of clusters. In the Lightcuts publication, [Walter et al.] describe an extension of their algorithm which they call ‘reconstruction cuts’. The idea is to compute light cuts sparsely on a grid in screen space, and then interpolate between grid points when they have similar light cuts, or resort to the regular Lightcuts algorithm otherwise. This idea can be extended to our sample points as well.

The low-frequent nature of indirect illumination also allows for interpolation in time, besides interpolation in space. For example, when one considers the direct illumination of a surface point by a spot light, there can be steep changes as a function of the occlusion of the light source. Indirect illumination typically changes more gradually. One way to exploit temporal coherence is to subdivide the set of sample points into $n$ disjoint subsets $0 \ldots n-1$. At frame $i$, only the points in set $i \mod n$ are updated. In the $n-1$ frames between two updates, points are interpolated between their previous and latest update to make the transition appear gradually. A disadvantage of this method is that the indirect illumination always lags at least $n$ frames behind the direct illumination. Whether this is perceptible or not depends on the frame rate (i.e. the length of the lag) and how rapidly the lighting conditions change. When this is to be combined with visibility checks proposed in the previous section, additional care has to be taken.
Bibliography


71


[38] Bruce Walter. personal correspondence, 2011. Cited on p. 33


Appendix A

Notation

Throughout this thesis, the following notation is used.

- Scalars are in lower-case italic. Examples are $i$, $j$, $n$;
- Angles are in lower-case Greek, e.g. $\theta$, $\phi$, $\omega$;
- Vectors and points are lower-case bold. Individual components of a vector are accessed as $\mathbf{v} = (v_x, v_y, v_z)^T$. For example: $\mathbf{p}$, $\mathbf{n}$;
- Sets are denoted with blackboard-typeset symbols, for example $C$ (for a Lightcuts cluster), $S$.

Furthermore, we borrowed the convenient clamped-cosine $\cos_{\phi}$ from [1]. It works like a normal cosine, except that negative output values are clamped to zero. In the Lightcuts description, hats above symbols are used to denote the upper bound. For example, $\hat{M}_C$ is the upper bound for the material term $M_C$. Tildes above symbols indicate the approximate version; $\tilde{L}_C$ is the approximation of the cluster radiance $L_C$. 