# INTERACTIVE DIRECT ILLUMINATION IN COMPLEX ENVIRONMENTS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Sebastian Pablo Fernandez

August 2004

© 2004 Sebastian Pablo Fernandez

ALL RIGHTS RESERVED

INTERACTIVE DIRECT ILLUMINATION IN COMPLEX ENVIRONMENTS

Sebastian Pablo Fernandez, Ph.D.

Cornell University 2004

Modeling the interaction of light with real-world environments is a difficult problem. In particular, the geometric and lighting complexity required to approximate reality are huge challenges. The "Ray Tracing" algorithm is well-suited to deal with geometric complexity since its performance is sub-linear in the number of geometric primitives. However, its computational cost is linear in the number of light sources. This leads to poor performance in environments with complex lighting.

In this thesis we present two algorithms that accelerate the rendering of direct lighting for complex environments within the context of a ray tracer. The first algorithm, "Local Illumination Environments" addresses direct lighting acceleration in scenes with up to a few dozen light sources. The second algorithm, "Hierarchical Light Clusters" accelerates direct lighting in models with hundreds to thousands of light sources.

The "Local Illumination Environments" algorithm reduces the cost of computing light visibility, the most expensive part of the direct lighting computation. It does so through an asynchronous process that caches, in a spatial data structure, the geometric primitives required to evaluate light visibility. This approach completely eliminates the cost of light visibility for fully visible and fully occluded light sources. It also substantially reduces the time to evaluate visibility from partially visible light sources by eliminating the cost of a traditional acceleration structure.

The "Hierarchical Light Clusters" algorithm reduces direct lighting computation in environments with very large numbers of light sources. This is done by using a single bright light to approximate the contribution of a group of lights. We present a locally adaptive technique that determines when this approximation is valid. We also introduce three algorithms that make use of this approach to provide varying levels of quality and performance.

"Local Illumination Environments" and "Hierarchical Light Clusters" both provide order-of-magnitude acceleration in the computation of direct lighting over traditional ray tracing approaches. Together, they can be used to interactively generate images of models of widely varying geometric and lighting complexity.

# Biographical Sketch

Sebastian Pablo Fernandez was born in Cordoba, Argentina in 1972. He immigrated to the United States in 1980 and grew up in Orange County, California. He obtained his Bachelor's degree in Electrical Engineering and Computer Science from the University of California at Berkeley in 1994. He then moved to Ithaca, New York where he obtained his Doctorate in Computer Science from Cornell University in 2004.

*For Isabel and Silvina*

# Acknowledgements

I would like to thank Kavita Bala, Bruce Walter and Moreno Piccolotto for their collaboration, friendship and coffee through all these years. Their respective lessons on hard work, cat-herding and collaboration-through-caffeine were particularly inspiring.

I'd also like to thank Don Greenberg for the opportunity to work at this lab and for his patience through this long and arduous process.

Thanks to Hurf Sheldon for maintaining an insanely heterogeneous system of computers without losing it, and for not constraining our creativity. Thanks to Linda Stephenson for always keeping me on Don's radar. Thanks to the dozens of PCG students and staff through the years who made it all very fun and interesting.

Finally, I would like to thank my mom, Isabel Correal, for helping me get here, and my fiancee, Silvina Dejter, for standing by my side and seeing it through with me.

# Table of Contents


**1 Introduction** **1**

**2 Background** **7**

2.1 Point lights . . . 12
2.2 Area lights . . . 13
2.3 Ray tracing . . . 15
2.4 Radiosity . . . 17
2.5 Two-pass algorithms . . . 18
2.6 Particle methods . . . 20
2.7 Many lights . . . 20
2.8 Image-based methods . . . 22
2.9 Conclusion . . . 23

**3 Local Illumination Environments** **24**

3.1 Local Illumination Environments . . . 27
3.1.1 LIE Construction . . . 28
3.1.2 Shading Using LIEs . . . 34
3.2 Masking . . . 34
3.3 System Description . . . 40
3.4 Results . . . 42
3.5 Conclusions . . . 48

**4 Local Illumination Environment Construction** **49**

4.1 Definitions . . . 49
4.2 Testing methodology . . . 55
4.3 Fixed cell size . . . 56
4.4 Adaptive subdivision . . . 60
4.5 Quasi-Monte Carlo sampling . . . 66
4.6 Regular sampling . . . 68
4.7 Three-state LIEs . . . 70
4.8 Error analysis . . . 75
4.9 Future Work . . . 78
4.10 Conclusion . . . 81

**5 Hierarchical Light Clusters** **83**
5.1 Concepts . . . . . 84
5.1.1 Light Clusters . . . . . 86
5.1.2 Cluster Hierarchy Tree . . . . . 88
5.1.3 Finding a Cut . . . . . 90
5.1.4 Error Estimates . . . . . 94
5.2 Dense Sampling Algorithm . . . . . 100
5.3 Sparse Sampling Algorithms . . . . . 102
5.3.1 Weighted Sum Reconstruction Algorithm . . . . . 103
5.3.2 Tree-Based Reconstruction Algorithm . . . . . 107
5.4 System . . . . . 116
5.4.1 Parallel System . . . . . 116
5.4.2 Nearest Samples . . . . . 117
5.4.3 Sample Priorities . . . . . 118
5.4.4 Storage . . . . . 118
5.5 Results . . . . . 121
5.6 Conclusion . . . . . 124

**6 Conclusion** **133**

**Bibliography** **137**

# List of Figures


1.1 Grand Central Station . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.2 Mosque de Cordoba . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
1.3 Gustavus Adolphus College Basketball Court . . . . . . . . . . . . . 5
1.4 Car show . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.1 Rendering Equation . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 Reformulation of Rendering Equation . . . . . . . . . . . . . . . . . 9
2.3 Direct Illumination Formula . . . . . . . . . . . . . . . . . . . . . . 10

3.1 LIE example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
3.2 State diagram for LIE construction . . . . . . . . . . . . . . . . . . 29
3.3 LIE construction example, part 1. . . . . . . . . . . . . . . . . . . . 32
3.4 LIE construction example, part 2. . . . . . . . . . . . . . . . . . . . 33
3.5 Shadow masking by other lights . . . . . . . . . . . . . . . . . . . . 35
3.6 Contrast sensitivity function . . . . . . . . . . . . . . . . . . . . . . 37
3.7 Rendering cost without taking advantage of masking . . . . . . . . 38
3.8 Rendering cost taking advantage of masking . . . . . . . . . . . . . 39
3.9 System structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.10 Science Center . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.11 Bar . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
3.12 Mosque de Cordoba . . . . . . . . . . . . . . . . . . . . . . . . . . 47

4.1 Shadow ray space . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.2 Shadow ray space with cells . . . . . . . . . . . . . . . . . . . . . . 53
4.3 View shadow ray space . . . . . . . . . . . . . . . . . . . . . . . . . 54
4.4 Child cells are never more complex than their parents . . . . . . . 57
4.5 Child cells require more samples than their parents . . . . . . . . . 57
4.6 Fixed tree depth (Bar) . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.7 Fixed tree depth (Grand Central Station) . . . . . . . . . . . . . . . 59
4.8 Fixed tree depth (Mosque de Cordoba) . . . . . . . . . . . . . . . . 60
4.9 Adaptive cell subdivision (Bar) . . . . . . . . . . . . . . . . . . . . 62
4.10 Adaptive cell subdivision (Grand Central Station) . . . . . . . . . . 63
4.11 Adaptive cell subdivision (Mosque de Cordoba) . . . . . . . . . . . 64
4.12 QMC vs. Random sampling . . . . . . . . . . . . . . . . . . . . . . 67
4.13 Regular sampling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

4.14 Blocker list vs. three-state LIEs on the Bar scene . . . . . . . . . . 72
4.15 Blocker list vs. three-state LIEs on the Grand Central Station scene 73
4.16 Blocker list vs. three-state LIEs on the Mosque de Cordoba scene . 74
4.17 $(1-\overline{s})^t$, for some values of $\overline{s}$ . . . . . . . . . . . . . . . . . . . . . 75
4.18 $(1-s)^t$, for some values of $t$ . . . . . . . . . . . . . . . . . . . . . . 76
4.19 Number of features times feature size . . . . . . . . . . . . . . . . . 77

5.1 Terms used in geometry factors. . . . . . . . . . . . . . . . . . . . . 85
5.2 Light clustering in a simple scene with four point lights . . . . . . 87
5.3 Cluster hierarchy tree and three example cuts for a simple scene . . 91
5.4 Finding a cut . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.5 Terms used in bounding distance and cosine . . . . . . . . . . . . . 96
5.6 The dense sampling algorithm . . . . . . . . . . . . . . . . . . . . 101
5.7 Sample generation in the weighted sum reconstruction algorithm . 104
5.8 Sample reconstruction in the weighted sum reconstruction algorithm 105
5.9 The Epanechnikov weighting function. . . . . . . . . . . . . . . . . 106
5.10 Irradiance discontinuities are common in environments with many lights. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
5.11 Common cluster irradiances can occur even when total irradiances differ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
5.12 The sample generation portion of the tree-based reconstruction algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113
5.13 The reconstruction portion of the tree-based reconstruction algorithm 114
5.14 The interactive system . . . . . . . . . . . . . . . . . . . . . . . . 117
5.15 Packing a sample cut into two arrays . . . . . . . . . . . . . . . . 120
5.16 Characteristics of the models used for testing. . . . . . . . . . . . . 122
5.17 Results for Grand Central Station . . . . . . . . . . . . . . . . . . 125
5.18 Difference images for Grand Central Station . . . . . . . . . . . . . 126
5.19 Results for Mosque de Cordoba . . . . . . . . . . . . . . . . . . . . 127
5.20 Difference images for Mosque de Cordoba . . . . . . . . . . . . . . 128
5.21 Results for Residential Kitchen . . . . . . . . . . . . . . . . . . . . 129
5.22 Difference images for Residential Kitchen . . . . . . . . . . . . . . 130

# Chapter 1

# Introduction

Global illumination describes the steady state distribution of light in an environment. Modeling global illumination in synthetic environments allows us to visualize what those environments would actually look like if they existed. Being able to model global illumination quickly would allow us to walk through those environments and be able to interact with a world that looks real. However, existing algorithms can take hours or even days to compute a global illumination solution, allowing for little interactivity.

This thesis focuses on interactive rendering and an important subproblem of global illumination, direct illumination. A solution to the direct illumination problem generates an image of what an environment would look like if photons were only able to bounce once off a surface before reaching our eye. Solving this problem is important for three reasons. One, it is commonly required to solve this subproblem when computing a global illumination solution. Two, it frequently provides a very reasonable first approximation to the global illumination solution and is "good enough" for many purposes. Three, computing accurate direct shadows is important for spatial perception.

As will be described in the Background chapter, there is a plethora of algorithms dealing with interactive direct illumination. However, the majority of these techniques address only simple environments. This thesis deals with interactive direct lighting in complex environments.

There are three components to complexity in rendering algorithms. There is geometric complexity, which refers to the number of objects in the scene and their composition (simple polygons or more complex parametric surfaces). There is material complexity which refers to the reflective properties of surfaces, ranging from diffuse reflection to glossy and bumpy surfaces. There is lighting complexity, which refers to the number of light sources, their emission properties, and whether indirect illumination is considered. As illustrated by Figures 1.1, 1.2, 1.3, and 1.4, the real world exhibits all of these complexities.

In this thesis, we will be addressing the problem of how to render these complex scenes interactively. We will be using an existing technique, called "ray tracing", to deal with geometric and material complexity. Our work will show how ray tracing can be adapted to also handle high lighting complexity.

Chapter 2 describes the problem of direct illumination. It provides a mathematical formulation of the problem. The chapter also discusses a wide range of previous approaches to computing direct and indirect illumination. We discuss the applicability of each approach to the problem.

Chapter 3 describes an algorithm called "Local Illumination Environments" for interactively rendering scenes of moderate lighting complexity. It also gives a description of the parallel software system developed to solve it and issues involved in the implementation of the solution.

Chapter 4 explores the parameter space of the algorithm described in Chapter 3

Figure 1.1: A picture of Grand Central Station in New York City (Bart Goddyn, http://users.skynet.be/bgoddyn/photo/ny17.jpg). This environment contains hundreds of lights, specular reflections from windows and glass, and glossy reflections from the floor. Note that almost almost all lights are visible from almost all surfaces, with minimal occlusion.

and basic extensions. We describe results and compare them to existing solutions to the direct lighting problem.

Chapter 5 describes "Hierarchical Light Clusters", an algorithm for dealing with scenes of very high lighting complexity. We describe several variants of the algorithm for different speed/quality requirements.

Finally, Chapter 6 concludes with a summary of the thesis and possible future extensions to the algorithms described therein.

Figure 1.2: Mosque de Cordoba (Leith Davis, http://www.sfu.ca/leith/cordoba.jpg). This environment contains highly detailed geometry and many lights and exhibits direct and indirect lighting.

Figure 1.3: Gustavus Adolphus College Basketball Court (http://www.gustavus.edu/ oncampus/athletics/ atr/images/gusyoungcourt.jpg). This environment contains dozens of lights and glossy reflections off the floor of the court.

Figure 1.4: Car Show (Donald P. Greenberg's slide collection). This environment contains hundreds of lights, highly detailed car geometries and specularly reflective materials.

# Chapter 2

# Background

The global illumination problem can be defined thus: Given viewing parameters, the geometry describing the scene, a set of materials representing light reflection and refraction properties on surfaces, and a set of light sources, compute the steady-state distribution of light in a way that can be displayed on a computer screen. Kajiya [Kaj86] formulated this problem as the Rendering Equation:

$$L(x \rightarrow \Phi) = L_e(x \rightarrow \Phi) + \int_{\Omega} L(x \leftarrow \Theta) f_r(\Phi, x, \Theta) cos\theta d\Theta \tag{2.1}$$

where

- $L(x \rightarrow \Phi)$ is the radiance exiting from a point $x$ in the direction $\Phi$
- $L_e(x \rightarrow \Phi)$ is the radiance directly emitted from a point $x$ in the direction $\Phi$ (this term is non-zero only when $x$ is on a light source)
- $L(x \leftarrow \Theta)$ is the radiance incident on a point $x$ from the direction $\Theta$
- $f_r(\Phi, x, \Theta)$ is the Bidirectional Reflectance Distribution Function (BRDF) and defines the fractional radiance reflected towards $\Phi$ at point $x$ that comes

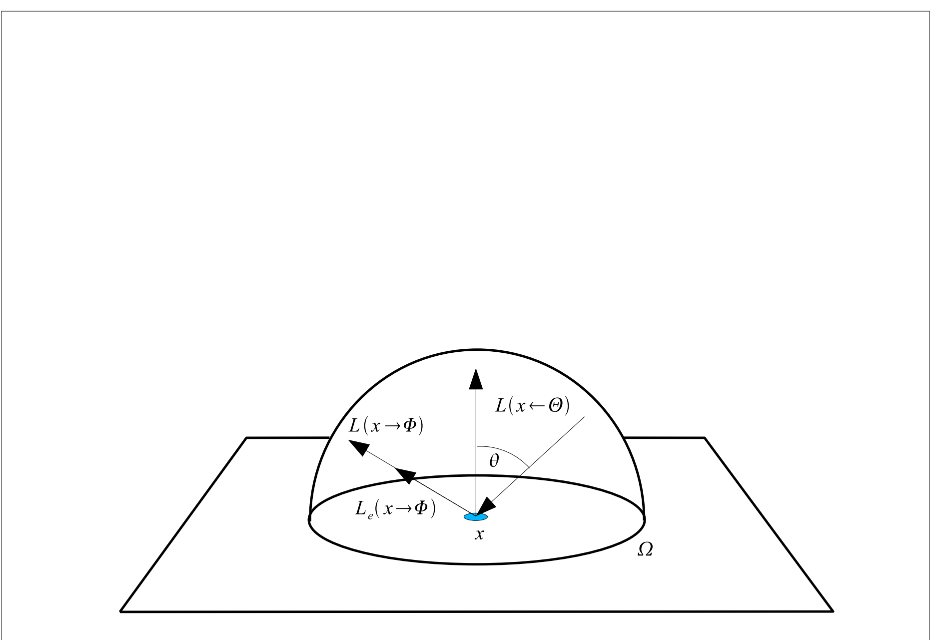



Figure 2.1: Rendering Equation

from the direction $\Theta$

- $\theta$ is the angle between $\Theta$ and the normal to the surface at $x$
- $\Omega$ is the integration domain, the set of directions represented by a hemisphere centered at $x$ and bounded by the surface being rendered

An important subproblem of global illumination is that of direct illumination, the illumination due to a single bounce of light from all sources to the eye. We obtain the Direct Illumination Formula from the Rendering Equation by first reformulating the Rendering Equation from an integral over the hemisphere of incident directions to an integral over all surface points:

$$L(x_1 \to x_0) = L_e(x_1 \to x_0) + \int_A L(x_2 \to x_1) f_r(x_0, x_1, x_2) \frac{cos\theta_1 cos\theta_2}{r^2} v(x_1, x_2) dx_2 \tag{2.2}$$

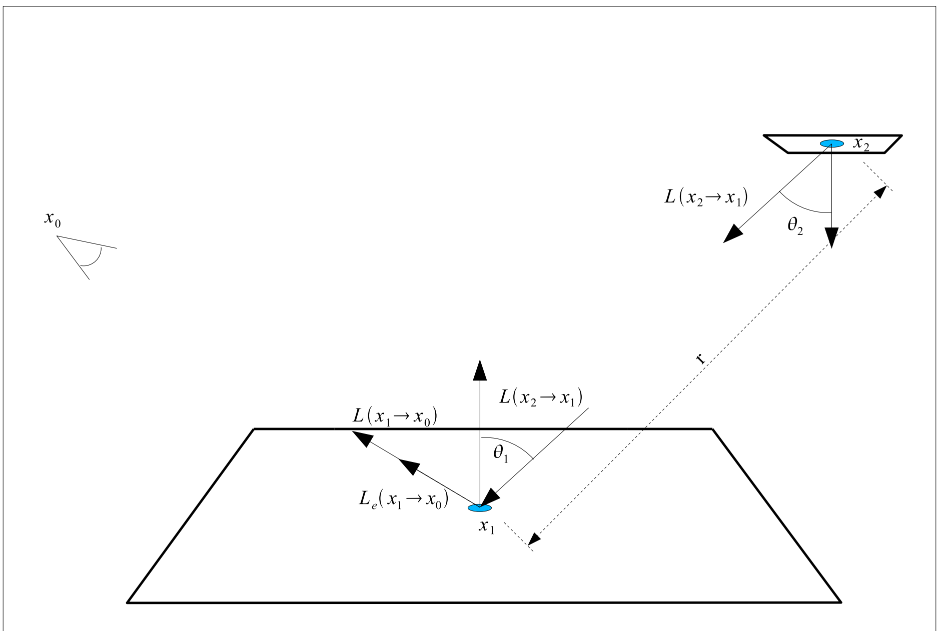



Figure 2.2: Reformulation of Rendering Equation

where

- $L(x_1 \to x_0)$ is the radiance from a point $x_1$ in the direction of the point $x_0$
- $L_e(x_1 \to x_0)$ is the radiance directly emitted from a point $x_1$ in the direction of the point $x_0$ (this term is non-zero only when $x_1$ is on a light source)
- $f_r(x_0, x_1, x_2)$ is the Bidirectional Reflectance Distribution Function (BRDF) and defines the fractional radiance reflected towards point $x_0$ at point $x_1$ that comes from point $x_2$
- $\theta_1$ is the angle between the ray from $x_1$ to $x_2$ and the normal to the surface at $x_1$
- $\theta_2$ is the angle between the ray from $x_2$ to $x_1$ and the normal to the surface at $x_2$

- $r$ is the distance from $x_2$ to $x_1$
- $v(x_1, x_2)$ is the visibility function and is defined as 1 if points $x_1$ and $x_2$ are mutually visible and 0 otherwise

Replacing the total radiance term, $L$, with an $L_e$ term, which represents only the radiance directly emitted by a surface, gives us the Direct Illumination Formula.

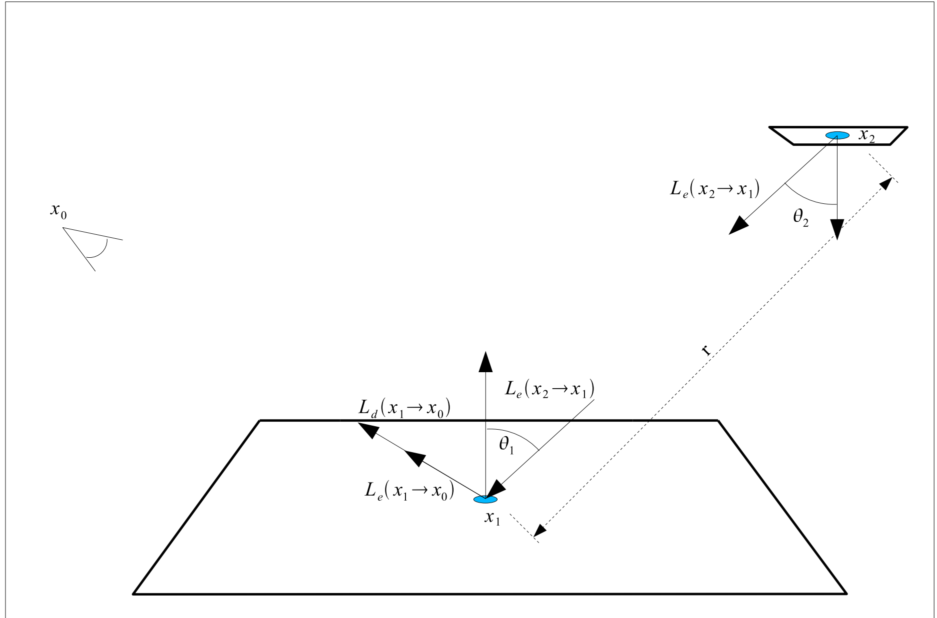



Figure 2.3: Direct Illumination Formula

$$L_d(x_1 \rightarrow x_0) = L_e(x_1 \rightarrow x_0) + \int_A L_e(x_2 \rightarrow x_1) f_r(x_0, x_1, x_2) \frac{cos\theta_1 cos\theta_2}{r^2} v(x_1, x_2) dx_2 \tag{2.3}$$

where

- $L_d(x_1 \rightarrow x_0)$ is the direct radiance from a point $x_1$ in the direction of the point $x_0$. Direct radiance is the radiance due solely to direct illumination and emission

- $L_e(x_1 \rightarrow x_0)$ is the radiance emitted from a point $x_1$ in the direction of the point $x_0$ (this term is non-zero only when $x_1$ is on a light source)
- $f_r(x_0, x_1, x_2)$ is the Bidirectional Reflectance Distribution Function (BRDF) and defines the fractional radiance reflected towards point $x_0$ at point $x_1$ that comes from point $x_2$
- $\theta_1$ is the angle between the ray from $x_1$ to $x_2$ and the normal to the surface at $x_1$
- $\theta_2$ is the angle between the ray from $x_2$ to $x_1$ and the normal to the surface at $x_2$
- $r$ is the distance from $x_2$ to $x_1$
- $v(x_1, x_2)$ is the visibility function and is defined as 1 if points $x_1$ and $x_2$ are mutually visible and 0 otherwise

Current methods of computing the Direct Illumination Formula suffer from a variety of drawbacks. Some approaches limit the type of scene that can be rendered, restricting the use of area light sources, glossy materials, and/or non-polygonal geometry. Other techniques are fully general but do not scale well with complex scenes. This thesis will present algorithms for computing the Direct Lighting Formula at interactive rates in highly complex scenes without restricting the characteristics of the scenes that can be rendered.

We will now describe some of the more important techniques for solving the direct illumination problem including approaches to solve the more general global illumination problem since that is a superset of direct illumination.

## 2.1 Point lights

The algorithms discussed in this section make the assumption that the light sources subtend a small solid angle to the surfaces they illuminate and thus behave as "point lights". This means that certain factors in the direct lighting equation can be assumed to be constant and can be taken out of the integral. For example, since a point light has zero area, the visibility factor is assumed to be constant.

Crow [Cro77] introduced shadow volumes. Shadow volumes construct polyhedra from the point light source and the edges of the polygonal model. Surfaces in the model are determined to be in shadow if they are completely contained by a shadow polyhedron. The problems with this approach arise from having to perform depth calculations with the surfaces of these polyhedra. If the scene is geometrically complex, this can be expensive. Additionally, elongated shadows can significantly increase rendering cost.

The Shadow Map technique was presented in [Wil78]. The scene is rendered from the point of view of the light source on a discrete grid. At each pixel in the grid, the distance from the light to the nearest surface point to the light through that pixel is cached. When rendering the scene, this distance is compared to the distance from the light of a surface point seen from the eye. If the point is farther away, it is considered to be in shadow, otherwise it is lit. This approach suffers from a few problems. First, it is limited to point light sources. Second, the assumption that the depth is constant over any pixel within the shadow map can lead to "jaggy" shadows or aliasing. Finally, the shadow map cache must be constructed for each light, making this technique expensive in environments with many lights.

Light Buffers [HG86] and Adaptive Shadow Maps [FFBG01] both address the

problem of aliasing. The Light Buffer approach does so by storing a list of geometry that causes the shadow within a particular pixel of the shadow map. Shadow tests can then be done analytically, removing any artifacts. Adaptive Shadow Maps take the approach of using a hierarchy of shadow maps, refining a shadow map to a required resolution based on the user viewpoint. Although both of these approaches remove the problem of discretization artifacts in shadows, they're both constrained to be used in environments with a small number of point light sources.

The algorithms discussed in this section can generate images quickly. However, they make the unrealistic assumption that lights are point sources. This assumption causes discontinuities in the direct illumination ("hard shadows") which leads to renderings that appear synthetic.

## 2.2 Area lights

In the real world, lights are not infinitesimally small. Lights subtend a non-zero solid angle, casting soft shadows. The following set of algorithms recognize this and attempt to render the effects expected from realistic lights.

Amanatides [Ama84] introduced Cone Tracing. Assuming circular or spherical light sources, a cone is constructed from the point to be rendered to the light source. The proportion of the cone obstructed by scene geometry indicates the level of intensity of the penumbra. This approach has two problems. First, it assumes spherical or circular light sources. Second, analytical intersections with cones can be hard to compute for general geometry.

Stewart [SG94] analytically computed the umbral and penumbral discontinuity events for every light against every object in the scene. This information could

then be used to generate a mesh that would not cross lighting discontinuities. This approach, however, is limited to purely polygonal environments. Furthermore, it identifies discontinuities which may not be perceptually apparent in an environment with many lights.

Soler and Sillion [SS98] approximated soft shadows for interactive viewing using an image-processing approach. They found objects that were at similar distances from the light plane and projected them onto a single plane. They then took advantage of the fact that when the light, occluder, and receiver lie on parallel planes, the shape of the shadow is a convolution of the shape of the light with the shape of the occluder. Using hardware convolution, they were then able to obtain soft shadows at interactive rates. Unfortunately, this approximation breaks down when the occluders are mostly perpendicular to the light source.

Agrawala et al. [ARHM00] presented two approaches to dealing with area lights. The first combines multiple shadow maps into a single layered shadow map. This allowed for fast soft shadows but introduced significant errors since interpolation of shadow maps can be an inaccurate approximation. Their second approach computes soft shadows by densely sampling the light source, but relying on coherence in the shaded points to reduce the number of actual shadow rays that have to be cast. The drawback to this approach is that it must reproject a potentially large number of occlusion points per light and thus does not scale well when the environment consists of a large number of lights.

Akenine-Moeller [AMA02] and Assarsson [AAM03] approximated the soft shadows cast by area lights through the use of “penumbra wedges”. The algorithm operates in two stages, first determining a hard shadow through shadow volumes as if only the center of the light was illuminating the scene. The second pass at-

tempts to correct the visibility by computing the amount of light coverage along the silhouette of the occluding object. This approach generates realistic-looking soft shadows at interactive rates for individual lights. However, because they generate the shadows cast by each object independently, they cannot correctly render shadows cast by multiple objects whose shadows overlap.

The preceding methods are a significant improvement over point-light approximations. However, they are still limited in use. Some of the techniques are geared towards single-image generation due to slow rendering speeds. The algorithms that are efficient enough to work at interactive rates are approximations which generate erroneous results under common conditions.

## 2.3 Ray tracing

Ray tracing, introduced by Whitted [Whi80], simulates the transport of light through the use of geometric optics. Rays of light are traced backwards from the camera through the image plane to find the first intersected surface. Based on material properties, the ray recursively bounces from specular surface to specular surface, modulating the color of the ray, until it lands on a diffuse surface. The color thus computed is used to generate a pixel value at that point on the image plane.

Cook [CPC84] showed that Monte Carlo integration could be used in combination with ray tracing to solve the Rendering Equation. This combination makes ray tracing a very general technique, allowing renderings that include any type of geometry, material, light source, or camera. This was an impressive result, providing the first full global illumination solutions, and the algorithm is, to this day,

generally used to produce the reference images against which other techniques are measured.

Although powerful in its generality, ray tracing is slow. This is due to the expense of computing ray-scene intersections. As a ray bounces around the scene a ray-tracing algorithm must determine the first visible surface from a point in a particular direction. This ray-scene intersection could be calculated, albeit naively, by computing the intersection point of the ray with every geometric primitive in the scene and finding the intersection closest to the source point.

Several techniques have been proposed to perform fewer intersections than this naive approach. These are usually in the form of "acceleration structures", such as regular grids, KD-trees, bounding volume hierarchies, etc. [Gla89], all of which reduce the cost of the ray-scene intersection. However, they suffer from a few drawbacks. First, they tend to be too conservative, not reducing the number of visibility tests as much as possible. Second, the cost of traversing the data structure can be a significant portion of the total computation time. Third, they do not have predictable memory access, making it hard to obtain the data the processor needs to work on ahead of time. As processor speeds quickly outpace memory latency, this last drawback becomes more and more important.

Haines et al. [HW94] showed how to compute the set of objects found in the shaft between two axis-aligned bounding boxes. By being able to precompute this set of objects, they avoid the overhead of a traditional acceleration structure while still significantly reducing the number of ray-primitive intersection tests that have to be performed. However, this method is too conservative. It requires intersection tests to be performed in the very common case of full occlusion between light and receiver and it can include many unnecessary blockers.

Bala et al. [BDT99] showed how to accelerate ray tracing by constructing four-dimensional interpolants and analytically determining when they could be used without introducing error above a given threshold. However, this method did not scale well with the number of light sources and was constrained to point lights.

While most of these techniques handle the full range of possible scenes, their rendering times are quite slow. For environments with many lights, these algorithms can only be used to generate individual images, not interactive walk-throughs.

## 2.4 Radiosity

Radiosity algorithms compute the radiosity of the surfaces in the scene and cache the resulting values on the surfaces themselves. Radiosity is a spatially variant but directionally invariant quantity defined as:

$$B(x) = \int_{\Omega} L(x \leftarrow \Theta) f_r(x) cos\theta d\Theta \qquad (2.4)$$

where

- $B(x)$ is the radiosity at point $x$
- $L(x \leftarrow \Theta)$ is the radiance incident on a point $x$ from the direction $\Theta$
- $f_r(x)$ is the Bidirectional Reflectance Distribution Function (BRDF), in this case varying over space but not direction
- $\theta$ is the angle between $\Theta$ and the normal to the surface at $x$
- $\Omega$ is the integration domain, the hemisphere centered at $x$ and bounded by the surface being rendered

Because it is directionally invariant, the radiosity value can be reused even as the camera changes positions. Thus, it is an object based rather than image based solution. However, the algorithm does assume that the surfaces are Lambertian ($f_r$ is not directionally variant).

The original radiosity paper [GTGB84] stored radiosity information directly on the polygons making up the model or on fixed-size meshes subdividing these polygons. However, although radiosity is directionally invariant, it does vary spatially and fixed meshes are inadequate in representing the spatial discontinuities in radiosity. Subsequent approaches [CCWG88] allowed for hierarchical meshes whose size adapted to match discontinuities. However they had the problem that a very large number of small mesh elements were needed to capture discontinuities that did not match the subdivision edges of the meshes. [HSA91] introduced radiosity transfer at different levels of the patch hierarchy in order to reduce the high computational cost associated with very fine patches.

Other approaches ([LTG92],[DDP97]) actually fit the mesh discontinuities to analytically calculated radiosity discontinuity locations. However, these approaches were limited in the complexity of the scenes they could support, as the number of potential discontinuity locations quickly exploded, leading to excessive computation and memory use.

## 2.5 Two-pass algorithms

Radiosity algorithms are constrained in only being able to represent Lambertian materials. As a result, algorithms were developed which used a two-phase approach, introduced by Wallace et al. [WCG87]. First, global illumination is com-

puted with a radiosity technique. Then, a ray-tracing phase is used on this radiosity solution to capture the effects of non-Lambertian materials.

Kok et al. [KJ94] presented a method for accelerating the gathering phase of a radiosity algorithm. Their algorithm selects a set of light sources to explicitly sample and determines how well each of those light sources should be sampled based on several criteria, such as whether the light source would require many shadow rays or whether there appears to be a significant gradient due to the light at the patch. Light sources not chosen to be finely sampled were instead sampled at the corners of the patch and thus were interpolated.

Scheel et al. [SSS01] took a similar approach, expanding the criteria for selecting explicitly sampled light sources to include perceptual metrics. Their algorithm also allowed the decision of interpolation vs. sampling to be made separately for the form factor and visibility terms.

While the techniques of both Kok et al. [KJ94] and Scheel et al. [SSS01] allowed certain light sources to be interpolated instead of sampled, they did not provide an acceleration technique for the cases in which lights do have to be sampled. Although Scheel et al. [SSS01] did permit interpolation of some partially occluded sources, this is possible only for broad penumbras, and not for sharper shadows. Scheel et al. [SSS02] further refined this approach by allowing interpolation to happen in object space, making the method more efficient in highly tessellated scenes.

Although these approaches do compute the global illumination solution, they are aimed at generating single images and can take many minutes to generate each image.

## 2.6 Particle methods

The Density Estimation [WHSG97] and Photon Maps [JC98] methods compute global illumination by modeling lighting as a flow of particles. Photons are sent out from the light sources and bounce around according to the geometric and material properties of the scene. Once the photons have been shot and accumulated on the surfaces, their density can be used in a second pass to compute the radiance being emitted from any point on a surface. The primary drawback to these approaches is that an excessive number of photons need to be shot in order to accurately capture sharp shadows. In addition, the particles cannot be displayed directly, requiring either a reconstruction phase to generate smooth functions for display, or some processing while rendering to query local particle density. These problems lead to non-interactive performance.

## 2.7 Many lights

Several of the techniques described above assume a single light. When multiple lights are present, the algorithm is simply repeated for each light and the results are accumulated. This can work well when the scene consists of a small number of lights, but does not scale well in more realistic scenes. The techniques described here attempt to reduce the amount of computation performed in scenes with many lights.

Ward [War94], accelerates the rendering of many lights using a user-specified threshold to eliminate lights that are less important. For each pixel in an image, the system sorts the lights according to their maximum possible contribution (assuming no occlusion). Occlusion for each of the largest possible contributors at the pixel

is tested, measuring their actual contribution to the pixel, and stopping when the total energy of the remaining lights reaches a predetermined threshold. This approach can reduce the number of occlusion tests, however it does not reduce the cost of occlusion tests that do have to be performed and does not do very well when illumination is uniform.

Shirley et al. [SWZ96] propose an approach that subdivides the scene into voxels and, for each voxel, partitions the set of lights into an important set and an unimportant set. Each light in the important set is sampled explicitly. One light is picked at random from the unimportant set as a representative of the set and sampled. The assumption is that the unimportant lights all contribute the same amount of energy.

To determine the set of important lights, they construct an "influence box" around each light. An influence box contains all points on which the light could contribute more than the threshold amount of energy. This box is intersected with voxels in the scene to determine when the light is important. This is an effective way to deal with many lights. However, the approach is geared towards static environments and produces single images since many samples per pixel are required to reduce the noise inherent in sampling the light set.

Paquette et al. [PPD98] present a light hierarchy for rendering scenes with many lights quickly. This system builds an octree around the set of lights, subdividing until there are less than a predetermined number of lights in each cell. Each octree cell then has a "virtual light" constructed for it that represents the illumination caused by all the lights within it. They derive error bounds which can determine when it is appropriate to shade a point with a particular virtual light representation and when traversal of the hierarchy to finer levels is necessary. Their algorithm

can deal with thousands of point lights. The major limitation of this approach is that it does not take visibility (i.e., occlusion) into consideration.

Wald et al. [WBS03] rendered complex environments of millions of polygons and thousands of lights at interactive rates. They did so by constructing a probability density function (PDF) of the light sources for the current image using a few paths in a path tracer. This PDF was then used to determine which lights to render for the current image. However, in order for this approach to work efficiently, they require environments with very high occlusion, where only a small number of light sources affect the lighting in any particular viewpoint.

These techniques can substantially reduce the amount of time to render scenes with high lighting complexity. However, most of them are aimed at generating individual images and are too slow for interactive walkthroughs. Wald's work, while interactive, places significant restrictions on the types of environments where interactive rates can be achieved.

## 2.8 Image-based methods

Finally, we mention some image-based methods. These methods cache rendering information on the image plane.

Hart et al. [HDG99] used an image-plane based flood-fill to propagate blocker information from pixel to pixel. With a list of the blockers affecting each light at each pixel, and under the assumption that the environment is polygonal, they were able to analytically compute soft shadows for complex environments. However, because the technique was image based, they were unable to use this blocker information from multiple viewpoints, so the technique lends itself only to single

image generation and not interactive walkthroughs.

Gershbein et al. [GH00] presented an image-based method for scene relighting. They render the scene from one viewpoint and store in the image enough information to be able to recalculate lighting if any of the light parameters change. They are thus able to change the lighting parameters of dozens of lights in a complex environment at interactive rates. However, this algorithm depends upon a static viewpoint and is therefore unsuitable for interactive walkthroughs.

## 2.9 Conclusion

We have discussed several methods for rendering direct and indirect lighting. Of these, ray tracing is by far the most flexible. It allows us to use any type of geometry, lighting, material, and camera. However, it is also the slowest. In the chapters that follow, we will introduce algorithms that allow us to render direct lighting of complex scenes at interactive rates without compromising the flexibility of ray tracing. It should be noted that our use of the term "direct lighting" includes shadows. This is in contrast to its use in the hardware rendering literature, where the light visibility term is omitted from the calculation.

Of course, we will only be addressing the problem of direct lighting, not the wider issue of global illumination. However, direct lighting is frequently a large portion of the computational budget of a global illumination algorithm, specially in the highly complex environments that will be dealt with in this thesis. Thus, this thesis also makes a valuable contribution to the solution of the global illumination problem.

# Chapter 3

# Local Illumination Environments

Computing high-quality direct illumination at interactive rates in scenes with many lights is a hard problem. Existing hardware approaches do not scale with many lights, have difficulties with shadows, and cannot easily deal with non-polygonal geometry. Software-based approaches, such as ray tracers, are more promising in their ability to scale with complex scenes and lighting [WS01]. However, such approaches also have difficulties dealing with large numbers of light sources.

In this chapter we will introduce *local illumination environments* (LIEs)[1], a world-space caching approach that accelerates direct illumination for a ray tracer in scenes with many lights. LIEs are associated with octree cells covering the volume of the scene. Each LIE caches geometric and radiometric information: for visible lights, the light is stored, for partially visible lights, the occluders that might occlude shadow rays for the region are stored, and for fully occluded lights no information is stored. We demonstrate an example of an LIE in Figure 3.1.

LIEs are based on two observations. First, the majority of the time in ray

[1] This is work done in collaboration with Prof. Kavita Bala and Prof. Donald P. Greenberg and published in [FBG02].

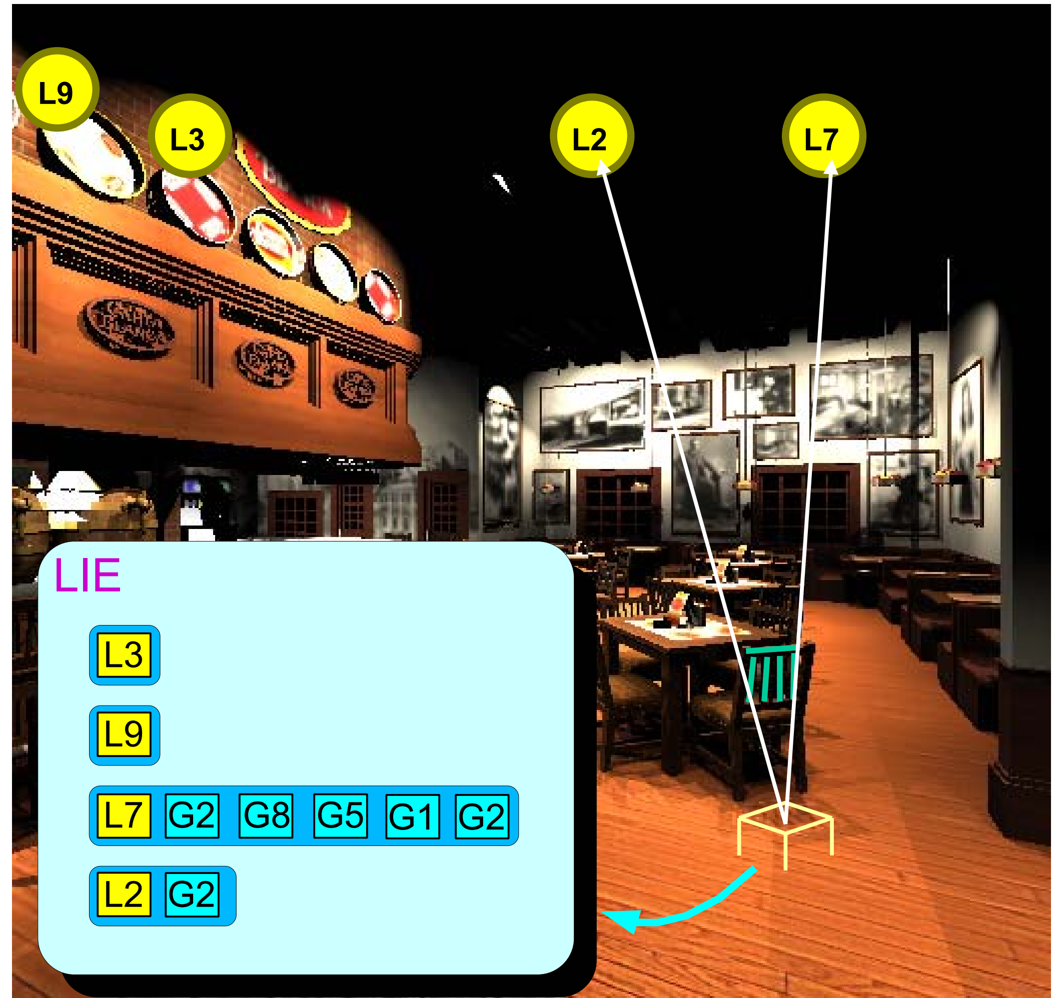



Figure 3.1: A sample LIE. LIEs are associated with world-space octree cells. Each LIE contains a list of lights that are at least partially visible from surfaces bounded by the cell. For each light, the LIE also maintains a list of geometry that occludes visibility to that light. In this example, lights L3 and L9 are fully visible from the cell on the floor. Light L7 is blocked by G2, G8, G5, G1, and G2 from the back of the chair. Light L2 is blocked by G2.

tracing is spent on evaluating the visibility of the light sources in order to render accurate shadows (typically 70-80%). Second, the direct lighting complexity actually observed in any particular region of a scene is usually not very high. This leads us to perform the following optimizations:

- Lights that are not visible at any point on a surface within a cell are completely ignored.
- Lights that are visible at all points on surfaces within a cell do not have any shadow rays cast to them.
- Lights that are visible at some points on surfaces within a cell and not visible at others have shadow rays cast to them, but these shadow rays do occlusion testing only against a minimal set of occluders.

Thus, LIEs accelerate rendering by decreasing the number *and* cost of the expensive visibility computations for shadow rays; it is these visibility computations that make direct illumination so expensive in scenes with many lights. Additionally, a simple perceptual metric based on Weber's law can be used to eliminate the contribution of fully and partially visible lights that are perceptually unimportant.

LIEs have three important properties. First, LIEs permit accurate computation of shadows because they include geometry (the occluders) in the partially occluded regions. Thus, they accelerate visibility without introducing error.

Second, LIEs can be easily integrated into a ray-tracing system. Because they only cache the set of potentially visible lights and the geometry which might occlude them, they do not compromise the flexibility of a ray tracer. In particular, LIEs allow the use of any type of geometric primitive and any type of material that

can be supported by a ray tracer. This also means that LIEs can be used in systems that require a ray tracer to perform direct lighting, such as global illumination systems.

Third, LIEs are designed to work interactively. They are constructed lazily in a view-driven manner as the user navigates the scene. Using 24 processors, we render scenes with hundreds of thousands of polygons with up to 100 lights at 1-2 frames per second, achieving performance improvements from 10× to 30× over a traditional ray tracer.

## 3.1 Local Illumination Environments

A local illumination environment (LIE) consists of a bounding box (cell) along with a set of visible lights, each with a (possibly empty) accompanying set of blocking geometry. LIEs are used to faithfully reproduce the incident radiance on surfaces bounded by the cell. The set of lights and the set of occluders are taken directly from the scene geometry.

In the sections to follow, we will refer to lights as being *unoccluded*, *partially occluded*, or *fully occluded*. We generate LIEs by creating a random point on a surface, a random point on a light source, and testing the visibility between these two points. This process repeats until the algorithm is terminated. A light is *unoccluded* with respect to the cell associated with the LIE if and only if every sampled point on a model surface contained within the cell can see every sampled point on the light. A light is *fully occluded* with respect to the cell associated with the LIE iff no sampled point on a model surface contained within the cell can see any sampled point on the light. Otherwise, a light is *partially occluded* with respect

to the cell associated with the LIE. As discussed in Section 3.1.1 , this approach to constructing LIEs is an approximation that converges to the correct LIE with enough samples.

### 3.1.1 LIE Construction

For optimal performance, local illumination environments should be as simple as possible. Ideally, the local illumination environment for a cell should include only lights that affect illumination for surfaces within that cell. Similarly, for each visible light in a local illumination environment, the associated occluder set should only include the occluders that actually occlude the light.

Computing this minimal occluder set is potentially expensive. Shaft culling [HW94] could be used to construct a set of occluders; however, shaft culling is typically too conservative, including a lot of occluders that are not actually relevant for illumination. Therefore, in our LIE construction, we *sample* the visibility between a surface and light to determine occluder lists, as demonstrated by Hart et al. [HDG99]. This technique is inexpensive and is guaranteed not to add any lights not seen from the cell nor any occluders not actually blocking a light from the cell. Sometimes a lot of samples are required to find all the required lights and occluders. Because of this, some artifacts are visible as LIEs are constructed. However, over time the LIEs converge to an accurate representation of visibility and the artifacts disappear.

For interactive walkthroughs LIEs are computed lazily using a view-driven approach. The LIE constructor picks a point randomly on the image plane and traces a ray from the eye to the closest visible point on a surface. The LIE constructor then constructs an LIE or improves an existing LIE for the smallest enclosing oc-

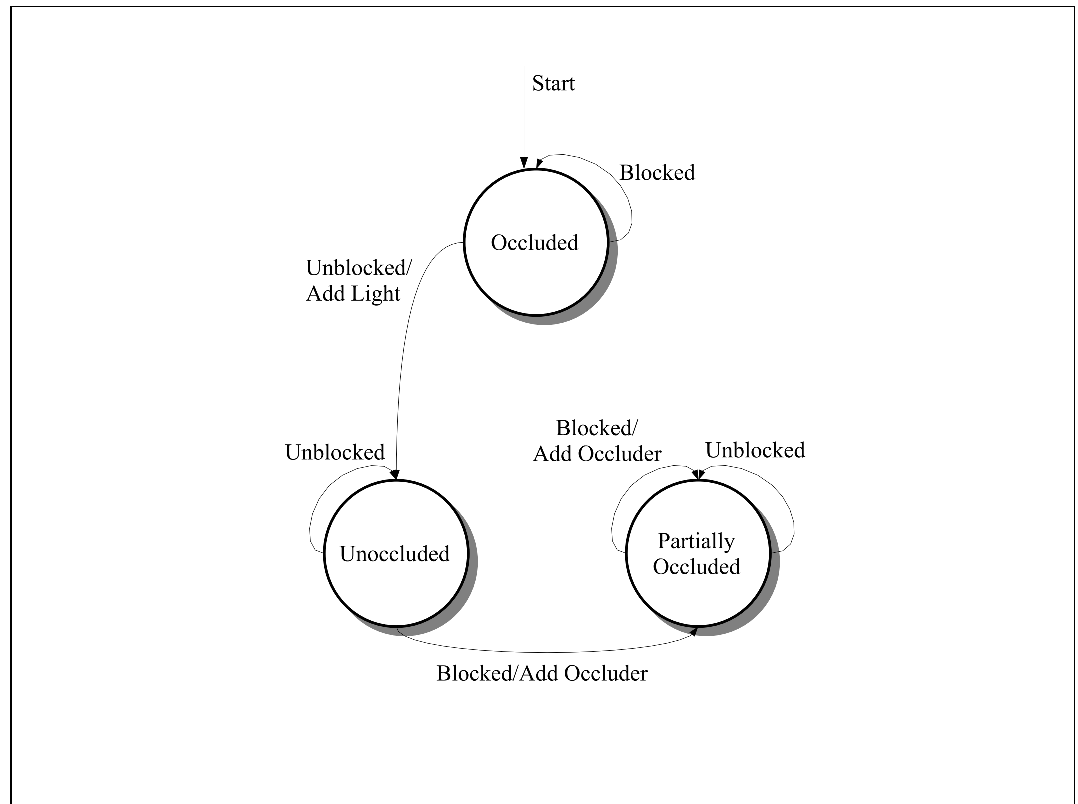



Figure 3.2: State diagram for LIE construction. A light within an LIE starts out in the *occluded* state. Any blocked shadow rays to the light leave it in the occluded state. An unblocked shadow ray puts it in the *unoccluded* state. Once in the unoccluded state, any unblocked shadow rays leave it in the unoccluded state. A blocked shadow ray in the unoccluded state moves the light into the final state, *partially occluded*. Blocked shadow rays in the partially occluded state cause the light to remain in this state but add blockers to the light's blocker list.

tree cell for that point. A ray is cast from the point to random points on all the lights in the scene. If the ray is unobstructed for a particular light and the LIE does not contain that light, that light is added to the LIE (unoccluded case). If the ray is obstructed and the LIE does not contain that particular light, the LIE is not modified (occluded case). If the ray is obstructed but the LIE does contain that light, the obstructing geometry is added to the set of occluders for that light (partially occluded case). This algorithm is shown as a state diagram in Figure 3.2.

We demonstrate LIE construction with an example, illustrated in Figures 3.3 and 3.4.

- **a)** Cast a ray which hits surface G4
- **b)** Traverse the octree to find the leaf cell containing the surface point.
- **c)** Obtain the LIE associated with the leaf cell
- **d)** Cast a shadow ray to light L1. Since the ray is unobstructed, add L1 to the LIE.
- **e)** Repeat the process for light L2.
- **f)** Cast a shadow ray to light L3. This time, the shadow ray is obstructed, so it is not added to the LIE.
- **g)** Repeat the process for another surface point.
- **h)** Find the same leaf cell that contained the previous point.
- **i)** Cast a shadow ray to light L1. The ray is unobstructed, but light L1 is already in the LIE, so it is not modified.

- **j)** G2 obstructs a shadow ray to light L2. Since light L2 is in the LIE, G2 is added to light L2's blocker list.
- **k)** Cast a shadow ray to light L3. Again, the shadow ray is obstructed so light L3 is not added to the LIE.
- **l)-p)** Repeat the processes for a third surface point. This time, G1 blocks a shadow ray to light L2, so G1 is also added to L2's blocker list.

Note that although G1 and G2 block shadow rays to light L3, there is no blocker list for light L3. A blocker list is only constructed if a light is found to be visible from *some* surface point in the cell and this list is empty if the light is visible from *all* surface points in the cell. For the very common case where a light is completely occluded, we can completely eliminate all shading calculations for this light. This is a key advantage of this algorithm since complete occlusion is common yet difficult to prove in general.

Different parts of the scene will have different illumination complexities. If it is determined that the LIE for a certain octree cell has become too complex, the cell is subdivided. We currently measure complexity as the total number of occluders in the LIE. The LIEs for the child cells are then generated from scratch. This allows us to maintain the invariant that only lights that can actually be seen from a cell are in an LIE and only occluders that occlude a cell's view of a light are in its occluder list. Currently we use a fixed maximum depth for the octree subdivision. We have found that the optimal maximum depth in terms of the LIE performance/LIE generation tradeoff varies as a function of the scene. For example, while a maximum depth of six was appropriate for a simple scene with a few thousand polygons and one light, a value of ten worked better for scenes with

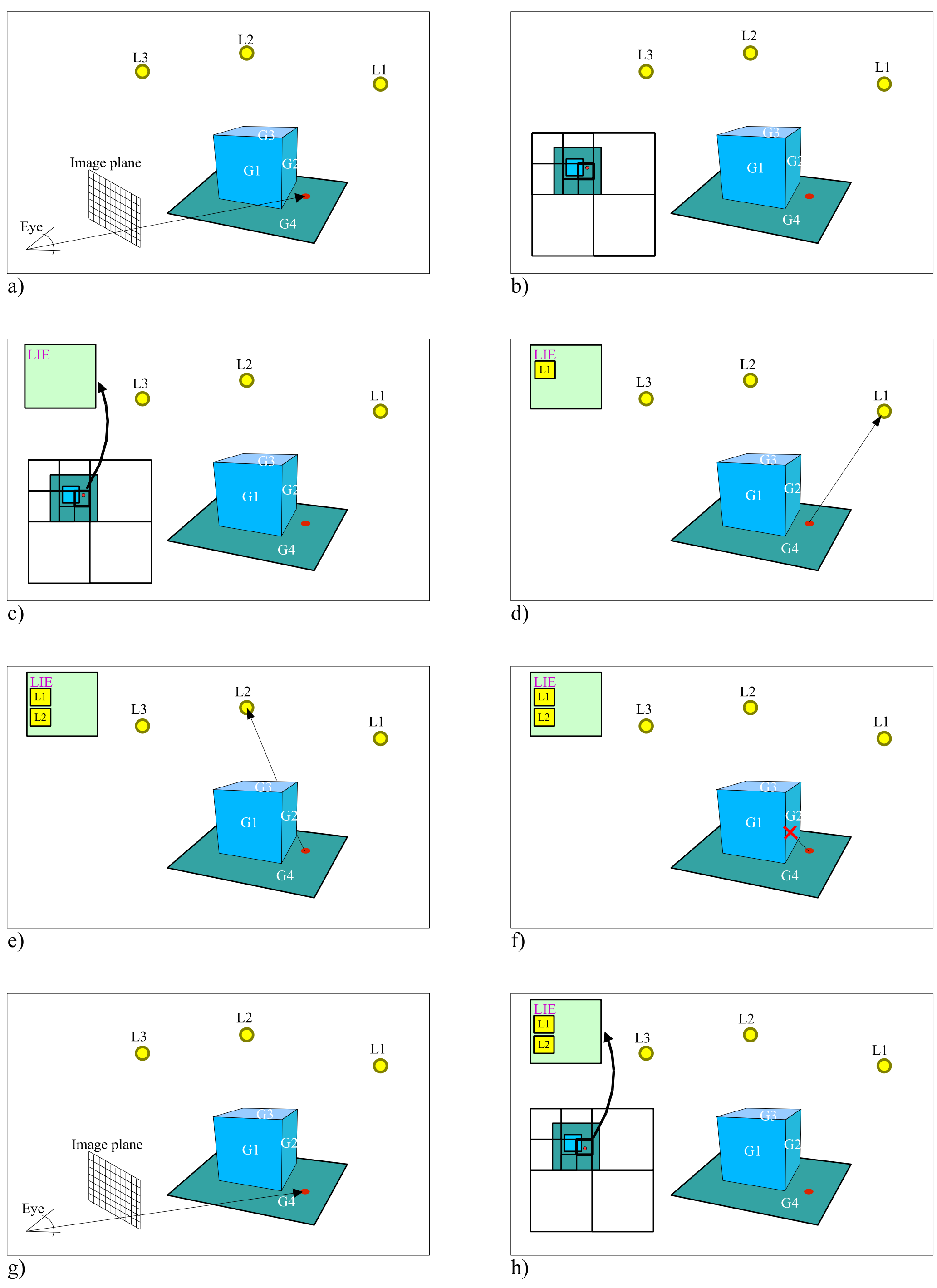



Figure 3.3: LIE construction example, part 1.

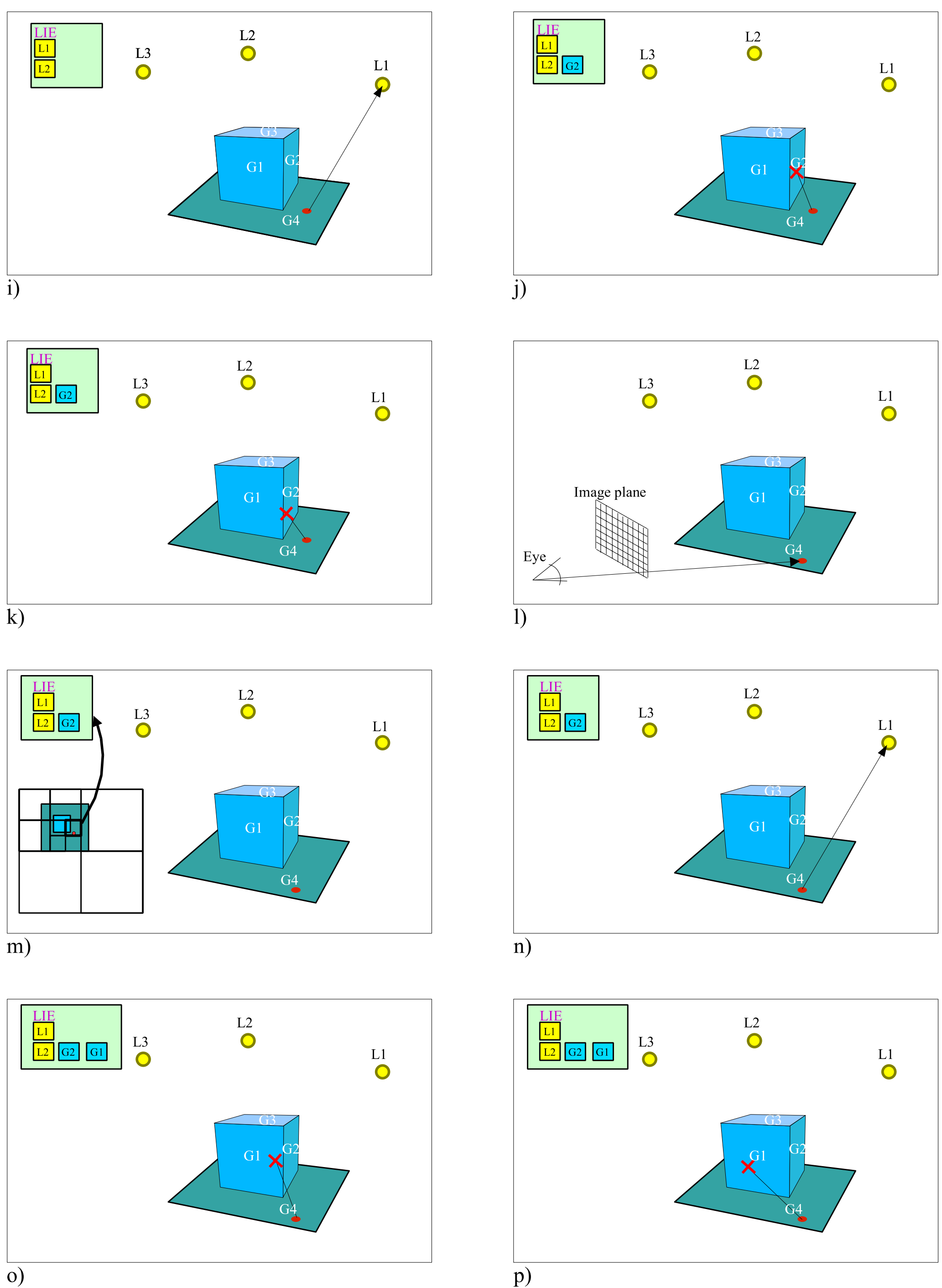



Figure 3.4: LIE construction example, part 2.

hundreds of thousands of polygons and dozens of lights.

### 3.1.2 Shading Using LIEs

When rendering a frame, the ray tracer is used to determine the closest visible object for each pixel. The appropriate LIE used for shading this point is found by descending the octree hierarchy. For each partially occluded light in the LIE a ray is cast from the point to be shaded to a sample point on the light. This ray is tested for intersection with the list of occluders for that light. If the ray is not blocked, the incident radiance from the light source is multiplied by the BRDF and the form factor and added to the exitant radiance. Monte Carlo sampling is used to integrate the contribution from area light sources. For fully visible lights in the LIE no visibility computation is required.

LIEs tend to be small resulting in fewer intersection tests than with traditional ray-tracing acceleration approaches. The simple structure of the occluder lists also incurs little overhead.

## 3.2 Masking

In scenes with large lighting complexity, parts of the scene could be illuminated with several lights that are significantly brighter than the rest of the lights illuminating that portion of the scene. These bright lights "mask" the effect that the dimmer lights have on the illumination of that region. Thus, the bright lights allow us to ignore the contributions of the dimmer lights. Ignoring these dim lights lets us further decrease the computational cost of rendering direct illumination.

We demonstrate this in Figure 3.5. All lights in the environment have been

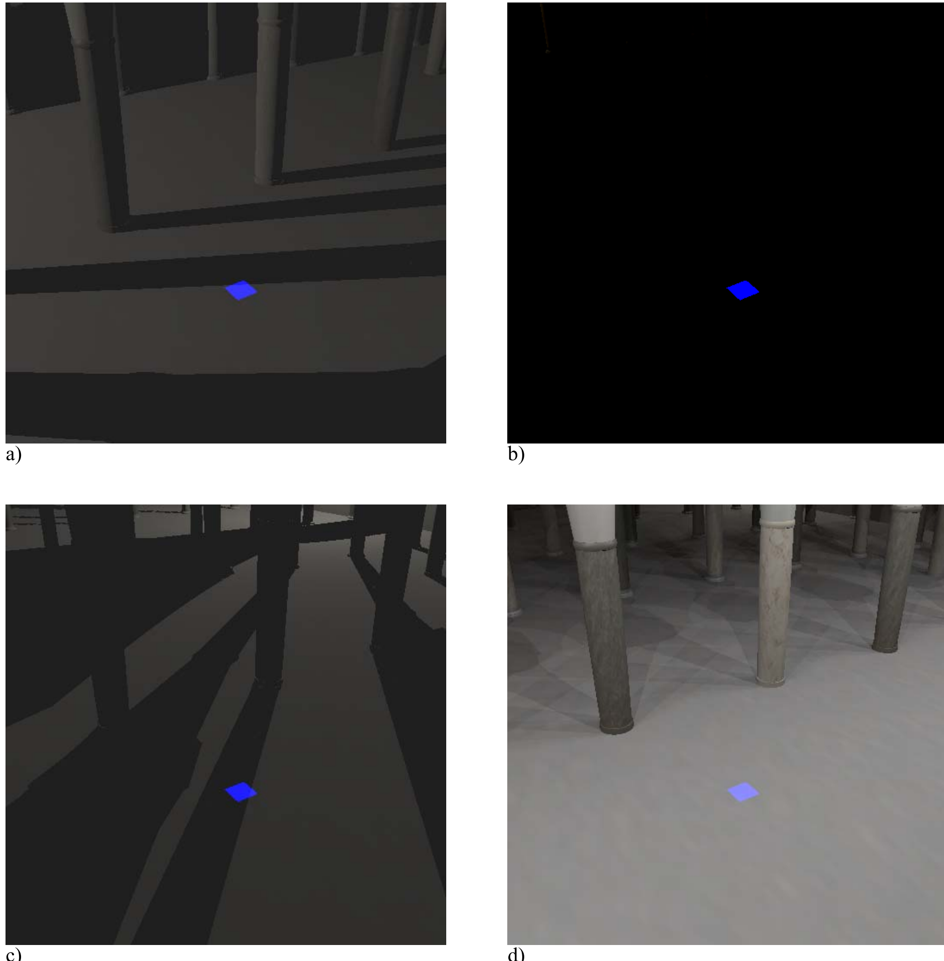



Figure 3.5: a-c) Images of Mosque de Cordoba with only one dim light on in each image. d) When all lights in the environment are turned on, the shadows cast by the dimmer lights are not visible.

turned off. We turn on one light in each of the first three images. One can see that the cell marked in blue has several shadows going across it from dim (distant) lights. However, when all lights are turned on, the energy of the brighter (nearby) lights is much higher than that of the dimmer lights and thus the brighter lights mask out the shadows of the dimmer lights. We would like to identify the cases where bright lights mask dimmer lights and avoid rendering the dim lights in those regions.

In order to determine exactly which dim lights can be safely ignored we use the contrast sensitivity function. The contrast sensitivity function, shown in Figure 3.6, describes the difference in illumination of a feature from its surroundings necessary for a viewer to "just notice" the feature. Weber's law [HF86],

$$\frac{\triangle I}{I} = k \tag{3.1}$$

where $I$ is the surrounding intensity in a region and $\triangle I$ is the "just noticeable difference" from this intensity, specifies that this ratio is a constant over a wide range of intensities. The value of this constant is about 1-3%.

To use Weber's law we have to obtain some estimate of the overall energy over an octree cell. As LIEs are constructed our system keeps track of the minimum and maximum exitant radiance over a region due to each light and sorts the lights based on their maximum exitant radiance. We then remove lights from the LIE starting with the light whose maximum contribution is smallest. We accumulate the error introduced, and stop removing lights when this error exceeds 2% of the minimum radiance for all the lights combined.[2]

[2] A similar approach is used by Ward [War94], on a per-pixel basis.

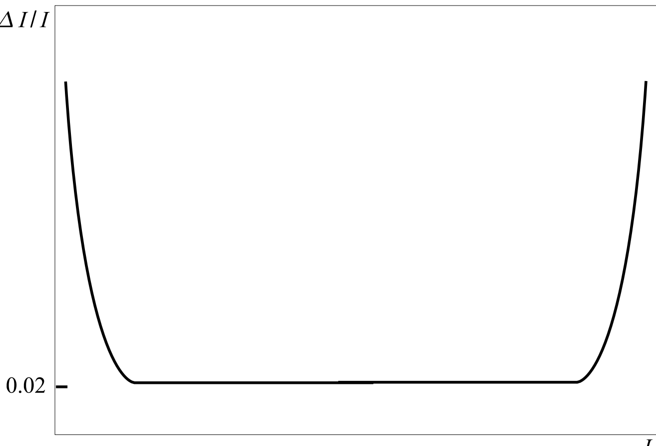



Figure 3.6: The *contrast sensitivity function* shows the minimum change in intensity ($\triangle I$) from the background intensity ($I$) that is noticeable by the human visual system. Over a broad range of intensities, this ratio is 0.02, only deviating in below-moonlight and above-sunlight lighting conditions.

Figures 3.7 and 3.8 show a particular section of the Mosque de Cordoba test scene. The scene happens to have a large number of relatively dim lights with a row of bright lights running down the main corridor seen in the picture. The columns surrounding the corridor cast shadows on the floor due to all the surrounding lights. However, the brighter lights above the corridor completely mask these shadows so that they are not visible in the image.

The bottom visualizations in each figure show the cost of rendering each pixel without and with masking. The generated images at the top of the two figures are perceptually identical. However, the cost to render the pixels in the corridor is significantly reduced when using masking as is particularly noticeable on the floor. The performance difference is shown in Figures 3.10, 3.11, and 3.12. Note that the difference image and results are for a converged set of LIEs. Non-converged LIEs would show visibility artifacts unrelated to masking.

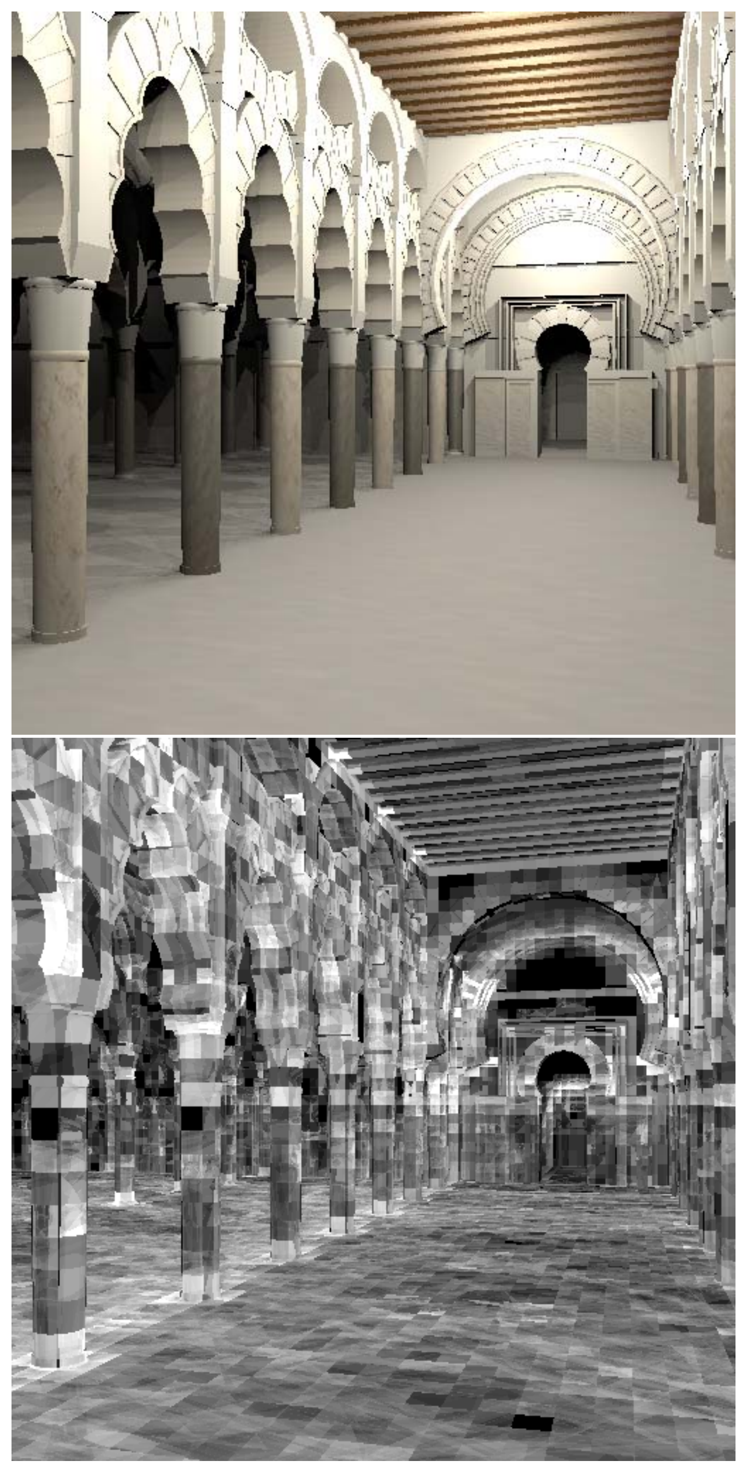


Figure 3.7: Top: Image of Mosque de Cordoba scene with no masking. Bottom: Cost to render each pixel without masking (whiter is more expensive).

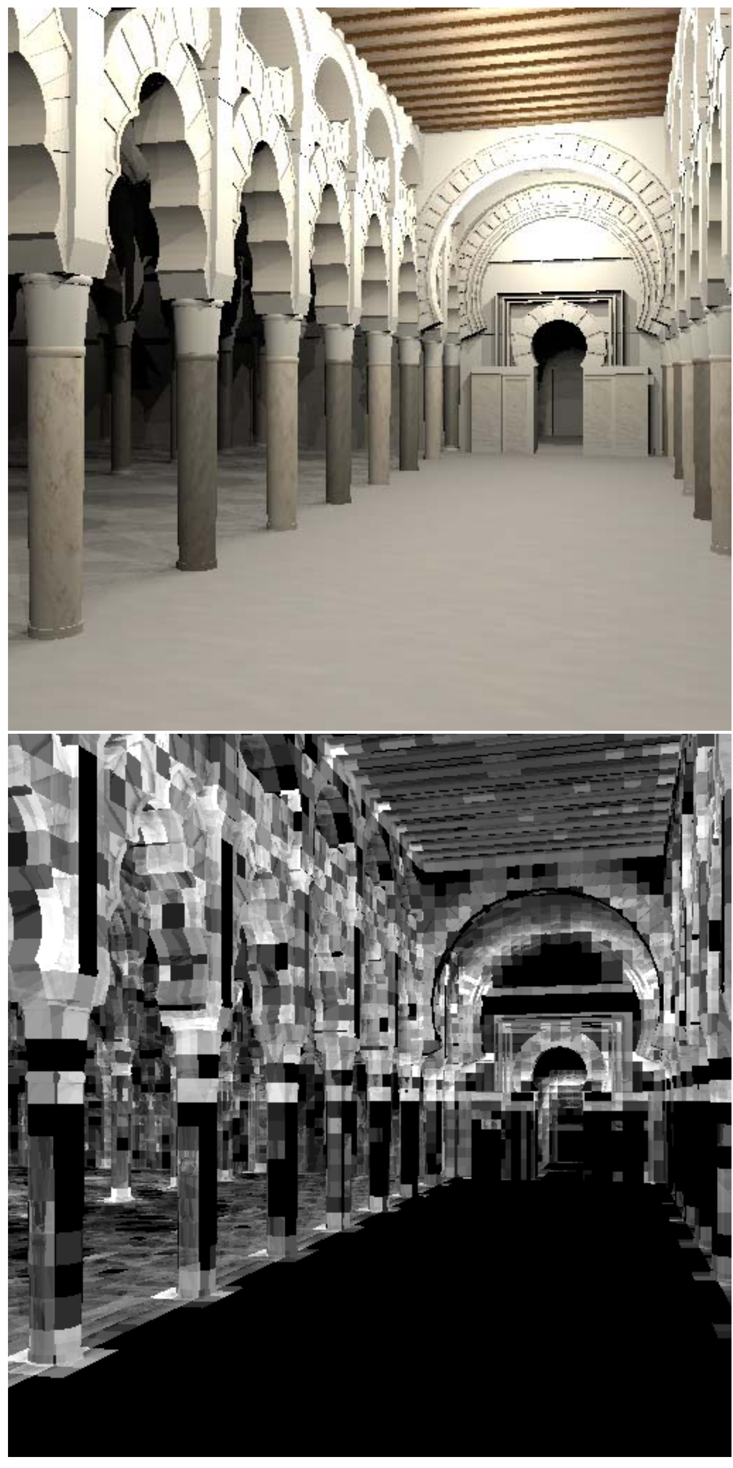


Figure 3.8: Top: Image of Mosque de Cordoba scene with masking. Bottom: Cost to render each pixel with masking (whiter is more expensive).

## 3.3 System Description

This section describes the overall structure of the system. The system is view-dependent, but caches data in a view-independent data structure that can later be reused by view-dependent renderers. It can make use of parallel processing to accelerate rendering. It also works online and can be used immediately upon loading the model if the user is willing to tolerate some error.

The system is split into two modules. As the user navigates the scene, the viewpoint is sent to the LIE constructor. This module continuously computes new LIEs if needed or refines existing LIEs. The LIE constructor typically runs on a single computer and communicates changes in the computed LIEs to the shaders which run on separate computers.

The LIE shaders consist of several parallel renderers that use LIEs to render the pixels assigned to them. The renderers are not synchronized with the LIE constructor, so the shaders do not have to wait on the results of LIE construction. Also, computing any one pixel does not depend on the computation of any other pixel, so there is no communication among the shaders. The computed pixels are sent over a local area network where they are assembled into an image.

At startup, the model is distributed to the LIE constructor and the LIE shaders. A regular grid acceleration structure is constructed and replicated on each system. This regular grid is used to accelerate both the ray casts needed to construct LIEs and the primary visibility ray computation used in both the constructor and the shaders.

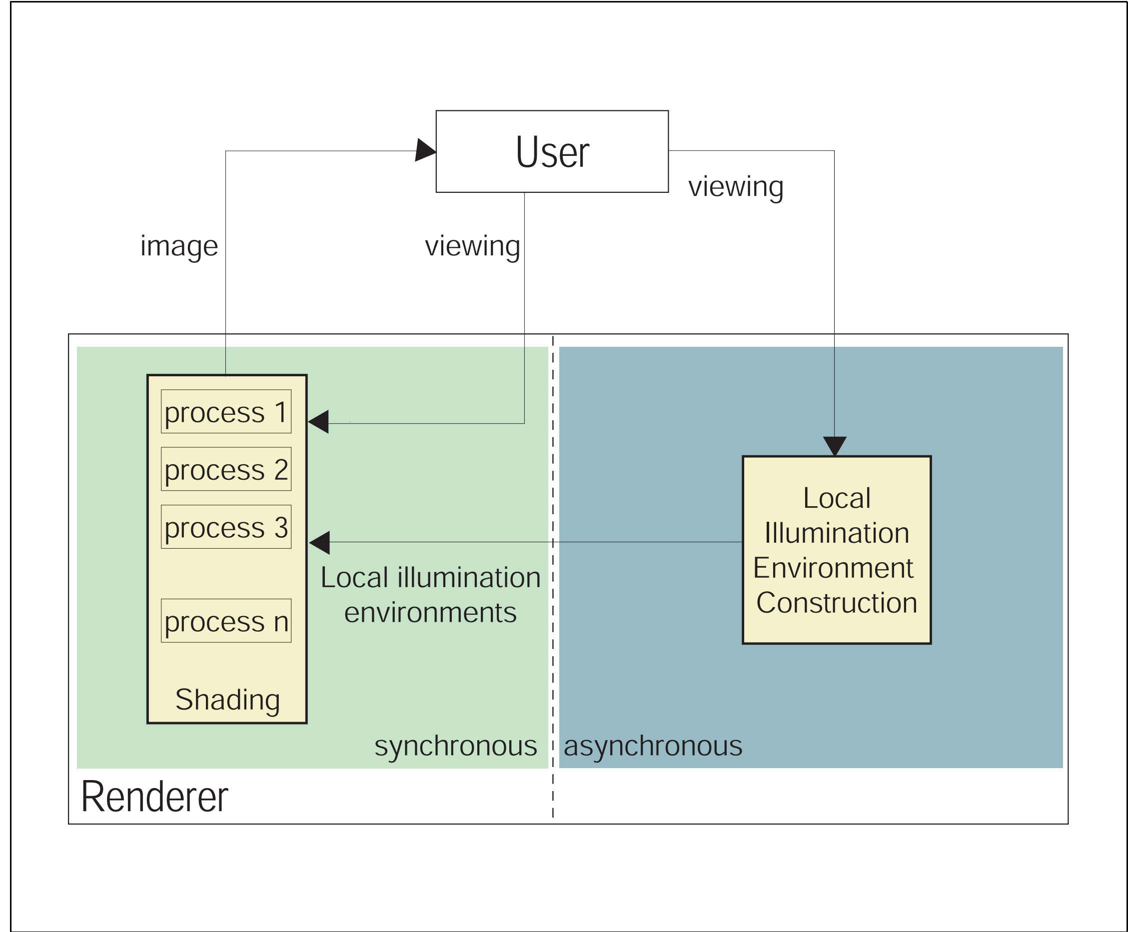



Figure 3.9: System structure. The user navigates on a display computer, sending camera positions to both the LIE constructor and the LIE shaders. The LIE shaders partition the image and work synchronously and in parallel to generate a rendered image to be sent to the display computer. The LIE constructor works asynchronously to generate LIEs to be used by the shaders.

## 3.4 Results

We tested our algorithm on several scenes. We present results for the following three scenes that differ in their lighting, materials, and complexity. The Science Center is a simple scene with 7,000 polygons and one large area light source, with completely diffuse materials. Bar is a scene with 240,000 polygons and 70 spotlights with mostly diffuse materials and a glossy floor.[3] Mosque de Cordoba is a scene with 980,000 polygons, 100 omnidirectional point light sources, and glossy columns.[4]

LIE construction was done on a single dual-processor Pentium-4 PC running at 1.7 Ghz. LIE shading was done on 10 dual-processor Pentium-4 PCs running at 1.7 Ghz. A 100 Mbit Ethernet network connects the machines. The LIE constructor communicates with the shaders on this network. Rendered pixels are also transmitted on this network. Both the standard ray tracer and the LIE system were written in Java and ran on the Sun Java Virtual Machine version 1.3.1.

Before obtaining timing results, the LIE data was precomputed by walking around the scenes until the LIEs were mostly converged. The Science Center required only seconds of precomputation, while Bar Carta Blanca required a few minutes and Mosque de Cordoba required about an hour. Timings are for particular views within the scenes and are representative of timings obtained for other views. Timings are in seconds per frame for a 512x512 image and scale linearly with the number of pixels.

For comparison, Figures 3.10, 3.11, and 3.12 show the performance of a standard ray tracer, the same ray tracer modified to use Ward's algorithm for many

[3] The Bar scene was modeled by Guillermo M. Leal LLaguno.

[4] The Mosque de Cordoba scene was modeled by Ivan Rossello and Yasemin Kologlu.

lights, and our LIE based ray tracer. Note that all these ray tracers have been parallelized. The timings shown are for one representative viewpoint within each scene. The algorithm shows consistent speedups over traditional ray tracing. The standard ray tracer uses a two-level hierarchical regular grid acceleration structure whose resolution adapts to the size of the model. As model complexity increases and shadow-ray costs for the standard ray tracer go up, our speedup increases, going from 11× for the Science Center to 29× for the more complex Mosque de Cordoba.

The Science Center scene (Figure 3.10) shows that we can achieve substantial speedups even in small environments where the cost of traditional shadow rays is not large. In this scene the performance improvement is mainly due to not having to cast shadow rays for fully occluded or fully visible lights. This cost savings can be substantial for scenes with large area lights because of the large number of shadow rays required.

The Bar scene (Figure 3.11) demonstrates the algorithm's performance on a complex scene with many point lights. In this scene, accelerating shadow computation for the partially occluded regions is more important than in the previous scene due to the high cost of shadow rays. LIEs are effective at accelerating these rays. Although this scene contains many lights, masking is not very effective. In this environment, far away lights tend to be fully occluded and thus incur zero cost in rendering.

The Mosque de Cordoba scene (Figure 3.12) demonstrates the additional performance improvements that can be obtained by taking advantage of light masking. The mosque is lit by a row of 10 bright lights in the central aisle, and 90 dimmer lights to the sides. Light masking is particularly effective in reducing the cost of

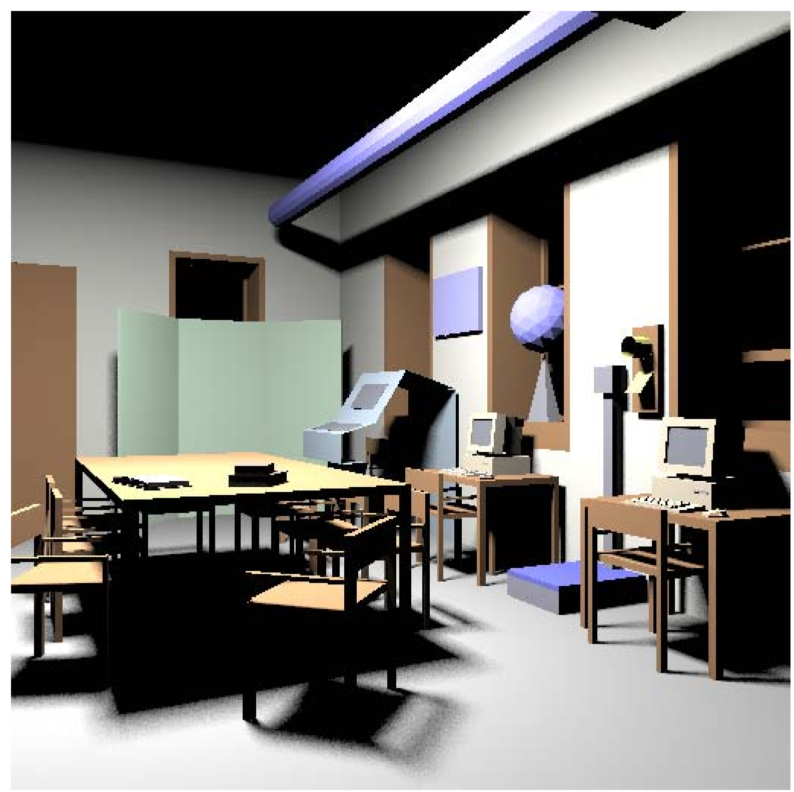
direct illumination in such a scene in the regions near the bright lights.

These tables break down the performance into three components. The visibility component is the cost of casting a ray from the eye to the first surface using a conventional ray tracer. This component is not accelerated by our algorithm. However, we present it as a point of comparison since the visibility computation is a lower bound on the time required to render a frame. The next component is the time spent rendering unoccluded lights. This involves a BRDF evaluation, a light emission evaluation and a form factor computation for each light. The final component is time spent rendering partially occluded lights. This involves intersection testing with the set of occluders in the LIE for each partially occluded light as well as the emission and BRDF evaluation if the light is determined to be visible. For area lights, these computations have to be done once for each sample point on the light.

The table shows the performance breakdown for LIEs without masking enabled. A significant effect from masking can be seen in the Mosque de Cordoba scene where enabling masking further decreases the time for rendering partially occluded lights by about 0.25 seconds.

The breakdown shows that in several cases the cost of performing the direct lighting computations is the same order of magnitude as the cost of performing the visibility computation. Thus, the LIE is effective at accelerating direct illumination. The table also shows that neither the rendering of fully visible lights nor the rendering of partially occluded lights is a performance bottleneck. Instead, both have to be further optimized for better performance.

Memory usage for the LIE data structure is not very high. Although it does have to store a collection of lights and occluders for each octree cell, these are



**Time comparisons**

| Method | Time | Speedup |
|---|---|---|
| Standard ray tracer | 15.0s | 1.0x |
| Ward method | 15.0s | 1.0x |
| LIEs without masking | 1.30s | 11.5x |
| LIEs with masking | 1.30s | 11.5x |

**Time breakdown**

| Aspect | Time |
|---|---|
| Visibility | 0.35s |
| Rendering unoccluded lights | 0.05s |
| Rendering partially occluded lights | 0.90s |

Figure 3.10: Science Center

**Time comparisons**

| Method | Time | Speedup |
|---|---|---|
| Standard ray tracer | 10.1s | 1.0x |
| Ward method | 10.1s | 1.0x |
| LIEs without masking | 0.69s | 14.6x |
| LIEs with masking | 0.69s | 14.6x |

**Time breakdown**

| Aspect | Time |
|---|---|
| Visibility | 0.48s |
| Rendering unoccluded lights | 0.14s |
| Rendering partially occluded lights | 0.07s |

Figure 3.11: Bar

**Time comparisons**

| Method | Time | Speedup |
|---|---|---|
| Standard ray tracer | 35.0s | 1.0x |
| Ward method | 11.5s | 3.0x |
| LIEs without masking | 1.45s | 24.1x |
| LIEs with masking | 1.20s | 29.2x |

**Time breakdown**

| Aspect | Time |
|---|---|
| Visibility | 0.59s |
| Rendering unoccluded lights | 0.34s |
| Rendering partially occluded lights | 0.52s |

Figure 3.12: Mosque de Cordoba

merely references to scene geometry and thus take up little space. For the 980,000 triangle Mosque de Cordoba model, 50 megabytes of data were used by the LIE data structure. This is less than the memory used for the model.

## 3.5 Conclusions

We have introduced an approach for accelerating direct illumination calculations using local illumination environments. LIEs spatially cache visibility and radiometric information in a scene. This caching allows fast rendering by reducing the number of shadow rays cast and by decreasing the cost of the shadow rays that must be cast. LIEs are flexible and can be easily introduced into a ray-tracing system. We have also shown how perceptual masking can be used in conjunction with LIEs to reduce the number of shadow ray casts in regions where shadows are hard to perceive.

We have implemented a system that renders direct illumination at 1-2 fps using LIEs on a cluster of PCs. Our system demonstrates performance improvements over conventional ray tracing of 10× to 30×. We believe LIEs can be easily integrated in other systems to accelerate shadow ray computations for interactive walkthroughs.

In the following chapter, we will explore in detail the parameters that go into generating LIEs. This will provide some guidance in generating optimal LIEs, as well as showing us the limits of their usefulness.

# Chapter 4

# Local Illumination Environment Construction

Once constructed, local illumination environments are fast and simple to use. However, the process of LIE construction can be time-consuming and complex. In this chapter, we will define the process of LIE construction more formally, explore different methods of constructing LIEs, and closely analyze the error involved in construction.

## 4.1 Definitions

First we will define the concepts of *shadow space region* and *view shadow space region*, as they are fundamental in the construction of LIEs.

Let $S$ be the set of all points on surfaces in the scene and let $L$ be the set of all points on light sources. We define *shadow ray space* as $S \times L$, the set of all shadow rays in the scene. LIE construction consists of sampling shadow ray space to find light sources and blockers.

Let us denote the set of all points on surfaces bounded by octree cell $i$ as $S_i$. Let us further refer to the set of points on the surface of light source $j$ as $L_j$. Let $G$ be the set of all geometric objects in the scene. We define $B : S \times L \rightarrow G \cup \perp$ as the function that describes the first geometric object in the scene to block a ray from a surface point to a point on the light source, with value $\perp$ if the ray is unoccluded. $B$ describes whether a light will be added to an LIE due to a sample (when $B$ is $\perp$) or, if the light is already part of an LIE, which blocker will be added to the blocker list for the light. We illustrate shadow ray space in Figure 4.1 and shadow ray space with cells in Figure 4.2.

The collection of sets

$$\{R_{ijk} \subset S \times L \text{ where } (s, l) \in R_{ijk} \text{ iff } s \in S_i, l \in L_j, \text{ and } B(s, l) = k\} \tag{4.1}$$

partitions shadow ray space into *shadow ray regions* of different sizes. Each region is the set of surface points in cell $i$ and light $j$ for which $k$ is the first occluder (or where there is no occlusion if $k = \perp$). One sample in each region of non-zero measure is sufficient and necessary to form a complete set of LIEs. Not sampling every region leads to erroneous LIEs. Sampling a region more than once is wasted work.

Let $v^c(p) \in S$ be the first intersection of a ray constructed from the focal point of camera $c$ and passing through the point $p$ on the image plane. The collection of sets

$$\{R^c_{ijk} \subset I \times L \text{ where } (p, l) \in R^c_{ijk} \text{ iff } v^c(p) \in S_i, l \in L_j, \text{ and } B(v^c(p), l) = k\} \tag{4.2}$$

partitions $I \times L$, which we'll refer to as *view shadow ray space* and illustrate in Figure 4.3. One sample in each region of non-zero measure of view shadow ray space is sufficient and necessary to form a complete set of LIEs for that camera

(that is, no incomplete LIEs are visible).

Thus, if we generate one shadow ray in each of these view shadow ray regions, we will have complete LIEs and have zero error in our generated image. Intuitively, each of these regions represents the projection onto the image plane of the intersection of a cell with the shadow caused by a particular piece of geometry and a particular light.

A random sampling of view shadow ray space will eventually sample each of these regions and therefore form a complete set of LIEs. However, random sampling will sample regions more than once, even though only one sample is required per region. This is the reason why error reduction via random sampling eventually levels off.

The number and size of these regions affect the rate of error reduction. The larger the number of regions, the larger the number of samples that are required to eliminate all errors. The smaller the regions, the lower the error reduction per sampled region and the lower the probability that a given region will be found.

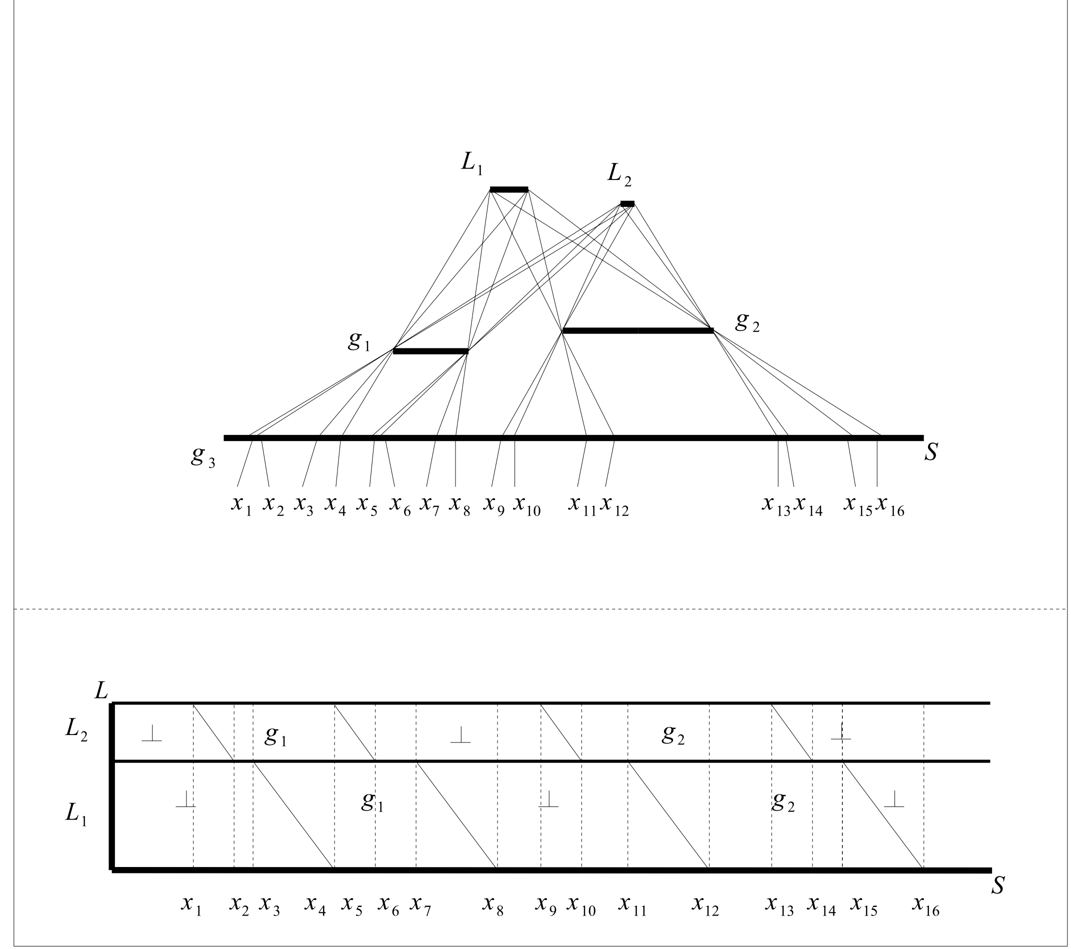



Figure 4.1: Top: A simple two-dimensional scene with two lights, two occluders, and a receiving surface; the $x_i$ represent points of shadow discontinuities on the receiving surface. Bottom: The shadow ray space representation of this scene. The shadow discontinuities partition this space into shadow ray regions that must be sampled.

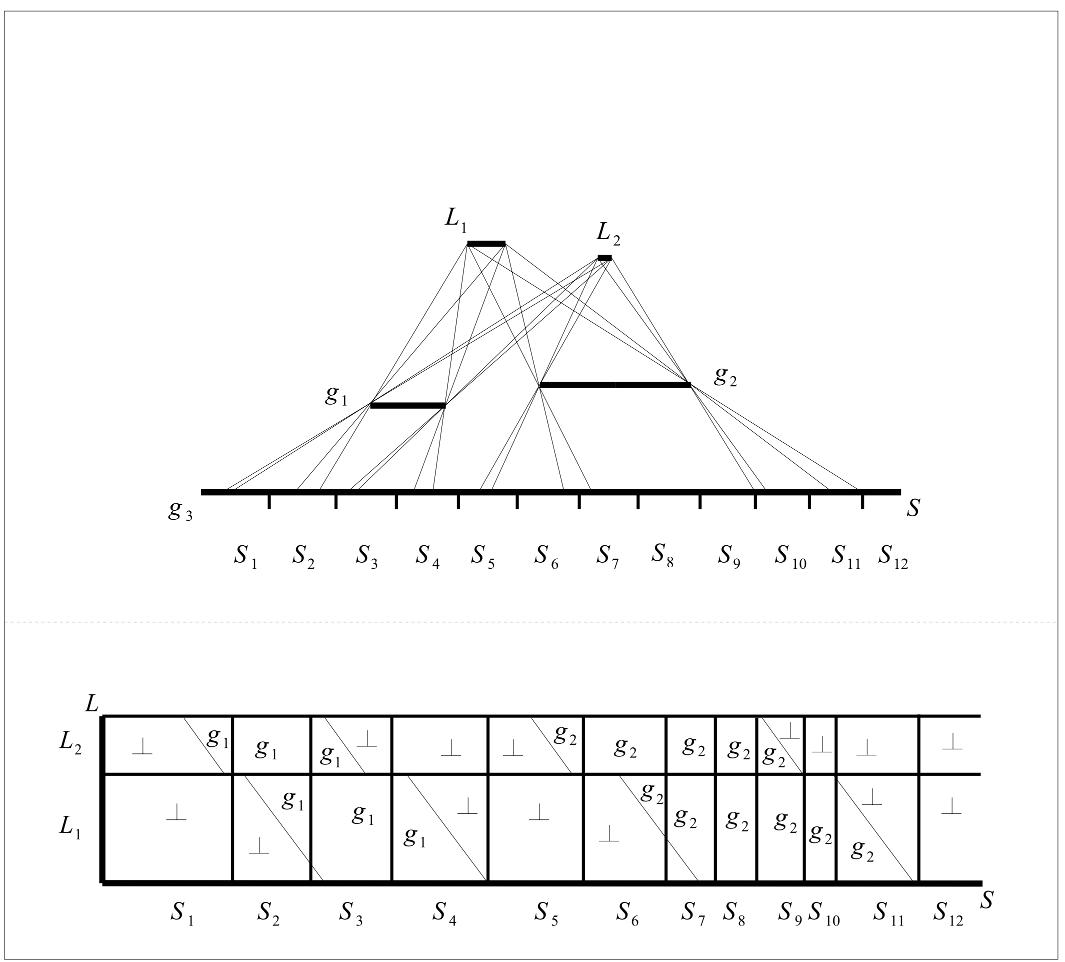



Figure 4.2: The introduction of cells ($S_i$) further partitions shadow ray space, requiring more samples.

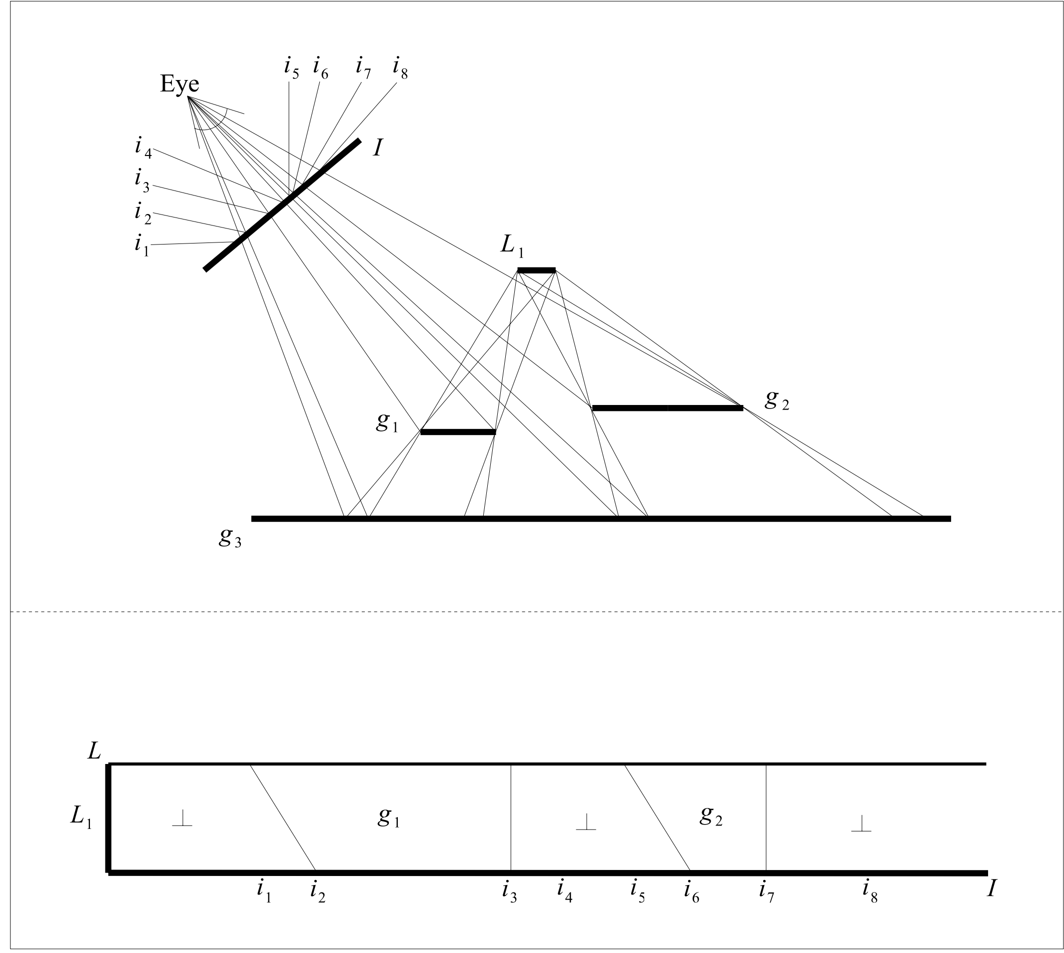



Figure 4.3: View shadow ray space is a reparametrization of shadow ray space. In this simpler version of the two-dimensional model, we see the view shadow ray regions generated by one light and two blockers on a receiving surface. The $i_i$ represent points of possible discontinuity either in shading or visible surface.

## 4.2 Testing methodology

This chapter describes several LIE construction experiments we performed. This section describes the testing methodology we used to evaluate these experiments.

We connected an LIE constructor running on a single processor with an LIE shader also running on the same processor. We alternated execution of the constructor and shader. All construction times take into account only time spent in the constructor, likewise for shading times.

We chose one representative viewpoint for each scene. We define the error function as

$$E = \overline{min(\frac{|L_{LIE} - L|}{L}, 1.0)} \tag{4.3}$$

where the average is over the pixels in the image, $L_{LIE}$ is the radiance computed at a pixel by using LIEs, and $L$ is the radiance computed at a pixel by a reference direct illumination shader. This error function gives us a relative measure of how far the generated image is from the reference image. The clamping to one ensures that no single pixel has an overwhelming influence on the total error. This is particularly important in regions where the reference image is very dark or black and the LIE-generated image is bright, due to an undiscovered occluder.

Unless otherwise noted, fixed-size cells were used (the leaves of a depth-6 octree). Image plane sampling was performed using a Halton sequence QMC algorithm, as covered by Glassner [Gla95].

We performed tests on three different scenes:

- **Bar.** A model of a bar. It contains 235,000 triangles, diffuse and glossy surfaces, 62 spot lights and 7 omnidirectional point lights. This is the same

bar model used in the previous chapter.

- **Mosque de Cordoba.** A model of the Mosque de Cordoba. It contains 950,000 triangles, diffuse and glossy surfaces, and 145 omnidirectional point lights. This is a version of the model used in the previous chapter with greater lighting complexity.

- **Grand Central Station.** A model of Grand Central Station. It contains 1.65M triangles, diffuse surfaces, 18 omnidirectional point lights and one spot light.[1]

## 4.3 Fixed cell size

One method of dividing the scene into cells is fixed-depth subdivision. An octree is constructed that bounds the scene and a fixed maximum depth for the tree is determined a priori. Whenever a surface point is sampled for LIE construction, we check to see whether a cell of the given fixed depth contains the surface point. If one does not, then the tree is subdivided until one does. LIEs are then stored only at the leaves of the tree at the given fixed depth. This gives us a fixed depth at which LIEs are stored without generating many leaf voxels in empty regions of space.

We experimented with different maximum depths. A larger depth implies more subdivision and therefore a smaller cell size at which LIEs are stored. Smaller cell sizes are desirable because the size of the blocker lists of a converged LIE for a given cell can be no larger than that of its parent cell, as demonstrated in Figure 4.4,

[1] The Grand Central Station scene was modeled by Anne Briggs, Dana Getman, Yasemin Kologlu, and Mike Donikian.

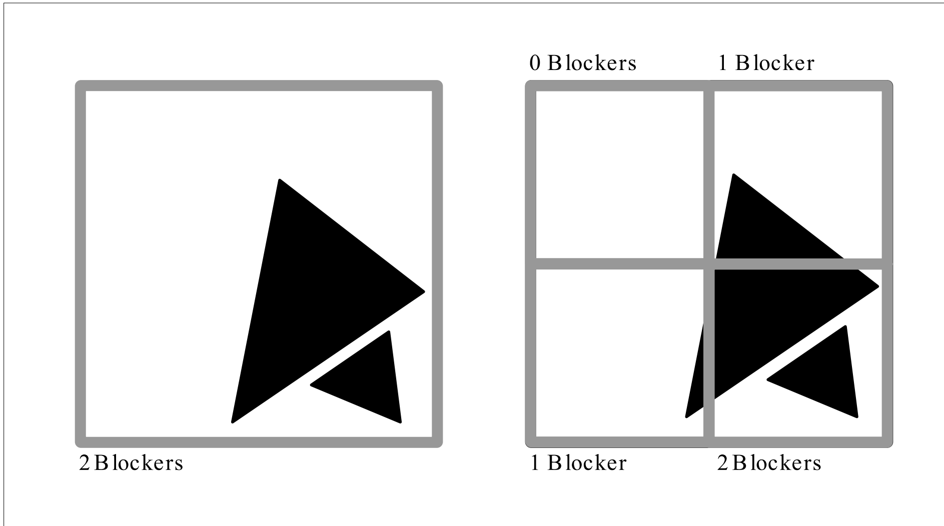



Figure 4.4: Child cells are never more complex than their parents. Here we show a cell with two blockers being subdivided. Each of the child cells ends up with two or fewer blockers.

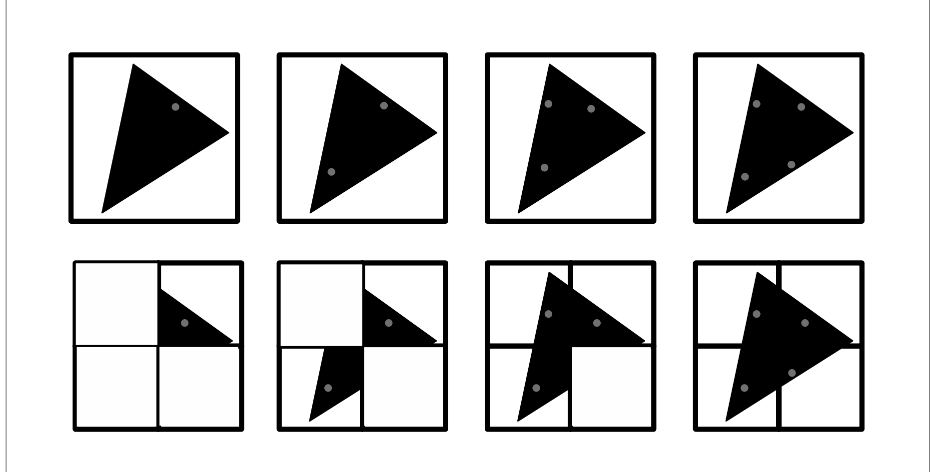


Figure 4.5: Child cells require more samples than their parents. Here we show a parent cell being sampled (top) and the same sampling pattern on its child cells (bottom). As soon as any sample falls within the blocker on the parent cell, the whole cell is rendered correctly. We need at least four samples to render the four child cells correctly.

and is frequently smaller. Smaller blocker lists are less expensive to render. On the other hand, smaller cells require more samples, as shown in Figure 4.5, and therefore lengthen construction time.

In the following pages we show the construction times and rendering rates for the three scenes:

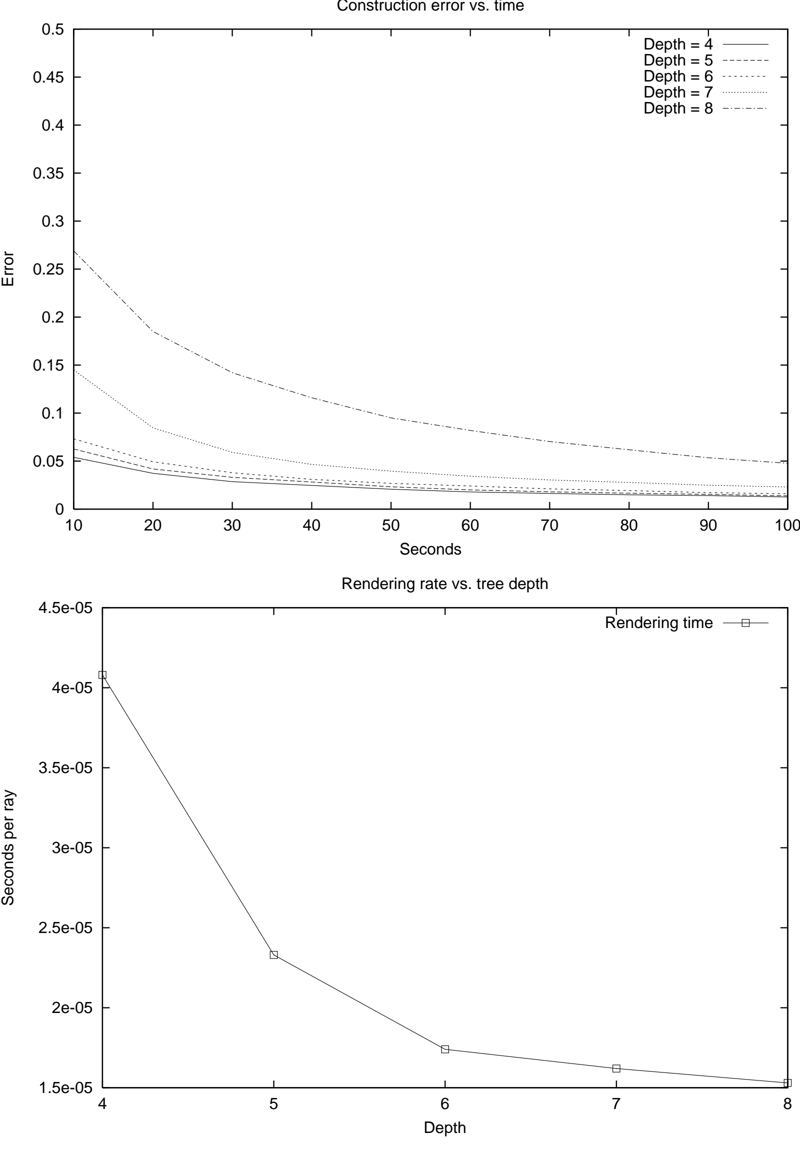



Figure 4.6: Fixed tree depth (Bar)

The results show that, as cell size decreases, it takes longer to reduce the error

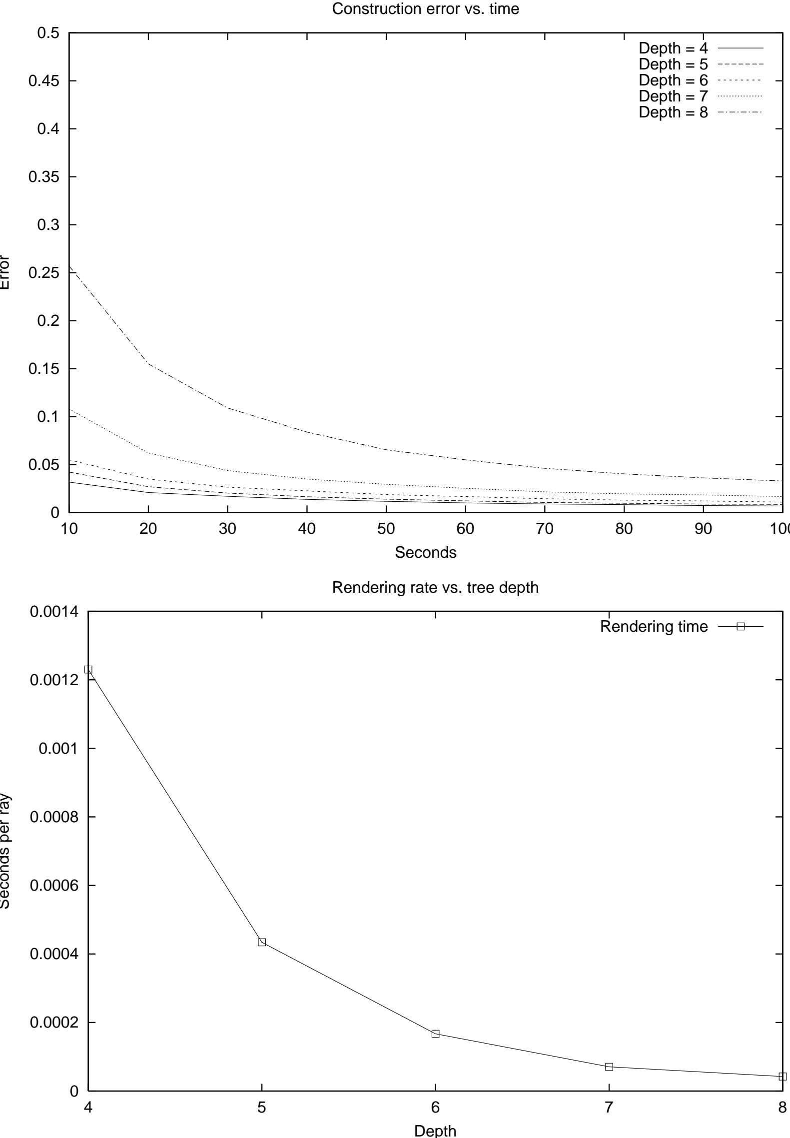



Figure 4.7: Fixed tree depth (Grand Central Station)

to a given level but the rendering rate increases as cell size decreases.

For these test scenes, a depth of six seems appropriate, since greater depths do not yield much improvement either in error or rendering rate. Although this depth is appropriate for these scenes, it may not necessarily be the correct value to use in different scenes. This is a drawback with fixed-depth subdivision as different fixed depths may need to be tried to determine the optimal one for the scene.

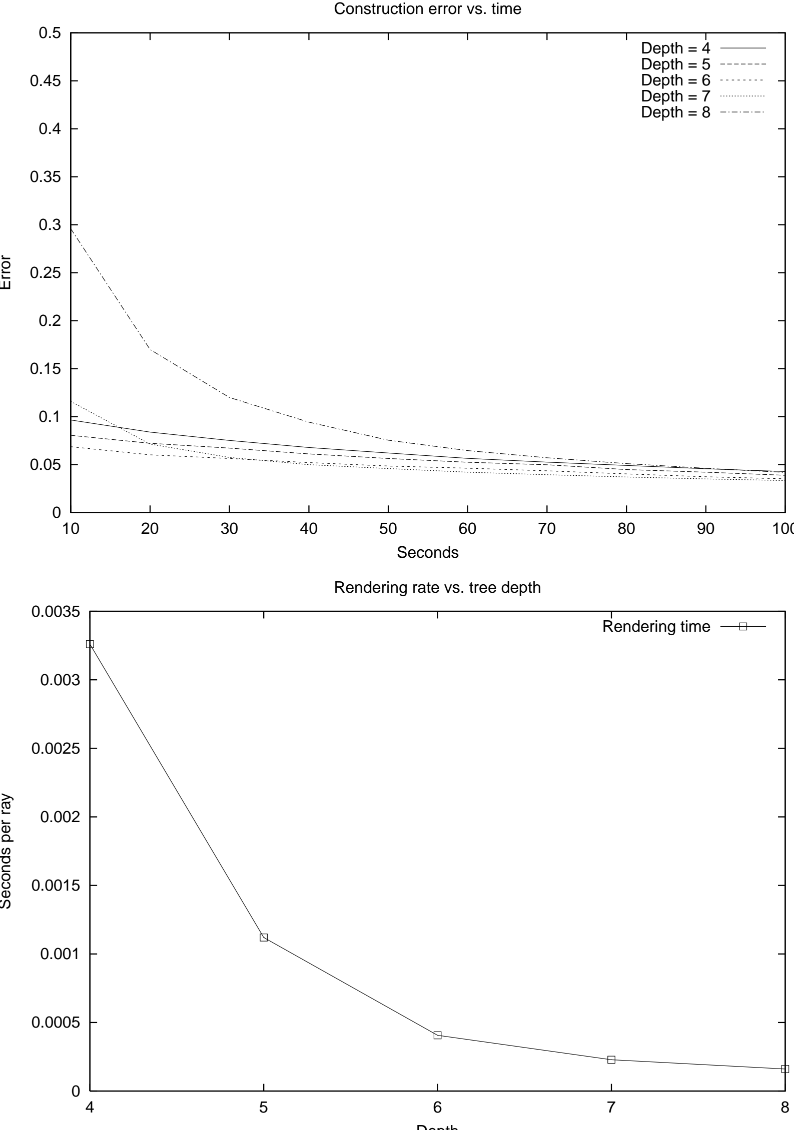



Figure 4.8: Fixed tree depth (Mosque de Cordoba)

## 4.4 Adaptive subdivision

Instead of using a fixed cell size, we can subdivide cells as their LIEs become more complex. This should provide us with small, easy-to-render cells where the lighting complexity is high. Regions with low lighting complexity should get large, easy-to-construct cells.

Lights and blockers are added to a cell as in fixed subdivision. When one of

the lights in the LIE gets more blockers than a given maximum blocker threshold, the cell is subdivided. Upon subdivision, the LIE associated with the parent cell is discarded and the LIEs associated with the child cells are initialized to be empty. Subdivision is limited to a certain maximum depth, in these tests this depth was eight.

At first glance, it would appear desirable to propagate the LIE of the parent cells to the child cells. However, doing so has several problems. If a light is partially visible in the parent cell, it may not be partially visible in the child cell. Assuming that it is partially visible would add the light to the child cell and perhaps unnecessarily add to its rendering time. If a light is fully occluded in the parent cell, we cannot assume that it is fully occluded in the child cell and never sample it, because the determination of full occlusion may just be due to insufficient sampling. Likewise, if a blocker is found in the parent cell's LIE, it may not belong in the child cell's LIE, and if it is not found in the parent cell's LIE, it may just be due to insufficient sampling. Because of these problems, we do not propagate LIE information from parent to child and instead start the child off with an empty LIE. However, this does mean that error increases initially after a cell subdivision.

We show here construction times and rendering rates for the three scenes, using adaptive subdivision of the cells. We measured construction times and rendering rates for various values for maximum length of the blocker list. We also compared the error of adaptive subdivision with that of fixed subdivision at depth six.

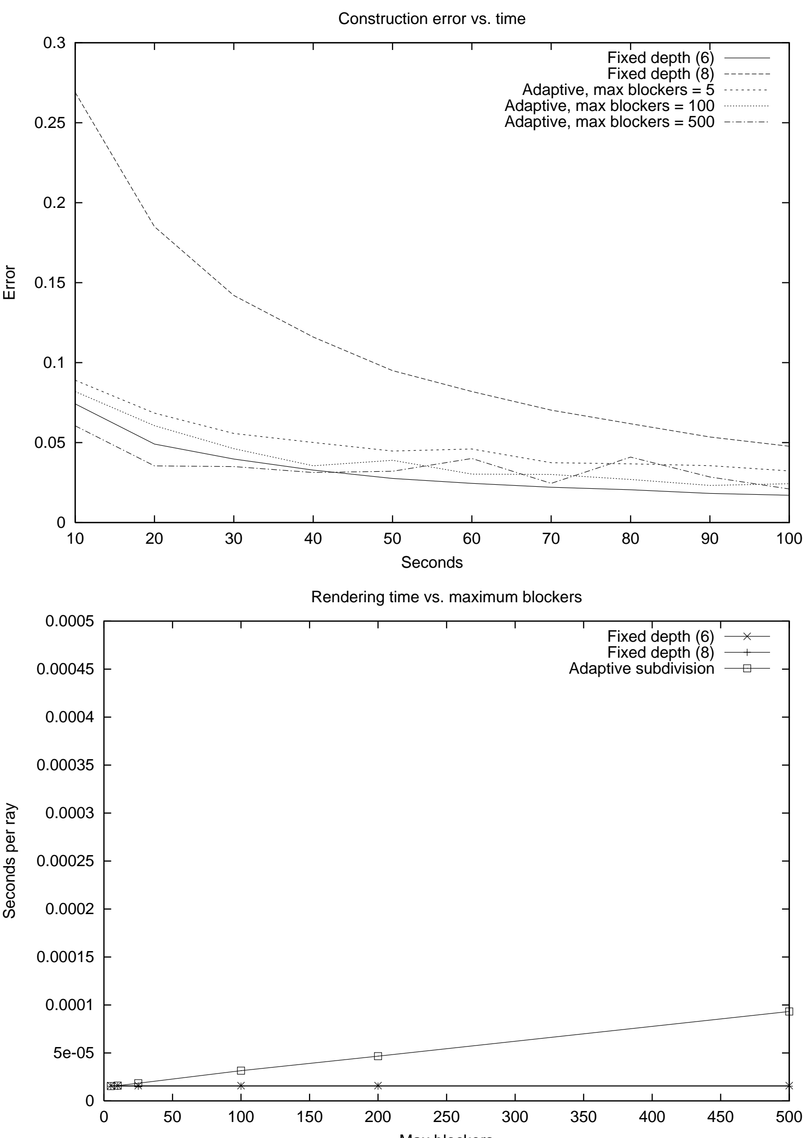



Figure 4.9: Adaptive cell subdivision (Bar)

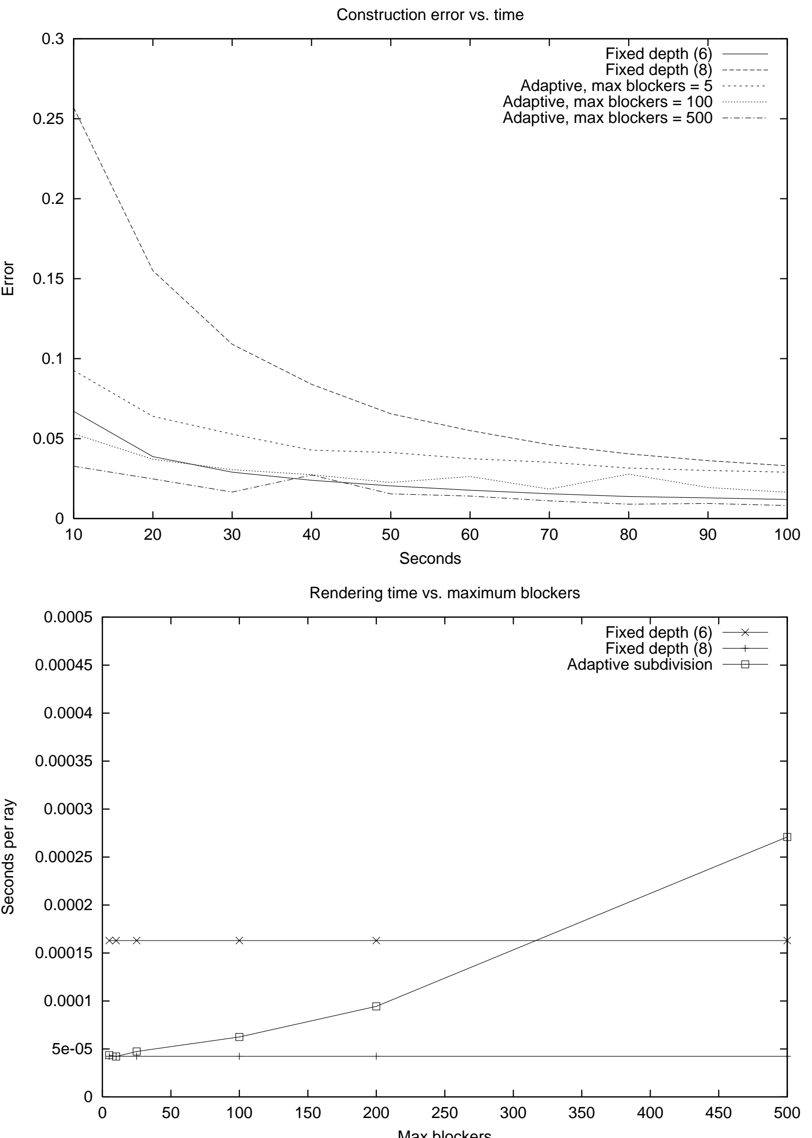



Figure 4.10: Adaptive cell subdivision (Grand Central Station)

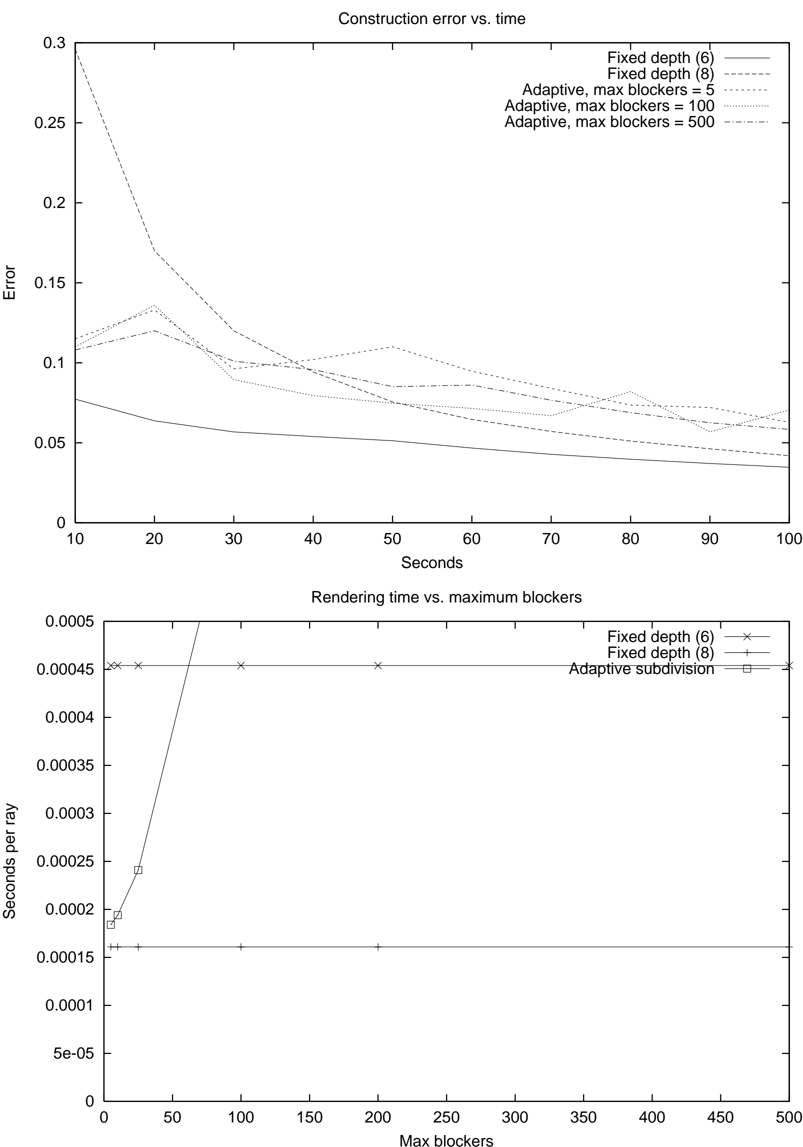



Figure 4.11: Adaptive cell subdivision (Mosque de Cordoba)

In general, given the same amount of time spent sampling, using a smaller blocker list threshold will produce more error than using a larger blocker list threshold. Small thresholds cause more frequent subdivisions, leading to smaller cell sizes and, as we saw in the previous section, smaller cells have greater error. However, smaller blocker list thresholds render at a faster rate than larger ones since the rendering rate is linear with the maximum size of the blocker list.

It is interesting to note that the error with adaptive cell subdivision does not decrease monotonically and is far noisier than the error associated with fixed subdivision. This is due to discarding the LIE upon subdivision. When a cell is subdivided, the children start with empty LIEs. This is equivalent to assuming all the lights are completely occluded. This assumption can have considerably higher error than the LIE associated with the parent.

In order to reduce error as quickly as possible in these scenes, it is almost always best to use a fixed depth of six. A fixed depth of eight produces the greatest amount of error. Adaptive subdivision with a maximum depth of eight tends to fall in between these two.

A fixed depth of eight or a small blocker list threshold will always give you the best rendering performance. In a smaller scene like the Bar, the performance difference between a fixed depth of six and a fixed depth of eight or a small blocker list threshold is negligible. In the other two larger scenes, however, the performance difference may be well worth tolerating a longer convergence time.

Ultimately, the subdivision approach to use depends on the scene being rendered. In the Bar scene, the best choice is a fixed depth of six. It produces less error than either a fixed depth of eight or adaptive subdivision. In addition, the difference in rendering performance is negligible.

In the Grand Central Station scene, the best choice is adaptive subdivision with a small blocker list threshold. It produces significantly less error than a fixed depth of eight while providing the same rendering performance, substantially better than a fixed depth of six.

In the Mosque de Cordoba scene, the best choice is a fixed depth of eight. It ultimately has less error than adaptive subdivision and has substantially better rendering performance than a fixed depth of six subdivision.

In general, the more complex the scene, the greater the subdivision depth required for optimal rendering performance. When rendering very complex scenes, adaptive subdivision can sometimes help in reducing the amount of error more quickly than fixed depth approaches. However, the fact that a lot of sample data is thrown away as parent cells are subdivided into child cells can hurt the convergence rate of adaptive subdivision.

## 4.5 Quasi-Monte Carlo sampling

Straightforward random sampling of shadow ray space can lead to clusters of samples, where some areas have too many samples and some too few. Quasi-Monte Carlo (QMC) sampling can provide low-discrepancy, meaning that samples are guaranteed to be well distributed over the domain. This can help us get our requisite one sample per shadow ray region more quickly.

We compared a linear congruential random number generator, with a QMC random number generator based on a Halton sequence with bases 2 and 3. These were used to generate samples in view shadow ray space. The results are shown in Figure 4.12.

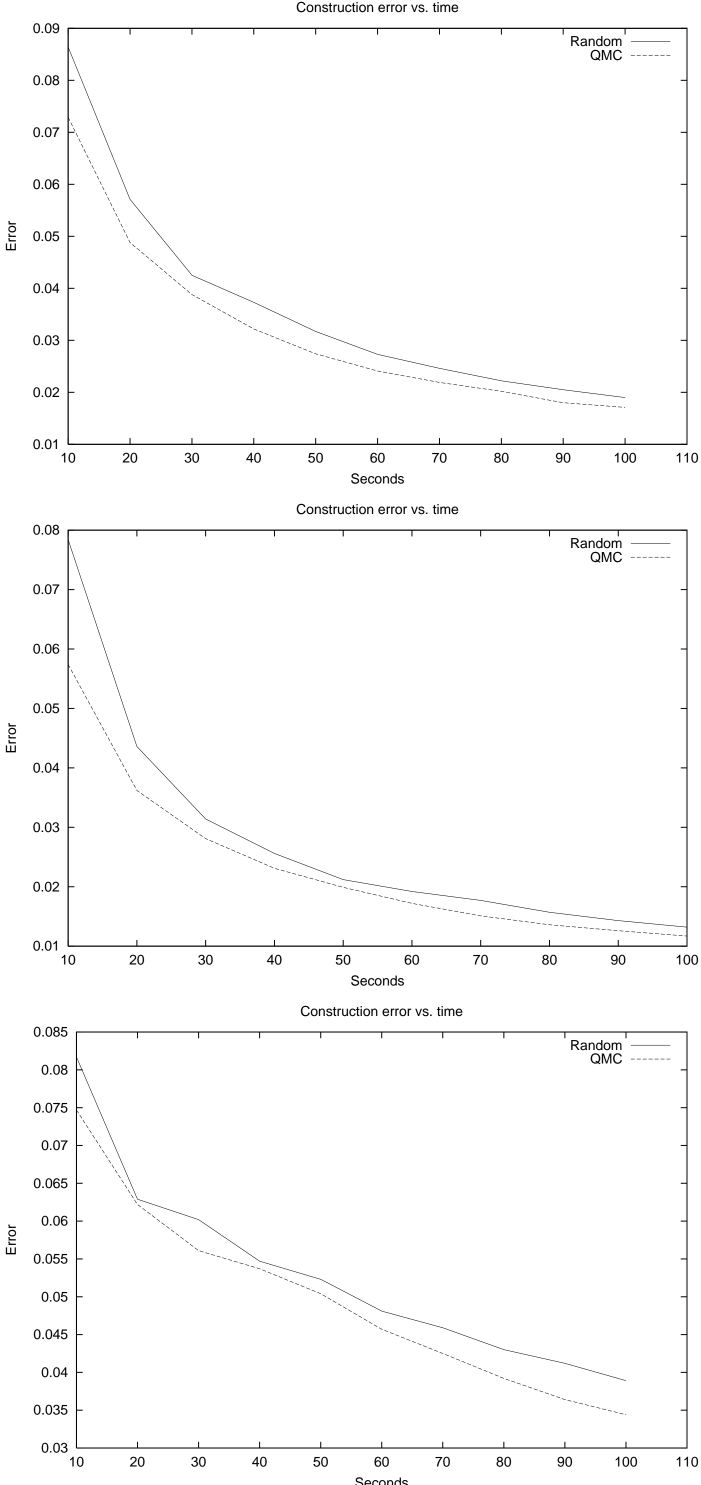



Figure 4.12: QMC vs. Random sampling on the three scenes. Top: Bar. Middle: Grand Central Station. Bottom: Mosque de Cordoba.

QMC sampling provides a small but definite reduction in error. Although the Halton sequence can be a bit more expensive to generate than conventional number random generation methods, the results seem to indicate it is worth using. Additional gains could be obtained by using less expensive QMC sequences.

## 4.6 Regular sampling

Instead of randomly sampling view shadow ray space, we can regularly sample it. We do this by subdividing view shadow ray space into fixed size sections and taking one sample at a fixed location in each section. One advantage of this approach is that once all samples have been taken for a particular viewpoint, one can stop sampling. On the other hand, it is not clear how to proceed once all samples have been taken if the error has not been driven down to zero.

In these tests, we have regularly sampled the image plane and light sources for the three scenes. The sampling was done at the rendering resolution, at one-quarter the rendering resolution, and at one-sixteenth the rendering resolution. Samples and rendering were done at the center of pixels. Although sampling at the rendering resolution is as expensive as conventional ray tracing when generating a single image, the resulting LIEs may be reused from frame to frame if the camera moves. We show the results in Figure 4.13.

For all resolutions, we get a roughly linear reduction in error as all the pixels are sampled. This will vary from a linear reduction if portions of the image are well represented by empty LIEs, which are rendered as black pixels. For example, for a viewpoint where the top half of the image is completely black and the bottom half is well illuminated, the error will start off at about one-half, stay constant for

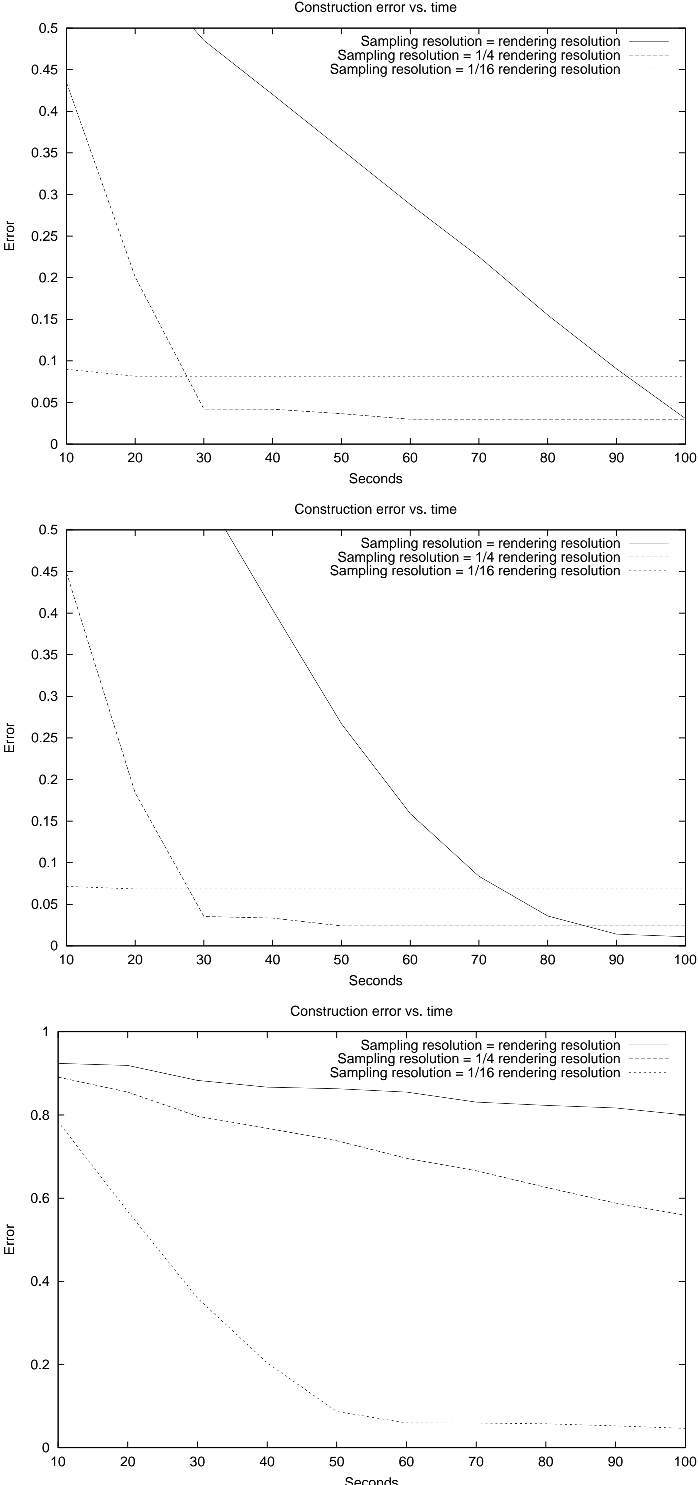



Figure 4.13: Regular sampling on the three scenes. Top: Bar. Middle: Grand Central Station. Bottom: Mosque de Cordoba.

some time, and then start decreasing when the regular sampling gets to the lower half of the image.

When sampling at one-quarter and one-sixteenth the rendering resolution, error levels off at a non-zero value once all samples have been taken. This error threshold is higher for lower sampling resolutions.

## 4.7 Three-state LIEs

While small lists of blockers can outperform conventional acceleration structures (due to little overhead), larger lists can significantly underperform them.[2] A blocker list should be limited in size and exceeding that size should trigger the use of a conventional acceleration structure.

One extreme form of this method is three-state LIEs. A three-state LIE behaves like a conventional LIE except that a conventional acceleration structure is used if there would be any blockers in the list for a given light. Lights in a three-state LIE do not need a blocker list. Instead they are in one of three states: fully occluded, partially occluded without a blocker list, or fully visible. Fully occluded lights are ignored. Fully visible lights are shaded without a visibility test. Partially occluded lights are shaded with a visibility test from a conventional acceleration structure, as in a regular ray tracer.

Three-state LIEs should take less time to construct than LIEs with blocker lists because there is no need to find each blocker in the list. A three-state LIE has at most two shadow space regions per light within each cell, one or zero unoccluded regions and one or zero occluded regions. In addition, three-state LIEs should use

[2] Conventional acceleration structures are are hierarchical and hence exhibit sub-linear performance.

less memory, since lists of blockers do not need to be stored. However, three-state LIEs should also not render as quickly, since a conventional acceleration structure is used instead of a short blocker list. In some cases, where the blocker list would have been long, three-state LIEs will render more quickly than blocker list LIEs. Such a case is shown in Figure 4.15.

We show in Figures 4.14, 4.15 and 4.16 the construction times and rendering rates for three-state LIEs and compare the approach to the use of blocker lists and fixed-depth subdivision at a depth of six. [3]

Construction error is significantly reduced by using three-state LIEs. The rendering times do increase in the case of the bar scene. In the case of the grand central scene, rendering times are actually lower for three-state LIEs than for blocker list LIEs, particularly for large cell sizes. This is due to blocker lists being slower than conventional acceleration structures when the list is long.

As the maximum depth of the tree increases and cell sizes become smaller, both the construction time and the rendering rate of three-state LIEs approach that of blocker list LIEs. With smaller cells, the size of the shadow ray regions, and hence construction time, is more influenced by the size of the cell than by the size of regions inside the cell. As cells become smaller, more of them become either fully occluded or fully unoccluded to the light sources. In these two cases, the rendering performance of three-state LIEs is equivalent to blocker-list LIEs. Three-state LIEs only have worse rendering performance than blocker-list LIEs when lights are partially occluded and the list of geometric blockers is more expensive to traverse than a conventional acceleration structure.

[3] We chose a depth of six because it provides a reasonable balance between convergence rate and rendering speed.

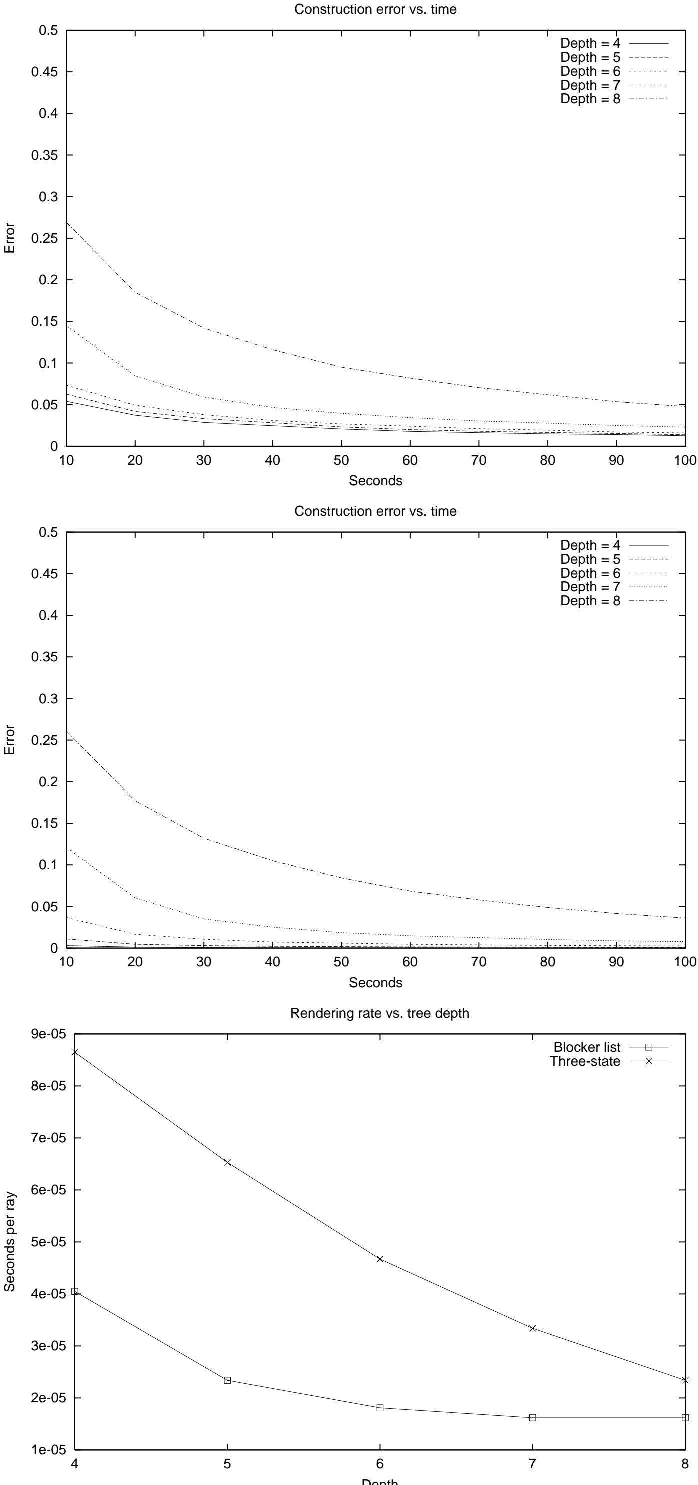



Figure 4.14: Blocker list vs. three-state LIEs on the Bar scene. Top: Construction time for blocker list LIEs. Middle: Construction time for three-state LIEs. Bottom: Rendering time for blocker list LIEs and three-state LIEs.

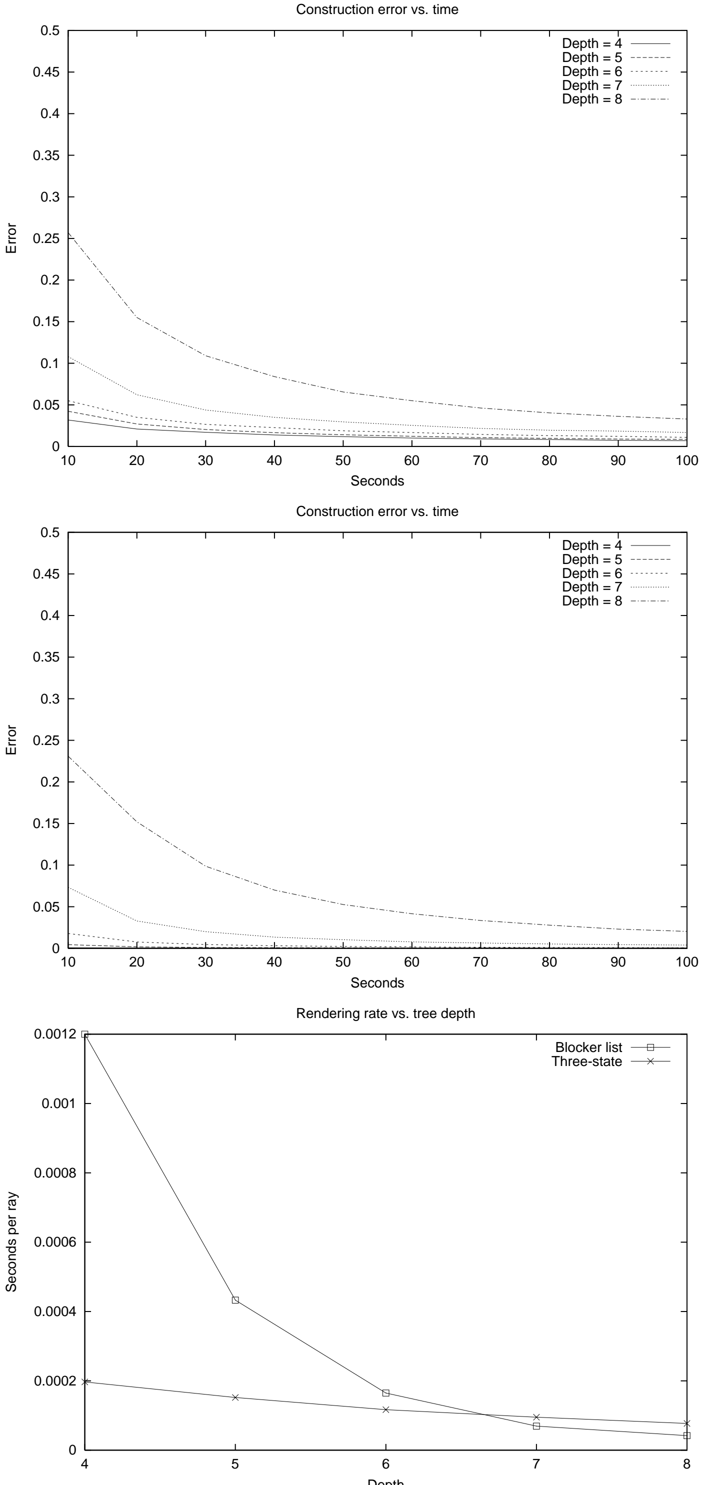



Figure 4.15: Blocker list vs. three-state LIEs on the Grand Central Station scene. Top: Construction time for blocker list LIEs. Middle: Construction time for three-state LIEs. Bottom: Rendering time for blocker list LIEs and three-state LIEs.

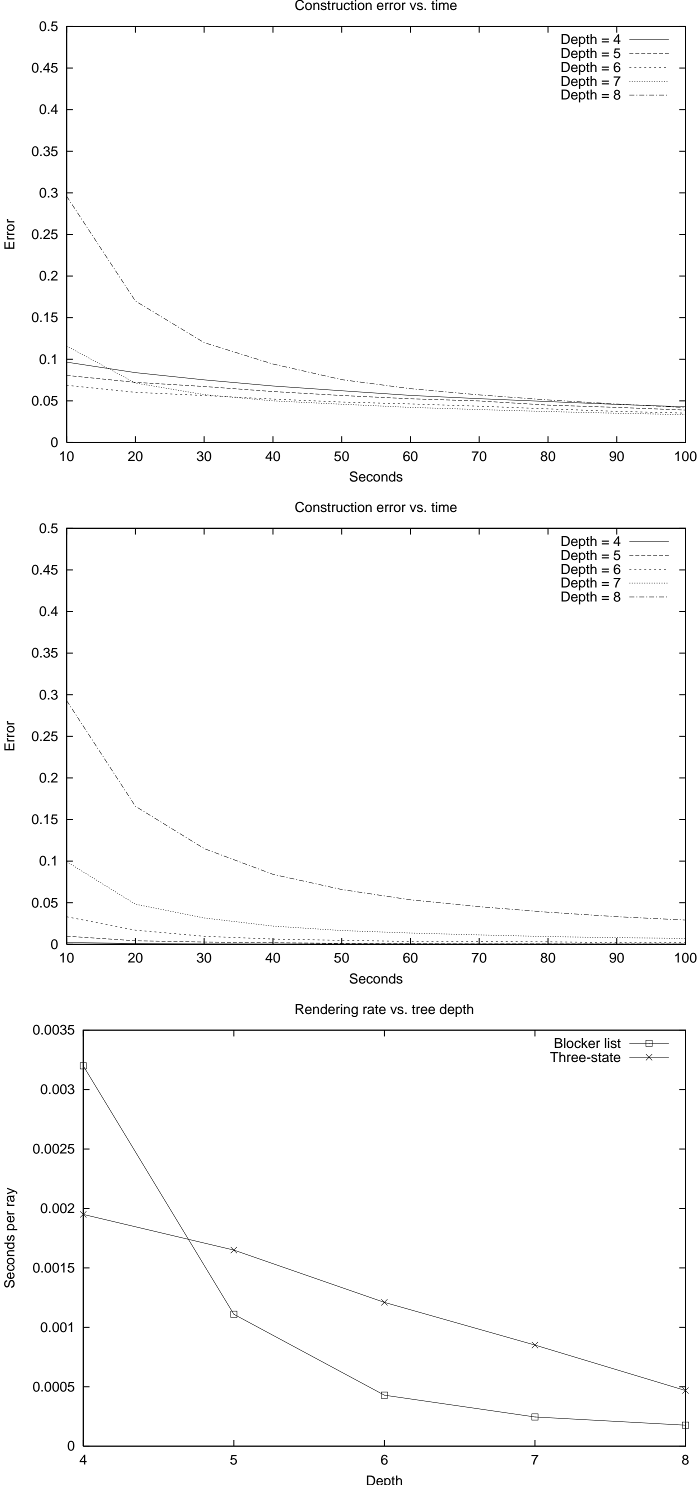



Figure 4.16: Blocker list vs. three-state LIEs on the Mosque de Cordoba scene. Top: Construction time for blocker list LIEs. Middle: Construction time for three-state LIEs. Bottom: Rendering time for blocker list LIEs and three-state LIEs.

## 4.8 Error analysis

In our experiments, the error curve has had a consistent shape, initially decreasing quickly, but later leveling off. In this section, we will explore the reason for this shape by constructing a simplified model of the LIE construction process and examining its behavior.

To simplify the analysis, we will assume that blockers are added to the blocker list as they are found, instead of waiting until visibility to a light has been proven.

Let the size of a region in view shadow ray space be defined by $s(I_x, L_y) = s(I_x) \times s(L_y)$, where $s(I) = 1$ and $s(L) = 1$. Let $U(t)$ be the sum of the size of all unsampled regions in view shadow ray space after $t$ samples have been taken. $U(t)$ is a reasonable error measure since it is equal to one when no regions have been sampled and the generated image is completely erroneous, zero when all regions have been sampled and the image is completely correct, and each decrease in $U$ usually leads to the removal of some visible artifact in the image, the larger the difference, the greater the correction.

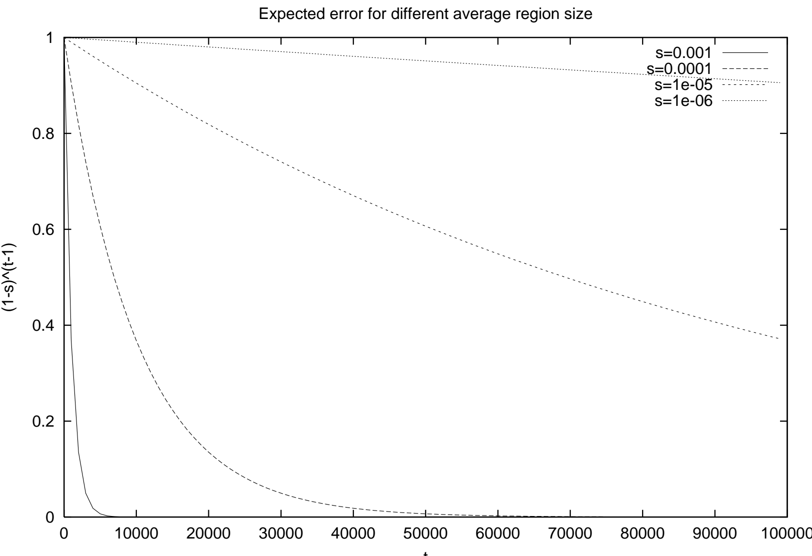



Figure 4.17: $(1 - \overline{s})^t$, for some values of $\overline{s}$

which is illustrated in Figure 4.17. This matches the shape of the error curves we're seeing.

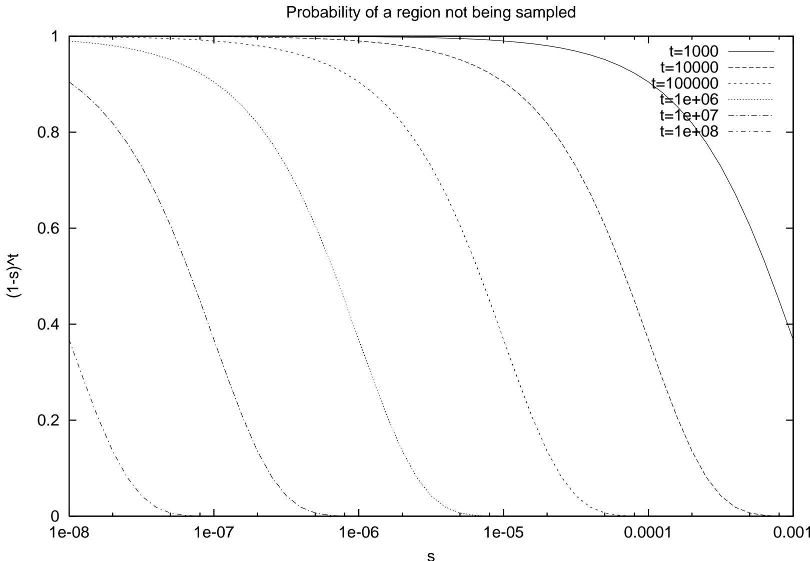



Figure 4.18: $(1-s)^t$, for some values of $t$

The expected value of $U(t)$ is simply a sum over all the regions of the region's size (denoted by $s(R)$) times the probabiplity that the region has not been sampled. The probability that a region $R$ has not been previously sampled after taking $t$ samples is $(1-s(R))^t$. Thus,

$$E(U(t)) = \sum_R s(R)(1-s(R))^t \tag{4.4}$$

If instead of summing over the regions, we sum over sizes,

$$E(U(t)) = \sum_s n(s)s(1-s)^t \tag{4.5}$$

where $n(s)$ is the number of regions with a given size. If we assume a single region size $\overline{s}$, this turns into

$$\begin{aligned} E(U(t)) &= \tfrac{\overline{s}(1-\overline{s})^t}{\overline{s}} \\ E(U(t)) &= (1-\overline{s})^t \end{aligned} \tag{4.6}$$

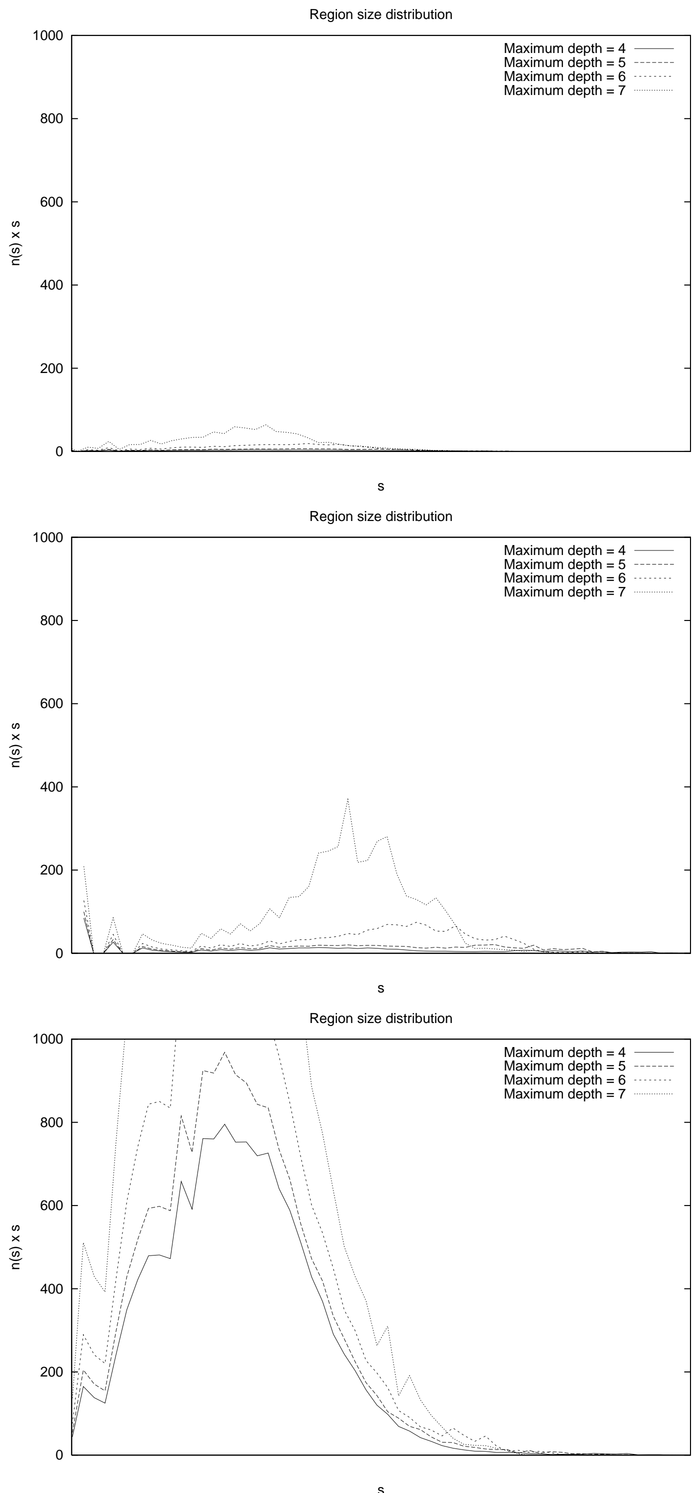



Figure 4.19: Number of features times feature size ($n(s)s$) for the three scenes. Top: Bar. Middle: Grand Central Station. Bottom: Mosque de Cordoba.

The regions are not, however, uniform in size. The distribution of $n(s)s$ is shown in Figure 4.19 for the three scenes. These plots were obtained by a very fine regular sampling of view shadow ray space. The plots of $n(s)s$ show us that complex environments with many lights tend to have a large number of small regions. The plots also show that, as depth increases and cell size decreases, we get more smaller regions.

Since $E(U(t)) = \sum_s n(s)s(1-s)^t$, if we overlay a plot of $(1-s)^t$ (Figure 4.18) with plots of $n(s)s$ for each model (Figure 4.19), we can get an idea of the expected error for a given number of samples and a given model. The $(1-s)^t$ term tells us that a very large number of samples are required to find small regions ($s$ is small). This term can be viewed as a filter on region sizes. For any given number of samples $t$, all region sizes to the left of the inflection point (on the plots of $n(s)s$) are unlikely to be sampled, where those to the right are likely to be sampled. As can be seen from Figure 4.19, on a model like Mosque de Cordoba, even ten million samples can leave us with some error.

## 4.9 Future Work

There are several other approaches that could be taken to construct LIEs. So far, we have described view driven approaches. However, there are several offline techniques that could be used to generate LIEs, to be later used by online shaders.

One such offline technique would be to generate LIEs based on sampling from the point of view of the light sources. For each light source, a set of rays would be generated in a given set of directions. The first surface that a ray hits would mark this light as visible in its LIE. Unlike conventional visibility rays, this ray would

then continue through the first surface, hitting additional surfaces. Any surfaces hit after the first surface would mark this light as not visible in their LIEs. If a light is marked as not visible with respect to an LIE, the blocker causing that mark is recorded in the LIE. A light would be considered fully visible with respect to a particular cell iff all marks were visible. If the light does not have any visible marks, then the light is considered fully occluded. Otherwise, the light is partially visible and the blocker list is used for shading.

This approach differs from the view-driven LIE generation approach in the following ways:

- Rays are generated from the lights, not the eye.
- It can be used offline or online.
- Some significant amount of work may be done in areas never seen by the viewer.
- The sampling resolution required is unclear if the LIE generation is performed offline.

Another offline approach that could be used is surface-driven LIE generation. Instead of iterating over points on the image plane or light sources, iteration occurs over surfaces. This works exactly as view-driven LIE generation, except that instead of choosing surface points based on rays from the eye through the image plane, a large set of surface points covering all the surfaces in the environment is chosen a priori.

The advantage of this approach is that it can be performed offline, since no camera position is needed.

One disadvantage of this approach is that, since no camera position is given, it is unclear how dense the samples have to be in order to not have artifacts. The lack of camera position also means that a significant amount of work may be done in areas not visible to the viewer.

The approaches discussed so far have been conservative about adding lights and blockers. A light is never added to an LIE unless it can be proven that a surface point within the LIE can see that light. A blocker is never added to an LIE unless it can be proven that it does block a partially visible light from being seen by some surface within the LIE region. Although this approach allows us to construct small LIEs that render quickly, it prevents us from reducing artifacts by inserting lights or blockers that we suspect, but have not proven, affect the LIE. There are a few "non-conservative" techniques that could be tried.

One such technique is flood filling. If a light has been proven visible in a cell, it is likely to be visible in adjacent cells. We could add this light to the LIEs in adjacent cells. However, this would lead to errors in cells where the light was not in fact visible. Therefore, the lights would have to be removed after some time if they were not proven visible.

A similar approach could be taken with blockers. If a blocker is found in the LIE of a cell, it could be added to adjacent cells. The difference in this case, however, is that a blocker added to an LIE incorrectly cannot cause rendering errors, just reduce performance. This is due to the fact that blockers in the blocker list are evaluated for occlusion at the point being rendered.

Similar to flood filling, inheritance propagates lights and blockers to cells where they have not been proven to affect the LIE. However, this works at the subdivision step. When an LIE cell is deemed to be too complex, it is subdivided. The child

cells could inherit the lights and blockers of the parent.

Both flood filling and inheritance have significant problems. While they would probably reduce errors caused by incorrect LIEs, the performance cost of rendering large LIEs could be high.

## 4.10 Conclusion

In this chapter, we've more closely analyzed the process of LIE construction. We've explored several LIE parameters and variants on the construction process.

We considered the use of fixed-depth cell subdivision and showed how the error and rendering rate changes with varying depth. We also found a reasonable maximum depth of six works well for the variety of scenes we tested.

We looked at the use of adaptive subdivision of the octree cells. We showed problems associated with subdivision, primarily that of throwing away sample data. We also showed the varying construction error and rendering rates associated with different subdivision threshold criteria. We determined that adaptive subdivision is most likely to be helpful in very complex environments where a high level of cell subdivision is required.

We also compared the use of Quasi-Monte Carlo sampling with random sampling and determined that, although the individual samples obtained are more expensive with QMC sampling than with random sampling, QMC sampling leads to a noticeable reduction in error.

We looked at the use of regular sampling as opposed to random sampling for LIE construction. Although regular sampling gave us a fixed number of samples for a particular view point, we found significant error if the sampling resolution

did not approach the rendering resolution.

We explored the use of three-state LIEs instead of blocker lists. We found that although construction error is considerably reduced with three-state LIEs, rendering time does increase significantly. We also noted that as cell size decreases, the differences between three-state and blocker list-based LIEs diminish.

Finally, we modeled the LIE construction process to explore the error curve associated with random sampling for LIE construction. We found that models with a large number of very small shadow space regions can take a very long time to reduce error below a certain level. We also measured the size distribution of shadow space regions in the different models and found a large number of small shadow space regions, especially in the more complex models. We also found that increased subdivision of cells increased the number of small regions, explaining why larger subdivision depths have larger error.

We found that LIEs can be a very efficient rendering method for medium-sized scenes of a few hundreds of thousands of rendering primitives and a few dozen lights, especially considering that it is an object-space caching method that can reuse information gathered from different viewpoints.

However, if zero error is required, or the scenes are very large, the error associated with LIE construction can be unacceptable. In the next chapter, we will explore methods that, although they do not render as quickly as LIEs for medium-sized scenes, can handle very large scenes well.

# Chapter 5

# Hierarchical Light Clusters

In this chapter, we introduce efficient approaches for evaluating the effect of large numbers of light sources. The techniques center around the concept of "Hierarchical Light Clustering"[1].

We first explain the concepts behind hierarchical light clustering. We define light clusters and how to simulate them with a single representative light. We then show how to build a hierarchy of such clusters. We explain the concept of a "tree cut" as a locally valid representation for illumination. We then discuss three algorithms built on these concepts.

This first algorithm uses dense sampling and generates high quality images, but offers only modest speed improvements over conventional ray tracers on scenes with a few hundred light sources. For this reason, we introduce a modification to the algorithm which samples the "tree cuts". We then develop two algorithms to reconstruct shading from sparse samples. The first is a simple interpolation technique which is quite fast, but blurs shadow boundaries. The second technique

[1] This is work done in collaboration with Dr. Bruce Walter, Mike Donikian, Prof. Kavita Bala, and Prof. Donald P. Greenberg.

is not as fast, but accurately captures illumination discontinuities.

We then define issues involved in building the system to support these algorithms, particularly the sampling and reconstruction approach, and demonstrate the parallel computer system used. We describe how to provide sample feedback so that samples can be placed at the locations where they can be the most beneficial. We also discuss how to compress samples, since the thousands of samples needed would otherwise occupy too much memory. We describe how to find nearby samples when using the sampling and reconstruction approach.

Finally, we show results for these algorithms, comparing their performance with each other and to conventional direct lighting techniques.

## 5.1 Concepts

In this section we introduce the basic concepts and tools used in our adaptive clustering approach.

The light $L_{\mathbb{S}}$ reflected by a surface from direct illumination by a set of non-directional light sources $\mathbb{S}$ is the sum of the contributions from each light. The contribution of an individual point light $i$ is the product of four factors: the surface's BRDF $f_{\mathrm{r},i}$ (Bidirectional Reflectance Distribution Function), the geometric factor $G_i$, the visibility $V_i$, and the intensity $I_i$ of the light.

$$L_{\mathbb{S}}(x,e) = \sum_{i \in \mathbb{S}} f_{\mathrm{r},i}(x,e)\; G_i(x)\; V_i(x)\; I_i \tag{5.1}$$

where $x$ is the surface point and $e$ is the position of the observer.

We will be initially concerned with computing this equation at a particular point on a surface and from a fixed viewpoint. The factors then depend only on

the characteristics of the lights and we abbreviate Equation 5.1 as

$$L_{\mathbb{S}} = \sum_{i \in \mathbb{S}} f_{\mathrm{r},i}\, G_i\, V_i\, I_i \tag{5.2}$$

The cost of evaluating this equation is linear in the number of lights. Although we express the equation only in terms of light indices, it is important to note that the BRDF values, geometric factors, and visibilities also depend on the relative position and orientation of the illuminated surface and viewer position and thus must be recomputed when these change. Visibility of a point light is either zero or one and is evaluated using shadow rays in our system. Intensities are assumed constant per light.

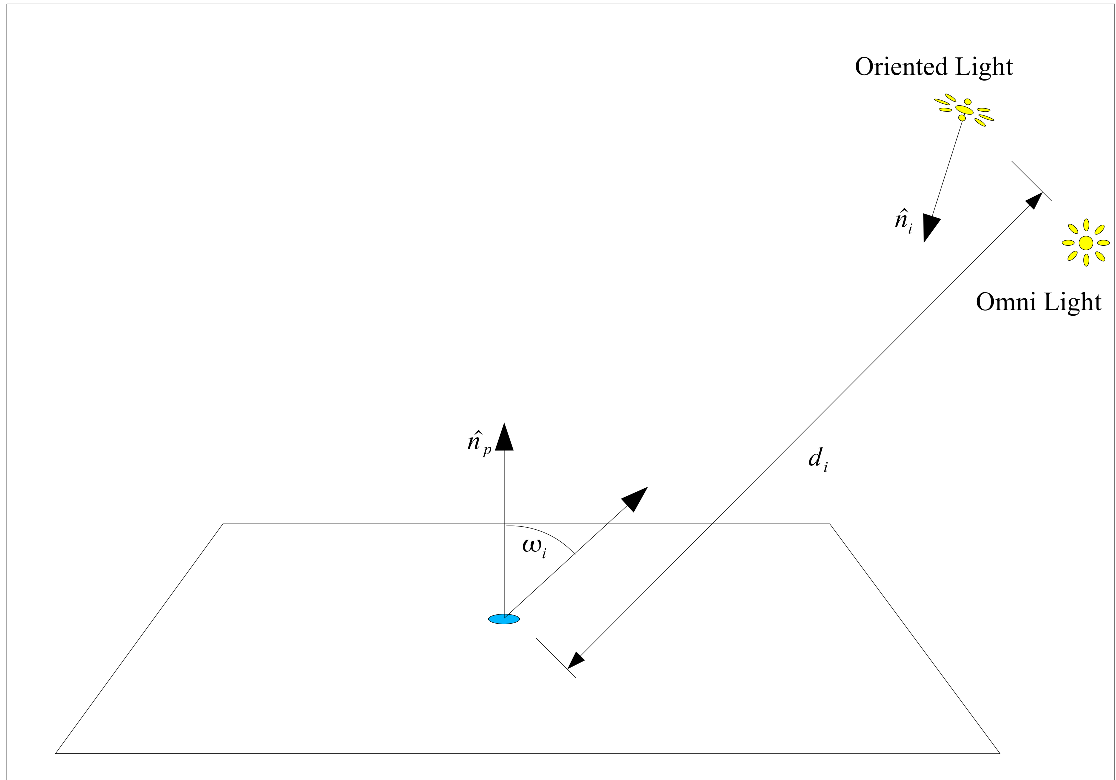



Figure 5.1: Terms used in geometry factors.

We use two types of lights, omnidirectional and oriented. Omnidirectional lights are point sources that are equally bright in all directions. Oriented lights are

also point sources but have an orientation and a cosine falloff from that orientation. Oriented lights are used to simulate area lights. We use $G_i$, the geometric factor [2], to capture the orientation of the surface as well as the attenuation with distance. For oriented lights, the $G_i$ term also captures the cosine falloff. The geometry factors for the two types of lights are shown below and illustrated in Figure 5.1. We define $\hat{n}_p$ as the normal at the point being illuminated, $\omega_i$ is the direction towards the light, $d_i$ is the distance to the light, and for oriented lights, $\hat{n}_i$ is normal at the light.

$$G_i = \begin{cases} \max(0,\, \hat{n}_p \cdot \omega_i)/{d_i}^2 & \text{for Omni lights} \\ \max(0,\, \hat{n}_p \cdot \omega_i) \max(0,\, -\hat{n}_i \cdot \omega_i)/{d_i}^2 & \text{for Oriented lights} \end{cases} \tag{5.3}$$

### 5.1.1 Light Clusters

In order to make the cost complexity sub-linear in the number of lights, we need a way to efficiently approximate the contribution of groups of lights without evaluating all the lights individually. Let us define a *cluster* $\mathbb{C}$ to be a subset of the lights. We can quickly approximate the contribution of a cluster $L_{\mathbb{C}}$ by choosing a representative light for that cluster and using the BRDF, geometric factor and visibility values evaluated at the representative light for the entire cluster as follows:

$$L_{\mathbb{C}} = \sum_{i \in \mathbb{C}} [f_{\mathrm{r},i}\; G_i\; V_i\; I_i] \;\approx\; f_{\mathrm{r},j}\; G_j\; V_j \sum_{i \in \mathbb{C}} I_i \tag{5.4}$$

where $j \in \mathbb{C}$ is the cluster's representative light and $I_i$ is a constant per light, and thus its sum can be precomputed and stored with the cluster as the cluster intensity $I_{\mathbb{C}}$. We will refer to Equation 5.4 as the *cluster approximation*. A simple example of such clustering is shown in Figure 5.2.

[2] We use the term geometric factor instead of form factor because it is more appropriate for point light sources.

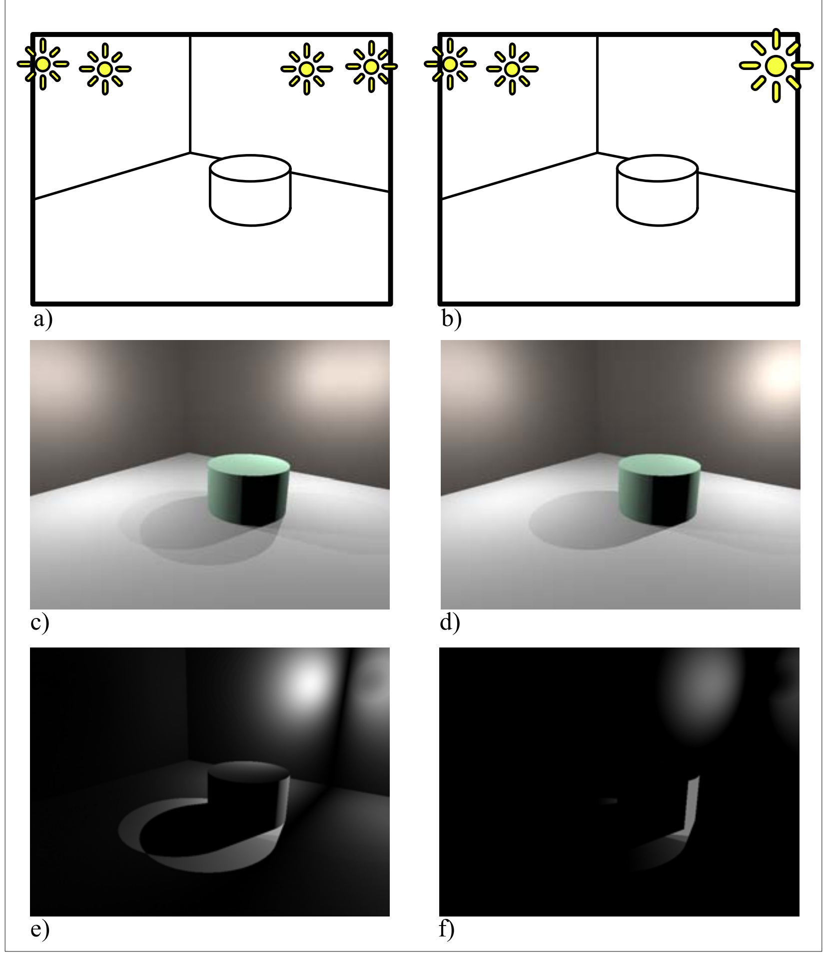



Figure 5.2: Light clustering in a simple scene with four point lights. In a), we illustrate a simple scene with four point lights. In b), one pair of point lights has been clustered into a single light. c) and d) show renderings of these two scenes respectively. In e) we have computed the absolute value of the difference between the images in the middle row ($abs(c - d)$) at each pixel and scaled the result by a factor of two to highlight the differences. We see that large parts of the difference image are dark, which implies very little difference between the images. In f), we use the same difference image, this time dividing by the image brightness at each pixel ($abs(c - d)/d$) to take into account the reduced sensitivity to error in bright regions.

Clustering allows us to estimate the contribution of an entire set of lights for the cost of just a single light evaluation. It also introduces some error which may or may not be acceptable depending on the local circumstances. Clustering is most effective when the BRDF, geometry, and visibility factors (as evaluated from the surface point and viewer orientation we are trying to render) vary little within the cluster or when the total contribution of the cluster is small compared to the rest of the lights. We would like to partition the lights into a small a set of large clusters because our rendering cost is proportional to the number of clusters. At the same time, we would like to create a large set of small clusters because small clusters will have less variation in illumination and therefore introduce less error. The challenge lies in finding a balance between these two goals.

### 5.1.2 Cluster Hierarchy Tree

To work effectively, our clustering scheme needs to be locally adaptive. No single partitioning of the lights into clusters is likely to work well over the entire image, but dynamically finding a new cluster partitioning for each point could easily prove prohibitively expensive. To solve this problem we use a two-step approach. First we build a global cluster hierarchy to generate a set of possible clusters to use. Second, we compute locally adaptive subsets (called *cuts*) of these possible clusters at each point to be rendered. We describe the global cluster hierarchy in this section and discuss cuts in Section 5.1.3.

A cluster hierarchy is a tree where the leaves are the individual lights and the interior nodes are light clusters that contain exactly the lights below them in the tree. We build the tree in a bottom-up fashion, starting at the leaves and building to the root. First we create a leaf node for each light in the scene. We then find

the two nodes with no parents closest to each other and create a new node with those two nodes as its children. This new node represents a cluster that is the union of the two clusters represented by its child nodes. This process of finding the two closest nodes and creating a new node from them is repeated until only one node is left with no parent, the root node.

In order to find the two closest nodes, we have to define a distance metric for nodes. The distance between a pair of leaf nodes is simply the cartesian distance between the two lights they represent. More generally, the distance between two nodes is the maximum distance between two points on the bounding spheres of the clusters they represent.

Each tree node (i.e. cluster) stores the total intensity of all the lights it represents, and points to its representative light and its children in the tree. To make our algorithms more efficient, we require that the representative light for a cluster have the same position and orientation as the representative light for one of its children. In particular, it shares the position and orientation of the representative light which is closer to the power-weighted centroid of the parent cluster. The representative light for an individual light is itself, naturally.

The above descriptions refer to point lights. However, the algorithm can easily be extended to area lights by approximating each area light as a set of oriented point lights. The number of point lights required to accurately represent an area light varies considerably depending on the local configuration. Surface locations near the area light or in its penumbra may require many point lights for an adequate result while further locations may require few. We can automatically handle this by integrating the area lights into our cluster tree, as follows.

For each area light, we generate a cluster subtree that progressively divides

the light into smaller subregions up to some maximum subdivision. We choose a point within each subregion to act as its representative point light and compute an intensity proportional to the area of the subregion. Cluster building then proceeds as described above.

For static environments, the cluster hierarchy is only computed once per scene and then can be reused at each pixel in finding an appropriate light clustering. When we decide which clusters to use to render a point (as described in the next section), only the clusters from this hierarchy are used.

### 5.1.3 Finding a Cut

A *cut* is a subset of the nodes in the cluster hierarchy tree. Cuts provide us with local adaptation, allowing us to choose which nodes from the previously built cluster hiearchy tree are appropriate for the point being rendered. More formally, a *cut* through the tree is a set of nodes such that every path from the root of the tree to a leaf will contain exactly one node from the cut. Each tree cut thus corresponds to a valid partitioning of the lights into clusters. An example cluster hierarchy and three different cuts are shown in Figure 5.3.

The algorithm for cut generation is illustrated in Figure 5.4. We have three types of nodes in the tree. Cut candidate nodes (shaded red) are nodes that have not yet been inspected but will be inspected before the final cut is determined. Cut nodes (shaded blue) are nodes that have been determined to be part of the final cut. Other nodes are unshaded.

We begin by marking the root node red, thus making it a cut candidate. The algorithm proceeds by picking some red node and determining whether it should be part of the final cut. We determine this by estimating (as described in the

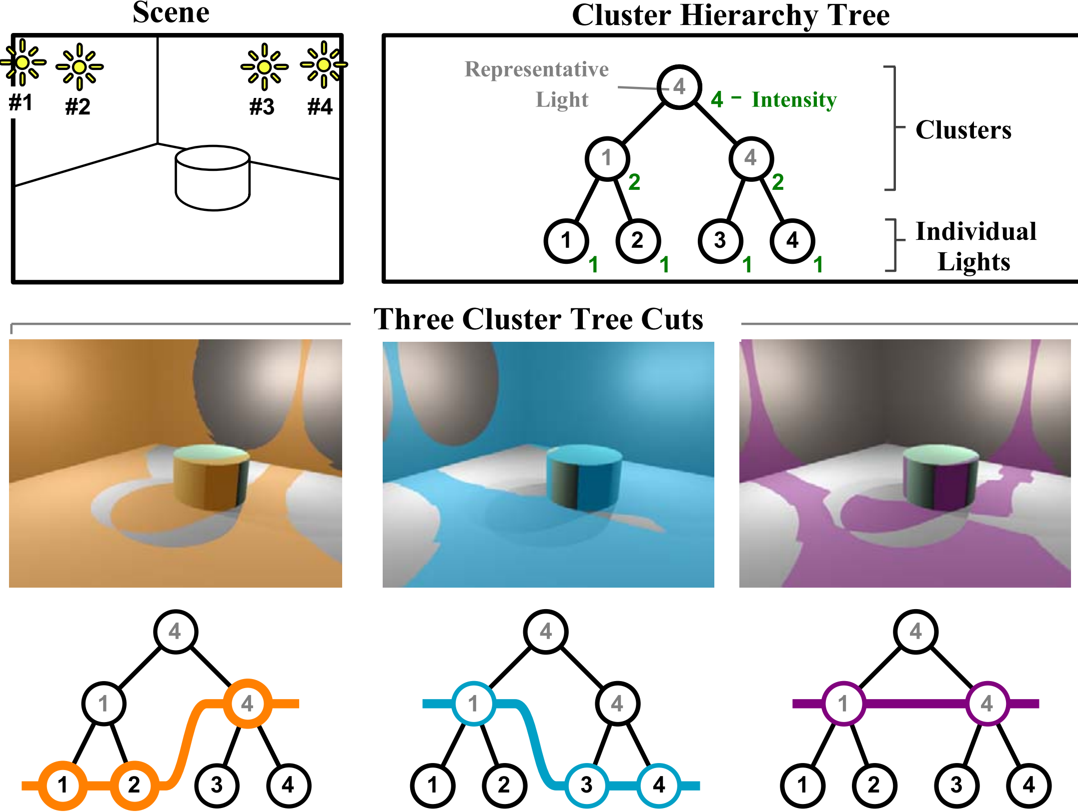



Figure 5.3: Cluster hierarchy tree and three example cuts for a simple scene with four lights. The cluster hierarchy is shown on the top along with the representative lights and cluster intensities for each node. The leaf nodes correspond to the individual lights while upper nodes correspond to progressively larger light clusters. The three cuts shown each represent a different partitioning of the lights into clusters (Note the orange cut is the same clustering as illustrated in Figure 5.2). Above each cut are highlighted in color the image regions where that clustering is virtually indistinguishable from the exact solution.

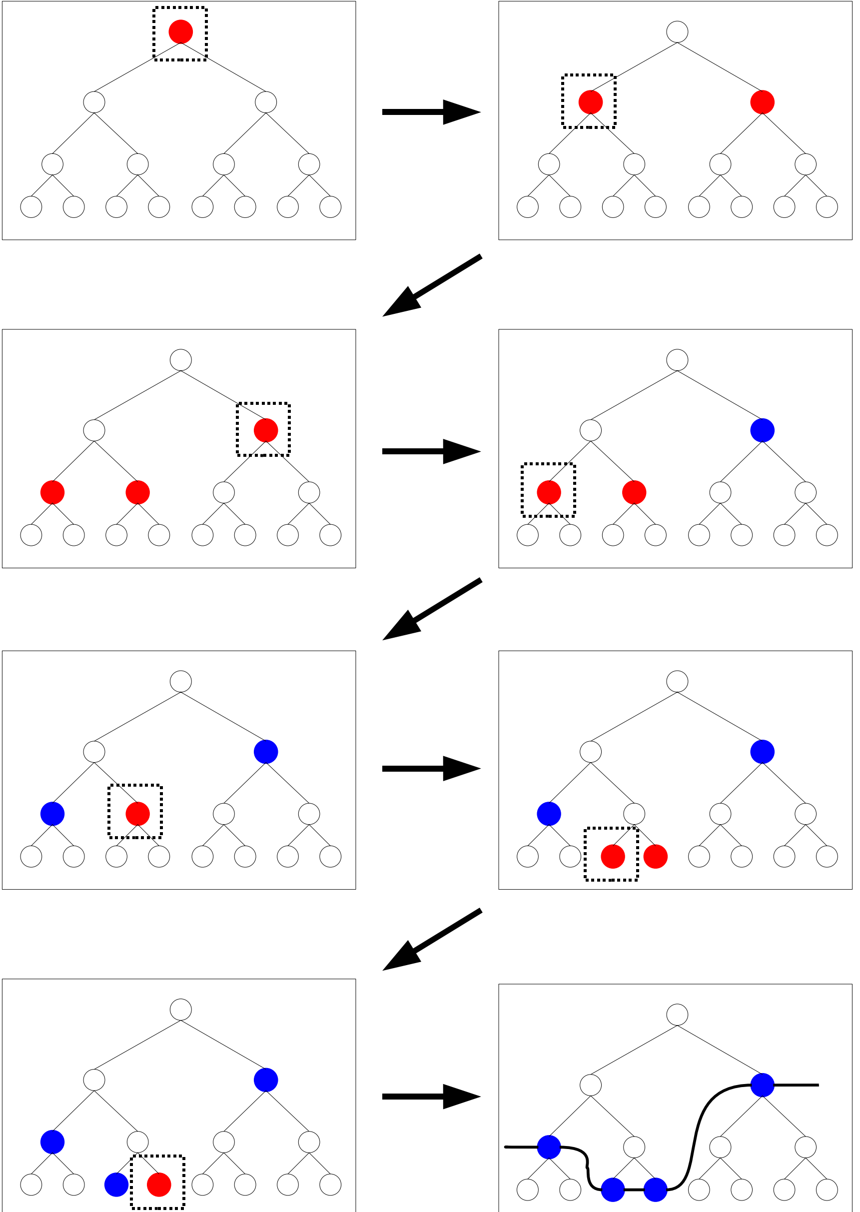


Figure 5.4: Finding a cut. Blue nodes are nodes in the final cut. Red nodes are nodes to be considered for inclusion in the cut. The algorithm begins with the root marked red. At each step, we select a red node (denoted by a square around the node) and decide whether it is acceptable for inclusion in the tree cut. If it is, it is marked blue. Otherwise, its children (if any) are marked red. The algorithm ends when there are no more red nodes. The blue nodes then denote our tree cut.

next section) the illumination of the cluster represented by the node and checking whether that falls below our threshold (e.g.,2% of the total illumination at the point). If the node illumination does fall below threshold or if the node is a leaf node, then it is considered part of the final cut and marked blue in the figure. If it does not fall below threshold and is not a leaf, then the node is removed from the set of red nodes and replaced with its children. Whether the node was marked blue or not, the algorithm iterates by choosing another red node and taking the same steps. The algorithm terminates when there are no red nodes left.

In order to decide whether a cluster's error is below our threshold, we need an estimate of the total illumination. We estimate this as the sum of the estimated contribution of all red and blue nodes. Whenever a red node is unmarked and its two children marked red, this estimate is updated.

If a node is not a leaf and its illumination is greater than threshold, we have to eventually compute the illumination of its children to determine whether they are above threshold. This means computing BRDF, geometry, and visibility factors for the representative lights of both of its children. However, we can reuse the factors already computed for the parent for one of its children because we required that the representative light of a cluster always be in the same position as the representative light of one of its children.

Weber's law (Equation 5.5) specifies that we are less sensitive to error in bright regions. This means, however, that we are *more* sensitive to error in *dark* regions. This effect can be seen in Figure 5.2f. Although the error close to the light is reduced compared to Figure 5.2e , the error on the side of the cylinder is magnified. Because we have a lower error threshold in dark regions, our algorithm can generate

excessively large cuts [3] in those areas. For example, if a point receives no light, then its error threshold will be zero and the cut would eventually be pushed down all the way to all the leaves resulting in wastefully shooting shadow rays to every single light. We can prevent this by setting a maximum cut size where cut refinement is stopped even if we have not found all the nodes in the final cut (there are still red nodes left)[4]. We have used a limit of 300 nodes and have found that it does not have any appreciable effect on image quality.

Although every tree cut corresponds to a valid partitioning of the lights into clusters, each resulting cut varies considerably in both cost and quality of the approximated illumination. Our goal for each surface point is to choose the cut with the least cost that will meet our error and quality criteria. The cost of a cut is proportional to the number of nodes in the cut, since this corresponds to the effective number of lights that have to be evaluated. The error, though, is more complicated to measure.

### 5.1.4 Error Estimates

Ideally, the errors introduced by our clusters should be imperceptible. Weber's law [Bla72]:

$$\frac{\triangle L}{L} = k \tag{5.5}$$

is a standard perceptual metric that says the minimum perceptible change ($\triangle L$) in a visual signal ($L$) is roughly equal to a fixed percentage ($k$) of the base signal. Under ideal conditions, humans can detect changes of just under 1%, though in practice the threshold is often a little higher. In our tests, keeping the allowed

[3] A large cut is a cut with many nodes.

[4] Alternatively, a threshold based on the illumination contribution of the nodes could have been used

error for each cluster under 2% to 3% of the total surface illumination produces images of very high quality with little to no noticeable noise.

We cannot efficiently calculate the exact error we introduce when approximating a cluster by its representative light, so we compute an estimate of the error. As explained below, we compute two error estimates using different techniques and combine them.

The first error estimate is conservative, based on the maximum possible error introduced by using a cluster. We use bounds on the $f_{\mathrm{r}}$, $G$, $V$, and $I$ factors possible within a cluster. The visibility $V$, to a point light is always zero or one, so we use can use one as a trivial upper bound. The light source intensities $I$ are known *a priori*.

The geometry factor $G$ depends on the distance between each light and the point on the surface being evaluated. To obtain a bound on this distance we store an axis-aligned bounding box with each cluster. The maximum value for the inverse of the distance squared occurs when the distance is smallest. We find a lower bound on the distance by finding the minimum possible value of each component. Thus:

$$
\begin{aligned}
x_{min} &= max(|c_x| - \frac{s_x}{2}, 0) \\
y_{min} &= max(|c_y| - \frac{s_y}{2}, 0) \\
z_{min} &= max(|c_z| - \frac{s_z}{2}, 0) \\
d^2_{min} &= x_{min}x_{min} + y_{min}y_{min} + z_{min}z_{min} \\
(\frac{1}{d^2})_{max} &= \frac{1}{d^2_{min}}
\end{aligned}
\tag{5.6}
$$

where $c$ is the vector from the point being shaded to the center of the bounding

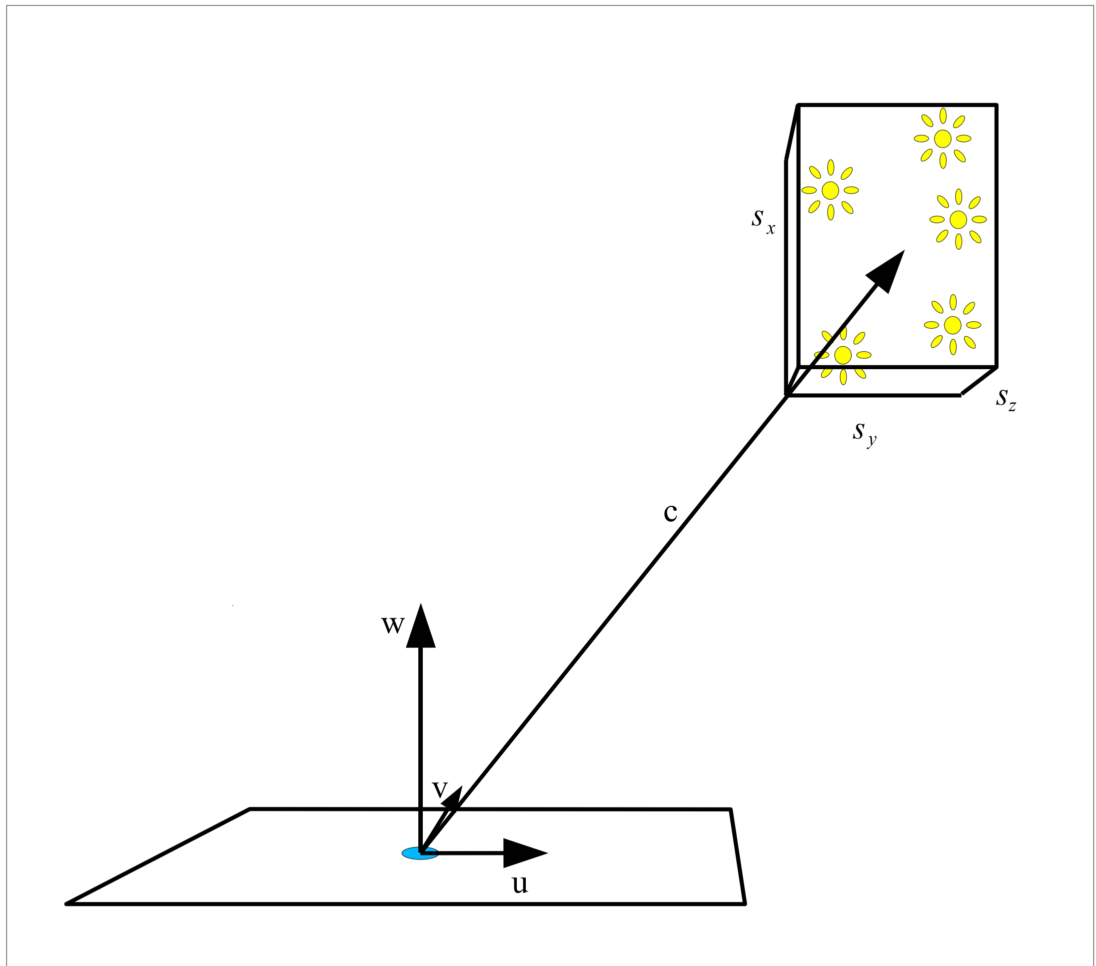



Figure 5.5: Terms used in bounding distance and cosine. $u, v, w$ form an orthonormal basis with $w$ as the normal at the surface. $c$ is the vector from the surface point to the center of the cluster bounding box. $s_x, s_y, s_z$ are the dimensions of the bounding box.

box and $s$ is the size of the bounding box, as illustrated in Figure 5.5.

We obtain an upper bound on the surface cosine term by finding the maximum component along the direction normal to the surface and the minimum component

along directions orthogonal to the surface:

$$
\begin{aligned}
u_{min} &= max(|u \cdot c| - (s_x|u_x| + s_y|u_y| + s_z|u_z|)/2, 0) \\
v_{min} &= max(|v \cdot c| - (s_x|v_x| + s_y|v_y| + s_z|v_z|)/2, 0) \\
w_{max} &= |w \cdot c| + (s_x|w_x| + s_y|w_y| + s_z|w_z|)/2 \\
cos_{max} &= \frac{w_{max}}{\sqrt{u_{min}^2 + v_{min}^2 + w_{max}^2}}
\end{aligned} \tag{5.7}
$$

where $w$ is the normal to the surface at the point being shaded, and $\{u, v, w\}$ form an orthonormal basis. To provide an intuition for this formula, we explain the derivation of the $u_{min}$ term.

In order to find the minimum $u$ component bounded by the box, we first obtain the distance to the center of the box, given by $|u \cdot c|$. In order to obtain the closest $u$ component bounded by the box, we need to subtract from this distance the $u$ component of the vector from the center of the box to the point $p$ on the box that minimizes $|u \cdot c| - |u \cdot p|$. This point $p$ will be one of the corners. The eight vectors from the center of the box to the corners of the box are denoted by $p = (\pm s_x, \pm s_y \pm s_z)/2$. We minimize $|u \cdot c| - |u \cdot p|$ by maximizing $|u \cdot p|$ over the eight possible corner points. This means maximizing each of the terms in the dot product. Since $p_x = \pm s_x$ and $s_x$ is positive, the maximum value for $u_x p_x$ is $u_x s_x$ if $u_x$ is positive and $-u_x s_x$ if $u_x$ is negative. In other words, the maximum value for $u_x p_x$ is $s_x|u_x|$. The other terms can be similarly derived.

It is also not immediately obvious that

$$
\frac{w_{max}}{\sqrt{u_{min}^2 + v_{min}^2 + w_{max}^2}}
$$

is an upper bound for

$$
\frac{w}{\sqrt{u^2 + v^2 + w^2}}
$$

In particular, one might expect

$$\frac{w_{max}}{\sqrt{u_{min}^2 + v_{min}^2 + w_{min}^2}}$$

instead. The latter is indeed an upper bound, but it is too conservative. We show here that the former is indeed an upper bound.

$$\begin{aligned}
w_{max} &\geq w, w \geq 0 \\
w_{max}^2 &\geq w^2 \\
w_{max}^2(u^2 + v^2) &\geq w^2(u^2 + v^2) \\
w_{max}^2(u^2 + v^2) + w_{max}^2 w^2 &\geq w^2(u^2 + v^2) + w_{max}^2 w^2 \\
w_{max}^2((u^2 + v^2) + w^2) &\geq w^2((u^2 + v^2) + w_{max}^2) \\
w_{max}^2 \frac{(u^2 + v^2) + w^2}{(u^2 + v^2) + w_{max}^2} &\geq w^2 \\
\frac{w_{max}^2}{(u^2 + v^2) + w_{max}^2} &\geq \frac{w^2}{(u^2 + v^2) + w^2} \\
\frac{w_{max}}{\sqrt{(u^2 + v^2) + w_{max}^2}} &\geq \frac{w}{\sqrt{(u^2 + v^2) + w^2}}
\end{aligned}$$

and of course, if $u \geq u_{min}$ and $v \geq v_{min}$ then

$$\frac{w_{max}}{\sqrt{u_{min}^2 + v_{min}^2 + w_{max}^2}} \geq \frac{w}{\sqrt{u^2 + v^2 + w^2}}$$

Using a technique similar to bounding the cosine term, we can quickly compute the maximum possible surface BRDF value. The bound is a simple constant for Lambertian materials. For Phong materials, we can use the same technique we used to find a bound on the cosine term, replacing the normal to the surface with the mirror reflection direction. Once the bound on the cosine term is found, the bound on $cos^n$ is trivial. Bounds for other materials could be determined in the future.

The upper bound-based error estimate can often be too conservative, so we instead use one-half the value of the conservative bound. However, sometimes

the conservative bound is accurate. For this reason, we also use a second error estimate, based on the representative light. This error estimate consists of actually evaluating, not bounding, all the terms in the lighting computation. However, we only do so for the representative light in the cluster, not for every light in the cluster. Our combined error estimate for a node $i$ is then given by:

$$Error(i) = max(0.5 \times Error_{bound}(i), Error_{rep}(i)) \tag{5.8}$$

$$Error_{bound}(i) = f_{\mathrm{r,max}} G_{max} V_{max} I_{max} \tag{5.9}$$

$$Error_{rep}(i) = f_{\mathrm{r,rep}} G_{rep} V_{rep} \sum_{i \in \mathbb{C}} I_i \tag{5.10}$$

The worst case overestimate of a node's contribution occurs when the representative light's contribution is significantly higher than that of the other lights in the cluster. This can happen when only the representative light is visible, when the representative light is particularly close to the surface being rendered, or on highly glossy surfaces when the representative light falls in the center of the BRDF lobe and the other lights in the cluster fall outside it. This overestimate can be arbitrarily high. However, since the error measure includes this overestimate and nodes whose error estimate is above threshold are not used in the final cut, the largest amount of error that can be introduced at a node due to overestimation is the threshold error.

The worst case underestimate of a node's contribution occurs when the representative light's contribution is significantly lower than that of the other lights in the cluster. This can happen when only the representative light is occluded, when other lights in the cluster are much closer to the surface being rendered than the representative light, or when the representative light falls outside a glossy BRDF lobe and other lights in the cluster do not. However, because 0.5 times the con-

servative upper bound must be below threshold, the largest amount of error that can be introduced at a node due to underestimation is twice the threshold error.

Ultimately, we're not guaranteeing an upper bound on the amount of error introduced by our estimates. Although we bound the amount of error introduced at any particular node, we do not bound the total. It is possible, for example, that from a particular point of view a large number of the representative lights are occluded while the rest of the lights in their clusters are visible, or vice versa. This type of scenario, however, rarely happens in practice.

## 5.2 Dense Sampling Algorithm

The first algorithm we describe is aimed at producing high-quality still images. We refer to it as the "Dense Sampling Algorithm" because it generates tree cuts at every pixel location (densely). This is in contrast with the other algorithms we will discuss later which produce samples more sparsely and reconstruct the illumination between the samples.

The algorithm is illustrated in Figure 5.6. We first cast a ray through a pixel on the image plane. This determines a point of intersection on a surface. We then generate a tree cut for this intersection point, as described in the previous section. The tree cut provides us with a set of clusters for which the cluster approximation is valid (Equation 5.4). We then compute the total radiance from that point in the direction of the viewer as the sum of the cluster approximations of the clusters contained in the cut. This radiance is used as the pixel color and the process is repeated until every pixel in the image has been computed.

This algorithm can generate images significantly faster than a standard ray

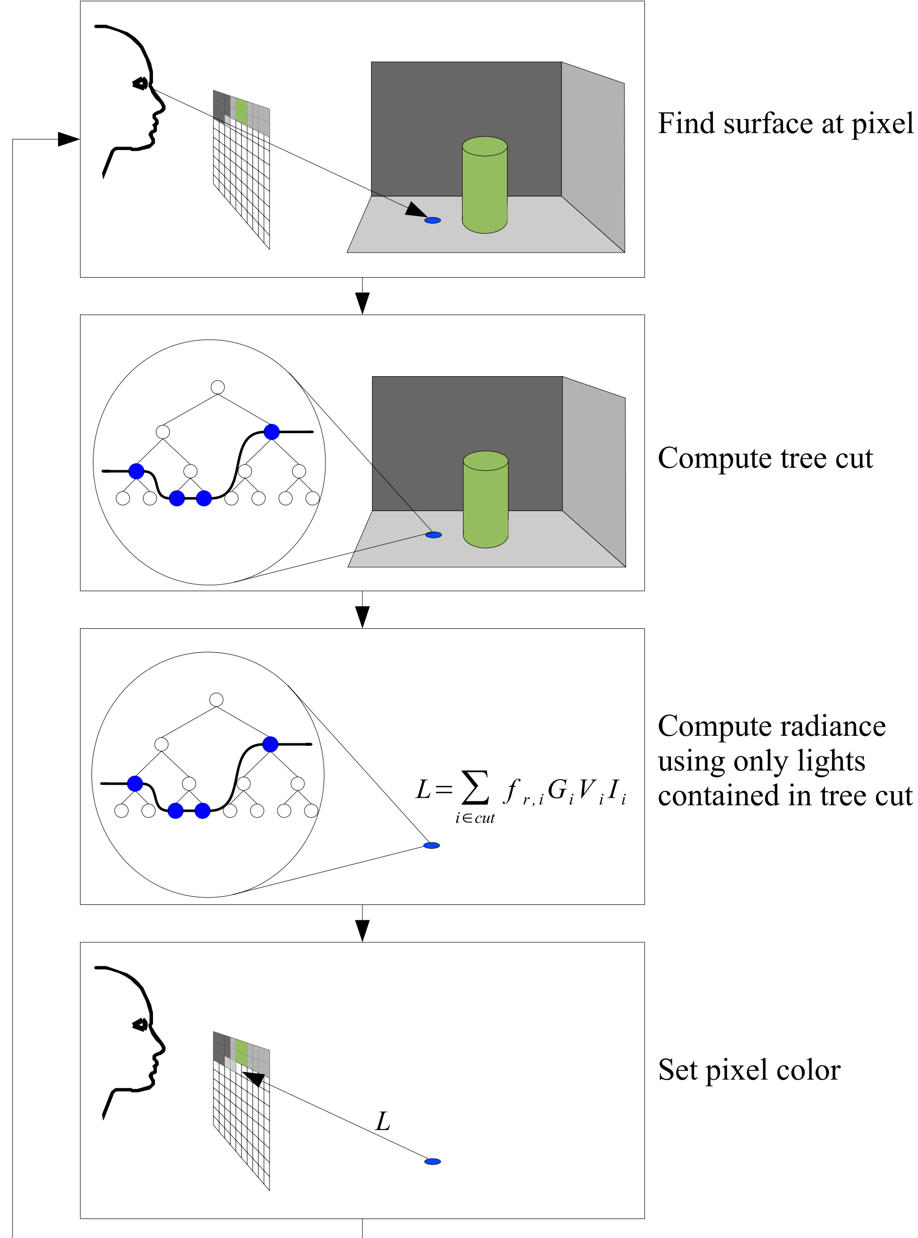



Figure 5.6: The dense sampling algorithm. We iterate through every pixel. At each pixel, we find a cut through the cluster tree. Only the nodes in the cut are used to compute the final color for the pixel.

tracer, as shown in the results section. Moreover, it does so without introducing noticeable artifacts.

## 5.3 Sparse Sampling Algorithms

Although the dense sampling algorithm provides substantial speedups over conventional rendering techniques, it is still too expensive for interactive use, even on a large cluster of PCs. In this section we will introduce a refinement of this algorithm that trades off some quality for a substantial performance improvement, allowing interactive walkthroughs on clusters of PCs.

We will describe two sparse sampling and reconstruction approaches. The first is a weighted sum reconstruction algorithm which is relatively quick but can only be used for smoothly varying illumination. The second is a tree-based reconstruction algorithm which can capture sharp shadow boundaries. Both methods rely on using tree cuts as described previously.

Our approach is to use the tree cuts (hereafter referred to as *sample cuts*) described in Section 5.1.3, but instead of generating them at each pixel location, generate them sparsely at visible surface locations in the scene. We can then apply reconstruction techniques to compute the shading between the samples. In general, we can compute these samples sparsely because of spatial coherence in the lighting. That is, the lighting usually does not change significantly from pixel to pixel. Storing the samples at surface locations allows us to reuse them from frame to frame, taking advantage of the temporal coherence when moving through a static environment.

Although these two approximation algorithms are significantly faster than the

dense sampling algorithm, they do suffer from sampling errors. For example, illumination features such as shadows, or specular highlights that are small enough to fall between samples can be lost. The algorithms are primarily suited for environments without small shadow details and composed of purely Lambertian materials.

### 5.3.1 Weighted Sum Reconstruction Algorithm

The weighted sum reconstruction algorithm is similar to the approach by [WRC88], sparsely sampling irradiance and interpolating between samples. The algorithm consists of two parts, sample generation and sample reconstruction. We will describe these two parts sequentially but, as described in Section 5.4.1, they operate in parallel.

Sample generation is illustrated in Figure 5.7. We first choose a random point on the image plane and send a ray through it. This ray intersects some point $x_i$ on a surface. This will be our sample location. We then generate a sample cut (as described previously) at the sample location. The cut describes which light clusters form a good approximation of the lighting at that sample location. We then use a modified cluster approximation (Equation 5.4) to compute the irradiance at the sample point:

$$E_{\mathbb{C}} = \sum_{i \in \mathbb{C}} [G_i \; V_i \; I_i] \; \approx \; G_j \; V_j \sum_{i \in \mathbb{C}} I_i \tag{5.11}$$

The total irradiance at the point is then the sum of $E_{\mathbb{C}}$ for all the clusters in the cut. This irradiance is then stored at the sample. The process is repeated to generate multiple samples.

Reconstruction is illustrated in Figure 5.8. For each pixel to be shaded, we shoot a ray through the pixel. This ray intersects some point $x$ on a surface. We

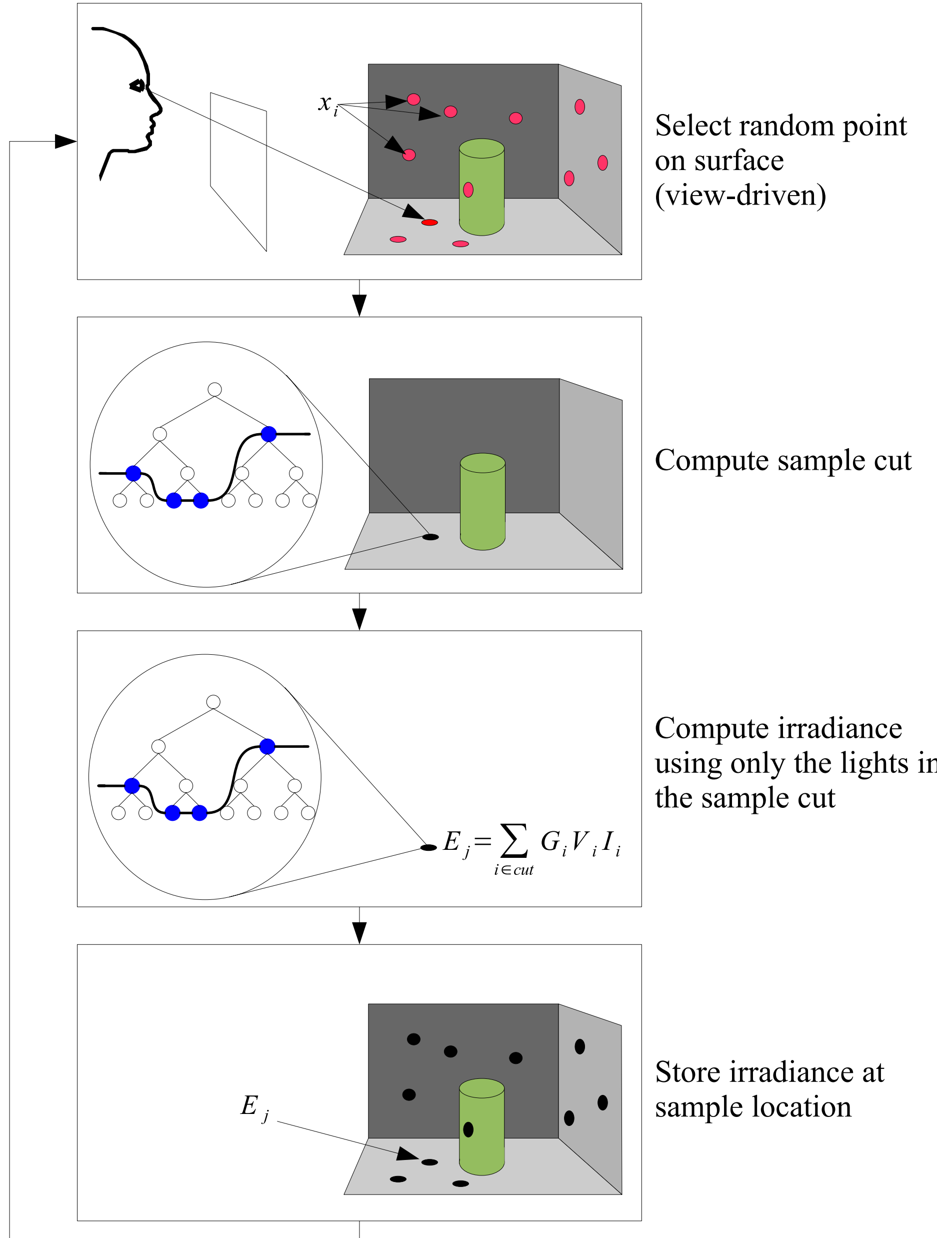



Figure 5.7: The sample generation portion of the weighted sum reconstruction algorithm. Samples are generated by selecting random points on the image plane (red samples). At each sample location, a sample cut is formed and these nodes are used to compute irradiance. The irradiance is then stored in the sample for use in the reconstruction portion of the algorithm (black samples).

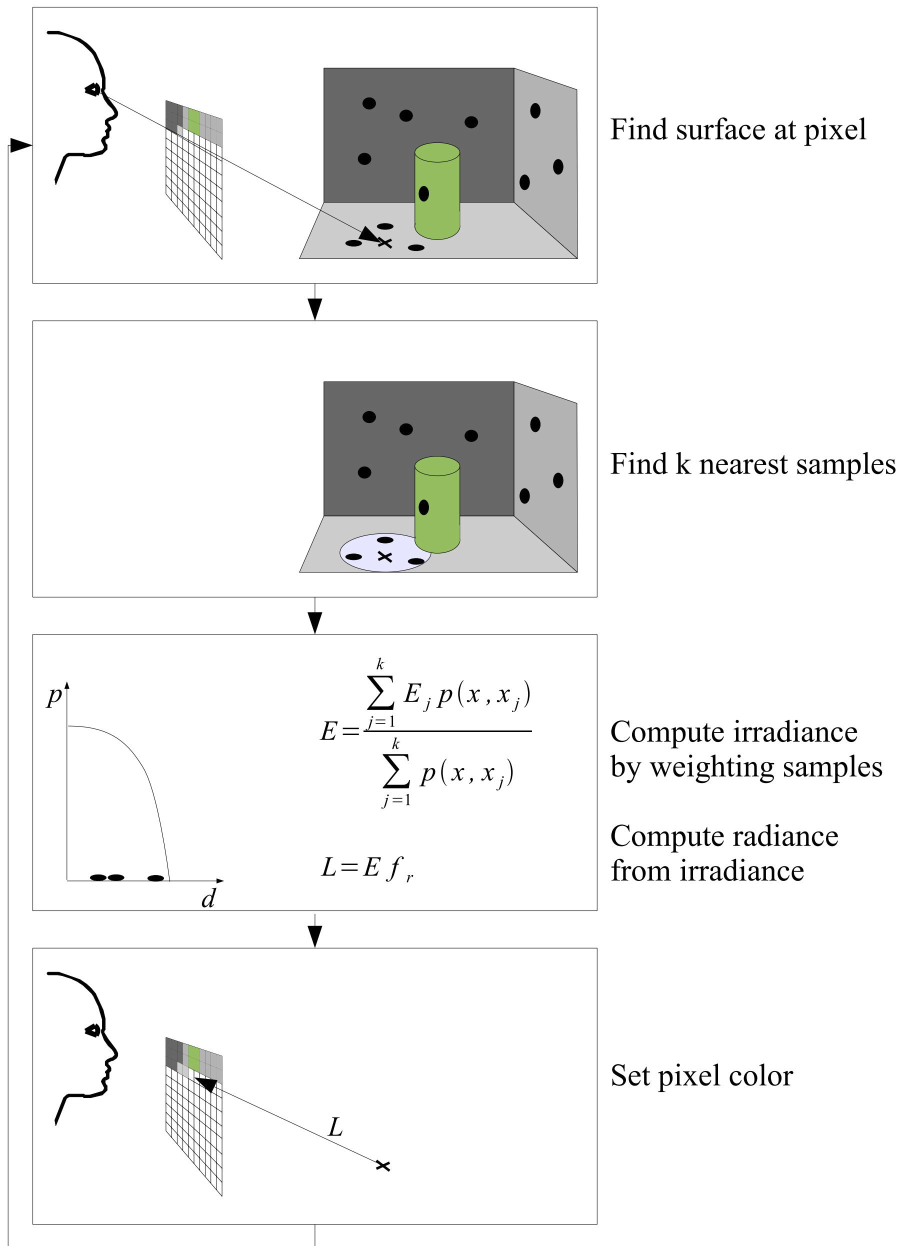



Figure 5.8: The reconstruction portion of the weighted sum reconstruction algorithm. When computing the color for a pixel, the $k$ nearest samples are used. Their irradiances are combined using the Epanechnikov kernel. The final radiance for the pixel is computed by multiplying the irradiance with the diffuse BRDF of the surface.

proceed to collect the $k$ nearest samples to $x$ (previously stored).[5] The irradiances at these samples are combined using the Epanechnikov kernel which has some beneficial characteristics as described below.

$$E(x) = \frac{\sum_i p(x,x_i) E_{x_i}}{\sum_i p(x,x_i)} \quad (5.12)$$

$$p(x,x_i) = \frac{d(x,x_k)^2 - d(x,x_i)^2}{d(x,x_k)^2} \quad (5.13)$$

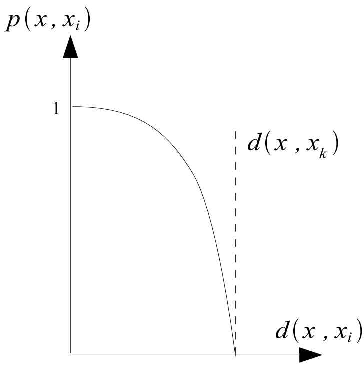



Figure 5.9: The Epanechnikov weighting function.

where $x_i$ are the samples, in order of distance from $x$, $E_{x_i}$ is the irradiance stored in sample $x_i$ and $d$ is the Cartesian distance between two points. The weighting function is illustrated in Figure 5.9.

This kernel assigns higher weights to nearby samples. The farthest sample, $x_k$ is not used in the averaging (it has a weight of 0) and is just used to establish a radius for the kernel. This means that the kernel is bounded. Furthermore, since the kernel radius is the distance to the $k$th nearest sample, the size of the kernel shrinks with high sample density and grows with low sample density. This means that if we do have a high sample density, we will not be incorporating distant

[5] As discussed in Section 5.4.2, the normal of the surface is also taken into account when finding the $k$ nearest samples.

samples into our irradiance estimate. The weighting function also has the nice property that it goes to zero at the boundaries, so there are no discontinuities at the boundary of the kernel.

Once we have computed our irradiance estimate $E(x)$, we multiply by the diffuse surface BRDF to compute the estimated radiance:

$$L(x) = f_r(x)E(x) \tag{5.14}$$

The radiance is then assigned as the pixel color and the process is repeated until an entire image is generated.

### 5.3.2 Tree-Based Reconstruction Algorithm

Irradiance interpolation using the weighted sum method described previously is appropriate for smoothly varying illumination; however it is not appropriate for sharp shadow boundaries. Using this approach in such a context results in light leaks and blurred shadows, as can be seen in the results section.

One approach we could take in order to benefit from the speed of irradiance interpolation without blurring shadows is the following. Inspect the irradiance samples for similar values. If the samples have similar values, it is likely that we are in a region with smoothly varying irradiance and can cheaply interpolate. If they are substantially different, then we can resort to a conventional ray-tracing of the pixel in question. This allows us to benefit from interpolation acceleration in smoothly varying regions yet obtain correct rendering at discontinuities [6].

The problem with this approach is that, in environments with many lights,

[6] This approach will, however, miss small details that fall completely between the samples. This can be solved with denser sampling.

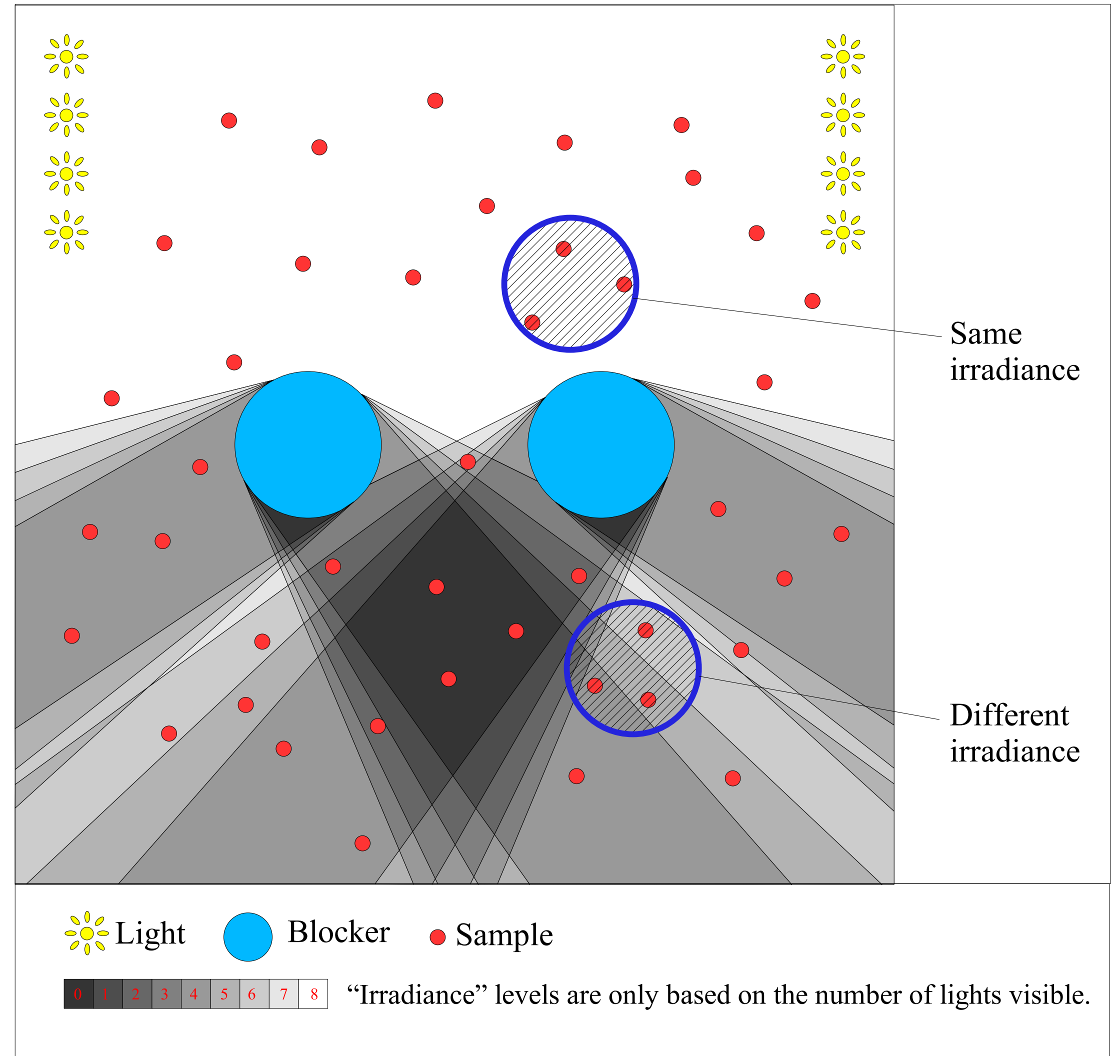



Figure 5.10: Irradiance discontinuities are common in environments with many lights.

illumination discontinuities are common and it will therefore be difficult to find regions where all samples have similar irradiance values. Figure 5.10 illustrates this problem. In this figure we have constructed a simple environment with eight lights of equal brightness, two objects that block the lights and some sample locations. To further simplify the concepts, we ignore the effects of the geometric factors when computing the “irradiance” in the figure and assume each light has unit intensity;

thus the "irradiance" does not vary with distance from the light source and is simply proportional to the number of lights visible at a point. As can be seen, even in such a simplified environment, the number of irradiance discontinuities is quite large. While an interpolation can work in the upper unoccluded regions, it is hard to find adjacent samples with similar irradiances in the lower part of the figure, due to the varying amounts of light occlusion.

However, even though the total sample irradiances differ, if we decompose the contribution by cluster nodes, as seen in Figure 5.11, we see some similarities. In this figure, we show the irradiance contribution due to each cluster at each sample. The top node differs for all three sample points, since the total irradiance (number of lights visible) is not the same, but some of the lower nodes are the same. Thus, we could interpolate portions of the illumination across samples for the nodes whose *cluster irradiance* ($E_{\mathbb{C}}$ as defined in Equation 5.11) does not vary among the samples and fully compute the irradiance due to those nodes where it does.

Our algorithm would then consist of first storing at each sample the cluster irradiance at all nodes of the tree. When rendering a pixel, we collect the nearest samples and compare their trees to each other, node by node starting at the root node. If a node has the same cluster irradiance for all the samples, that value is used as the irradiance contribution from that cluster. Otherwise, the node's children are compared. If we reach a leaf without finding the same irradiance at all the samples, the contribution of that light is directly evaluated through a shadow ray cast.

One problem with this approach is that the geometry factors *will* vary from sample to sample, as the distance and orientation to the cluster and the surface

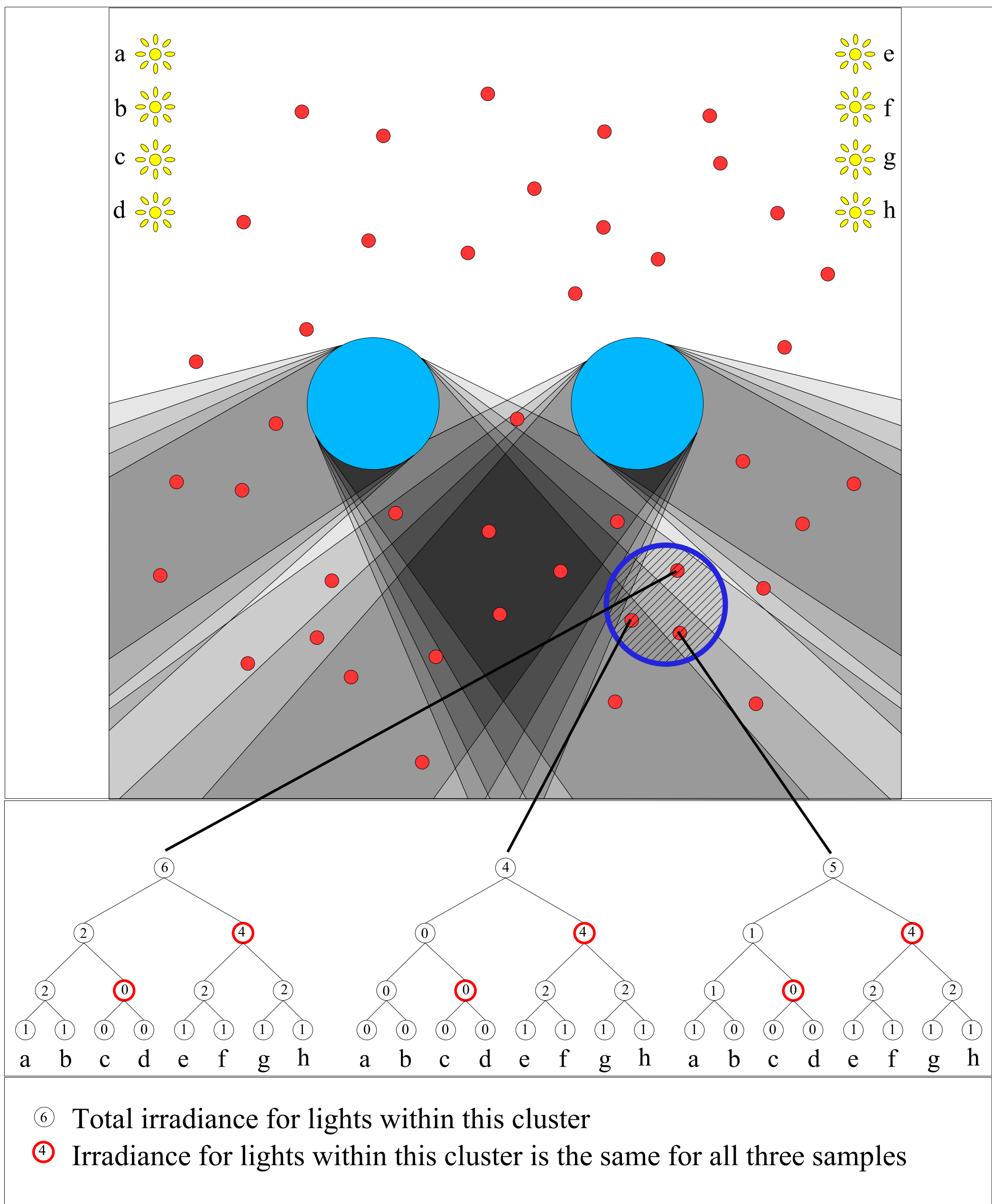



Figure 5.11: Even though total irradiances differ among the samples, they share some common cluster irradiances. A value of 1 in the leaf nodes indicates the specific light is visible.

orientation change. We will therefore rarely get samples with exactly the same cluster irradiance. Even if we allow for some error, performing simple interpolation of the cluster irradiance will lead to visual artifacts, particularly if there is surface normal variation among the samples.

We therefore instead store at each node a *relative cluster irradiance*, which we define as

$$R_{\mathbb{C}}(x_i) = \frac{E_{\mathbb{C}}(x_i)}{G_j(x_i)\sum_{i\in\mathbb{C}} I_i} \tag{5.15}$$

where $j$ is the representative light for the cluster and $x_i$ is a sample point.

Relative cluster irradiance divides cluster irradiance by the geometry factor of the representative. Since, at least at a distance, the geometry factor of the representative light should vary in the same way as the geometry factors for the lights in the cluster, this should reduce this source of variability. We also divide the cluster irradiance by the total intensity of the cluster to reduce the dynamic range of the cluster irradiance and make storage less expensive.

Of course, when rendering a point $x$ that is not a sample point, we actually want $E_{\mathbb{C}}(x)$. This is approximated inexpensively via

$$E_{\mathbb{C}}(x) = R_{\mathbb{C}}(x_i)G_j(x)\sum_{i\in\mathbb{C}} I_i \tag{5.16}$$

where $x_i$ is the nearest sample point.

### Algorithm

We now describe the "reconstruction cut" algorithm. It consists of two parts, sample generation and reconstruction. As in the weighted sum reconstruction algorithm, these parts which will be described operate in parallel.

Sample generation is illustrated in Figure 5.12. We first choose a random point on the image plane and send a ray through it. The ray intersects some point $x_i$ on a surface. This will be our sample location. We then generate a sample cut at the sample location. At this point, instead of computing the total irradiance at the sample point, we compute the relative cluster irradiance for every node. This is a relatively inexpensive operation since we have already computed the cluster irradiances for the tree when computing the sample cut. This tree of relative cluster irradiances is compressed, as described in Section 5.4.1, and stored with the sample point.

Reconstruction is illustrated in Figure 5.13. For each pixel to be shaded, we shoot a ray through the pixel. This ray intersects some point $x$ on a surface. We collect the $k$ nearest samples to $x$ and combine them to produce a reconstruction cut, as described in the following paragraphs. The cluster irradiances at the nodes of the reconstruction cut are then summed to produce a total irradiance for point $x$:

$$L = f_r \sum_i E_{\mathbb{C}_i} \tag{5.17}$$

The irradiance is then multiplied by the diffuse surface BRDF to compute the estimated radiance. The radiance is used as the pixel color. The process is repeated until an entire image is generated.

### Finding Cuts

Reconstruction cuts are built in a manner similar to sample cuts, illustrated in Figure 5.4, with two differences. First, two types of nodes are generated in the final cut: interpolated nodes and evaluated nodes. Second, we use a different oracle to determine whether a node belongs in the reconstruction cut.

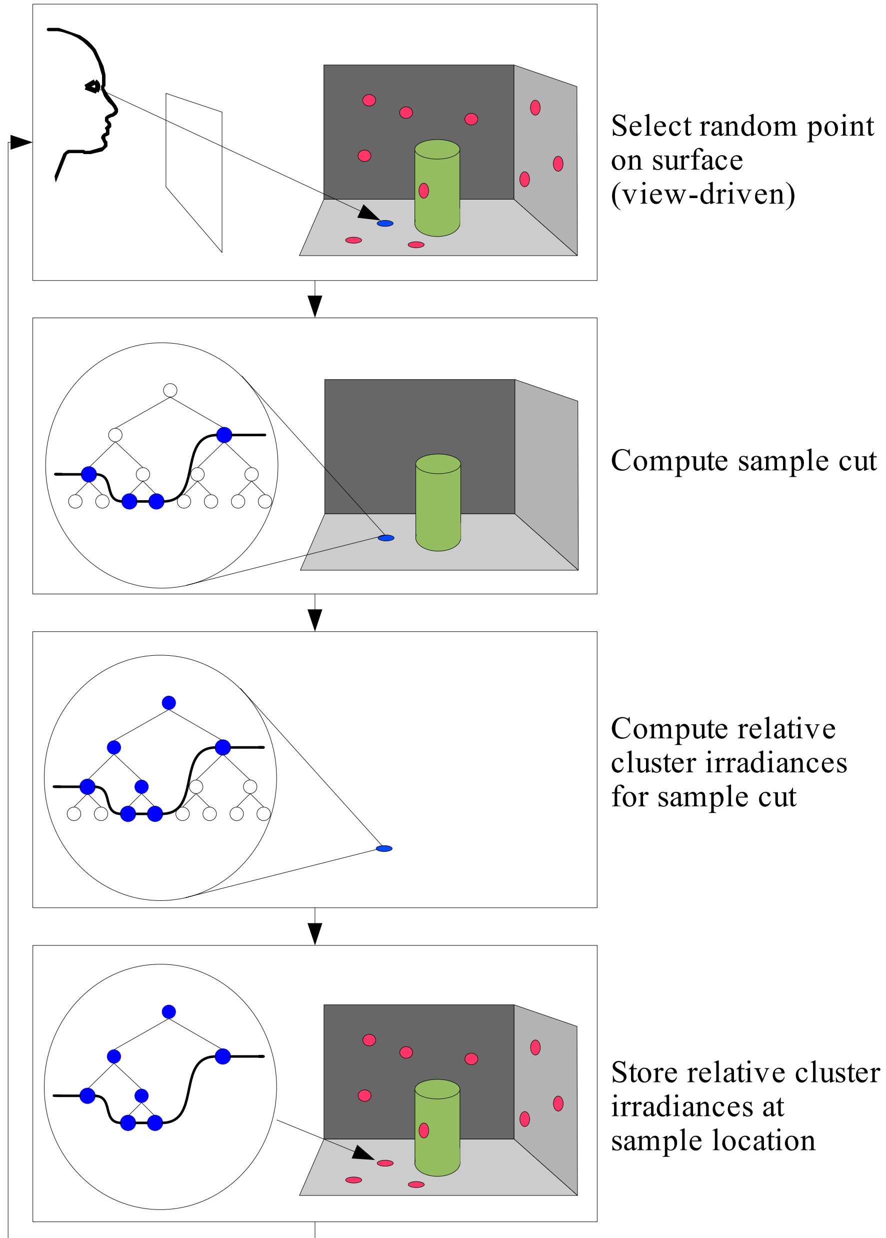



Figure 5.12: The sample generation portion of the tree-based reconstruction algorithm. Samples are generated by selecting random points on the image plane. At each sample location, a cut is formed. The nodes in the cut are used to compute a tree of relative cluster irradiances. This tree is then stored in the sample for use in the reconstruction portion of the algorithm.

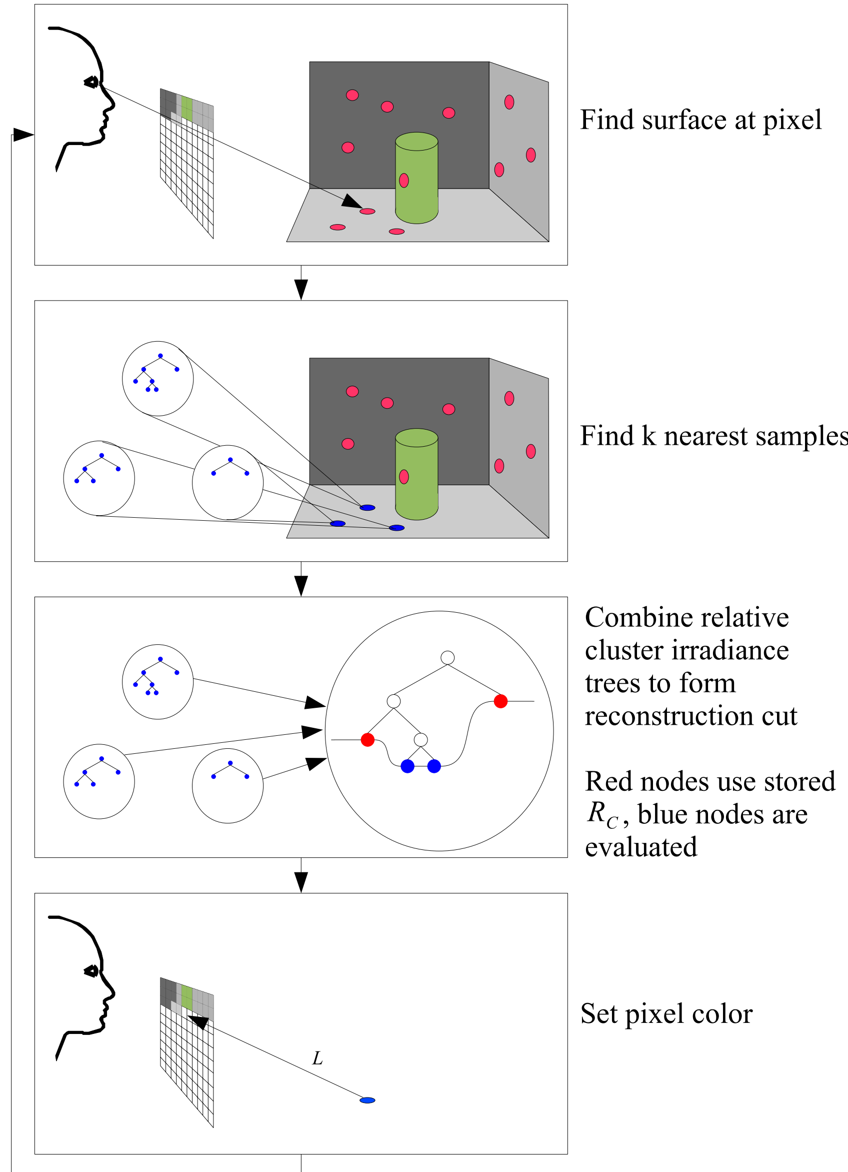



Figure 5.13: The reconstruction portion of the tree-based reconstruction algorithm. When computing the color for a pixel, the $k$ nearest samples are used. Their relative cluster irradiance trees are combined to form a reconstruction cut. The cluster irradiance of the nodes in the reconstruction cut is computed either through evaluation or interpolation. The cluster irradiances are then summed and multiplied by the diffuse surface BRDF to obtain a radiance value for the pixel.

The oracle first tests to see whether a node is an evaluation node. Evaluation nodes will have their cluster irradiance determined via the modified cluster approximation (Equation 5.11) i.e. based on the visibility and geometric factors of the representative light. A node is determined to be an evaluation node if

$$I_{\mathbb{C}_i} G_{\hat{i}} < t_{eval} \tag{5.18}$$

where $I_{\mathbb{C}_i}$ is the total intensity of the cluster, $G_{\hat{i}}$ is the geometric factor of the cluster's representative light and $t_{eval}$ is 2% of the total irradiance at the surface. This is similar to the oracle used to determine whether a node is part of the sample cut in the dense sampling algorithm, except that we do not compute conservative bounds and we do not evaluate the representative light's visibility, both for performance reasons.

If a node is determined to be an evaluation node, then its cluster irradiance is computed by

$$E_{\mathbb{C}_i} = I_{\mathbb{C}_i} G_{\hat{i}} V_{\hat{i}} \tag{5.19}$$

If a node is not an evaluation node, then we check whether it is an interpolation node. A node is determined to be an interpolation node if

$$\frac{(max_j(R_{x_j,i}) - avg_j(R_{x_j,i}))}{avg_j(R_{x_j,i})} < t_{interp} \tag{5.20}$$

where $R_{x_j,i}$ is the relative cluster irradiance for node $i$ at sample $x_j$ and $t_{interp}$ is a parameter to the algorithm (we used 0.05 in our renderings).

If a node is determined to be an interpolation node, then its cluster irradiance is currently computed by

$$E_{\mathbb{C}_i} = I_{\mathbb{C}_i} G_{\hat{i}} R_{x_0,i} \tag{5.21}$$

although a better approximation might be obtained by interpolating the relative cluster irradiances instead of using the relative cluster irradiance of the nearest sample.

If a node is neither an evaluation node nor an interpolation node, then it does not become part of the reconstruction cut and its children are checked. Leaf nodes that fall in this category are considered to be evaluation nodes.

## 5.4 System

We have built an interactive parallel system to implement these algorithms. In this section, we describe this parallel system, along with some details of the implementation.

### 5.4.1 Parallel System

Although the sampling algorithms are substantially faster than generating sample cuts at every pixel, they are still not fast enough to provide interactive performance on a single machine. To provide this performance we use a parallel cluster of machines as illustrated in Figure 5.14. Samples are generated asynchronously on a single machine and distributed to the rest of the machines for reconstruction and shading. The shaders generate pixel colors which are sent to a computer for final assembly and display. The shaders also generate sample priorities (covered in the next section) which are sent back to the sampler.

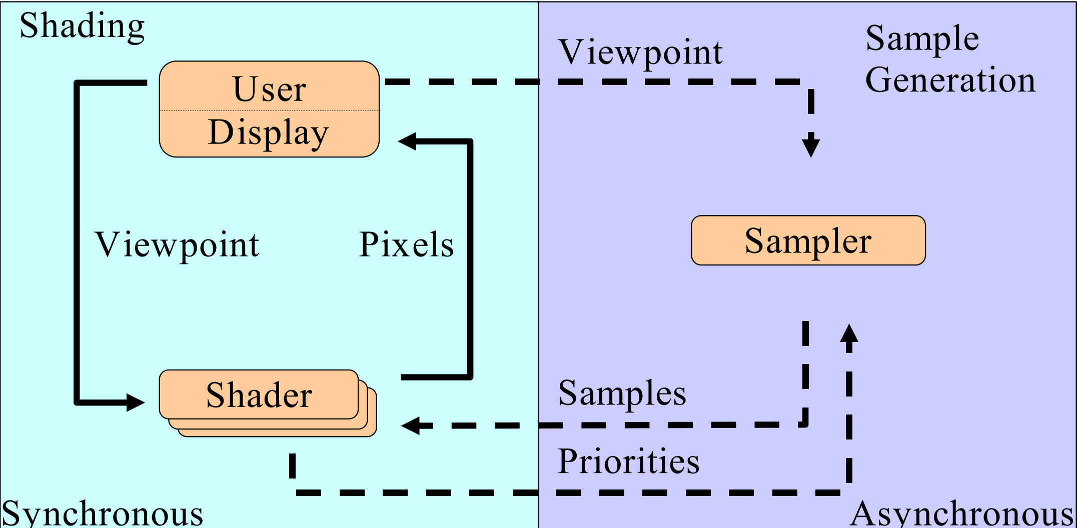



Figure 5.14: The interactive system. The shaders operate synchronously to generate images from the samples. The samplers generate samples asynchronously from the shaders. The current viewpoint is used to drive sample generation.

### 5.4.2 Nearest Samples

When reconstructing illumination from samples, we use the $k$ nearest samples to the surface point in question. However, the $k$ spatially nearest samples may not be appropriate. We would like to take the surface normal of the sample into account, so that we do not attempt to use samples that may have a very different surface normal. We define our distance metric as

$$d(x,s) = \begin{cases} ||x-s||/(\widehat{n_x}\cdot\widehat{n_s}) & \text{if } \widehat{n_x}\cdot\widehat{n_s} > 0 \\ \infty & \text{otherwise} \end{cases} \tag{5.22}$$

where $\widehat{n_x}$ is the normal to the surface at point $x$ and $\widehat{n_s}$ is the normal to the surface at sample $s$. A kd-tree data structure is used to quickly find the $k$ nearest samples.

### 5.4.3 Sample Priorities

When combining samples to reconstruct shading at a point, it is sometimes the case that the nearby samples are too dissimilar. This can be due to differences in the irradiances in the case of interpolation or very different sample cuts when trying to create a reconstruction cut. These dissimilarities can usually be fixed by a higher sample density near the problem point. We thus create a feedback mechanism so that the interpolation and reconstruction cut processes can request additional sampling.

Each shader considers each point it shades as a potential sample location. The points are assigned a priority and forwarded to the sample generation machine. The sample generator picks from all the requests randomly with probability proportional to their priority. The priority is based on a simple formula: the size of the reconstruction cut last evaluated at the pixel times the distance to the nearest sample. This encourages samples at expensive reconstruction cuts and discourages samples from clustering together.

Although this mechanism helps guide our sampling to regions that need it, we can still miss details such as small shadows. For this reason, we also choose some samples uniformly over the image.

### 5.4.4 Storage

Storing and transmitting a tree of relative cluster irradiances per sample naively could get quite expensive, since our trees normally have hundreds of nodes. We use several mechanisms to reduce sample size.

We only store relative cluster irradiances for nodes on or above the sample cuts. Values for nodes below the sample cut are assumed to be equal to that of

the nearest ancestor of the node within the sample cut.

We encode the tree as two arrays, as illustrated in Figure 5.15. The first array holds the relative cluster irradiances for every node above the sample cut. The second array holds the index of the left child of that node into both arrays.

We first assign a “sample cut index” to each node at or above the sample cut. This is the index of the node into the two arrays. The root is assigned an index of 0. We then proceed down the tree in a depth-first manner, incrementing a counter used to assign the index. We maintain the property that the right child of a node is always assigned an index that is one greater than the left child of the node. We can do this because cuts have the property that either both children are at or above the cut or neither of them are.

We then place the relative cluster irradiance for each node at its sample cut index in the relative cluster irradiance array. We also place the sample cut index for each node's left child (the right child's index is implicitly one more than the left child's index) into the left child index array at that node's sample cut index. Nodes without children at or above the sample cut get a zero for their left child index entry.

The left child index array is used to reconstruct the tree when needed. Given that we're visiting any particular node in the tree, the left child index array tells us the index into both arrays of its children, if any. This of course requires that we know the index of the node itself, but this was obtained when we visited its parent. This approach requires that we visit a node's parent before we visit the node, but this is our normal traversal approach anyways.

We also save memory and bandwidth by compacting the relative cluster irradiance values. Instead of storing each value as a 4-byte floating point number, we

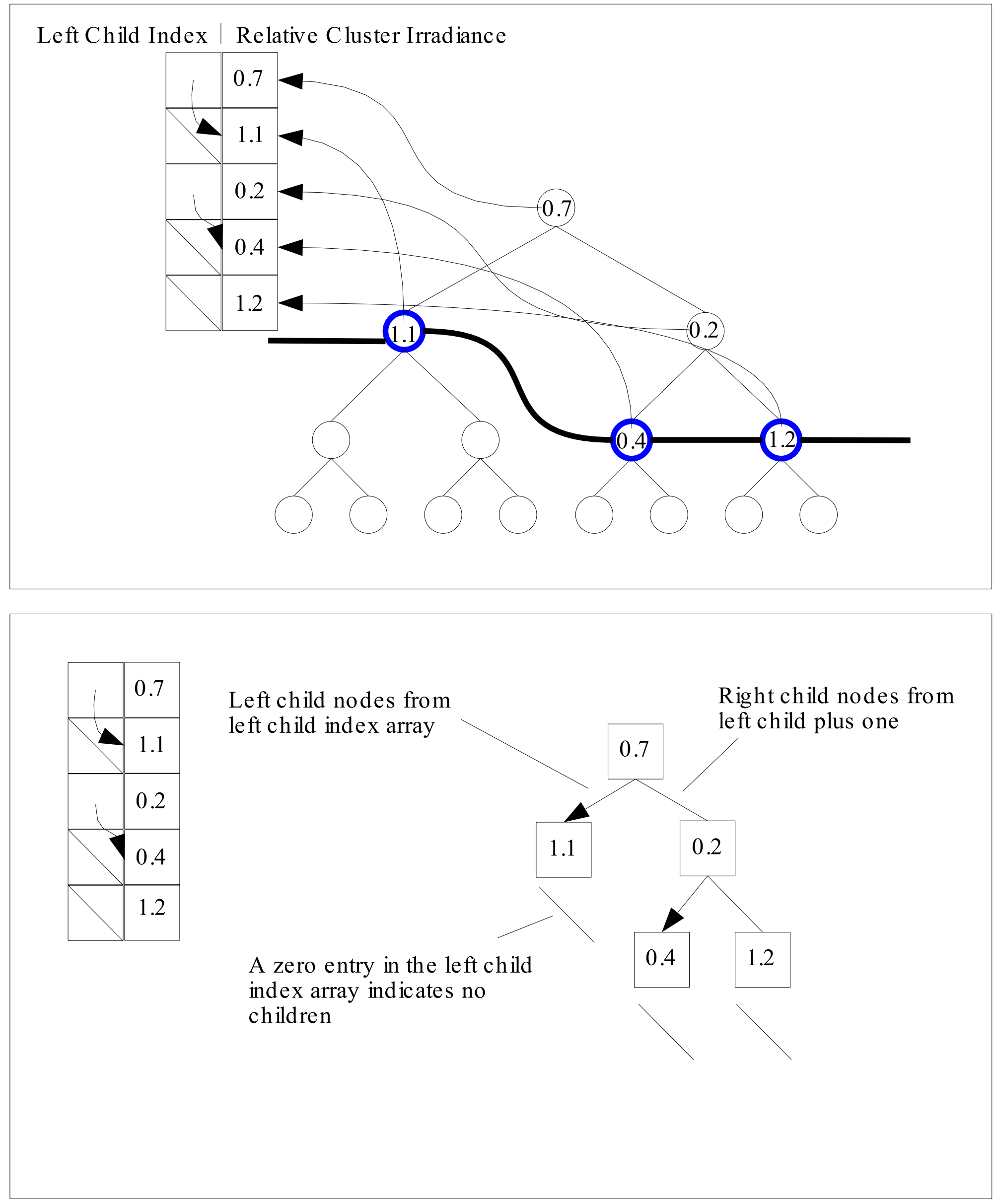



Figure 5.15: Packing a sample cut into two arrays. In the top figure we show a tree with some random values for relative cluster irradiance and a given cut. We first place the relative cluster irradiances into the relative cluster irradiance array. We then create a left child index array which indicates the position in the arrays of the left child of a given node. In the bottom figure, we show the tree being reconstructed from the array.

store it as a 2-byte fixed point value. This does not introduce much error since we designed relative cluster irradiance to not have great dynamic range. Left child indices are also stored as a 2-byte quantity. This approach would have to be modified when dealing with scenes of more than 32,768 lights.

Since relative cluster irradiance will be extrapolated to nodes below the sample cut anyways, whenever a node above the sample cut has the same value as its two children at the sample cut, we collapse the sample cut up one level. This technique is applied until there are no more nodes to collapse. This approach can be particularly effective in parts of the tree that have many zero values due to occlusion.

All these techniques combined allow us to compress the relative cluster irradiance trees by a factor of six over naive storage of the entire tree. Sample sizes for the Grand Central model, for example, are less than 1 KB per sample.

## 5.5 Results

In this section we describe the results from using these algorithms. We tested our system on three models each with many lights. These models are described in Figure 5.16. On the following pages, we show images and timings of each model run under the following four algorithms:

- **Standard Ray Tracer:** A conventional ray tracer with a KD-tree-based acceleration structure is used. The ray tracer does not perform pixel anti-aliasing, and sends one primary ray for each pixel. The ray tracer evaluates every light at a point being rendered except for those lights that are not facing a surface or that are below the horizon of a surface (zero geometry fac-

| Model | Characteristics |
|---|---|
| Grand Central Station | **1,526K triangles** **831 point lights** Open environment, most lights are indirectly visible from most views. Light configuration consists of a few chandeliers with 80 lights each and two rows of lights along the ceiling. |
| Mosque de Cordoba | **1,096K triangles** **839 point lights** An environment with many cylindrical columns casting shadows. Lights are arranged in small circles of a few lights around the top of each column. A stronger set of lights is placed along the central corridor. |
| Residential Kitchen | **388K triangles** **72 disk area lights** **(4,608 generated oriented virtual lights)** Most of the lights here are placed on the ceiling with some lights located under the counters. The area lights cast soft shadows that harden on contact above the counters. This scene was modeled by Jeremiah Fairbank and William Stokes. |

Figure 5.16: Characteristics of the models used for testing.

tor). Glossy and diffuse materials are supported for direct lighting. Indirect lighting is only supported through the specular path. Using Heckbert notation [7] [Hec90], $ES*(D|G|S)?L$ paths are supported. Texture anti-aliasing is supported through the use of ray differentials [Ige99].

- **Dense Sampling:** This is the dense sampling algorithm discussed in Section 5.2. Primary and shadow rays are computed using the same acceleration structure as the standard ray tracer. As in the standard ray tracer, $ES*(D|G|S)?L$ paths are supported.

- **Sparse Sampling/Interpolated:** This is the sparse weighted sample algorithm discussed in Section 5.3.1. The glossy direct lighting component is not computed, nor is indirect through a specular bounce. Primary and shadow rays are computed using the same acceleration structure as the standard ray tracer. $ED?L$ paths are supported.

- **Sparse Sampling/Reconstructed:** This is the sparse reconstruction cut algorithm discussed in Section 5.3.2. Primary and shadow rays are computed using the same acceleration structure as the standard ray tracer. $ED?L$ paths are supported.

For the sparse sampling algorithms, we used approximately 5,000 samples for each image. We break down the statistics for each of the sampling algorithms into sample generation and sample reconstruction and report those below the total statistics for each algorithm. For sample generation, we amortize the cost over all the pixels in the image. The total time spent in sampling and reconstruction is

[7] We actually use an extension of Heckbert notation, where $G$ is used for glossy reflections and ? is used for zero or one bounces of the previous type.

also reported.

Timings are given for a single-processor 1.7Ghz Pentium IV rendering a 512 × 512 image. Times scale linearly with the number of pixels and inversely with the number of processors.

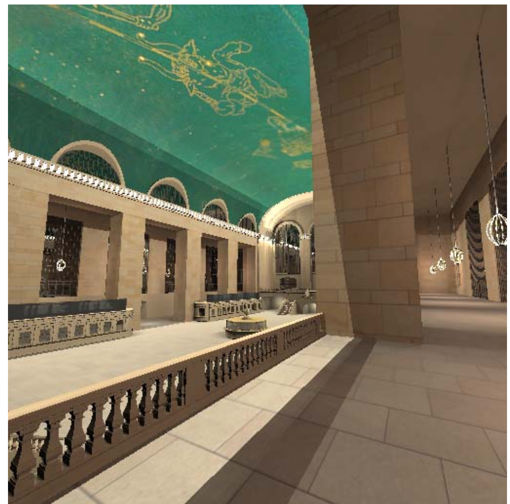
The results show that the dense sampling algorithm provides a modest speedup over the conventional ray tracer for models with several hundreds of lights. As the number of virtual lights increases (e.g. the Kitchen model), the speedup is more substantial. Note that the dense sampling algorithm generates images that are almost indistinguishable from the conventional ray tracer.

The sparse sampling algorithms provide a very large speedup over the conventional ray-tracer and the dense sampling algorithms, allowing interactive walk-throughs on a cluster of CPUs. The sparse/interpolated algorithm is faster than the sparse/reconstructed algorithm, but has substantial artifacts, particularly around shadow boundaries. The sparse/reconstructed algorithm captures some shadow boundaries but still exhibits some artifacts due to undersampling. Both sparse sampling algorithms have difficulty with glossy and specular surfaces.

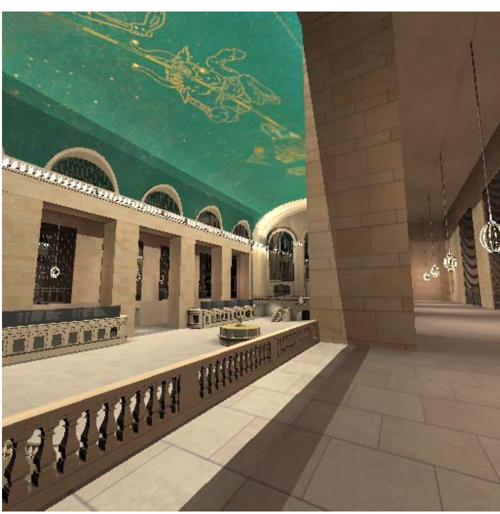
## 5.6 Conclusion

In this chapter, we presented three related direct lighting algorithms that can efficiently render environments with hundreds of lights. The dense sampling algorithm can be used to produce still images that are nearly indistinguishable from a reference image, but take less time to produce. The two sparse sampling algorithms can be used for interactive walkthroughs on clusters of computers, but the quality is often not sufficient for final production.

Standard Ray Tracer

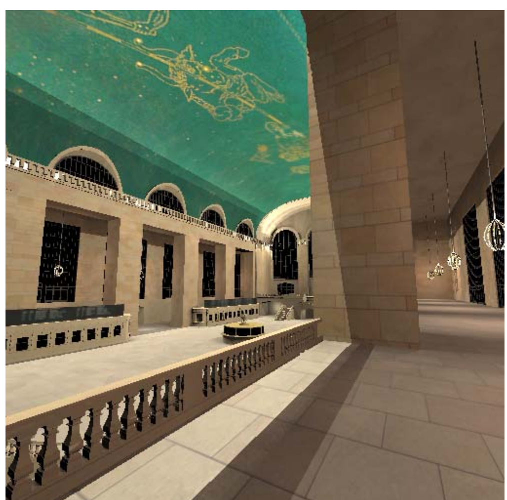


Dense Sampling



Sparse Sampling/Interpolated

Sparse Sampling/Reconstructed



| Algorithm | Number of Lights Evaluated per Pixel | Number of Shadow Rays Cast per Pixel | Total Time for Image | Speedup |
|---|---:|---:|---:|---:|
| Standard | 831 | 626 | **1694s** | **1x** |
| Dense | 118 | 118 | **612s** | **2.8x** |
| Sparse/Interpolated | 2.2 | 2.2 | **20.3s** | **83.4x** |
| Sampling | 2.2 | 2.2 | 11.7 | |
| Interpolation | 0 | 0 | 8.6s | |
| Sparse/Reconstructed | 27.7 | 6.1 | **50s** | **33.9x** |
| Sampling | 2.2 | 2.2 | 11.7s | |
| Reconstruction | 25.5 | 3.9 | 38.3s | |

Figure 5.17: Results for Grand Central Station. Note the blurring of the shadow in the sparse/interpolated algorithm, corrected by the sparse/reconstructed algorithm.

Standard

abs(Standard-Dense)

abs(Standard-Interpolated)

abs(Standard-Reconstructed)

Figure 5.18: Difference images for Grand Central Station. The dense sampling algorithm is mostly identical to the standard ray tracer. Both sampling algorithms exhibit error at specular surfaces, such as the windows, since they do not handle specularity. We also see error close to the light sources since irradiance is rapidly changing and requires high sample density. The sparse/interpolated algorithm exhibits error around the sharp shadow in the center of the image, while the sparse/reconstructed algorithm does not, as expected.

Standard Ray Tracer

Dense Sampling

Sparse Sampling/Interpolated

Sparse Sampling/Reconstructed

| Algorithm | Number of Lights Evaluated per Pixel | Number of Shadow Rays Cast per Pixel | Total Time for Image | Speedup |
|---|---|---|---|---|
| Standard | 839 | 610 | **993s** | **1x** |
| Dense | 54 | 54 | **257s** | **3.86x** |
| Sparse/Interpolated | 1 | 1 | **15.5s** | **64.1x** |
| Sampling | 1 | 1 | 4.9s | |
| Interpolation | 0 | 0 | 10.6s | |
| Sparse/Reconstructed | 39.7 | 8.4 | **64.9s** | **15.3x** |
| Sampling | 1 | 1 | 4.9s | |
| Reconstruction | 38.7 | 7.4 | 60s | |

Figure 5.19: Results for Mosque de Cordoba. The dense sampling algorithm captures gloss on the columns not seen in the sparse sampling algorithms.

Standard

abs(Standard-Dense)

abs(Standard-Interpolated)

abs(Standard-Reconstructed)

Figure 5.20: Difference images for Mosque de Cordoba. The dense sampling algorithm is almost identical to the standard ray tracer. The differences in the sparse sampling algorithms are primarily due to not handling glossy surfaces. The interpolated algorithm also exhibits error at the sharp shadows at the base of the columns.

Standard Ray Tracer

Sparse Sampling/Interpolated

Sparse Sampling/Reconstructed

| Algorithm | Number of Lights Evaluated per Pixel | Number of Shadow Rays Cast per Pixel | Total Time for Image | Speedup |
|---|---|---|---|---|
| Standard | 4608 | 976 | **7200s** | **1x** |
| Dense | 141 | 141 | **626s** | **11.5x** |
| Sparse/Interpolated | 2.7 | 2.7 | **21.9s** | **328x** |
| Sampling | 2.7 | 2.7 | 11.9s | |
| Interpolation | 0 | 0 | 10s | |
| Sparse/Reconstructed | 67.3 | 21.9 | **141s** | **51.1x** |
| Sampling | 2.7 | 2.7 | 11.9s | |
| Reconstruction | 64.6 | 19.2 | 129s | |

Figure 5.21: Results for Residential Kitchen. The sparse/reconstructed algorithm renders a smooth soft shadow below the cabinet, compared to a blurry shadow from the sparse/interpolated algorithm.

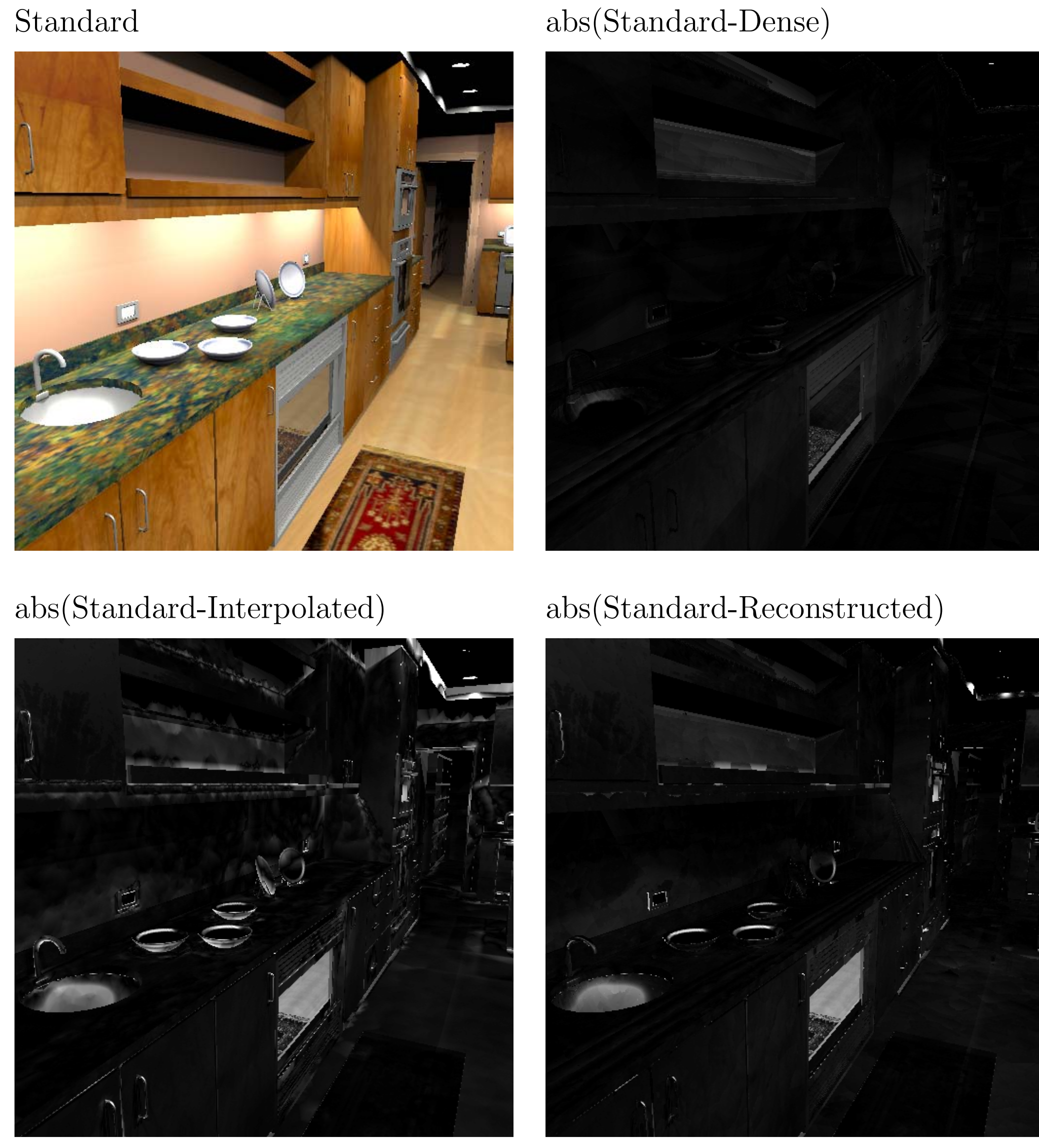



Figure 5.22: Difference images for Residential Kitchen. In the dense sampling algorithm we see some error in the darker regions, where the bound on the maximum number of nodes takes effect. The sparse sampling algorithms show errors on glossy and specular regions such as the sink and oven door. The sparse/interpolated algorithm also shows errors at shadow boundaries.

These algorithms do have weaknesses. The dense sampling algorithm primarily suffers from being too slow. Although faster than the standard ray tracer, it is not fast enough to produce images at an interactive rate. It also has difficulties dealing with very dark regions, where we are most sensitive to error. We have dealt with this problem by placing an arbitrary cut-off on the size of the tree cut, but this can sometimes lead to visible error.

The weighted sum reconstruction algorithm is comparatively fast, but introduces errors. It only deals with the direct diffuse illumination component, so scenes with specular or glossy materials will not look right. It suffers from sampling error; features smaller than the sampling density will be missed. It also blurs shadow boundaries so hard shadows, caused by small or distant lights, are rendered incorrectly. Soft shadows that harden on contact between two surfaces (an important visual cue) also cannot be rendered correctly.

The reconstruction cut algorithm can correctly capture hard shadows and soft shadows hardening on contact. However, it is somewhat slower than the weighted sum reconstruction algorithm. It also does not address the problems of sampling error nor the problem of non-diffuse surfaces.

Some of these problems have straightforward solutions. Problems in dark regions for the dense sampling algorithm could be reduced by keeping track of when the maximum possible total contribution of the remaining nodes falls below the minimum representable pixel value (instead of having an arbitrary number-of-nodes limit). Both sparse sampling algorithms could deal with specular surfaces if the sample generation implementation followed the specular path when generating sample locations.

Some of the other problems have more difficult but feasible solutions. The

reconstruction cut sparse sampling algorithm might be able to deal with some low-gloss materials by computing the upper-bound of the BRDF values for the lights in a cluster and not classifying the node as an interpolation node if the bound is too high. The weighted sum reconstruction algorithm might also be able to deal with low-gloss surfaces by storing vector irradiance at sample points. Vector irradiance represents not just the total irradiance at a point, but also the average direction the illumination is coming from.

One problem does, however, require significant research to solve. That problem is the sampling error introduced by the sparse sampling algorithms. Any illumination features that are completely missed by the samples will not be rendered. A solution to such a problem would have to involve discovering where small, visible features are, and ensuring a sample is located there.

Although these algorithms have weaknesses, they also have significant strengths. The algorithms are not sensitive to geometric complexity. They do not depend on polygonal geometry nor do they slow down significantly as the number of objects in the scene increases. They are also easily parallelizable, allowing us to increase performance by adding CPUs. Finally, and most importantly, they can render realistic scenes with hundreds of light sources.

In conclusion, we have provided the ability to move around in a realistic, complex environment at interactive rates. If a user finds a view that is interesting, they can pause and generate a high quality image in a fraction of the time that would be taken by conventional algorithms. This functionality can be used by architects, lighting designers and movie makers to work on realistic models with ease.

# Chapter 6

# Conclusion

Ray tracing is the best known technique for handling high scene complexity. It can deal with high geometric complexity because it is sub-linear in the number of geometric primitives used and because it is not restricted to triangular geometry. It can also cope with material complexity, allowing for a variety of materials to be used. However, since its computation times are linear with the number of light sources, it does not deal well with complex direct lighting situations. Architects and lighting designers have to wait for hours to obtain accurate renderings of complex scenes.

In this thesis we have introduced several algorithms for accelerating the rendering of direct lighting in ray tracers. We have described a technique for reducing the computational cost of "shadow rays" used to determine the expensive visibility component of direct lighting. We have also developed several related techniques for reducing the number of shadow rays in highly complex scenes with many lights.

We described a method for accelerating shadow rays called "Local Illumination Environments." This algorithm subdivided a scene spatially with an octree data structure. At each octree cell, we computed, through view-driven sampling, a list

of all geometry necessary to compute shadow rays from surfaces within that cell. This list was then cached at the cell and used by a conventional ray tracer to avoid the traversal through an acceleration structure for each shadow ray, as well as to avoid many unnecessary ray-geometry intersection tests. We demonstrated over an order of magnitude acceleration over conventional shadow rays in scenes with a few dozen light sources.

We also described a method for reducing the number of shadow rays cast called "Hierarchical Light Clusters". This algorithm built a hierarchy of groups of lights. It then decided which levels of the hierarchy to use at each point to be rendered based on simple perceptual metrics. We showed how to use this mechanism in a high-quality setting, where image quality is of paramount concern, and in an interactive setting, where the time it takes to generate an image is most important. This algorithm showed significant speedups over conventional ray tracing in very complex scenes with thousands of light sources.

These techniques do have limitations. The sampling-based algorithms are constrained by the fact that they miss illumination features that are not sampled. This means that small or thin shadows and lights are not portrayed in rendered images until the sampling density becomes high enough. This problem is particularly apparent in the "Local Illumination Environments" algorithm, where the cell subdivision of the environment makes features even smaller than they would normally be. In Chapter 4, we showed that this problem means that it is unreasonable to use LIEs in very complex environments. The "Hierarchical Light Clusters" sampling algorithm also has difficulty finding small features. However, the problem is greatly reduced by not relying on a spatial subdivision of the scene.

"Hierarchical Light Clustering" does have its own unique problems though. It

relies on the assumption that if a cluster makes a small enough contribution to the lighting at a point, it is reasonable to approximate it with the lighting just from its representative light. However, under certain geometric configurations of occluders, e.g. geometry precisely aligned with the representative lights, the error from multiple small clusters becomes coherent, and we get noticeable artifacts.

Another limitation of these algorithms is their computational cost. Although significantly less expensive than conventional ray tracing, these techniques require clusters of computers to yield interactive rates. However, we believe that within a few years processing power will have increased enough to run these algorithms on a single workstation.

There are several natural extensions to this research. For example, the reconstruction cut algorithm in "Hierarchical Light Clusters" could be adapted to non-diffuse surfaces. This would be done by computing bounds on the surface reflection properties, as done for the high-quality algorithm, and using these bounds to determine when interpolation among samples is appropriate.

Another possible extension is to use off-line techniques for all the sampling algorithms described. The goal would be to generate and store samples a priori instead of while the user is moving around. This would be done by establishing a fixed sampling density for the entire environment and generating samples over all surfaces. Alternatively, if a movement path is known ahead of time, samples could be focused on those parts of the environment visible from the path.

One possible route to make the "Local Illumination Environments" algorithm faster would be to take advantage of SIMD instructions available in today's PCs. Highly optimized ray tracers currently take advantage of SIMD instructions by either intersecting multiple triangles with a single ray or multiple rays with a single

triangle. We would make use of the multiple simultaneous triangle intersections to walk through the list of potential "blockers" more quickly. The algorithm is particularly well suited to this type of optimization since all the geometry we will be intersecting against for shadow testing is known as soon as we intersect the surface. This knowledge allows for prefetching of the geometry data, reducing the effects of memory latency.

Since the "Hierarchical Light Clusters" algorithm is not very sensitive to lighting complexity, it could be adapted to handle indirect illumination, not just direct illumination. This would be done by automatically generating a large appropriately distributed set of virtual point lights to mimic indirect illumination. This technique is already utilized by movie studios where lighting designers spend a great deal of time setting up small numbers of lights that mimic real illumination.

Real environments are vastly complex. Modeling and rendering scenes with realistic lighting in anything but "toy" scenes is currently a slow, difficult process. Architects and lighting designers must wait hours for renderings, manually fake the lighting to reduce complexity, or accept inaccurate renderings. It is our hope that the work in this thesis can be used to make the modeling of real-world environments more accurate and less labor-intensive.

# Bibliography


[AAM03] Ulf Assarsson and Tomas Akenine-Möller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics*, 22(3):511–520, July 2003.

[Ama84] John Amanatides. Ray tracing with cones. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18 of *Annual Conference Series*, pages 129–135, 1984.

[AMA02] Tomas Akenine-Möller and Ulf Assarsson. Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *13th Eurographics Workshop on Rendering*, pages 297–306, June 2002.

[ARHM00] M. Agrawala, R. Ramamoorthi, A. Heirich, and L. Moll. Efficient image-based methods for rendering soft shadows. In Kurt Akeley, editor, *Computer Graphics (SIGGRAPH '00 Proceedings)*, Annual Conference Series, pages 375–384, 2000.

[BDT99] Kavita Bala, Julie Dorsey, and Seth Teller. Radiance interpolants for accelerated bounded-error ray tracing. *ACM Transactions on Graphics*, 18(3):1–45, July 1999.

[Bla72] H. Richard Blackwell. Luminance difference thresholds. In *Handbook of Sensory Physiology*, volume VII/4: Visual Psychophysics, pages 78–101. Springer-Verlag, 1972.

[CCWG88] Michael Cohen, Shenchang Eric Chen, John R. Wallace, and Donald P. Greenberg. A Progressive Refinement Approach to Fast Radiosity Image Generation. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22 of *Annual Conference Series*, pages 75–84, August 1988.

[CPC84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145, 1984.

[Cro77] Franklin C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, volume 11 of *Annual Conference Series*, pages 242–248, San Jose, California, July 1977.

[DDP97] Frédo Durand, George Drettakis, and Claude Puech. The visibility skeleton: A powerful and efficient multi-purpose global visibility tool. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, Annual Conference Series, pages 89–100, Los Angeles, California, August 1997. ACM SIGGRAPH / Addison Wesley.

[FBG02] Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Local illumination environments for direct lighting acceleration. In *13th Eurographics Workshop on Rendering*, pages 7–14, June 2002.

[FFBG01] R. Fernando, S. Fernandez, K. Bala, and Donald P. Greenberg. Adaptive shadow maps. In *Computer Graphics (SIGGRAPH '01 Proceedings)*, Annual Conference Series, pages 387–390, August 2001. E. Fiume, editor.

[GH00] Reid Gershbein and Patrick M. Hanrahan. A fast relighting engine for interactive cinematic lighting design. In *Computer Graphics (SIGGRAPH '00 Proceedings)*, Annual Conference Series, pages 353–358. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, July 2000. ISBN 1-58113-208-5.

[Gla89] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, San Diego, CA, 1989.

[Gla95] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1995.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18 of *Annual Conference Series*, pages 212–222, July 1984.

[HDG99] D. Hart, P. Dutré, and D. Greenberg. Direct illumination with lazy visibility evaluation. In *Computer Graphics (SIGGRAPH '99 Proceedings)*, Annual Conference Series, pages 147–154, August 1999.

[Hec90] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24 of *Annual Conference Series*, pages 145–154, August 1990.

[HF86] D. C. Hood and Finkelstein. Volume 1: Sensory processes and perception. In *Handbook of perception and human performance*, New York, NY, 1986. John Wiley & Sons.

[HG86] E. Haines and D. Greenberg. The light buffer: A shadow-testing accelerator. *IEEE Computer Graphics & Applications*, 6(9):6–16, September 1986.

[HSA91] Pat Hanrahan, David Salzman, and Larry Aupperle. A rapid hierarchical radiosity algorithm. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25 of *Annual Conference Series*, pages 197–206, July 1991.

[HW94] E. Haines and J. Wallace. Shaft culling for efficient ray-traced radiosity. In P. Brunet and F. W. Jansen, editors, *Second Eurographics Workshop on Rendering*, New York, NY, 1994. Springer-Verlag.

[Ige99] Homan Igehy. Tracing ray differentials. In *Computer Graphics (SIGGRAPH '99 Proceedings)*, Annual Conference Series, pages 179–186, August 1999.

[JC98] Henrik Wann Jensen and Per H. Christensen. Efficient simulation of light transport in scenes with participating media using photon maps. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, Annual Conference Series, pages 311–320, Orlando, Florida, July 1998. ACM SIGGRAPH / Addison Wesley. ISBN 0-89791-999-8.

[Kaj86] James T. Kajiya. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, volume 20 of *Annual Conference Series*, pages 143–150, August 1986.

[KJ94] A. J. F. Kok and F. W. Jansen. Source selection for the direct lighting computation in global illumination. In P. Brunet and F. W. Jansen, editors, *Photorealistic Rendering in Computer Graphics*, pages 75–82, 1994.

[LTG92] Daniel Lischinski, Filippo Tampieri, and Donald P. Greenberg. Discontinuity Meshing for Accurate Radiosity. *IEEE Computer Graphics and Applications*, 12(6):25–39, November 1992.

[PPD98] E. Paquette, P. Poulin, and G. Drettakis. A light hierarchy for fast rendering of scenes with many lights. *Eurographics '98*, 17(3), September 1998.

[SG94] A. James Stewart and Sherif Ghali. Fast computation of shadow boundaries using spatial coherence and backprojections. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, volume 28 of *Annual Conference Series*, pages 231–238, 1994.

[SS98] Cyril Soler and Francois X. Sillion. Fast calculation of soft shadow textures using convolution. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, Annual Conference Series, pages 321–332, August 1998.

[SSS01] A. Scheel, M. Stamminger, and H.-P. Seidel. Thrifty final gather for radiosity. In *12th Eurographics Workshop on Rendering*, pages 1–12, June 2001.

[SSS02] A. Scheel, M. Stamminger, and H.-P. Seidel. Grid based final gather for radiosity on complex clustered scenes. In *Eurographics '02*, 2002.

[SWZ96] P. Shirley, C. Wang, and K. Zimmermann. Monte carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1), January 1996.

[War94] G. Ward. Adaptive shadow testing for ray tracing. In *Second Eurographics Workshop on Rendering*, pages 11–20, New York, 1994. Springer-Verlag.

[WBS03] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Interactive global illumination in complex and highly occluded environments. In *14th Eurographics Workshop on Rendering*, pages 74–81, June 2003.

[WCG87] John R. Wallace, Michael F. Cohen, and Donald P. Greenberg. A two-pass solution to the rendering equation: a synthesis of ray tracing and radiosity methods. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21 of *Annual Conference Series*, pages 311–320, July 1987.

[Whi80] Turner Whitted. An improved illumination model for shaded display. *CACM*, 23(6):343–349, 1980.

[WHSG97] Bruce Walter, Philip Hubbard, Peter Shirley, and Donald Greenberg. Global illumination using local linear density estimation. *ACM Transactions on Graphics*, 16(3):217–259, July 1997.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, volume 12 of *Annual Conference Series*, pages 270–274, Atlanta, Georgia, August 1978.

[WRC88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. A ray tracing solution for diffuse interreflection. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22 of *Annual Conference Series*, pages 85–92, August 1988.

[WS01] I. Wald and P. Slusallek. State of the art in interactive ray tracing. In *State of the Art Reports, Eurographics 2001*, pages 21–42. Eurographics, Manchester, United Kingdom, 2001.