Approval Sheet

Title of Thesis: **Interactive Illumination Using Large Sets of Point Lights**

Name of Candidate: **Joshua David Barczak**
Master of Science, 2006

Thesis and Abstract Approved: _____

Marc Olano
Assistant Professor
Dept. of Computer Science and Electrical Engineering

Date Approved: _____

Curriculum Vitae

Name: Joshua Barczak

Permanent Address: 14 S. Belle Grove Rd, Apt. B. Catonsville, MD 21228

Degree and Date Conferred: Master of Science, 2006

Date of Birth: July 22, 1981

Place of Birth: Baltimore, MD

Secondary Education: Perryville High School
Perryville, MD
Graduated May 1999

Collegiate Institutions Attended:

University of Maryland, Baltimore County. Master of Science. 2006
University of Maryland, Baltimore County. Bachelor of Science. 2003

Major: Computer Science

Professional Publications:

NEHAB, D. BARCZAK, J. SANDER, P. 2006. Triangle Order Optimization for Graphics Hardware Occlusion Culling. Proceedings of the SIGGRAPH 2006 Symposium on Interactive 3D Graphics and Games (to appear)

Professional Positions Held:

Research Intern ATI Research, Inc. Marlborough, MA (June-August 2005)
Research Intern. ATI Research, Inc. Santa Clara, CA (June-August 2004)

Abstract

Title of Thesis: Interactive Illumination Using Large Sets of Point Lights

Joshua Barczak, Master of Science, 2006

Thesis Directed by: Marc Olano, Assistant Professor, Computer Science and Electrical Engineering

There are a number of techniques in the field of computer graphics in which the illumination in a scene is approximated by a set of point light sources. These techniques are not well-suited to interactive applications, because evaluating the contribution of large sets of point lights can require a great deal of computation.

In this work we present an approach to rendering scenes containing large numbers of lights, using programmable graphics hardware. Our approach uses a geometric visibility determination algorithm in order to avoid performing visibility computation at the pixel level. In addition, by pre-computing and storing visibility information, and amortizing the update cost over several frames, we are able to support scenes containing moving objects.

We apply our techniques to the problems of rendering soft shadows from area light sources, and diffuse indirect illumination, and are able to render screen-sized images of scenes with several hundred lights at interactive speeds. Our optimization techniques allow us to render a significantly larger number of shadowed point lights than previous interactive systems. We present experimental results showing a significant performance improvement over a more straightforward baseline implementation.

Interactive Illumination Using Large Sets of Point Lights

By.

Joshua David Barczak

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2006

Acknowledgements

Thanks to my parents for their continued love and support, especially to my father, for proofreading this entire thing.

Thanks to my fiancée for her constant encouragement, and for not tolerating any procrastination.

Table Of Contents

List of Figures

List of Tables

1 Overview

Although the field of computer graphics has had great success in rendering plausible illumination in interactive applications, the quality of the illumination leaves much to be desired. Interactive computer graphics still struggles with the problem of reproducing global illumination effects, in which the objects in the scene influence each other's appearance by changing the amount of light that reaches the surfaces. One important example is the state of shadow rendering. Modern interactive applications can render direct shadows from a small number of point light sources, but the shadows that are produced typically have rigid, geometric edges. These types of shadows are often referred to as "hard" shadows. In real scenes, most shadows are "soft", and exhibit a gradual transition from light to dark regions (see Figure 1-1).

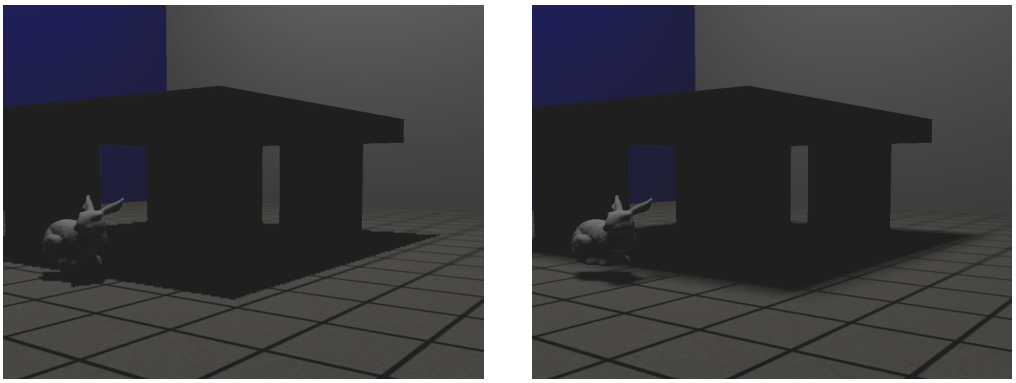

Figure 1-1 : Direct illumination with hard shadows (left) and soft shadows (right).

Indirect illumination is another effect which is important in real life, but which is typically badly approximated in interactive applications. Indirect illumination occurs when a surface is illuminated by light that is first reflected by another surface, rather than emitted directly from the source. Much of the illumination that humans observe is indirect, especially in indoor environments. The traditional (and least correct) solution to the indirect illumination problem is to add a constant term to every illumination computation to take into account the "ambient" illumination in the environment. This is often better than ignoring indirect illumination, but the results are obviously inaccurate, especially when compared to a rendering which better approximates the indirect illumination (Figure 1-2). In static environments, illumination can be pre-computed using a global illumination algorithm and used to render interactive walkthroughs (light mapping), but this method is only correct if light sources and objects remain in a fixed configuration. In scenes with moving objects, light mapping will be inaccurate. Many interactive applications, especially games, use light mapping for static geometry such as walls, and less accurate techniques for characters and moving objects.

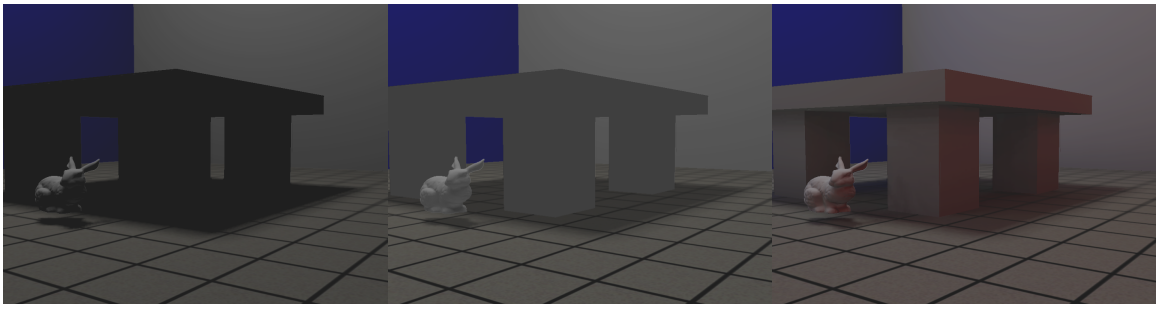

Figure 1-2 : A scene rendered with direct illumination (left), constant ambient illumination (middle), and our method (right). The red light in the right image is due to reflection from a red wall, which is not visible.

Indirect illumination computations are especially problematic if occlusion is to be considered. Although some previous work has chosen to ignore shadowing in indirect illumination, we believe that its effect is far too important to ignore, as shown in Figure 1-3.

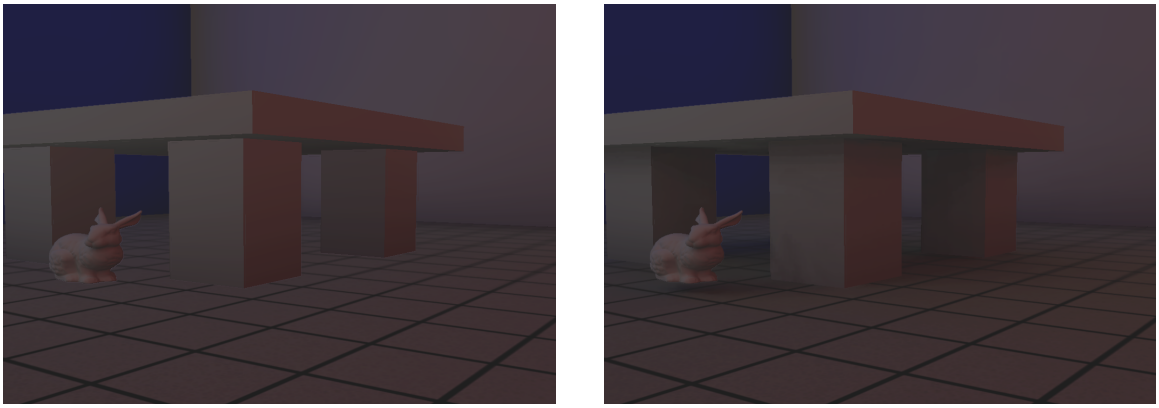

Figure 1-3 : A scene rendered with only indirect illumination. Left: Without occlusion. Right: With Occlusion.

In this master's thesis, we present a rendering system which is able to handle soft shadowed direct illumination, and indirect illumination with occlusion, while rendering at interactive rates. We accomplish this by adapting the instant radiosity algorithm [Keller 1997] to run efficiently on modern programmable graphics hardware. Our solution to the problems of indirect illumination and soft shadows is to simply apply standard direct illumination techniques on a large scale. We employ a number of optimization techniques which allow us to achieve an order of magnitude speedup over a naïve, brute force implementation for scenes with several hundred lights. We are able to support moving objects, to some extent, by carefully limiting the amount of re-computation that is performed in response to a change in the scene, and by amortizing the cost over multiple frames.

The remainder of this thesis is organized as follows: Chapter 2 describes the capabilities of programmable graphics hardware, and provides a primer on global illumination. Chapter 3 provides a review of relevant prior work. Chapter 4 describes the techniques employed in our rendering system. Chapter 5 provides an analysis of our rendering performance and image quality. Chapter 6 describes the limitations of the current system and suggests avenues for futures work.

2 Background

We begin by presenting a basic introduction to the capabilities of modern graphics hardware. We then give a brief description of global illumination, as it is understood in computer graphics. We describe the classical global illumination algorithms, and recent attempts to accelerate global illumination techniques using graphics hardware.

2.1 Graphics Hardware

In this section, we present a brief, high-level overview of the capabilities of modern graphics accelerators (GPUs). We begin by giving a high-level description of the rendering pipeline that is implemented by modern GPUs, such as the NVIDIA GeForce 5900 or 6800 series, or the ATI Radeon X800 or X1800 series. We then discuss the user-programmable portions of the pipeline in more detail, and present an overview of the programming model that is exposed by graphics APIs such as DirectX.

2.1.1 Modern GPU Architectures

Commercial graphics accelerators use specially designed hardware to produce images by rendering lines or polygons using a scanline rendering (rasterization) algorithm. The rendering pipeline that these accelerators implement can be

broken down into several stages, some of which are user-programmable. In the first stages, polygon vertices are fetched from GPU memory, and are transformed to align the polygons to a particular viewpoint and account for perspective. In addition to the transformation of vertex positions, other computations such as animation or lighting are sometimes performed at this stage, using additional information that is stored with the vertex positions. On modern graphics accelerators, there are generally a number of floating-point vector processors which perform vertex processing in a MIMD fashion. Modern accelerators also contain a dedicated cache to store the results of the computation for several previous vertices, allowing redundant computation to be skipped if vertices in a model are repeated (as they often are).

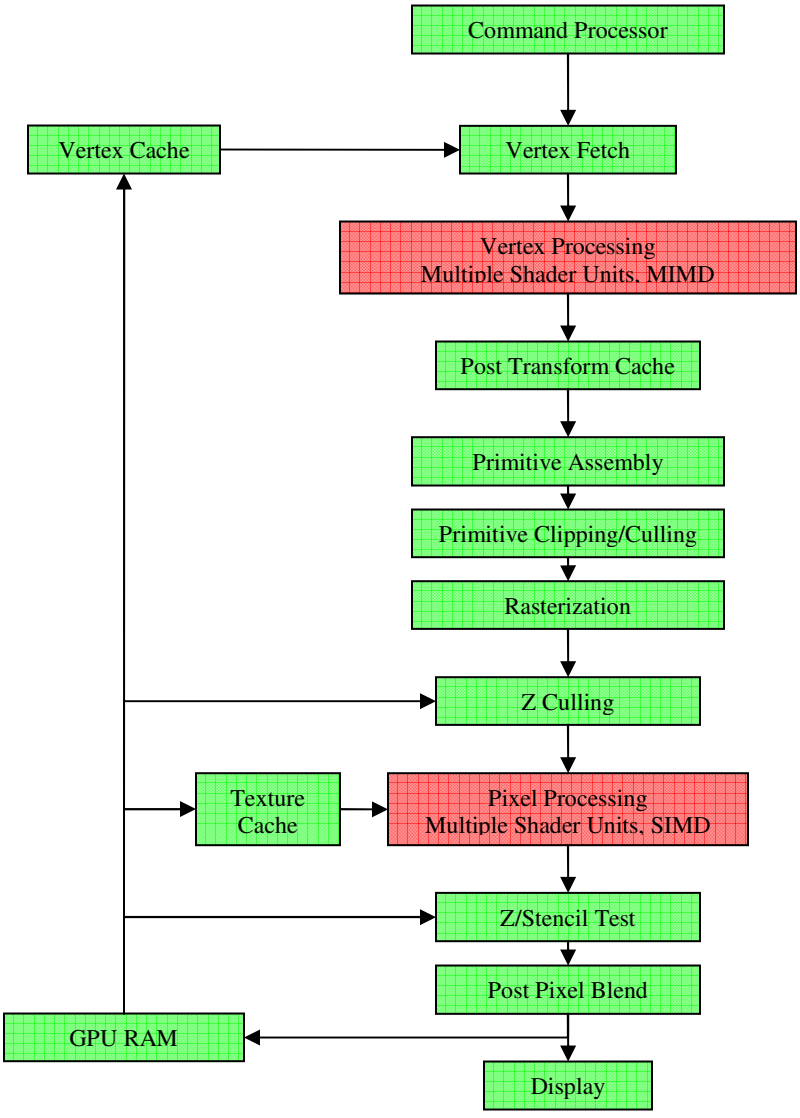

Figure 2-1: The rendering pipeline implemented by a modern GPU. Components colored green are implemented using fixed hardware. Components colored red are user-programmable.

After vertex processing, the vertices of each triangle are collected and passed to dedicated rasterization hardware, which is responsible for determining the set of pixels covered by a given primitive. In addition to positions, additional values such as colors, surface normals, and texture coordinates can be generated for each vertex by the vertex processors, and interpolated between the vertices as the pixels of the polygon are plotted. Modern hardware is typically able to interpolate up to thirty-two floating-point values, organized as eight four-component vectors, in addition to the vertex position. Prior to rasterization, polygons which are only partly visible have their invisible portions discarded, through a process called *clipping*. In addition, polygons which are completely outside the screen area, or which face away from the viewer, are detected and discarded. In addition to the fixed-function clipping and culling, modern hardware allows the user to specify user-defined clipping planes, which can be used to clip geometry in a variety of ways. We make use of these clipping planes during our shadow map rendering.

After rasterization, additional computation must be performed for each rasterized pixel in order to determine its final color. Most of the transistors in modern GPUs are dedicated to pixel processing. Pixel-processing is carried out by a set of vector processors (between 12 and 24 on current models), which operate on blocks of pixels in a SIMD fashion. The pixel processors can perform arbitrary computations using the interpolated outputs of the vertex processors, and can also access additional data by reading pixels from a set of images known as texture maps. Texture maps are blocks of image data which can be stored in a variety of formats. This functionality is typically used to map images onto a polygon in order to add fine detail to the scene. For example, a surface which represents a wall may have a brick image mapped onto it. There are a myriad of other uses for texture mapping, such as storing illumination values (light mapping) or applying detailed, small-scale displacements to a surface (bump mapping). Modern graphics hardware typically supports a variety of filtering operations on texture data in order to eliminate certain kinds of artifacts that can occur in the image, but filtering is generally not supported on all texture formats.

In addition to rasterization, graphics hardware must also perform hidden surface elimination, in order to ensure that polygons which overlap on the screen are properly occluded, regardless of the order in which they are drawn. This is accomplished by using a technique called Z-buffering. A screen-aligned buffer is used to store a depth value for each pixel, which represents the distance from the eye to the point on the nearest polygon. The depth value is a function of the Z-component of the pixel's position, relative to the camera. When a pixel is plotted, its depth value is compared to the depth value currently in the Z-buffer, and if its depth value is higher, the pixel is discarded. On modern graphics hardware, the Z test can usually be performed early, prior to pixel processing, which allows the potentially expensive pixel processing operations to be skipped for occluded pixels. Current GPUs also employ hierarchical Z buffering techniques [Green et al. 1993, Xi and Schantz 1999] which allow them to efficiently discard occluded pixels during rasterization.

In addition to the Z-buffer, modern accelerators offer an additional buffer which is known as the stencil buffer. The name is derived from the fact that the intended use of this functionality was to mask out pixels which are covered by user-interface elements, and which should not be modified when rendering a frame. The stencil buffer is essentially a single-byte counter for each pixel, which can be set to automatically increment or decrement whenever a pixel is plotted in the given location. In addition, a test can be specified for the stencil value to disable writing to pixels under certain conditions. The most common use of this functionality is for shadow rendering, using the shadow volume technique [Heidmann 1991, Everett and Kilgard 2002], but it can be used for a variety of other purposes. Stencil testing occurs immediately after Z testing, and the stencil bits are often stored in the same memory word as the depth buffer bits (using 24 bits for depth and eight for stencil).

After pixel processing, a variety of post-processing operations are available which can be used to combine the colors of the resulting pixel with the colors already in the render target. This post-processing stage is commonly referred to as the alpha-blending stage, because it is most often used to implement transparency by using an 'alpha' term to interpolate between two colors. Alpha blending can also be used to sum the results of several rendering passes, or to modulate one set of pixel values by another.

After post-processing, the final values for all pixels are written into a fixed location in GPU memory (the exact location is determined by the rasterization process). Current graphics accelerators do not have the ability to perform so-called "scatter" writes, in which a shader unit writes to an arbitrary memory location. In addition to writing to the frame buffer that is used to drive the display, modern hardware can also write pixels to a texture map, so that the results of the computation can be used in subsequent rendering operations.

2.1.2 Programmable Shaders

During the past few years, the flexibility of commercial GPUs has been significantly increased by the advent of user-programmable architectures. Modern graphics APIs such as DirectX or OpenGL expose this functionality to application developers by providing an interface to supply user-defined programs (*shaders*), written in a special-purpose language. The shader languages range from abstract instruction sets that resemble assembly language, to high level languages. The most popular shader languages, at the time of this writing, are HLSL, Cg, and the OpenGL shading language, GLSL.

Shaders can be used to control the computation that is performed by the vertex or pixel processing stages. Shading languages typically operate on scalar values, or vectors with up to four components. The parameters to a shader can be classified into *uniform* parameters, which have a constant value that the application may change between rendering

passes, and *varying* parameters, whose values depend on the vertex or pixel being processed. The numbers of uniform and varying parameters available to a shader are limited, and depend on the number of registers available in the underlying hardware. The languages offer a variety of built-in functions for mathematical operations such as dot products, cross products, vector normalization, square roots, and texture access. These operations are mapped by the GPU driver to its native instruction set. Shaders can perform any computation they desire, but are generally required to produce a minimal set of outputs, such as a vertex position for vertex shaders, or a color value for pixel shaders.

The instruction sets of current hardware suffer from a number of limitations. Older hardware has limited memory available for storing shader instructions, and imposes a limit on the length of the shader program. These limits are typically higher for vertex programs than for pixel programs, and have become less of an issue on the newest hardware. While older hardware was limited to 96 instructions in the pixel stage, the limit on the newest hardware is about sixty-five thousand. Although the shader languages support iterative or conditional constructs, these are not always usable due to the lack of flow control support. Most hardware can only support a conditional selection instruction, which can be used to implement statements of the form: X = (y == 0) ? a: b;

This instruction can be used by the shader compiler to implement arbitrary conditional statements, but this requires executing both branches of the conditional and selecting between the results. Data-dependent branching and looping is supported on the newest GPU architectures in both the vertex and pixel stages, but it did not exist on previous GPUs, and it can sometimes incur a serious performance penalty due to the SIMD nature of the underlying architecture.

2.2 Global Illumination

Computer graphics practitioners often characterize phenomena related to light reflection by placing them into one of two distinct categories. The first category, local illumination, deals with light emitted from a source being directly reflected by an object to the camera. The second category, global illumination, is concerned with the effect that every object in the scene has on every other object. Global illumination takes into account visual phenomena that are caused when one object changes the way light reflects onto another. Examples of global illumination effects include shadows, mirror reflections, indirect illumination, and caustics (the focusing of light by reflection or refraction from a curved surface). These effects are a very important factor in producing realistic images, but accurately synthesizing images containing global illumination effects is a difficult task because the interactions between objects in a scene can quickly become very complex.

The standard formulation of the global illumination problem in the computer graphics literature is to view the problem as an integral equation, for which a numerical solution is being sought. For each point visible in the scene, the amount

of light reflected from the point can be expressed as an integral over the other points in the scene. This equation was originally presented to computer graphics by James Kajiya [1986], and is referred to as "The Rendering Equation."

The rendering equation is:

$$L(x,v) = L_e(x,v) + \int_{x'} L(x',(x'-x))P(v,(x'-x))V(x,x')G(x,x')dx'$$

Where:

L(x,v) is the total light energy transferred from point x in direction v

L_e(x,v) is the energy emitted from point x in direction v (if the point is on a light source).

P(v,v') is the bi-directional reflectance distribution function (BRDF) at point x.

V(x,x') is a delta function which is equal to one if the point x' is visible from x, and zero otherwise

G(x,x') is a term which depends on the relative orientation of the two surfaces, and is defined as follows:

$$G(x,x') = \frac{(N \cdot L)(N' \cdot L')}{\|x'-x\|^2}$$

Where:

L and L' are the normalized directions from x to x' and x' to x, respectively.

N and N' are the surface normal vectors at x and x'

It is important to realize that the rendering equation is only an approximation. It ignores a number of potentially important optical phenomena, such as the scattering or absorption of light by the medium in which it travels. It also ignores effects such as polarization and diffraction, which can be important under certain circumstances. In addition, the form of the equation is such that an exact, analytical solution is not possible, and thus the methods used to solve the problem are, in effect, approximations to the approximation. In computer graphics, however, numerical inaccuracy is often overlooked if the visual results are aesthetically pleasing.

The BRDF is an important aspect of global illumination. The BRDF is a function of two directions whose value represents the fraction of incoming light from the first direction that is reflected in the second direction. The BRDF of a surface generally depends on the physical properties of the material that the surface is composed of. BRDFs in computer graphics are typically analytically derived reflectance models, but it is also possible to use tabulated representations of measured data.

Computer graphics practitioners often make a distinction between different classes of BRDFs. Specular BRDFs reflect light primarily along the mirror direction (that is, the reflection of the incident direction about the surface normal). Specular reflection is exhibited by extremely glossy surfaces such as glass, metals, or smooth, polished materials. A mirror is an example of an ideal specular reflector. Diffuse BRDFs tend to scatter light in all directions above the

surface, in a roughly uniform fashion. This type of reflection is caused by small-scale inter-reflections between microscopic bumps or cracks in the surface. Ideal diffuse surfaces are often used in computer graphics to approximate the reflectance from rough surfaces. Very few real materials actually exhibit ideal diffuse reflection, though chalk comes close.

Most real materials, (for example, layered materials such as plastic) often exhibit a combination of these behaviors, and as a result, most reflection models in computer graphics use a combination of a diffuse term and a specular term. Because these two classes of reflectance produce different visual phenomena, they are often reproduced using different algorithms. In the next section we will give a brief description of the fundamental global illumination algorithms.

3 Related work

In this section, we review some of the relevant related work. We begin by discussing traditional algorithms for computing global illumination. We then discuss shadow generation techniques, and recent attempts to compute global illumination in interactive applications.

3.1 Classical Global Illumination Algorithms

Perhaps the most famous of the global illumination techniques is ray tracing, originally developed by Turner Whitted [1980]. This technique is well suited to modeling specular transport effects such as mirror reflection. Ray tracing works by projecting a line from the focal point of the camera through each pixel in the image, and calculating the intersection of the line with each object in the scene. Once the location of the first hit point is known, effects such as mirror reflection or refraction can be handled by recursively firing another ray from the hit location and adding its weighted contribution to the final image. Shadows can also be computed in a ray tracing system by shooting rays from the hit point to the light source to determine whether any other object blocks the light.

In addition, it is possible to use ray tracing to generate a sampled reconstruction of caustics and indirect illumination by using a technique known as Monte-Carlo Path Tracing [Kajiya 1986]. The main problem with ray tracing is that the

number of ray-object intersection tests becomes extremely large as scene complexity increases. In order for rendering times to be reasonable, complex data structures are required to avoid performing unnecessary intersection tests. The text by Watt and Watt [1992] devotes several chapters to ray tracing and the various techniques which have been developed to make it practical.

A related technique is photon mapping [Jensen 2001] which works by repeatedly tracing rays from the lights into the scene, and storing a database of the points at which the rays strike the objects. The rays are reflected and refracted in random directions (chosen according to the reflectance of the objects they hit), until their energy is depleted. The database of photon hits, known as a photon map, can be thought of as a sampled representation of all of the light in the scene. The light reflected from a surface at any point can be approximated either by examining the density of photons near the point of interest, or by using a technique called 'final gathering', in which all surfaces in the scene are treated as light sources, and their contributions are sampled by firing rays through a hemisphere over the point of interest.

A fundamentally different technique is radiosity [Goral et al. 1984], which is a method for simulating the interaction of light between diffusely reflecting surfaces. Radiosity works by producing a fine discretization of the surfaces in the scene into small patches, and calculating the amount of energy that each surface patch contributes to every other surface patch (assuming diffuse BRDFs). To take into account shadowing between patches, ray tracing can be used, but a large number of rays may need to be shot for each patch in order to produce accurate results.

The computer graphics literature is overflowing with various modifications, tweaks, and enhancements to the radiosity algorithm, and it would be beyond the scope of this work for us to present a comprehensive treatment of the evolution of this technique. For a more thorough review, we refer the reader to the book by Cohen and Wallace [1993]. Some of the more notable enhancements are hierarchical radiosity [Hanrahan et al. 1991], which adaptively subdivides the scene to reduce the number of light interactions which must be computed; hemi-cube radiosity [Cohen and Greenberg 1985], which works by rendering the scene repeatedly from the point of view of each patch; and discontinuity meshing [Lischinski et al. 1992], which subdivides the scene along shadow boundaries, allowing a more accurate solution with less subdivision.

The main problems underlying all of these classical global illumination algorithms are their high computational cost, their high memory requirements due to the need for a full scene representation in memory, and the fact that they do not scale well to dynamic scenes. In the classical global illumination algorithms, the entire computation must usually be repeated from scratch in response to any scene changes.

3.2 Shadowing Algorithms

Because shadows are such an important part of a visual scene, the most common interactive global illumination algorithms are shadow generation algorithms, which are design to handle shadows caused by the occlusion of direct illumination. Although ray tracing can be used for this purpose, it is not well-suited to interactive applications. The two main shadowing techniques used in interactive applications are shadow maps and shadow volumes.

The shadow map algorithm [Williams 1978] is very straightforward, and in addition to its use in interactive applications, it is also extremely common in offline rendering for film production. The algorithm works by first rendering a depth map from the point of view of the light source. The depth map is an image where each pixel contains the distance from the rendered object to the light. When rendering the scene from an arbitrary viewpoint, the point under each pixel is transformed into the shadow map viewing space, and its depth is compared with the depth in the map. If the map's depth is lower, the point is shadowed. This algorithm is illustrated in Figure 3-1. The main drawback to this technique is that the limited resolution of the shadow map image can produce jagged shadow edges, which detract from the realism of the image. A simple filtering technique called percentage closer filtering [Reeves et al. 1997] can be used to soften the appearance of the edges, and other, more involved techniques have also appeared recently in the graphics literature [Stamminger and Drettakis 2002, Sen et al. 2003].

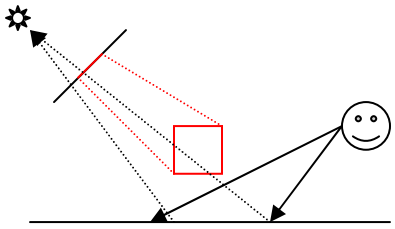

Figure 3-1 : Shadow Mapping. A depth-image, taken from the light, is used to test for occlusion. The right point is in shadow, the left point is not

The shadow volume algorithm [Crow 1977, Heidman 1991] is more involved than shadow maps, but does not suffer from the jagged edge problem. This algorithm works by first locating the silhouette edges of a polygonal object, with respect to the light, and projecting those edges to infinity, creating a polyhedral volume. Any point which lies inside this volume is shadowed by the object. The technical report by Everett and Kilgard [2002] presents a number of techniques for robust implementation of shadow volumes. A key drawback to this technique is that in a graphics hardware implementation the volumes must be rasterized in order to determine whether each pixel is inside or outside the volume. This, together with the silhouette edge detection and geometry processing, typically makes the shadow volume technique much slower than using shadow maps, but the higher shadow quality often justifies the reduced rendering performance.

Both of the algorithms described above are limited to producing what graphics practitioners call “hard shadows”. This means that the shadows have sharp, geometric edges. Most real-life shadows are “soft”, meaning that there is a gradual transition between occluded and unoccluded regions. Soft shadows occur because most light sources are not simple point or directional sources, but are in fact extended, light emitting surfaces. The only physically correct way to render soft shadows is to compute the fractional visibility of the light, with respect to the shaded point. This is generally done by point-sampling its surface and computing the fraction of samples which are visible from the point of interest. In order to produce accurate shadows, a large number of sample points are required. There has been a great deal of recent work which has attempted to produce soft shadows from area light sources in interactive applications. We discuss a few of the most well-known methods here. For a more thorough review, we refer the reader to the recent survey paper [Hasenfratz et al. 2003].

The most intuitive technique for rendering soft shadows is to take a set of samples over the light emitting surface, and compute the illumination contributed by each sample point. When visibility is taken into account, this produces an accurate shadow image, but a large number of samples are typically needed, especially if the illumination produced by the light is not uniform over its surface. Various techniques such as importance sampling [Ward 1991] have been used to improve this process, but it can still be quite expensive. In our work we show how to efficiently compute the illumination from a point-sampled area light using programmable graphics hardware.

One recent technique approximates these shadows by extruding small, flat quadrilaterals, called “Smoothies” out of the silhouette edges of the object, rendering these into a texture from the light position, similar to a standard shadow map [Chan and Durand 2003]. Any surface that is occluded by the smoothie is assigned a visibility value that is computed based on the distance between the blocker and the light, and the distance between the receiver and the light. This technique, combined with standard shadow mapping for the umbra region, produces approximate soft shadows which are visually pleasing, but physically inaccurate. This technique is very similar to another approach called “Penumbra Mapping”, which was developed independently and published at the same time [Wyman et al. 2003].

Soft shadowing techniques based on shadow volumes have also been developed [Assarson et al. 2003]. These methods augment standard shadow volumes by using additional geometry (Penumbra wedges) to enclose the penumbra region. Fractional visibility is estimated for each pixel under the wedge by projecting the corresponding silhouette edge onto the light source and estimating fractional visibility. The visibility estimation is performed in a pixel shader, and the estimated visibility is accumulated into a screen-aligned texture which stores fractional visibility. The technique produces accurate shadows for rectangular or spherical lights, under most circumstances, but the per-pixel computation required can be expensive. Large area light sources amplify this problem, because they cause the creation of larger

wedges. The original work was targeted at interactive implementation on graphics hardware, but a variant of the technique has also been used to accelerate soft-shadow computation in a ray tracer [Laine et al 2005].

3.3 Global Illumination in Interactive Applications

Interactive implementations of ray tracing have been achieved for dynamic scenes [Parker et al. 1999, Wald et al. 2001], but a complex parallel computing architecture is usually required, and the algorithms used do not scale down to a single workstation with a commercial graphics accelerator. Specialized hardware architectures have been proposed for ray tracing, [Schmitteler et al. 2003, Woop et al. 2005] but these architectures do not yet achieve the same raw rendering performance as standard rasterization-based GPUs. Commodity GPUs have been used to accelerate the implementations of many of the traditional global illumination algorithms [Purcell et al. 2002, 2003, Carr et al. 2002, Cohen and Greenberg 1985, Coombe et al. 2004], but these implementations do not scale well to dynamic scenes, and even for static scenes they do not always achieve interactive frame rates even on modern GPUs.

An early attempt to adapt radiosity to dynamic scenes used a hierarchical structure of the patches of the scene, so that pairs of patches that were affected by a change to the scene could be detected and updated [Drettakis and Sillion 1997]. The main drawback of this work is that it is based on hierarchical radiosity, and thus requires a very fine subdivision of the surfaces in the scene in order to yield high-quality results.

Selective photon tracing [Dmitriev et al. 2002] is an improved photon mapping algorithm which is designed to support interactive rendering in scenes with moving objects. In this algorithm, a fixed number of photons are used to represent the indirect illumination, and are divided into a set of groups. One photon from each group, called a pilot photon, is shot during each frame in order to detect possible scene changes. When a pilot photon detects a scene change, all other photons in the group are re-shot and their contributions to the indirect illumination are adjusted.

A more recent work by Larsen and Christensen [2004] uses techniques similar to selective photon tracing, but performs the shading using graphics hardware. Photons are traced adaptively on the CPU in response to scene changes, and a final gathering approach based on hemi-cube radiosity [Cohen and Greenberg 1985] is used to compute a coarse texture map for indirect illumination, which is then applied to each surface. Caustics are also supported by using a screen-space density estimation technique over the set of caustic photons. The main limitation of this work is that the textures used for indirect illumination must be very small, because the textures are generated by rendering the scene once for each texel. In order to achieve high frame rates the authors had to use very coarse light maps and progressively update the illumination over several frames whenever it changed. This restriction to coarse textures limits the accuracy of the indirect illumination, but their method is able to efficiently render caustics.

Apart from the classical techniques, a variety of techniques have been developed specifically for handling global illumination effects in interactive applications. These techniques are too specialized to handle the general problem of accounting for all global illumination effects in dynamic scenes, but they typically provide an acceptable solution for specific cases.

Pre-computed radiance transfer [Sloan et al. 2001] is a technique which has attracted a great deal of attention in recent years. This technique supports interactive rendering of fixed objects under changing lighting conditions, and is able to capture a number of subtle effects such as interreflection and self-shadowing, which are difficult to compute using other methods. The main problems with this method are that it requires an expensive offline pre-processing step for each object, and dynamic changes to the object's shape are difficult to support. We make use this method in our work, and will discuss it in more detail in Section 4.4.

A recent paper by researchers at the University of Central Florida [Nijasure et al. 2005] is probably one of the most effective examples of interactive indirect illumination on programmable graphics hardware. Their technique is to subdivide the volume of the scene into a coarse three dimensional grid, and compute a global illumination solution by repeatedly rendering images of the scene as viewed from each face of the grid. The images on the grid faces are then projected onto a spherical harmonic basis, and trilinear interpolation between grid points is used to compute the approximate indirect illumination for any point in the image. Because of the limited resolution of the cube grid, the algorithm does not correctly handle shadows in cases where indirect illumination is blocked by a nearby object, and it can also suffer from discontinuities in the illumination at grid boundaries.

3.4 Instant Radiosity

In this section, we introduce an indirect illumination technique called "Instant Radiosity" [Keller 1997], which our work will be heavily dependent on. This technique takes a different approach to indirect illumination, and despite its name, it is actually more closely related to monte-carlo ray tracing methods [Kajiya 1986]. Instead of subdividing the surfaces in the scene, instant radiosity begins by firing rays from the light into the scene, in the same fashion as photon mapping. The instant radiosity algorithm then treats each of the ray hit points as a point light source, which represents some of the indirect illumination in the scene. By rendering an image with shadows for each of these lights using *direct* illumination techniques; and accumulating the contribution of each image, a plausible approximation to indirect illumination can be computed. Keller's original paper used graphics hardware and the shadow volume algorithm to render these individual light images, but did not achieve very high frame rates due to the large amount of per-pixel computation that is required.

Other researchers have picked up on the idea of representing indirect illumination as a set of point sources. Udeshi and Hansen [1999] present a modified version of this technique which estimates the amount of indirect light reflected by each scene polygon into the scene. The polygons are sorted according to the amount of light they reflect back to the scene, and the point lights are placed at the centroids of the first N polygons. The main problem with their work is that their implementation does not consider occlusion, that is, it does not take into account shadows caused by scene objects occluding these indirect lights. It is also very expensive to apply the polygon sorting technique to more than one bounce of indirect light, since this computation would be at least $O(N^k)$ for N polygons and k bounces.

Wald et al. [2002] have also implemented global illumination algorithms using a modification of Instant Radiosity, but their technique uses a cluster based ray tracer [Wald et al. 2001] and is not designed to run on a single workstation with a GPU.

Another technique called reflective shadow maps [Dachsbacher and Stamminger, 2005] takes a creative approach to this problem. Reflective shadow maps are images rendered from the light's point of view, in which each pixel acts as a point light source. When rendering an image with indirect illumination, the RSM is sampled several dozen times per pixel, and the contributions of the sampled pixels of the RSM are used to approximate the indirect illumination in the scene. The main limitation of this technique is that it does not consider occlusion. In addition, the technique assumes that pixels close to a point's projection into the RSM will contribute more than pixels which are further away, and this assumption is not always accurate. Finally, the technique is limited to only a single bounce of indirect illumination.

A recently published technique called Light Cuts [Walter et al. 2005] has been used to optimize instant radiosity in a ray tracing context by performing a clustering over sets of point lights. In this method, a hierarchical clustering is performed over all of the light samples in the scene, and a search through this tree is performed for each rendered pixel to locate the smallest set of clusters that meets a certain error bound. This method can reduce the number of light samples used from several thousand to several hundred, while still maintaining good image quality. The main drawbacks to this method are the fact that the initial set of lights must be very large (on the order of tens of thousands), and the fact that the clustering must be performed for each rendered pixel, which makes implementation on graphics hardware difficult.

4 Method

In this section we describe our techniques for rendering shadowed scenes using large numbers of point lights. We begin by discussing how point lights are distributed in our scenes. We describe our method for rendering omni-directional shadow maps in Section 4.2. In Sections 4.3 and 4.4, we discuss our approach to rendering the scene geometry. In Section 4.5, we describe how to progressively update the illumination computation in response to a change in the scene configuration.

4.1 Light Sample Generation

The first step in our technique is to generate a set of point lights to sample the illumination in the scene. Our method for doing this is a modification of Keller's original instant radiosity technique [1997]. Our goal is to generate a set of oriented lights which represent the illumination transported by the surfaces in the scene. Each light sample will have a position, the normal to the surface on which it resides, and an intensity value.

For each light emitting surface in the scene, we generate two sets of point light samples. The first represents the direct illumination, and the second represents the indirect illumination. The relative sizes of these two sample sets can be adjusted according to the relative importance of direct and indirect illumination in the scene, and the total number of

lights can be increased if higher image quality is desired, or decreased to reduce rendering time and memory consumption.

The direct illumination samples are computed by selecting random positions on the surface of each emitter. The intensity of each direct illumination sample is equal to the power of the light source divided by the number of samples. Because we use a sampling of the area light source, our system implicitly computes accurate soft shadows for the direct illumination in the scene.

Our second set of lights represents the indirect illumination, and is computed by a random walk through the scene. A ray is fired from a random position on the light source in a random (cosine-weighted) direction, and a point sample is created on the first surface hit by the ray. The intensity of this sample is equal to the power of the light source scaled by the BRDF of the surface at the hit location. The process is repeated by recursively firing another random ray from the hit location until a user-specified bounce limit is reached, at which time a new ray is generated from the light source. The ray-shooting continues until the specified number of indirect light samples have been generated, at which point each sample's intensity is divided by the total number of paths that were generated.

When distributing indirect illumination samples, we make one key assumption to simplify our algorithm. We assume that the effect of the moving objects on the distribution of indirect illumination is negligible. This assumption allows us to compute the set of sample points as a pre-process, using only the static geometry of the scene. This assumption will not be valid for scenes which contain very large movers, but it is a reasonable approximation for most scenes, since the majority of the indirect light is typically reflected by large stationary surfaces such as walls. This assumption also allows us to assume that the positions and orientations of the light samples in our scenes will not change.

4.2 Shadow Mapping

We use shadow maps, rather than shadow volumes, for our visibility computations, because we believe that they are inherently more efficient. Shadow volumes require a great deal of geometry processing for each light, since the silhouette edges of the objects must be extracted in order to generate the shadow geometry. More importantly, shadow volumes can also require a great deal of pixel processing, since the volumes tend to cover a large area of the screen. Clamping and culling techniques have been developed that will reduce the fillrate requirement [Lloyd et al. 2004], but these may not scale effectively to large numbers of lights. Using shadow maps also allows our graphics hardware implementation to compute the contribution from several lights in one rendering pass, which reduces both the memory bandwidth cost and GPU driver overhead. This is not possible with shadow volumes, since the volumes for each light must be rasterized in order to isolate the shadowed pixels.

4.2.1 Shadow Map Parameterization

One of the most serious limitations of shadow maps is that they are difficult to use with point light sources. Standard shadow maps are implemented by rasterizing the scene geometry from the point of view of the light source in order to generate the map. If the field of view that is needed to cover the scene is very large, then standard shadow maps break down because graphics hardware cannot rasterize projections onto a full hemisphere (Figure 4-1). Because of this limitation, we must use a variant of shadow mapping which is able to represent the occlusion over the full hemisphere.

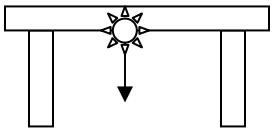

Figure 4-1 : A case where standard shadow maps break down. The shadow map must cover the entire hemisphere below the light to represent occlusion from both table legs. Graphics hardware cannot rasterize projections onto a hemisphere.

Cube mapping is a technique for representing a function over the sphere by mapping a set of images onto the faces of a cube surrounding the sphere. In order to map a direction to a cube-map texel, the largest component of the direction vector is used to determine the appropriate face, and the remaining two components can be used to access the appropriate texel. All modern GPUs contain special-purpose hardware to accelerate this mapping operation. It is possible to implement omni-directional shadow maps by rendering the shadow map into a cube texture. There are, however, two problems with doing so. The first problem is that rendering into the six faces (five in our case) requires a large number of rendering passes. The main problem, however, is that cube maps make inefficient use of shadow map memory. In order to accurately represent shadows from distant occluders, the cube map resolution must be extremely high. In our case, this problem is amplified by the fact that the bottom face of the cube map will never be needed, (since our lights only illuminate over one half of the sphere) but there is no way to prevent the graphics hardware from allocating memory for it. Standard texture maps could, in principle, be used to simulate a cube map and avoid allocating the redundant memory, but this would be a very inefficient use of the API.

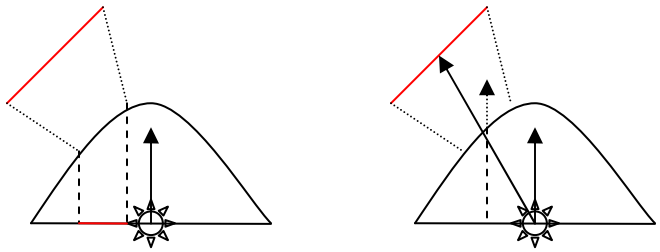

Figure 4-2 : Parabolic Shadow Mapping. Left: Projection of a line onto the shadow map. Lines in three dimensions can project to curves. Right: Accessing the map to test visibility for a particular direction.

Paraboloic mapping [Brabec et al. 2002] is another technique which can be applied to functions over a hemisphere, and is more memory-efficient than cube mapping. In this technique, the scene is projected onto the surface of a paraboloid centered at the light position (Figure 4-2**,** left), and the resulting map is stored in a standard two-dimensional texture. When testing for visibility, the pixel shader computes the point on the paraboloid that reflects the direction of interest in the direction of the light's normal, and uses this point to access the texture (Figure 4-2, right). Rendering into a paraboloid map requires only a single rendering pass with a relatively cheap vertex shader, and accessing the map during illumination computations can also be implemented very efficiently. However, because the projection of the scene onto the paraboloid is non-linear, it is not possible to generate an accurate shadow map without heavily tessellated geometry, because graphics hardware cannot accurately rasterize a non-linear projection. We have found this error to be a much more severe problem than the relevant literature indicates. The projection error is especially problematic if the shadow casting object is a large planar surface such as a wall, because the errors can result in thin slivers of light leaking through the bottom of the wall (see Figure 4-3). In this case, the artifacts cannot be eliminated without tessellating the walls to a size at which each triangle covers roughly one shadow map pixel. This level of tessellation is prohibitively expensive for our purposes.

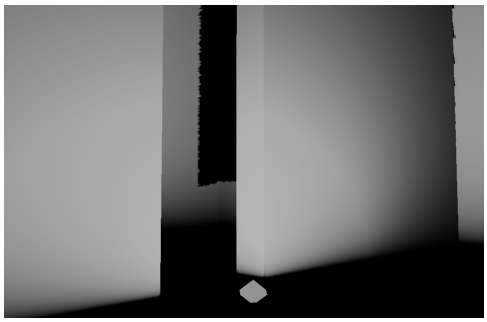

Figure 4-3 : Light leaking artifacts due to inaccurate projection in a paraboloid map. The shadow on the far wall should extend to the floor. The walls are subdivided into a 25x25 vertex grid, but the artifacts are still prominent. A tessellation level of 60x60 eliminates the artifact, but is too expensive for our purposes.

For our system, we use a technique which is combines the strengths of each of these two methods. We render our shadow maps by projecting the scene geometry onto the top four faces of an octahedron, and mapping the result onto a square as shown in Figure 4-4. Because the projections are linear, this mapping does not suffer from the artifacts that occur in parabolic mapping, and no tessellation of the scene geometry is required. This method requires up to four rendering passes for each shadow map, but this is still preferable to the five passes required for cube mapping, and the octahedral map is much more memory-efficient. Each rendering pass projects the blocker geometry onto one of the four faces of the octahedron, and user-defined clipping planes are used to constrain the rendering to the corresponding triangular slice of the render target. Computing the mapping between a point on the hemisphere and coordinates in the

map can be expensive in a pixel shader, since it involves a conditional projection onto one of four octahedron planes. Depending on the performance characteristics of the GPU, this cost could be reduced by pre-computing the texture coordinates corresponding to each direction on the hemisphere and storing the result in a cube map. As long as the cube map resolution is sufficiently high, no errors are introduced. This reduces the cost to one additional cube texture lookup for each shadow test. The memory cost of this cube map is negligible, since only one copy is needed.

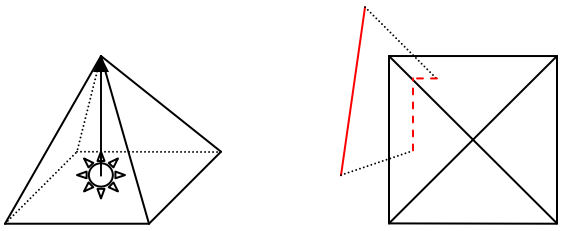

Figure 4-4 : Octahedral Shadow Mapping. Left: An octahedron centered at a light source. Right: Projection of a line onto the shadow map. Note that the projection is linear over each face, but discontinuous at the edges.

4.2.2 Shadow Map Rendering

Because of the large number of shadow maps that our system must deal with, we take great care to ensure that the shadow map rendering cost is as low as possible. We begin by taking the entire set of occluding objects and transforming the models into world space. We exclude from this set any surfaces which cannot block other surfaces (such as the outer walls of rooms). We do this transformation on the CPU, rather than on the GPU as it is normally done, because we cannot afford the overhead of specifying a different transformation matrix for each occluder in the shadow map. To do so would require changing vertex shader constants for each occluder and issuing a separate rendering command. This does not generally incur a great deal of overhead, but in our case, that small overhead is multiplied by a large constant factor. Performing the transformation on the CPU allows us to render all occluder geometry using no more than four rendering passes per shadow map.

After the transformation into world space, we perform a simple culling test for each shadow map face to determine if any of the four faces does not contain occluders. If this is the case, we are able to avoid rendering the projection onto this face, and in some scenes this produces a significant performance improvement.

Our shadow mapping vertex shader consists of two matrix multiplies. The first transforms the geometry into a light space coordinate system, (which is a rotation to align the Y axis with the light normal). The second projects the result onto the render target. User clipping planes are used during each pass to restrict the rendering to the triangular slice corresponding to the current octahedron face. Our pixel shader computes and stores the distance to the computed light-space position. This value is written out as the depth-value that is used for Z-testing.

4.3 Rendering Illuminated Scenes

Given our previously computed set of light samples, and assuming diffuse BRDFs for all surfaces (which are constant over the hemisphere), our goal at render time is to evaluate the following equation for each point x that is visible in the scene:

$$L(x) = \sum_{i \in Lights} A(x) L_i V(x, p_i) G(x, n, i)$$

Where:

$$G(x, x') = \frac{(N \cdot L)(N' \cdot L')}{\|x' - x\|^2}$$

L_i is the intensity of light i

p_i is the position of light i

n_i is the surface normal at point x

$n_{\underline{i}}$ is the normal of the surface on which light i is located

d_i is equal to $p_i - x$ (the direction from x to the light)

$V(x,y)$ is a visibility function between two points x and y

$A(x)$ is the reflectance of the surface at point x

There are several ways to optimize the above computation. The most obvious is to factor out the A(x) term, which is constant over the entire set of lights. We can implement this by rendering the diffuse albedo of the visible surfaces into a screen-aligned texture, and modulating the computed illumination by the albedo by using alpha blending. However, it is still expensive to evaluate the summation directly for each pixel. We employ a number of optimization techniques in order to reduce the cost of this computation. These techniques, taken together, can produce a significant speedup over a naïve implementation, as our results demonstrate.

The visibility function in the above equation is implemented by projecting each rendered point into the shadow map for each light. Because this must be done on a per-light, per-pixel basis, it can become prohibitively expensive. In order to reduce the cost of the visibility computation, we use a geometric algorithm to determine, for each object, which lights are potentially occluded, and which ones are definitely not. Because shadow map access is only required for partially occluded surfaces, we are able to avoid a great deal of redundant work by skipping the visibility computation for surfaces that can be shown to be free of occlusion. We use a hierarchical subdivision of the scene polygons in order to increase the effectiveness of this optimization.

Another avenue for optimization is to avoid repeated evaluation of the *G* function, since it is not view-dependent. This is accomplished by pre-computing the sum of $L_i G(x,n,i)$ into a texture map for each surface and ignoring visibility. At render-time, we take the interpolated illumination value from the texture map and subtract out the contributions of the occluded lights. This reduces our computational burden because it allows us to skip the illumination computation for the non-occluded lights.

We also simplify the computation by interpolating illumination from a low-resolution image, rather than re-computing it for each image pixel. This approximation is effective because illumination typically changes very slowly between pixels, especially when the surfaces in the scene are planar. The interpolation can produce unacceptable aliasing at object boundaries, but we can correct this aliasing by repeating the computation at full resolution for pixels that lie near an edge. Because interpolation is hundreds of times cheaper than re-computing illumination, we are able to drastically reduce the fillrate demand of our rendering system.

In addition to the algorithmic changes described above, we also structure our implementation to minimize the graphics API overhead that is entailed by repeated multi-pass rendering. When computing shadows, we process as many lights as possible in a single rendering pass, and we introduce an object clustering technique to minimize the required number of API calls.

4.3.1 Visibility Determination

As the number of lights in the scene grows, the large number of shadow map accesses per pixel quickly becomes a limiting factor. Because our rendering speed is so heavily tied to the pixel fillrate, we can obtain a performance boost by determining, for each element of the scene, which lights are potentially occluded and which lights are definitely not. This allows us to avoid shadow map accesses for unblocked lights on surfaces which obviously do not need them.

For each rectangular surface patch in the scene, we perform this visibility determination by connecting each vertex of the patch to the light-source position, forming a skewed pyramid. Any object which does not intersect this pyramid cannot possibly cast a shadow on the patch. We can therefore separate the lights into unblocked, partially blocked, and fully blocked sets by testing each blocker object for intersection with the occlusion pyramid of each light. A light without any intersections is guaranteed not to be blocked over the surface of the patch. For complex models with a large number of polygons, we implement visibility determination at the object level, by widening the occlusion pyramid until it completely encloses the bounding sphere surrounding the object. The geometry of the occlusion pyramid is illustrated in Figure 4-5.

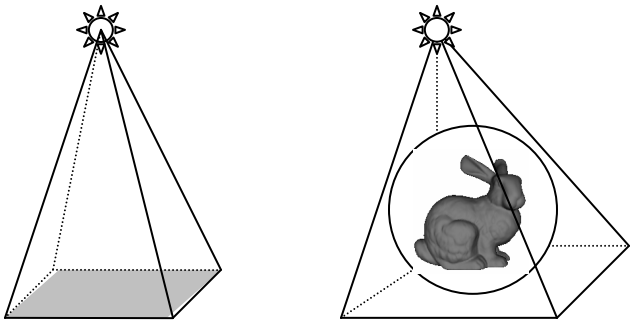

Figure 4-5 : Occlusion pyramids for planar surface patches (left) and complex objects (right).

Depending on the complexity of the blockers, we can choose to test each individual polygon against the pyramid, or to test a bounding volume such as a sphere. If the blocker contains a small number of polygons, it is advantageous to test the individual polygons, rather than the bounding volumes, because this yields more accurate visibility determination. In addition, it is possible to detect cases in which the entire object is occluded by a single large polygon. In such cases, the light can be ignored entirely when rendering the occluded object. For complex blockers, however, testing individual triangles against the pyramid can quickly become very expensive, and a bounding volume test can generally yield an adequate, conservative result. In our implementation, we make a distinction between simple surface patches like walls, which we test directly against the pyramid, and complex objects such as bunnies and teapots, for which we test only the bounding sphere against the pyramid.

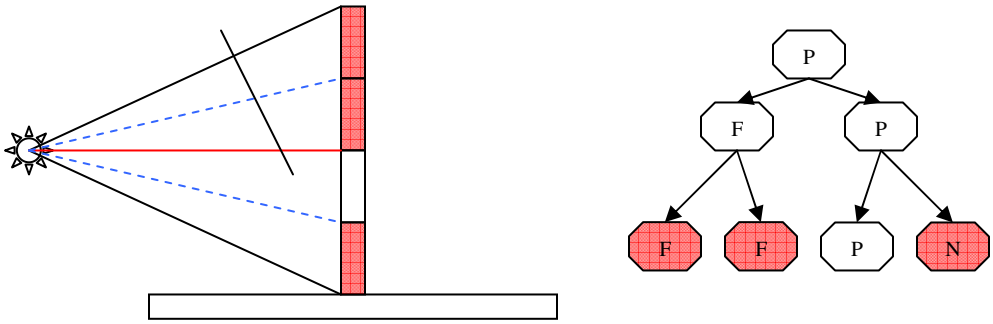

Figure 4-6 : A 2D example illustrating hierarchical visibility determination. An object is recursively split two times, and the results of occlusion testing are shown for each node of the tree (F-full occlusion, P-partial occlusion, N-no occlusion). Leaves colored red do not require shadow map access.

For very large surface patches such as walls, it is beneficial to subdivide the surface into smaller pieces. This allows shadow map accesses to be concentrated in regions of the surface where they are needed, which can reduce the rendering cost for the scene. We can make visibility determination more efficient by imposing a tree structure on the objects. For each light, we first perform the occlusion test for each of the parent patches, after first discarding any lights which are behind the patch. If a potential occluder is found for the parent patch, we recursively test the sub-patches until we reach a leaf, or until we find a sub-patch which is either fully blocked or unblocked. In this case, the

occlusion testing is stopped and the result is propagated down the tree to the leaves. Our current implementation only performs this recursive subdivision on planar surfaces such as walls or floors, but it could in principle be applied to meshes as well by recursively splitting the mesh faces into clusters. Our subdivision strategy for patches is illustrated in Figure 4-6.

In our implementation, we found that subdividing large patches up to three times could produce performance improvements, but that excessive subdivision was generally not beneficial, since the overhead of adding additional rendering passes eventually outweighs the performance gained by not accessing redundant shadow maps. The level of subdivision for each surface in the scene should be set according to the degree of occlusion that is likely to occur on that surface. In our scenes, we used a fixed subdivision level for each surface, but we manually adjusted the subdivision according to the role of the surface in the scene. For example, we typically subdivided floors more heavily than ceilings. An adaptive subdivision could also be performed at run-time, using the number of occluded lights as a subdivision metric.

For static scenes, visibility determination for each light can be performed as a pre-processing step and stored. However, if the positions of objects change, then dynamic re-computation becomes necessary. We discuss how to handle moving objects in Section 4.5.1.

4.3.2 Light Mapping

For large, planar surfaces, such as walls, the illumination does not change rapidly over the surface (shadows notwithstanding). It is therefore possible to pre-compute the contributions of each light, to avoid having to compute light intensities at each pixel. We create a light map texture for each patch in our scene. This light map stores a sampling of the accumulated light intensity for the patch, for all lights which affect the patch and which are either unoccluded or partially occluded. We found that in order to prevent interpolation artifacts at the boundaries of sub-patches with differing illumination, it was necessary to maintain a separate light map for each sub-patch.

At run time, we first render the contents of these light maps into the frame buffer, using bilinear interpolation to smooth the illumination. The result of the first pass produces an illuminated image of the scene, without taking into account occlusion from partially blocked lights. We then make a second pass over the scene and use the shadow maps to subtract the contributions of occluded lights. The results of our visibility determination algorithm are used to optimize this process by applying a shadow map to a patch only if the light in question is partially occluded over the surface of that patch. We use this approach, rather than baking the shadows into the light maps, because it prevents our shadow

accuracy from being limited by the light map resolution, and because it reduces the need for light map updates when a shadow casting object moves.

Light mapping is based on the assumption that the surface has a diffuse BRDF. For non-diffuse BRDFs, this technique is not applicable, because the reflectance from the surface is view-dependent. If our surface BRDFs used a combination of diffuse and specular terms, as is often the case in practice, we could use the light maps to store the diffuse term, and compute the specular term dynamically, which would still allow us to avoid some of the computation. It may also be possible to store a compressed representation of the view-dependent reflected radiance function, using, for example, a spherical harmonic basis. However, this would require a great deal of storage, and dynamic updates would be difficult.

4.3.3 Lowres Pass

Even with visibility determination, the per-pixel computation that is required to render the image can still be very high. Fortunately, for large numbers of lights, the illumination changes between pixels on a flat surface are smooth enough that it is possible to compute a low-resolution sampling of the image and interpolate it into the high resolution image. This is similar in spirit to the irradiance caching technique which is commonly used to accelerate monte-carlo global illumination algorithms [Ward et al. 1988, Tabelleon and Lamorlette 2004]. This technique has been used before in interactive applications with a high fillrate demand [Dachsbacher and Stamminger 2005], and has proven to be a very effective means of boosting rendering speed.

Figure 4-7 : Interpolation without edge detection. Although the illumination is smooth, the edges have an unacceptable jagged appearance.

Even though interpolation can produce a reasonable reconstruction of the illumination, it will also create jagged edges at object boundaries (Figure 4-7). This class of image artifacts is due to a phenomenon called *aliasing,* which is a common problem in computer graphics, and is caused by the limited resolution of a rendered image. To solve this problem, we render our images in two phases. We first render a low resolution image of the illumination, and then perform an edge detection operation to locate regions in which the surface orientation, and therefore, illumination,

changes rapidly. We will then perform a second rendering of the shadows at full resolution, which renders only to these boundary pixels.

We perform the edge detection by examining the texture maps which contain surface positions and normals for each pixel, and locating regions where the plane equations of the underlying surfaces change. When performing edge detection, it is necessary to mask out pixels in a wide region around the edges in order to prevent "color leaking" artifacts caused by the interpolation, as shown in Figure 4-8. We can implement this efficiently by stepping over a larger distance when examining neighboring pixels during edge detection. Although this simple approximation may problems at very thin discontinuities between surfaces, these cases are sufficiently rare in practice to justify its use. During the edge detection pass we set the depth value to zero at non-edge pixels, and one otherwise. This creates a depth mask, which we use for re-rendering the illumination at full resolution to avoid edge aliasing. Because the graphics hardware is able to quickly discard pixels which fail the depth test, we can use the result of the edge detection pass to restrict rendering to the edge regions.

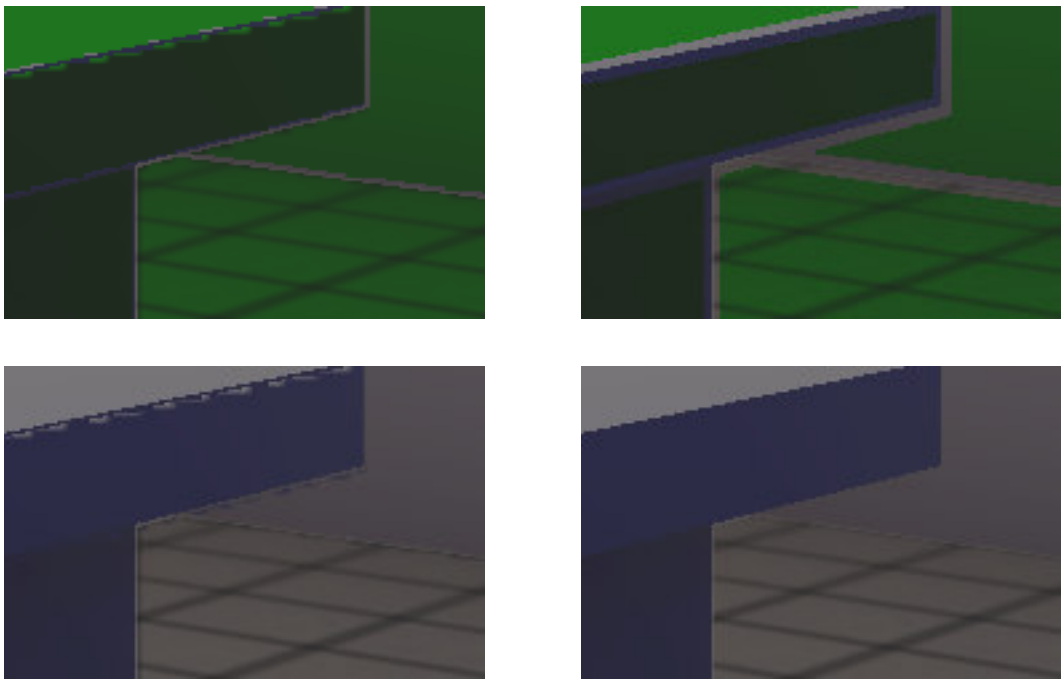

Figure 4-8 : Edge detection artifacts. Left: Stepping one pixel in each direction causes color to bleed across the edges during interpolation. Right: Stepping several pixels in each direction corrects the problem. Top images show interpolated pixels in green

4.3.4 Multi-pass Rendering implementation

Because of the limited shading resources in graphics hardware, our system must use multiple rendering passes to compute the shadows. During the shadow computation, our vertex shader performs a transformation of each vertex

from world coordinates into light-space coordinates, which are the coordinates used for shadow mapping. We do this in the vertex shader for efficiency, and also because our pixel shading hardware lacks sufficient registers to store the transformation matrices in the pixel stage. The light-space positions are passed from the vertex shader to the pixel shader as interpolants. On an ATI X800 GPU, we have enough interpolators to handle up to seven lights in a single pass.

The pixel shader uses the light-space positions to access the shadow map for each light, and computes the illumination for each pixel by looking up the world-space positions and normals from screen-aligned textures. The positions and normals are stored in textures in order to free up interpolators to allow a higher number of lights per pass.

In order to avoid quantization artifacts, we must use a floating point texture to store the accumulated results from each rendering pass. Since the X800 does not support alpha blending when rendering to floating point surfaces, we are forced to emulate it by using a double-buffered scheme. Each rendering pass reads the previous result from one texture using a texture lookup, and outputs the new pixel color to a different texture. The textures are then swapped for the next pass. Using a texture lookup to read the accumulated result can produce rendering artifacts near boundaries due to floating-point error in the computation of intermediate pass texture coordinates. If floating-point alpha blending were available, however, these artifacts would disappear, since the rasterizer would guarantee that the correct pixel addresses are always read at each pass.

A more serious problem with this double-buffered scheme is the fact that the buffers must be kept consistent between rendering passes. Consider the example presented in Figure 4-9. In this example, we have a pair of buffers, each containing two pixels. We first perform three passes which increment the value stored in the left pixel, but which do not touch the right pixel. This is analogous to rendering geometry which only covers a portion of the pixels in a render target. After the first three passes, we switch geometry and begin to increment the right pixel. Because the value in the left pixel is not touched by this operation, the value produced by the third increment pass is not carried over, and the buffers become inconsistent.

In order to maintain consistency between the buffers, it is necessary to ensure that the number of rendering passes for each piece of geometry is even. This forces us to introduce redundant rendering passes which simply copy the accumulated results from one texture to the other. This problem would also be eliminated by floating-point alpha blending.

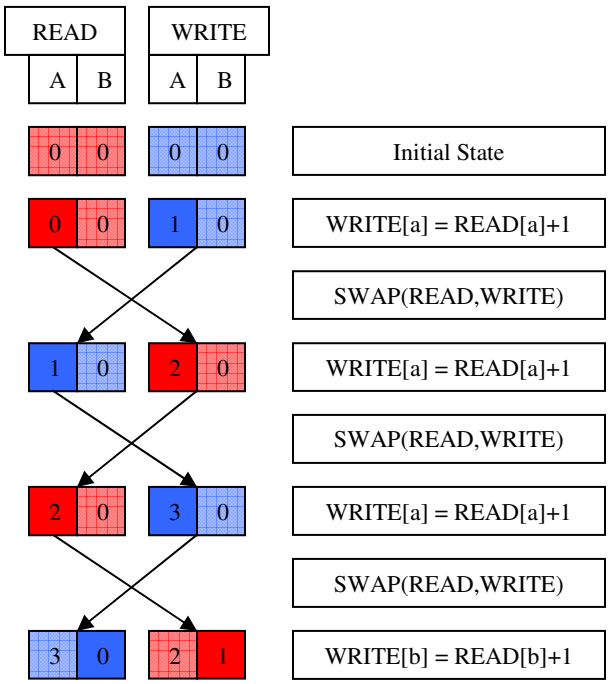

Figure 4-9 : Simple example illustrating buffer consistency problems. The red and blue boxes represent two different buffers, each containing two pixels. The left buffer is the buffer that is read during each step, the right buffer is written to. Affected pixels are hi-lighted. Note that an odd number of writes to the left pixel causes the buffers to become inconsistent.

4.3.5 Render Pass Optimization

Although we are able to process the effects of several shadow maps in a single pass, the number of rendering passes required to render the objects can become very large if the objects are rendered separately. As an example, consider a scene in which 20 objects are lit by 300 lights each. This translates to 50 rendering passes per object, for a total of 1000 passes. In addition, our low-resolution first pass requires that the objects be rendered twice, which raises the number in our example to 2000. Because each rendering pass incurs a certain amount of CPU overhead to pass the commands to the GPU driver, this can quickly become a performance problem.

Fortunately, we can considerably reduce the number of rendering passes by grouping objects together and issuing rendering passes for the groups, rather than for the individual objects. We have developed an algorithm to do this, which is able to significantly reduce the number of rendering passes. Our algorithm operates on *elements*, which for our purposes are defined as sets of objects, paired with sets of lights that illuminate the objects. Initially, we place each object into its own element, and initialize the element's light set with the set of potentially occluded lights for the object. We then optimize the set of elements by repeatedly merging elements in a greedy fashion, as long the merging results in a reduction in the number of rendering passes. When two elements *e1* and *e2* are merged, we create a new element containing the union of their object sets and the intersection of their light sets. The lights contained in the

intersection are then removed from each of the component elements' light sets. Elements which contain no lights after a merge are removed from the set. This essentially means that we will process lights common to the two objects simultaneously, rather than processing them separately for each element.

Figure 4-10 shows pseudo-code for the algorithm. After running this algorithm, we take each of the resulting elements and treat it as a single object, issuing the required set of rendering passes for each. Although the running time of this algorithm is at least $O(N^2)$ in the number of elements (depending on the number of iterations of the second loop), we have found it to be sufficiently fast in practice. In all of our test scenes, the loop runs approximately N times, which incurs only a modest per-frame overhead, and the optimization pays for itself by producing a substantial reduction in the number of rendering passes that our system must perform, as shown in Figure 4-11.

```
OptimizeElements( elems )

   pairs = GenerateAllPairs(elems, elems);
   for( each x in pairs )
      // count passes saved, discard pairs which don't' give improvements
      x.savings = CountPassesSaved(x);

      if( x.savings <= 0 )
         RemovePair(x, pairs);

   while( Size(pairs) > 0 )
      best = FindBestPair(pairs);        // merge the best pair of elements

      Element merge_element;
      merge_element.objects = Union(best.e1.objects, best.e2.objects);
      merge_element.lights = Intersection(best.e1.lights, best.e2.lights);
      best.e1.lights = Difference(best.e1.lights, merge_element.lights);
      best.e2.lights = Difference(best.e2.lights, merge_element.lights);
      AddElement(elems, merge_element);

      if( best.e1.lights == 0 )  // remove elements with no lights
         RemoveElement(elems, best.e1);
      if( best.e2.lights == 0 )
         RemoveElement(elems, best.e2);

      // generate new pairs involving the merge element
      pairs = Union( pairs, GenerateAllPairs( merge_element, elems ) );

      // score all new pairs,
      // rescore all pairs involving one of the merged elements
      for( each x in pairs )
         if( best.e1 or best.e2 in x || merge_element in x )
            x.savings = CountPassesSaved(x);
            if( x.savings <= 0 )
               RemovePair(x, pairs);
```

Figure 4-10 : Psuedo-code for the render-pass optimization algorithm

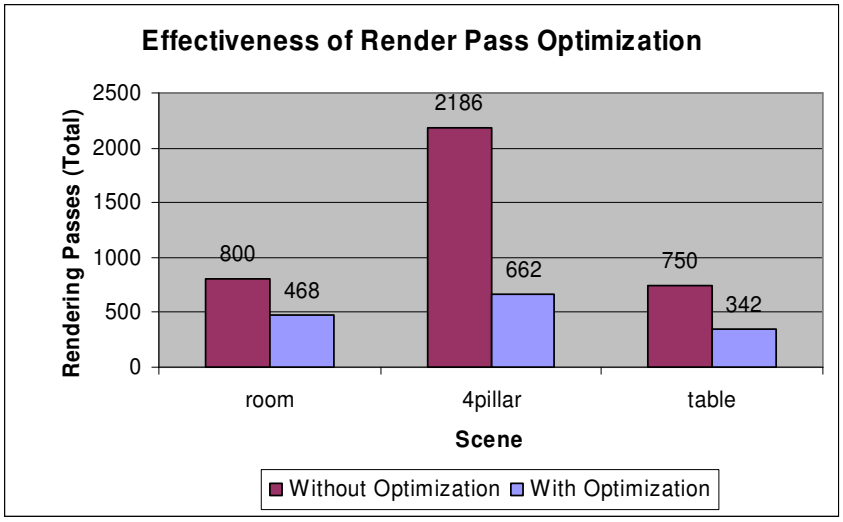

Figure 4-11 : Effectiveness of render pass optimization for selected views of several scenes.

This optimization is most effective in situations which are strongly CPU-limited, such as the one shown in Figure 4-12. In such cases, we have seen performance improve by almost thirty percent, despite the poor asymptotic performance of the algorithm, and despite the fact that our implementation is not very well optimized. However, in most cases our rendering is fillrate limited, and the optimization does not show a significant benefit. As GPU power increases, however, the render pass optimization could become increasingly important. Our current implementation performs the full optimization every frame, but it may be possible to develop an alternative algorithm which incrementally updates the solution, which would further increase the benefit.

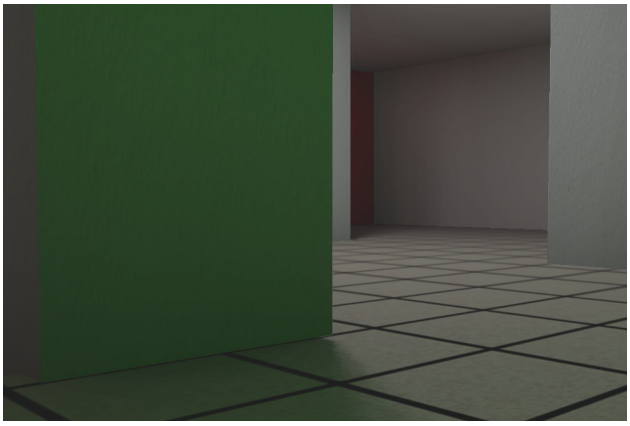

Figure 4-12 : A case in which pass optimization does well. A large portion of the screen is covered by the pillar in the foreground, which does not need shadow map access. Performance goes from 18fps (1563 passes) to 24fps (447 passes)

4.4 Rendering Complex Meshes

The techniques described above are effective for rendering large polygonal surfaces such as walls, but they have some problems when used for general polygonal objects. One key problem is that not all such objects admit a global parameterization, which means that applying a light-map texture to the object can be difficult. These objects are also problematic because they can easily contain thousands of triangles, which results in a heavy vertex processing cost when rendering the objects dozens of times, as our method must do. Another problem with these objects is the fact that their surface normals vary continuously, which forces us to render them at full resolution, instead of using our low-resolution optimization (see Section 4.3.3). The main problem, however, is that our method does not take into account self-shadowing in these complicated objects. A proper treatment of self shadowing would negate the effect of our visibility determination and force us to access every shadow map for each pixel of the moving object. This becomes very expensive in situations where the user is looking very closely at such an object and it covers a large area of the screen.

4.4.1 Precomputed Radiance Transfer

In order to solve this problem, we render the illumination for complex meshes by using pre-computed radiance transfer (PRT) [Sloan et al. 2002]. PRT works by expressing the diffuse light transport for an object as a function of the incident illumination. This preprocess can take into account a variety of intra-object global illumination effects; such as diffuse inter-reflection, sub-surface scattering, and self-shadowing. An offline simulator is used to sample the light transport function at each vertex of the object, and the function is represented by expressing it in terms of some set of basis functions over the sphere. Spherical harmonics are the most common basis used for this purpose. At run time, the object's lighting environment is computed and projected onto the same basis as the transfer function, and the shading computation is reduced to a dot product between vectors of basis function coefficients.

In our system, we use 5^{th} order spherical harmonics as our basis, which requires 25 coefficients per color channel to represent the lighting environment, and 25 more coefficients per object vertex to represent the transfer function, which in our system is not wavelength dependent (we assume white objects). CPCA compression of the PRT coefficients can be used to remove this restriction [Sloan et al. 2003], but we have not implemented this. In our examples, we have used the PRT sample application provided with the Microsoft DirectX SDK [Microsoft 2005] to perform the preprocessing on our objects.

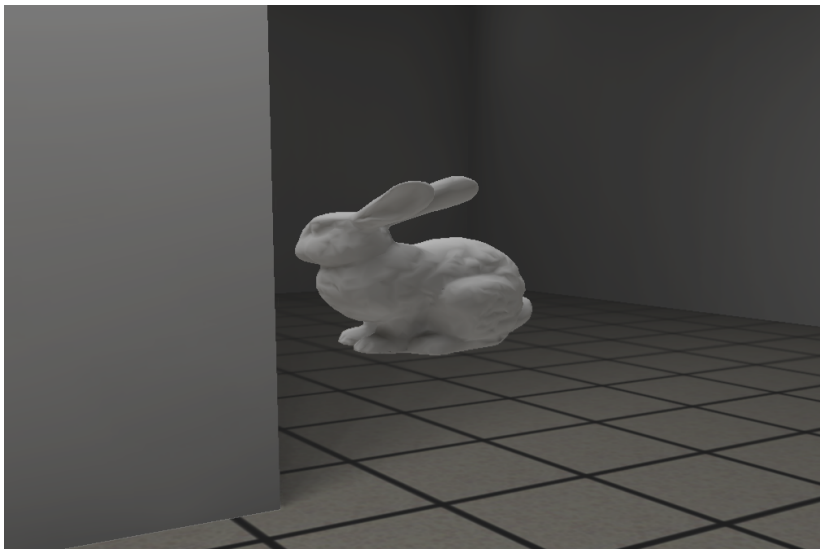

Figure 4-13 : Bunny rendered using Precomputed Radiance Transfer. Note the detailed self-shadowing behind the head and above the feet.

Given our set of point light sources, we approximate the lighting environment by computing the radiance field at the center of each object's bounding sphere. For each light sample, we compute its intensity as:

$$L_i = V(P_i, C)\frac{(\overline{D}_i \cdot N_i)}{\| D_i \|^2}$$

Where:

P_i is the position of light i

C is the sphere center

D_i is the direction from the sphere center to the light ($P_i - C$)

N_i is the surface normal associated with the light

$V(x,y)$ is a visibility function

We use ray tracing during this process to evaluate the visibility function. Although ray tracing is generally very costly, its limited use in this context does not pose a performance problem, since the number of rays cast is quite small, and the results of visibility determination for the mover can be used to avoid casting unnecessary rays.

We compute a spherical harmonic projection for each sample by treating the sample as a directional light with direction D_i and intensity L_i. Our implementation uses the utility functions provided in the D3DX utility library [Microsoft 2005] to perform this projection. After computing the spherical harmonic projection for each light sample, we sum the coefficients to produce the spherical harmonic approximation to the full lighting environment. Our vertex shader then uses this projected radiance field, along with a set of per-vertex transfer coefficients, to compute the exitant radiance, and passes the result to the pixel shader, which simply interpolates it into the frame buffer.

4.4.2 Limitations of PRT

Although PRT is a very powerful technique for rendering illumination on complex objects, the method has several limitations. The main limitation is that it is restricted to static objects under rigid body transformations (scaling, translation, and rotation). It is difficult to generalize PRT to moving objects such as animated characters, though there have been recent related techniques [Sloan et al. 2005, Zhou et al. 2005] which attempt to overcome this limitation.

Another key limitation is that projecting the lighting environment onto a spherical basis is an approximation which assumes that the lighting environment is distant. This approximation breaks down if the relative orientation of a light source can change rapidly over the surface of the object (as is the case with a point light source), because the approximation of the lighting environment is valid only at the center of the object. The spherical harmonic approximation is also inaccurate if there are high frequency changes in the illumination, such as sharp shadow boundaries. In this case, the number of coefficients in the spherical harmonic basis is not sufficient to accurately represent the change in illumination. Other basis functions, such as wavelets, are better suited to such situations [Ng et al. 2003].

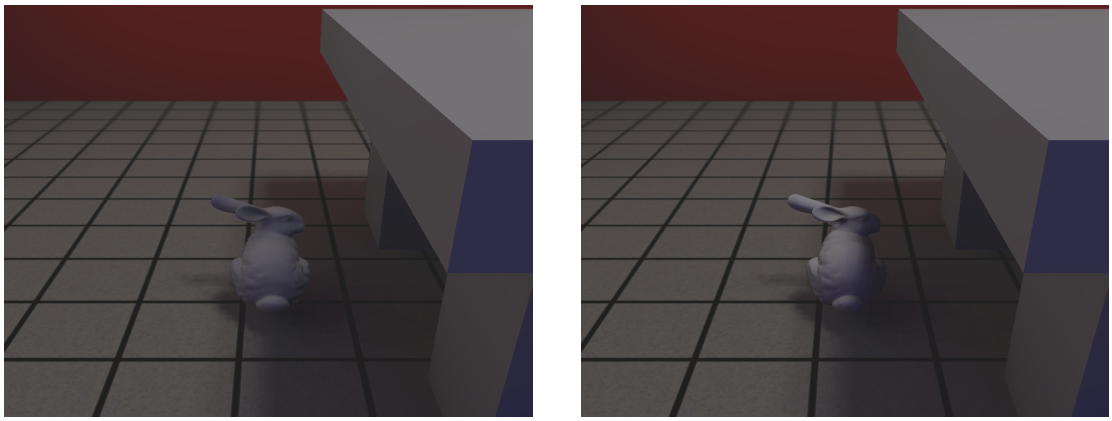

Figure 4-14 : Shadow accuracy with PRT. Left image: Result of using PRT. Right image: Brute-force shadow computation. Note that PRT fails to produce an accurate shadow boundary on the object

The main visual error that is observable in our system is that shadows cast on objects are not well-represented (see Figure 4-14). This is because we sample the incident illumination at only one location for the entire model, and this produces a poor reconstruction of the shadow boundary. It may be possible to use a grid structure to sample the radiance field around the object in a way that is able to better approximate shadow boundaries. This type of sampling has been used in previous work [Greger et al. 1998, Nijasure et al. 2005, Oat 2005]. This would improve the shadow quality to some extent, but a dense sampling would be required to reach a reasonable level of accuracy, and it would be impractical to compute such a sampling on the CPU. It may be possible to use the GPU for this computation, using the

shadow maps instead of ray tracing to determine visibility. Efficiently performing this computation on the GPU would be a useful direction for future work.

4.5 Handling Dynamic Objects

One of the advantages of our method is that most of the data we compute, such as visibility determination results, light maps, and shadow maps, can be computed once and then re-used to render subsequent frames. However, if objects in our scene are able to move, then it becomes necessary to update this stored information. In this section, we discuss how to efficiently perform these updates when objects in the scene move.

4.5.1 Updates to Visibility Information

Recall from Section 4.3.1 that we use a hierarchical visibility algorithm to avoid unnecessary shadow map accesses. When an object in the scene moves, it becomes necessary to update the visibility information in the scene. Rather than re-compute this information each frame, we maintain a database of visibility results, and update only the information that may be affected by a particular change in the scene configuration. Updating only the relevant information greatly decreases the update cost.

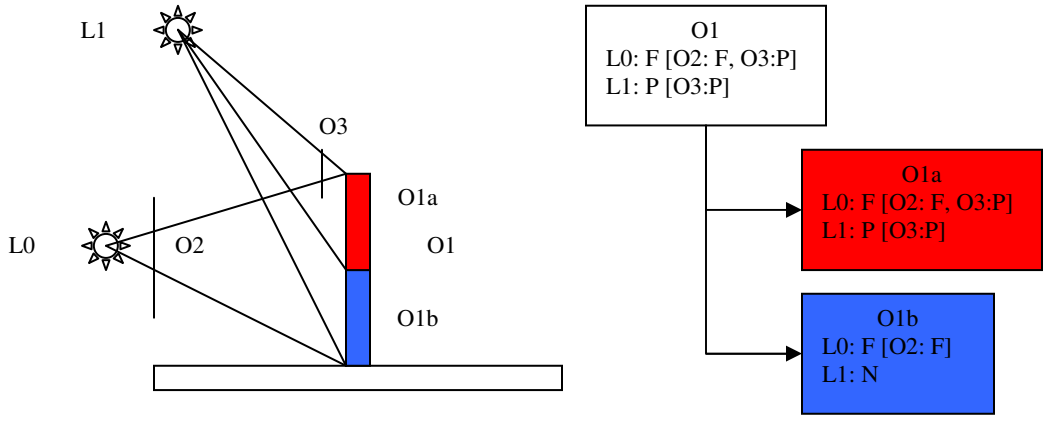

Figure 4-15 : Example to illustrate the visibility database. Object O1 is split into sub-objects O1a and O1b. Visibility information is maintained for the object as a whole, and also for each child individually. Each light is fully occluded (F), partially occluded (P), or not occluded (N) over a given node. A set of blockers for each node is also maintained.

We compute the initial visibility database as a pre-processing step For each object in the scene, we store the occlusion state of each light (fully occluded, partially occluded, or un-occluded). We also store the set of blockers which occlude each light, and we store a flag indicating whether each blocker fully or partially occludes the object. For objects which have been subdivided, the information is stored in a tree structure and light state is maintained separately for each node

of the tree. For example, if a node is fully occluded and its parent is partially occluded by the same blocker, the blocker is stored in both nodes, once as a partial occluder and once as a full occluder.

When an object moves, we first re-compute all of the visibility information for this object in its new position, using the hierarchical occlusion test described in Section 4.3.1. We exclude from this process lights which do not face the object in either its old or new positions. These lights are treated as though they are fully occluded.

After updating the visibility information for the changed object, we must also determine what effect, if any, this change had on the other objects. When performing this computation, it is only necessary to consider the occlusion that is caused by the moving object. Occlusion tests for other blockers are not necessary. For each node in the scene, we compute the new occlusion state of the mover, and update the information accordingly. There are several cases to consider here:

- If a node was fully occluded by the mover and it is still fully occluded, or if it was un-occluded by it and remains un-occluded by it, then no changes are necessary.
- If a node was partially occluded by the mover, and the mover still partially occludes it, then the status of that node is unchanged, but the status of the child nodes may have changed to un-occluded or fully occluded. Therefore, we must perform recursive updates on the child nodes. Recursion terminates when the traversal reaches a leaf node, or an un-occluded or fully-occluded inner node.
- If a node that was un-occluded or fully occluded by the mover becomes partially occluded by it, then we modify the node's blocker list, and then perform recursive updates on the child nodes.
- If any node becomes fully occluded by the mover, and it was not previously, we add the mover to the blocker list (or change its status if it was partially occluding before), and propagate the change down the hierarchy. Additional occlusion tests for the child nodes are not necessary in this case and can be skipped.
- If any node was previously occluded by the mover (fully or partially), and it becomes un-occluded by it, we remove the mover from the blocker list for that object and infer a new occlusion state from the states of the remaining blockers at that node. We also propagate the change down the object hierarchy. Additional occlusion tests are, again, unnecessary.

These visibility determination cases are summarized in Table 4-1.

	NEW BLOCKER STATE		
OLD BLOCKER STATE	*Full*	*Partial*	*None*
Full	No change	Recursive Update	Change state Propagate to children
Partial	Change state Propagate to children	Recursive Update	Change state Propagate to children
None	Change state Propagate to children	Recursive Update	No change

Table 4-1 : Visibility update cases

4.5.2 Light maps

Recall from Section 4.3.2 that we compute and store a light map for each surface in the scene. These light maps store the total illumination from each light in the scene that is unoccluded or partially occluded over the surface, without taking into account the visibility term. The contributions of fully occluded lights are not stored in the light maps. In order to facilitate efficient light map updates, we store, for each light map, a list of each of the lights which contribute to it. During the update of the visibility data, we also detect any necessary changes to the light maps. If a light that was visible before the change becomes fully occluded, we remove the contribution of that light from the light map. If a light that was fully occluded becomes visible, we add its contribution to the light map. Changes from un-occluded to partial and from partial to un-occluded, which are the most common in scenes with complex movers, do not require light map updates.

4.5.3 Shadow Maps

Updating shadow maps is the most difficult problem for our system to contend with. Shadow map updates are extremely expensive, due to the large number of rendering passes involved. When an occluding object is moved, we determine which shadow maps in the scene are affected by the change. The affected shadow maps are those for which the blocker is visible from the light's point of view. The results of visibility determination are used to avoid updating shadow maps in which the moving object is fully occluded.

Even though visibility determination can be used to reduce the number of shadow map updates, the number of updates required can still become quite large, which leads to an unacceptably low frame rate. We have found that we can increase the frame rate, at some cost in visual accuracy, by amortizing the cost of the update over multiple frames. We do this by using a simple importance metric to assign an update priority to each shadow map, and updating a fixed number during each frame, in priority order.

Our importance metric is the intensity of the brightest color channel of the light, multiplied by the number of frames that the map has been out of date. This metric ensures that extremely bright lights, which are likely to cast more prominent shadows, are updated very quickly, and also guarantees that no shadow maps will starve.

The number of lights to update per frame must be chosen carefully. If this number is too small, then shadows will appear in incorrect places in the image. If it is too large, performance will suffer. In general, we have observed that updating all of the lights over the course of approximately two or three frames does not result in objectionable artifacts, even when our system is running at low frame rates (Figure 4-16). In our test scenes, we typically used 300 light samples, which translates to between 100 and 150 updates per frame. More sophisticated importance metrics, which take into account the visual importance of the scene surfaces, might be able to achieve even better results.

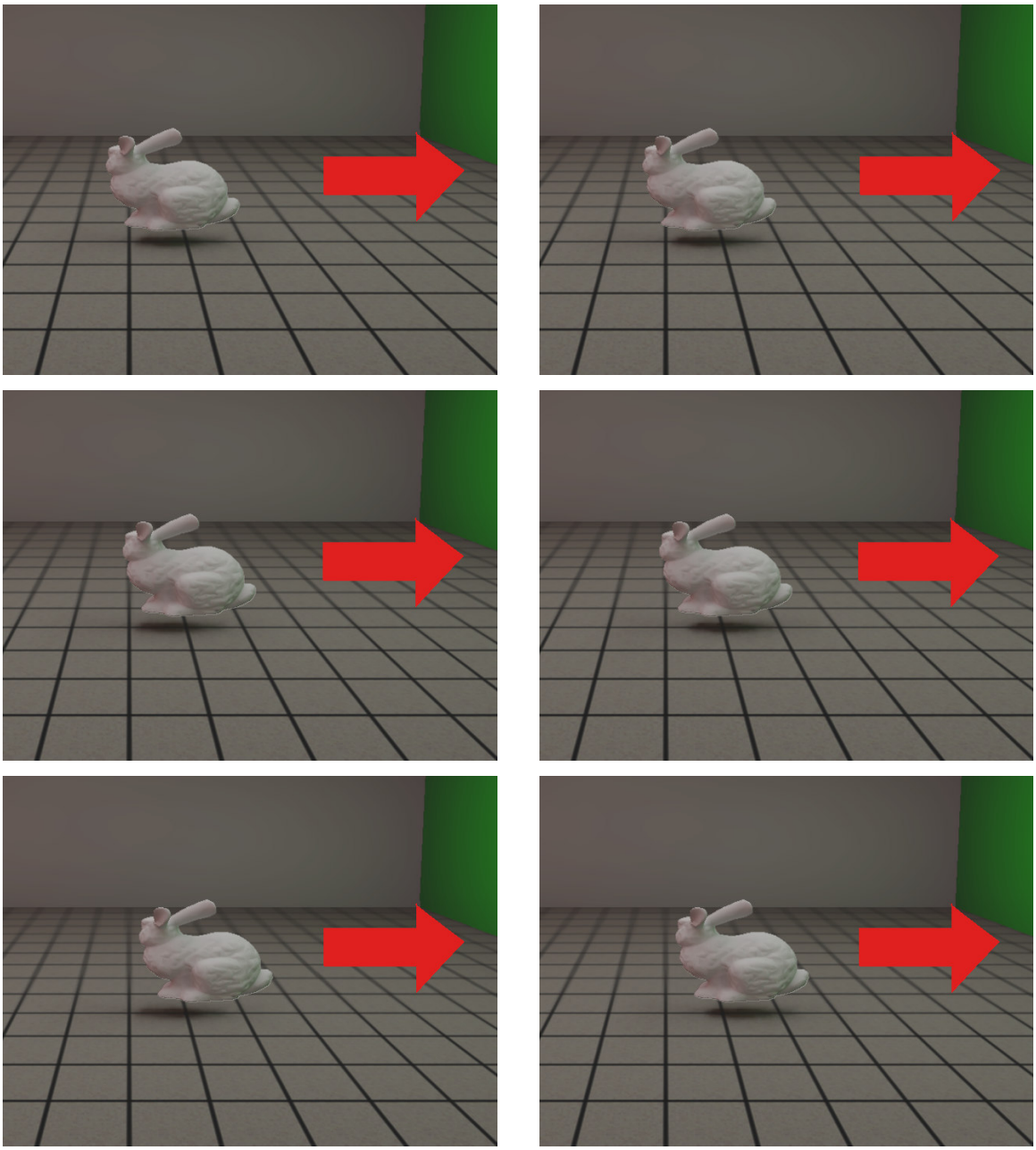

Figure 4-16 : Visibility updates with moving objects. The scene contains 300 lights. Left: 50 updates per frame. Right: 100 updates per frame.

5 Results

In this section, we analyze the rendering speed and image quality produced by our rendering system. Our test application was implemented in C++ using the Direct3D 9.0 API. All performance analysis was conducted on a PC with a 3.2GHz Pentium 4 CPU, and an ATI Radeon X800 XT GPU. All performance test images were rendered in full-screen mode at a resolution of 1024x768 pixels.

5.1 Static Scene Performance

In this section, we discuss the rendering performance of our system for static scenes. We show the effect of each of our optimization strategies on the rendering performance, and examine how performance scales with increasing numbers of light samples. For our analysis, we consider the performance from a number of static viewpoints, shown in Figure 5-1.

We begin by examining our system performance as a function of the number of illumination samples used. We measured the rendering performance of our system for each of the scenes in Figure 5-1. The results (Figure 5-2) indicate that our system performance scales linearly with the number of lights. We achieve interactive frame rates for up to 300 lights in each case, even in the more complex pillar scene.

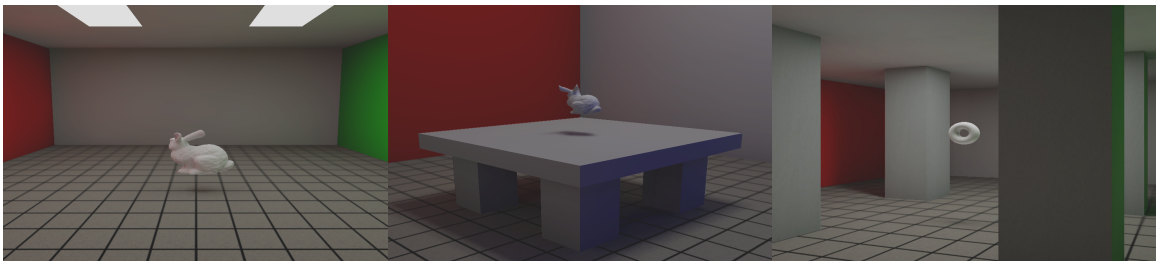

Figure 5-1 : Scenes used for performance testing. From left to right: A room with large area lights, a room with a table, and a room with four pillars

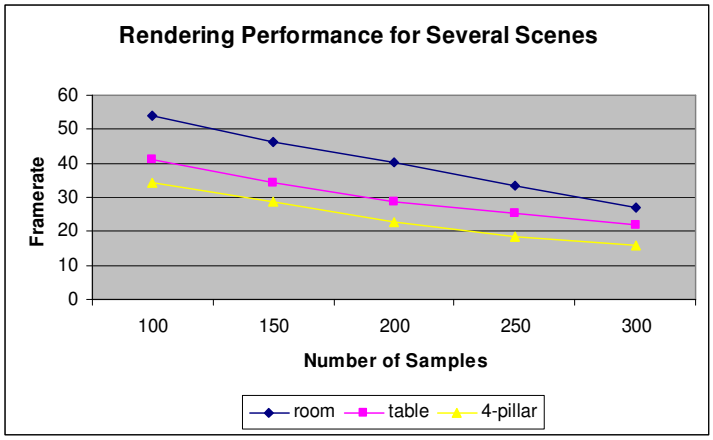

Figure 5-2 : Rendering performance for fixed views of each test scene.

Next, we consider the effect of each optimization technique on the rendering performance of our system. For this experiment, we used the area light room scene (Figure 5-1, left). This scene contains two large area light sources. Large area lights cast very subtle, soft shadows, and tend to require a much denser sampling to produce accurate shadows than smaller area lights would. This test allows us to compare our results against other shadowing algorithms for direct illumination. The results of this experiment are presented in Figure 5-3. The results indicate that our rendering system scales much more effectively than simple brute-force rendering, due to the various optimizations that we employ. Visibility determination alone is effective in reducing the time required to render the scene, even in the case of only two light samples. Our low-resolution first pass is not helpful until the number of samples reaches a certain threshold, because of the overhead associated with edge detection and the additional rendering passes. Once this threshold is crossed, however, the performance improvements begin to increase substantially. Our pass optimization algorithm is not very effective in this scene, for the reasons previously discussed (see section 4.3.5). The performance for indirect illumination and direct illumination is similar, given equivalent numbers of samples, but with indirect illumination, the occlusion is much less coherent, and more pixels will require shadow map access, which results in lower performance than direct illumination.

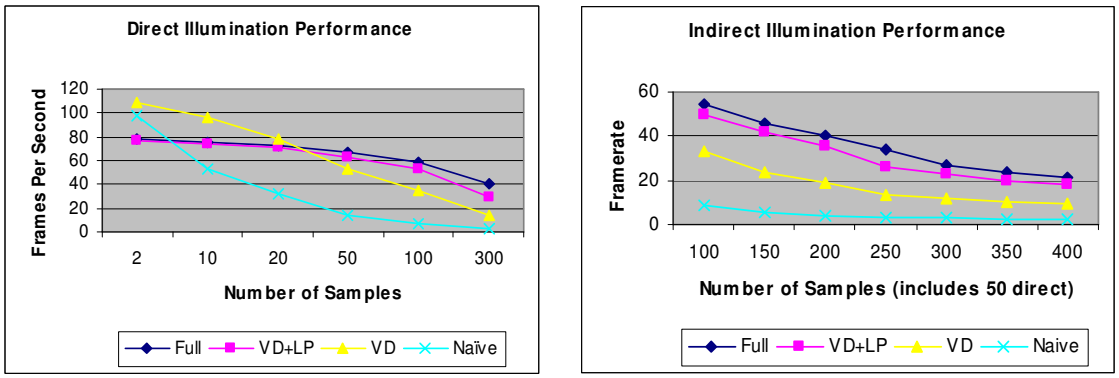

Figure 5-3 : Effect of each optimization on rendering performance. In this figure, *Full* indicates all optimizations. *VD+LP* indicates visibility determination and the low-resolution pass. *VD* indicates only visibility determination (rendering at full resolution), and *Naïve* indicates brute-force rendering

5.2 Dynamic Scene Performance

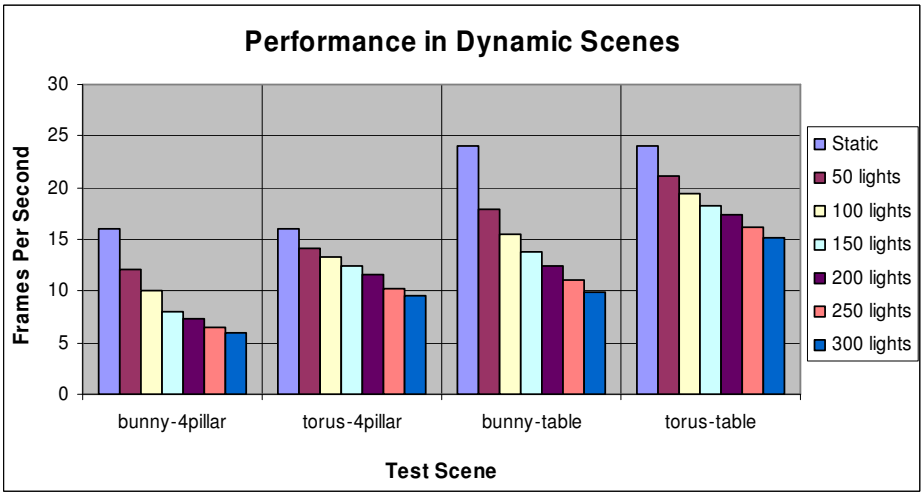

Figure 5-4 : Rendering performance in dynamic scenes, as a function of the number of visibility updates per frame. Performance for a static view is given for reference. All tests used a total of 300 light samples.

Figure 5-4 shows the degradation in rendering performance as a function of the number of lights updated per frame, for several test cases. These tests were conducted by continually moving the object over a fixed path, and averaging the results over 200 frames (for a fixed camera position). Although moving objects incur a performance penalty, our system is still able to achieve interactive frame rates, even when updating hundreds of lights per frame. The frame rates for the table scene are higher than for the four pillar scene, because many of the shadow maps in this scene only require rendering to one or two of the four octahedron faces, and our system is able to avoid some of the rendering work. The test models used were a bunny model with 3000 vertices, and a torus model with 820 vertices. In each case, the performance penalty is linear in the number of lights.

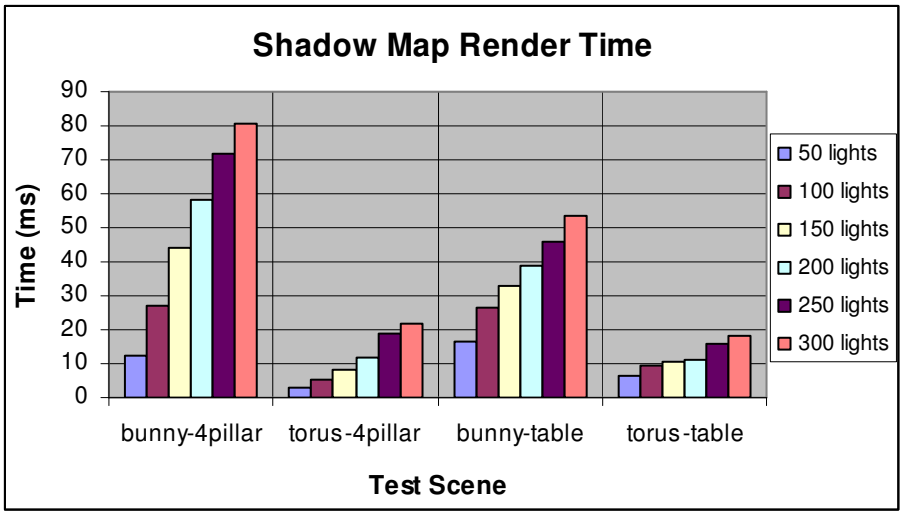

Figure 5-5 : Shadow map rendering times for each test case. Note that rendering time correlates well with the number of vertices in the moving occluder.

Figure 5-5 shows the time spent on rendering shadow maps in each test case. Note that although there is a linear increase, the performance degrades much faster for the bunny model. Note also that ratio of bunny to torus performance is approximately equal to their relative vertex counts. This indicates that vertex processing is the limiting factor in our shadow map rendering. This result is not surprising, because each vertex must be transformed up to four times for each shadow map update, and the vertex transform rate of most current GPUs is not as high as the pixel fillrate.

Figure 5-6 shows the fraction of rendering time occupied by shadow map updates and visibility determination for the bunny-4pillar test case. The results show that the cost of shadow map rendering gradually begins to dominate the frame time as the number of updates per frame increases. The cost of visibility determination also increases, but since there is only one moving object, it is relatively low compared to the shadow map cost. It should be noted, however, that in more elaborate scenes with many moving objects, visibility determination could quickly become the bottleneck.

The performance characteristics of our implementation make a strong case for a unified shading architecture in future GPUs, in which the allocation of functional units to vertex or pixel shading tasks can be dynamically adjusted according to the current workload. During shadow map rendering, the vertex units of our GPU are saturated, while the pixel units are under-utilized. During illumination rendering, the reverse is true. If the same number of functional units were available in a dynamically scheduled configuration, we would be able to achieve much higher rendering speeds, because we would make much more efficient use of the hardware resources. Previous examples of reconfigurable architectures exist in the graphics literature. The PixelFlow shading system [Eyles et al. 1997] used reconfigurable processing elements, but the configuration had to be specified ahead of time. More recent work [Chen et al. 2005] has

investigated graphics pipeline implementations which can be dynamically configured at run-time. A commercial architecture with dynamic load balancing has already been prototyped by the ATI Xenos GPU [Baumann 2005], and this type of system may become more widely available in the next few years.

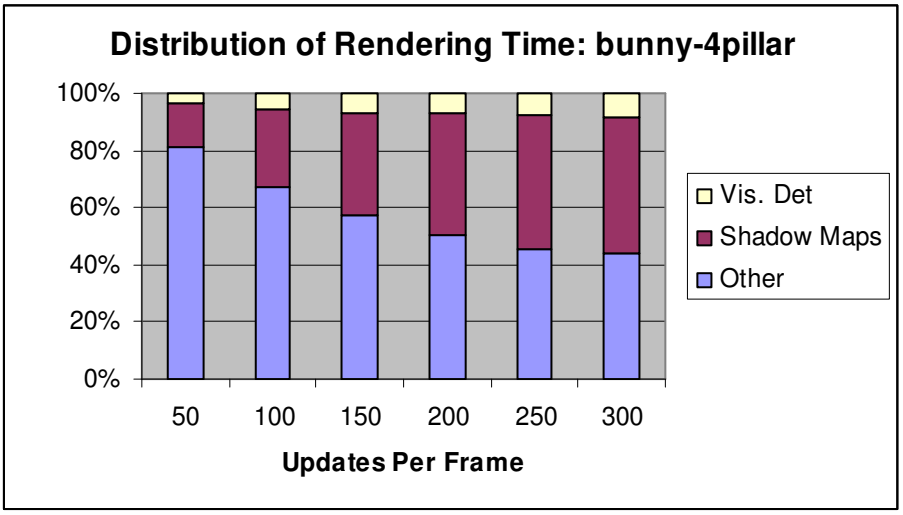

Figure 5-6 : Fraction of rendering time taken by visibility updates. Note that updates dominate performance for high light counts

5.3 Direct Illumination Quality

Figure 5-8 shows a series of images rendered with only direct illumination, and examines the effect of modifying both the shadow map resolution and the number of lights. The images suggest that a high shadow map resolution is needed to accurately reproduce the shape of the cast shadows (as we would expect). They also show that a relatively high light sample count is needed to completely remove aliasing due to under-sampling of the source. Some aliasing is also visible due to the limited resolution of the shadow maps. Percentage closer filtering [Reeves et al. 1987] could be used to soften these edges, but because this requires three additional shadow map samples, it causes a substantial performance penalty. Dedicated hardware support for PCF could help to reduce the performance penalty and allow us to greatly improve the quality of our images, but many current GPUs lack this support.

We have observed, however, that the aliasing in these images may not present as much of a problem in practice. In most real applications such as computer games, surfaces typically have texture maps applied to add details to a surface. If the texture maps contain enough high frequency details, we have observed that these details can act as a natural filter to reduce the effect of the aliasing. Consider the right half of Figure 5-7. These images were produced under the same conditions as the left images, except that a texture map was used to change the color of the floor. This effect may reduce the need for percentage closer filtering in practice, but it is also dependent on the shape of the occluder and its position relative to the light source.

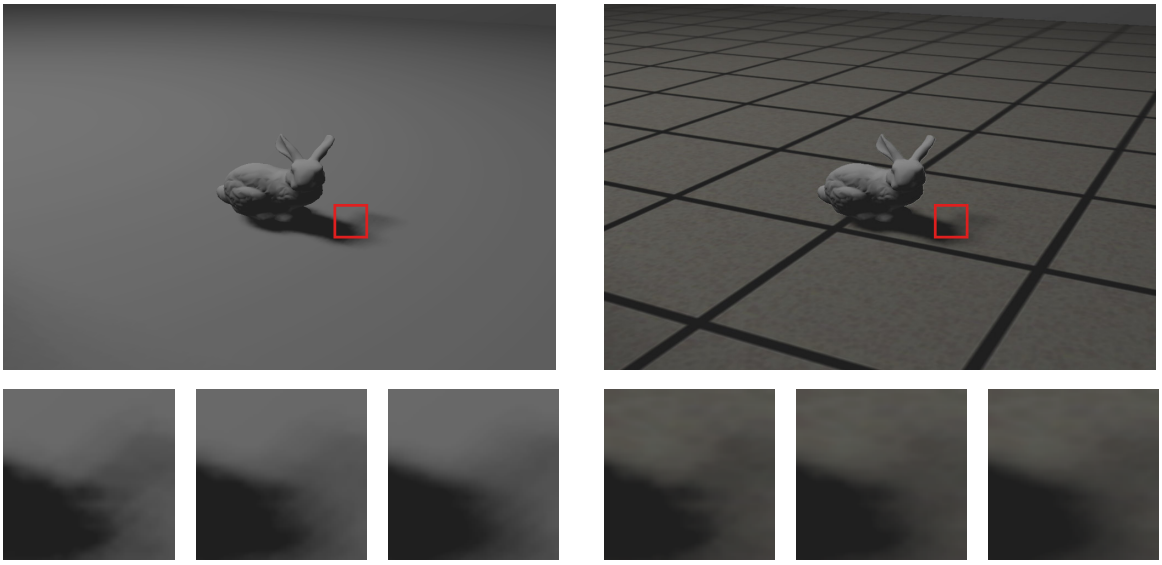

Figure 5-7 : Shadows from direct illumination. The scene is rendered using 512x512 shadow maps for 25, 50, and 100 lights. Note that aliasing in the shadow is less perceptible in the textured image on the right. Filtering was not used in the right images.

In scenes containing only direct illumination, our results indicate that our method can produce soft shadows from area lights at speeds comparable to those obtained by previous methods [Assarson et al. 2003]. In addition, because our technique samples the light source directly, it can easily be extended to lights of arbitrary shape, and can produce physically accurate shadows from any area source given a sufficiently high sample count. Some of the faster interactive soft-shadow techniques [Chan and Durand 2003, Wyman 2003] do not produce physically accurate shadows, and most of the existing algorithms do not generalize well to lights of arbitrary shape.

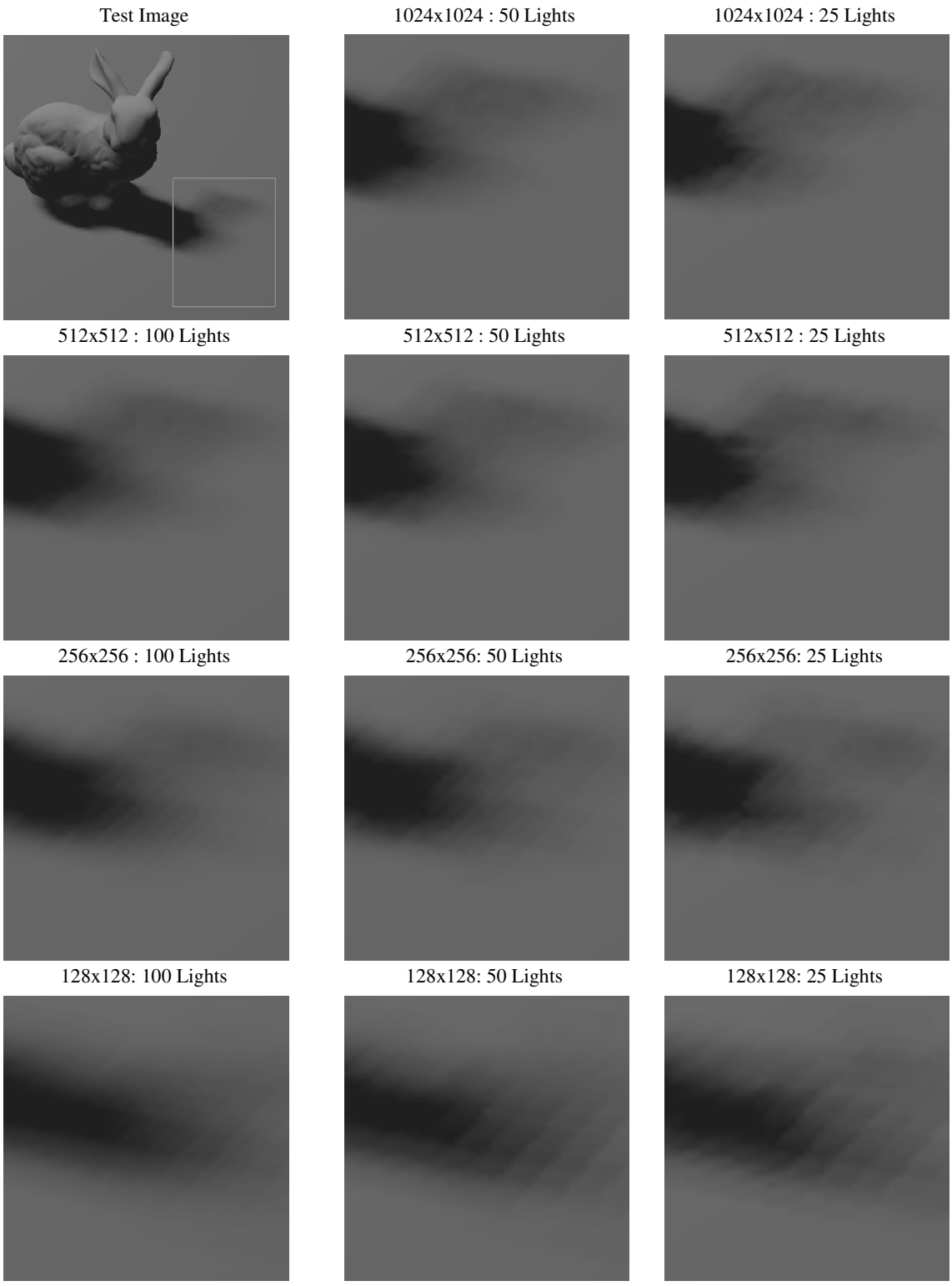

Figure 5-8 : Effects of shadow map resolution and light count on image quality. Higher map resolutions result in less aliasing and a more accurate shadow shape. Higher sample counts reduce aliasing but do not improve shadow shape.

5.4 Indirect Illumination Quality

By implementing instant radiosity using our method, we are able to effectively capture a number of indirect illumination effects at interactive rates. One such effect is color bleeding, which occurs when a bright, colored surface reflects illumination onto a grey or white surface. In the table scene (Figure 5-1, center), strong color bleeding can be seen from a blue wall (not shown), onto the bunny's tail, and onto the sides of the table. Color bleeding from the red wall is also visible in this image, but is much more subtle.

Our method is also one of the first interactive techniques which can accurately handle the occlusion of indirect illumination. Unlike previous techniques [Kontkanen and Laine 2005], we do this without performing any pre-processing on the occluders, which means that occlusion from dynamically changing objects, such as skinned characters, could eventually be handled. Figure 5-9 prevents an example illustrating the importance of indirect occlusion. In this figure, color bleeding on a moving object disappears when the object moves behind an occluding pillar. Figure 5-10 presents a more interesting example. In this example, the scene is illuminated using 25 direct samples and 275 indirect samples, with a bounce limit of five. Note the contact shadow which the bunny casts onto the ground, and the significant occlusion caused by the grey block.

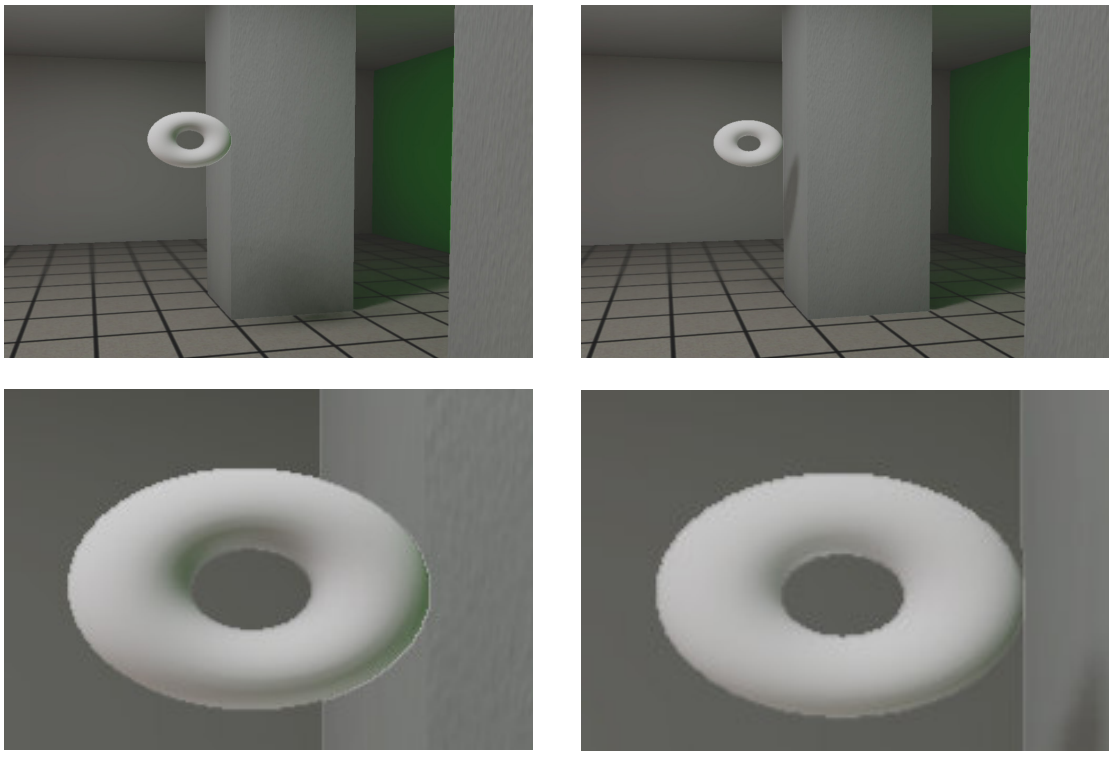

Figure 5-9 : Changes in color due to occlusion. Color bleeding from the green wall disappears when the object moves behind the pillar. Bottom images show close-ups of the object.

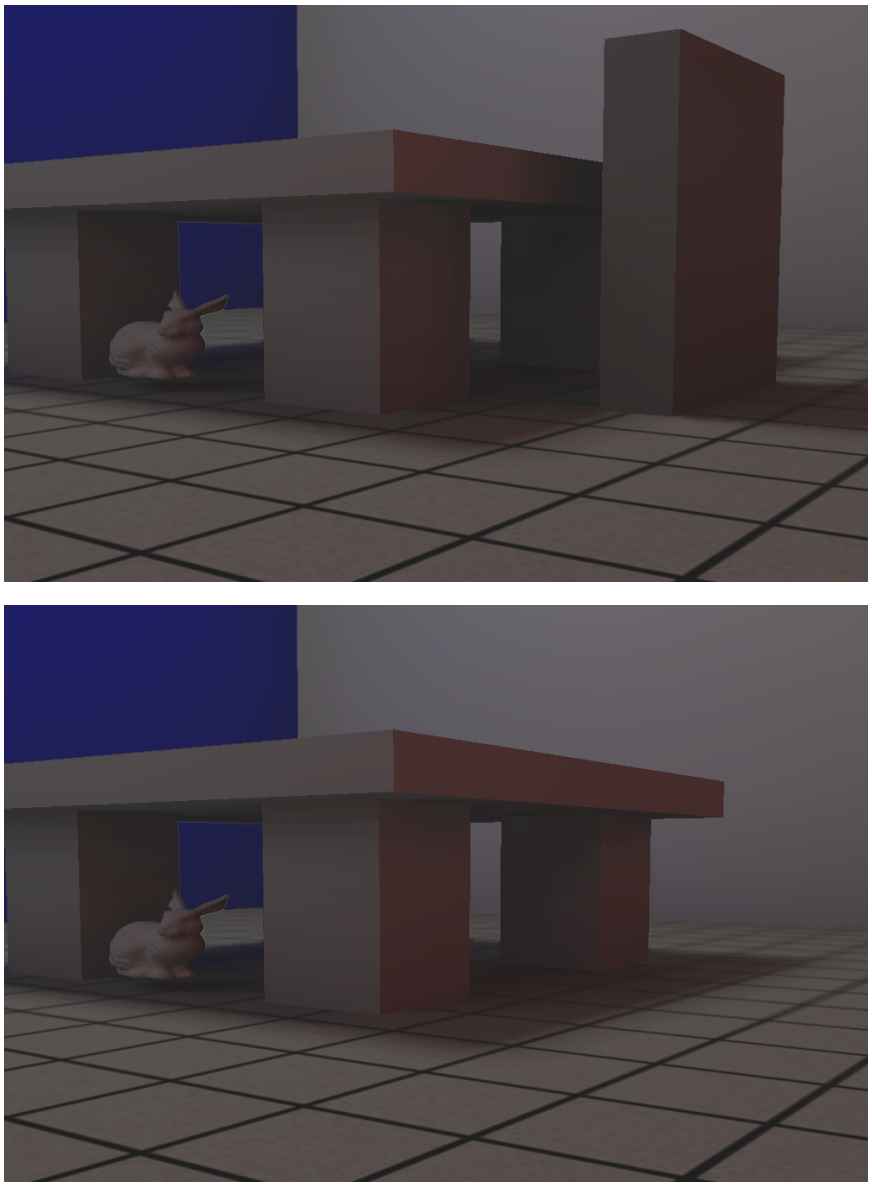

Figure 5-10 : Shadows in indirect illumination. Note the shadow cast by the bunny onto the ground, and the darkening caused by the large grey block. For comparison, the bottom image shows the scene with the block removed.

6 Conclusions, Limitations and Future Work

We have presented an interactive rendering system based on large point light sets. Our system is able to render physically accurate soft shadows from arbitrary area lights at speeds comparable to existing soft shadow algorithms. We can also render indirect illumination by utilizing instant radiosity. We present a number of optimization techniques which provide an order-of-magnitude performance boost over a naïve implementation, and allow us to achieve interactive frame rates. Our system is also able to maintain interactive frame rates in scenes containing moving objects, by spreading the update costs across several frames. In this section, we briefly discuss some of the limitations of our current system, and suggest directions for future work.

6.1 Limitations of Instant Radiosity

Although we are able to achieve compelling indirect illumination by using instant radiosity, there are a number of situations in which we produce unacceptable images due to limitations inherent in this method. A key problem with the instant radiosity approach is that the use of a fixed set of point samples does not provide an adequate sampling over the hemisphere for all points in the scene if the set is very small. In our system, we can only handle a few hundred lights at most, and this number of samples has sometimes proven to be inadequate.

Consider the example presented in Figure 6-1. In this image, the object is located in a dark area which is accessible only through a narrow gap in the wall. This is a case in which our light placement strategy is not effective. Because we are using a limited number of indirect illumination samples, very few of these samples are able to pass through the gap in the wall, and the ones which make it through do not provide an adequate sampling over the space of possible light directions. The result is that occlusion in one of the sampled directions inappropriately biases the solution.

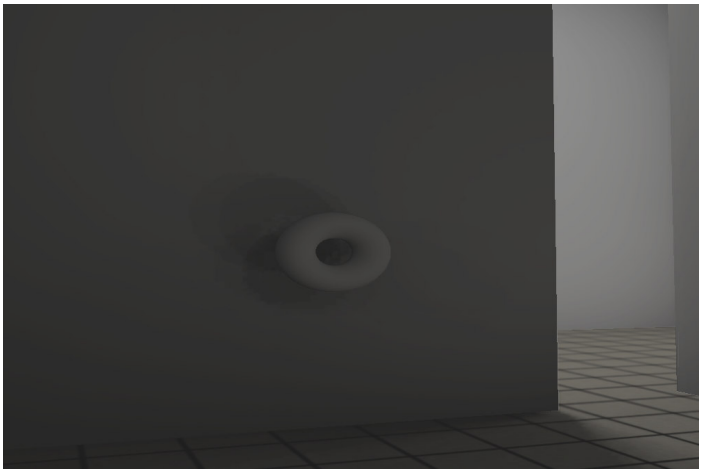

Figure 6-1 : Sampling artifacts: Not enough point lights illuminate the wall, and the discrete shadows from each are visible. This effect occurs because the samples do not fully sample the space of incoming illumination directions.

The problems in this image could be potentially be solved by a smarter sampling strategy. Instead of distributing light samples by a random walk, as we do in our system, and as was done in the original work [Keller 1997], the samples could instead be distributed uniformly in space and weighted according to the fraction of random walk samples which reach a particular region. This would create a set of samples which uniformly samples the hemi-sphere of incident directions, but the relative intensities of the lights would still accurately reflect the energy distribution in the scene.

A related, and more difficult problem, occurs when an occluder passes too close to a light source, as in Figure 6-2. In this image, the light on the floor is occluded by the bunny, and a large bunny-shaped silhouette is cast onto the far wall. This occurs because this nearby light contributes a disproportionate amount of illumination to the wall. A more accurate image would not have such a prominent silhouette, because additional illumination from the area around the bunny would cancel out its effect. The only effective solution to this problem is to increase the number of samples to a level that is well beyond what our method is capable of handling efficiently. We estimate that thousands or perhaps tens of thousands of samples would be needed, based on the test scenes which have been used in previous work [Walter et al. 2005]. Apart from the obvious performance penalty, the main problem with using so many lights is that it would become difficult to store such a large number of shadow maps. An interleaved sampling approach [Wald et al. 2002] may provide a better solution.

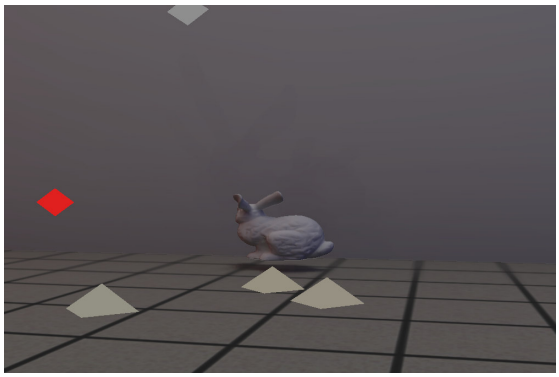

Figure 6-2 : Inappropriate silhouettes occur when a moving object passes too close to a light sample. Colored pyramids indicate light positions.

6.2 Improving Scalability

While not a problem in our test scenes, our visibility determination algorithm can be expensive, and is likely to become a bottleneck in scenes which contain a large number of moving objects. Hierarchical space partitioning structures such as BSP trees [Fuchs et al. 1980] could produce significant performance gains. Techniques which exploit frame-to-frame coherence in the relationships between surfaces [Drettakis and Sillion 1997] [Teller and Hanrahan 1993] are another possibility. It may also be worthwhile to use a hierarchical clustering over the lights to accelerate this process. Rather than perform occlusion tests for each light individually, a larger bounding shape could be used to test for visibility between objects and clusters of lights.

Another problem which we have not considered is the problem of applying our method to large, elaborate environments. Large environments would require a large number of light samples, and it would be impossible to store all of the shadow maps in memory at the same time. Some partitioning of the scene into discrete cells, using a representation such as an adjacency graph [Teller and Sequin 1991] would need to be devised in order to detect the set of lights whose shadow maps are needed. As the user moves through the environment, shadow maps could be re-generated on the fly as the user approaches their area of influence, and discarded when they are no longer needed. If this process were done gradually, over the course of perhaps ten frames, the impact on rendering performance would be modest, as demonstrated in Section 5.2.

6.3 Dynamic Light Sets

Another limitation of our system is that it does not currently support dynamic changes to the set of point lights. This can become a problem if the moving objects have a significant amount of influence over the distribution of the indirect illumination (for example, a brightly colored object bleeding color onto the walls). We could support dynamic updates of the light sample set by adapting the selective photon tracing techniques which have been used in previous work

[Dmitriev et al. 2002, Larsen and Christenson 2004]. This would require ray tracing, but the number of rays traced per frame would be relatively low, and the cost would be manageable. Changes to light samples would require a full re-computation of the visibility information for the changed lights, and would also require a large number of light map updates for each frame, in addition to the shadow map updates. Light maps, however, are much easier to update than shadow maps, since the effects of many light changes can be processed in one rendering pass. In addition, multiple render targets could be used to modify several light maps in a single pass.

7 References

ASSARSON, U., DOUGHERTY, M., MOUNIER, M., AKENINE-MOLLER T. 2003. An Optimized Soft Shadow Volume Algorithm with Real Time Performance. Proceedings of The ACM/Eurographics Annual Conference on Graphics Hardware.

BAUMANN, D. 2005. ATI Xenos: X Box 360 Graphics Demystified. Online at: http://www.beyond3d.com/articles/xenos/. June 13, 2005.

BRABEC, S., ANNEN, T., AND SEIDEL, H. 2002. Shadow mapping for hemispherical and omnidirectional light sources. In Proc. of Computer Graphics International.

CARR, N., HALL, J., HART, J., 2002. The Ray Engine. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. pp. 37-46

CHAN, E., DURAND, F. 2003. Rendering fake soft shadows with smoothies. In Eurographics Symposiumon Rendering. pp 208-218

CHEN J., GORDON, M. THIES, W. ZWICKER, M. PULLI, K. DURAND, F. 2005. A Reconfigurable Architecture for Load-Balanced Rendering. Proceedings of the ACM/EUROGRAPHICS Workshop on Graphics Hardware. pp 71-80

COHEN, M. GREENBERG, D. The Hemi-Cube: A Radiosity Solution for Complex Environments. Proceedings of SIGGRAPH 1985. pp. 31-40.

COHEN, M. WALLACE, J. 1993. *Radiosity and Realistic Image Synthesis*. Morgan Kaufmann Series in Computer Graphics. ISBN: 0121782700, 1993.

CROW, F. C. 1977. Shadow algorithms for computer graphics. In Proceedings of ACM SIGGRAPH (1977), pp. 242-248.

COOMBE, J. HARRIS, M., LASTRA, A. 2004. Radiosity on Graphics Hardware. Proceedings of Graphics Interface 2004. pp. 161-168.

DACHSBACHER, C., STAMMINGER, M., Reflective Shadow Maps. 2005. Proc. SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games. pp 203-231

DMITRIEV, K., BRABEC, S., MYSZKOWSKI, K., SEIDEL, H.P., 2002. Interactive Global Illumination Using Selective Photon Tracing. Rendering Techniques 2002 (Proc. 13th Eurographics Workshop on Rendering), pp. 15-24.

DRETTAKIS, G. SILLION, F. 1997 Interactive Update of Global Illumination Using a Line-space Hierarchy. Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques. (SIGGRAPH '97). pp 57-64

EVERITT C., KILGARD M. J. 2002: Practical and robust stenciled shadow volumes for hardware-accelerated rendering. http://developer.nvidia.com/object/robust_shadow_volumes.html.

EYLES J., MOLNAR S., POULTON, J. GREER, T., LASTRA, A. ENGLAND, N. WESTOVER, L. 1997. PixelFlow: the realization. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on graphics hardware. pp 57-68

FUCHS, H. KEDEM, ZVI. NAYLOR, B. 1980. On visible surface generation by a priori tree structures. Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques. (SIGGRAPH '80). pp 124-133.

GORAL, C., TORRANCE, K., GREENBERG, D., BATTAILE, B. 1984. Modeling the Interaction of Light Between Diffuse Surfaces. Proceedings of SIGGRAPH 1984. pp. 213-222.

GREEN, N. KASS, M. MILLER, G. 1993. Hierarchical Z-Buffer Visibility. Proceedings of the 20th annual conference on Computer graphics and interactive techniques. (SIGGRAPH '93). pp 231-237.

GREGER, G. SHIRLEY, P. HUBBARD, P. GREENBERG, D. 1998. The Irradiance Volume. IEEE Computer Graphics and Applications. 18(2) pp 32-43

HANRAHAN, P., SALZMAN, D., AUPPERLE, L. 1991. A Rapid Hierarchical Radiosity Algorithm. Proceedings of ACM SIGGRAPH. pp. 197-206.

HASENFRATZ, J. LAPIERRE, M. HOLZSCHUCH, N. SILLION, F. 2003. A Survey of Real-Time Soft Shadows Algorithms. Computer Graphics Forum: 22(4), pp 753-774.

HEIDMANN, T. 1991. Real Shadows Real Time, IRIS Universe, Number 18, pp. 28-31.

JENSEN, H. 2001. Realistic Image Synthesis Using Photon Mapping. A.K. Peters.

KAJIYA, J. 1986. The Rendering Equation. Proceedings of ACM SIGGRAPH 1986. pp. 143-150.

KELLER, A., Instant Radiosity. 1997. Proceedings of ACM SIGGRAPH 1997. pp. 49-56.

KONTKANEN, J. LAINE, S. 2005. Ambient Occlusion Fields. Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games. pp. 41-48

KOK, A.J., Grouping of Patches in Progressive Radiosity. 1993. Proceedings of the Fourth Eurographics Workshop on Rendering. pp 221-231.

LAINE, S. AILA, T. ASSARSSON, U. LEHTINEN, J. AKENINE-MOLLER, T. 2005. Soft Shadow Volumes for Ray Tracing. ACM Transactions on Graphics. 24(3). pp 1156-1165

LARSEN, B.D., CHRISTENSEN, N.J. 2004. Simulating Photon Mapping for Real-time Applications. Proc. Eurographics Symposium on Rendering (2004).

LISCHINSKI, D., TAMPIERI, F., GREENBURG, D. 1992. A Discontinuity Meshing Algorithm for Accurate Radiosity. IEEE Computer Graphics and Applications. 12(6): 1992. pp. 25-39.

LLOYD B., WENDT J., GOVINDARAJU N.,MANOCHA D. 2004. CC Shadow Volumes. In Proceedings of the Eurographics Symposium on Rendering

MICROSOFT. 2005. DirectX 9.0C Programmer's Guide. Online at: http://msdn.microsoft.com/library/en-us/directx9_c/dx9_graphics_programming_guide.asp

NIJASURE, M., PATTANAIK, S., GOEL, V. 2005. Real-Time Global Illumination on GPU. Journal of Graphics Tools. Vol. 7. November 2005.

NG, R. RAMAMOORTHI, R. HANRAHAN, P. 2003. All-Frequency Shadows Using Non-linear Wavelet Lighting Approximation. ACM Transactions on Graphics. 22(3). pp 376-381.

OAT, C. 2005. Irradiance Volumes For Games. Game Developer's Conference, Oral Presentation. Slides available at: http://www.ati.com/developer/gdc/GDC2005_PracticalPRT.pdf

PURCELL T.J., BUCK I, MARK W.R., HANRAHAN P. 2002. Ray tracing on programmable graphics hardware. Proceedings of SIGGRAPH 2002.

PURCELL T.J., DONNER C., CAMMARANO M., JENSEN H.W., HANRAHAN P., 2003. Photon Mapping on Programmable Graphics Hardware. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. pp. 41-50

PARKER S., PARKER M., LAVNAT Y., SLOAN P.P., HANSEN C., SHIRLEY P., 2002. Interactive Raytracing for Volume Visualization. In IEEE Transactions on Computer Graphics and Visualization, 1999, 5(3):238-250

REEVES, W. T., SALESIN, D. H., COOK, R. L 1987. Rendering antialiased shadows with depth maps. In Computer Graphics (Proceedings of ACM SIGGRAPH 87), ACM SIGGRAPH, 283–291.

SCHMITTLER, J. WALD, I. SLUSALLEK, P. 2002 SaarCOR: A Hardware Architecture for Ray-Tracing. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. pp 27-36.

SEN, P. CAMMARANO, M., HANRAHAN, P. 2003. Shadow Silhouette Maps, Proceedings of ACM SIGGRAPH 2003. p. 521-526.

SLOAN, P.P, KAUTZ, J, SNYDER, J. 2002. Precomputed Radiance Transfer for Interactive Rendering in Dynamic, Low Frequency Lighting Environments. Proceedings of the 29th annual conference on Computer graphics and interactive techniques. (SIGGRAPH 2002). pp. 527-536, San Antonio, 2002.

SLOAN, PP. HALL, J. HART, J. SNYDER, J. 2003. Clustered Principle Components for Precomputed Radiance Transfer. ACM Transactions on Graphics, 22(3). pp 382-391.

SLOAN, PP. LUNA, B. SYNDER, J. 2005. Local, deformable pre-computed radiance transfer. ACM Transactions on Graphics, 24(3). pp 1216-1224.

STAMMINGER, M., DRETTAKIS, G. 2002. Perspective shadow maps. ACM Transactions on Graphics 21, 3, 557–562. 2002.

TABELLION, E., LAMORLETTE, A. 2004. An Approximate Global Illumination System for Computer Generated Films. ACM Transactions on Graphics: Proceedings of the 2004 SIGGRAPH Conference. 23(10) pp. 469-476.

TELLER, S. SEQUIN, C. 1991. Visibility Preprocessing for Interactive Walkthroughs. Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques. (SIGGRAPH '91). pp 61-70.

TELLER, S. HANRAHAN, P. 1993. Global Visibility Algorithms for Illumination Computations. Proceedings of the 20th annual conference on Computer graphics and interactive techniques. (SIGGRAPH '93). pp 239-246.

UDESHI, T., HANSEN, C. 1999. Towards Interactive Photorealistic Rendering of Indoor Scenes: A Hybrid Approach. In Rendering Techniques 1999 (Proc. 10th Eurographics Workshop on Rendering), pp. 63-76

WALD, I., SLUSALLEK, P., BENTHIN C., WAGNER M. 2001. Interactive Rendering with Coherent ray tracing. Computer Graphics Forum 20, 3 (2001), pp. 153-164.

WALD, I., KOLLIG, T., BENTHIN, C.,WAGNER, M., SLUSALLEK, P., 2002. Interactive Global Illumination Using Fast Ray Tracing. Rendering Techniques 2002 (Proc. 13th Eurographics Workshop on Rendering), pp. 15-24.

WALTER, B. FERNANDEZ, S. ARBREE, A. BALA, K., DONIKIAN, M, GREENBERG, D. 2005. Lightcuts: A Scalable Approach to Illumination. ACM Transactions on Graphics, 24(3) pp 1098-1107.

WARD, G. RUBINSTEIN, F, CLEAR, R., 1988. A Ray Tracing Solution for Diffuse Interreflection, Computer Graphics, Vol. 22, No. 4

WARD, G. 1991. Adaptive Shadow Testing for Ray Tracing, Second EUROGRAPHICS Workshop on Rendering, Barcelona, Spain, April 1991.

WATT A.,WATT M. 1992., *Advanced Animation and Rendering Techniques*. ACM Press.

WHITTED, T. 1980. An Improved Illumination Model for Shaded Display. Communications of the ACM, 23(6): 343-349, June 1980.

WILLIAMS, L. 1978. Casting Curved Shadows on Curved Surfaces. Computer Graphics (Proceedings of SIGGRAPH 78), 12(3):270–274, August 1978

WOOP, S. SCHMITTLER, J. SLUSALLEK, P. 2005. RPU: A Programmable Processing Unit for Realtime Ray Tracing. ACM Transactions on Graphics, 24(3). pp 434-444.

WYMAN, C. HANSEN, C. 2003. Penumbra maps. Proceedings of the Eurographics Symposium on Rendering,2003.

ZHOU, K. YAOHUA, H. LIN, S. GUO, B. SHUM, H. 2005. Pre-computed Shadow Fields for Dynamic Scenes. ACM Transactions on Graphics. 24(3). pp 1196-1201