Real-time Cinematic Design Of Visual Aspects In Computer-generated Images

Juraj Obert
University of Central Florida

REAL-TIME CINEMATIC DESIGN OF VISUAL ASPECTS IN COMPUTER-GENERATED IMAGES

by

JURAJ OBERT
B.S. Slovak University of Technology, 2004
M.S. Czech Technical University, 2007

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2010

Major Professor:
Sumanta N. Pattanaik

Abstract

Creation of visually-pleasing images has always been one of the main goals of computer graphics. Two important components are necessary to achieve this goal — artists who design visual aspects of an image (such as materials or lighting) and sophisticated algorithms that render the image. Traditionally, rendering has been of greater interest to researchers, while the design part has always been deemed as secondary. This has led to many inefficiencies, as artists, in order to create a stunning image, are often forced to resort to the traditional, creativity-baring, pipelines consisting of repeated rendering and parameter tweaking.

Our work shifts the attention away from the rendering problem and focuses on the design. We propose to combine non-physical editing with real-time feedback and provide artists with efficient ways of designing complex visual aspects such as global illumination or all-frequency shadows. We conform to existing pipelines by inserting our editing components into existing stages, hereby making editing of visual aspects an inherent part of the design process.

Many of the examples showed in this work have been, until now, extremely hard to achieve. The non-physical aspect of our work enables artists to express themselves in more creative ways, not limited by the physical parameters of current renderers. Real-time feedback allows artists to immediately see the effects of applied modifications and compatibility with existing workflows enables easy integration of our algorithms into production pipelines.

To those who have always believed in me.

Acknowledgments

Most and foremost, I would like to thank my advisor, Sumanta Pattanaik, for his support, direction and help over the past three years. It was him and Jaroslav Křivánek who introduced me to UCF and the life of PhD students. I thank all former UCF graphics students for their friendship, fun in the lab and help with my research: Mais Alnasser, Kévin Boulanger, Mark Colbert, Jared Johnson, Adrià Forés Herranz, Jaakko Konttinen and Musawir Shah.

I also thank Fabio Pellacini for being a fantastic person to work with and a great expert in our field, Charles Hughes for his advice and corrections of this text, Joseph LaViola for providing many constructive suggestions and improvements and Alex Zelenin for his great 3D models.

Special thanks goes to all of my friends in Florida who are making my life here more than amazing: Elena, Miko, Kamran, Irina, Ruben, Paula, all UCF breakdancers, all of my friends scattered around the world, my former colleagues at EA Tiburon and my family in Slovakia. I also thank Facebook for letting me keep in touch with all of you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Computer graphics has progressed rapidly since its inception in the early 1960's and is now used extensively in computer-generated movies and games. Today, computer-generated photo-realistic imagery enhances every new movie and allows computer games to deliver stunning visual effects and immersive gameplay. This, however, brings along a new set of challenges for the design of lighting, shadows and materials. In this dissertation, we focus on leveraging the computational power of Graphics Processing Units (GPUs) in order to allow artists to design visual aspects of movies and games with ease. As we shall see, these tasks would be very difficult otherwise.

1.1 The Problem

Traditionally, research in computer graphics has focused mainly on rendering of photo-realistic images, a process known to be very time-consuming even on specialized hardware. Due to long rendering times and lack of real-time feedback, lighting design has become a tedious task consisting of repeated rendering, waiting, reviewing and adjusting of lighting parameters. Ultimately, the ideal lighting design application would allow the user to change all visual aspects of a scene while retaining real-time rendering feedback. Unfortunately, due to lack of computational power, this is not yet achievable.

This problem has been addressed by the introduction of many precomputation-based techniques for relighting. These techniques allow artists to precompute static data and reuse it later in interactive design sessions. For example, in *material design applications*, lighting is considered to be static and only material parameters can be changed in the design session. Similarly, in *relighting applications*, materials are considered to be fixed and only lighting can be adjusted. The great advantage of being able to tweak lighting parameters in real-time, however, often comes at the high cost of long precomputation times. Another disadvantage stems from the fact that internal representations of performed edits are very application-specific and therefore cannot be easily transferred to other renderers.

Very often, artists wish to modify visual aspects of a scene in a way that has no equivalent to the real-world laws of light physics. For example, artists might wish to deform a shadow without deforming an object or apply different reflectance properties to an object on each light bounce. Since the vast majority of rendering techniques available today focus on physically-correct rendering, such *non-physical* edits are very hard to achieve (unless one is willing to do a significant amount of hacking around the code of the production renderer), yet they are very desirable for artists [Bar97].

Consider a typical production pipeline as depicted in Figure 1.1 and a relatively simple task description asking the artist to modify the color of reflected indirect light in a dark corner of a room. If we assume the best case scenario, when the renderer, which is based

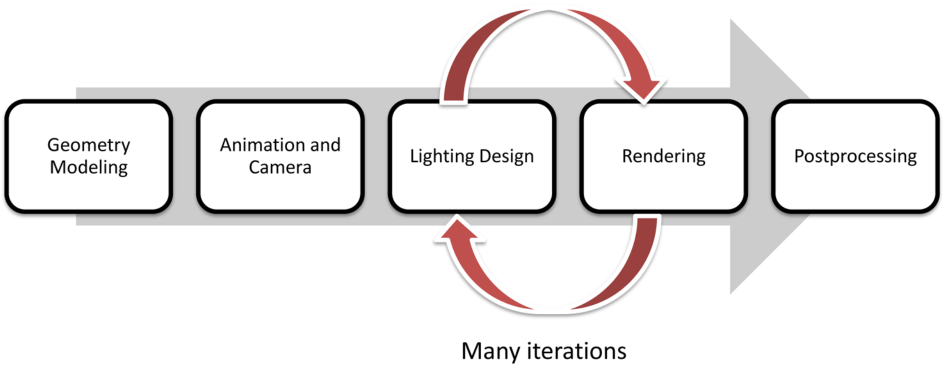


Figure 1.1: A traditional production pipeline.

on a physically-correct simulation of light transport, supports global illumination, then the task of modifying the color of reflected indirect light comes down to repeated tweaking of light parameters and rendering. This process can take many iterations and is very inefficient for many reasons:

1. Physically-correct simulation of global illumination is computationally expensive.
2. Most renderers do not provide any tools to modify indirect lighting.
3. The edit itself is non-physical, which creates a strong disconnect between the design goal and the physically-correct rendering algorithm.

Now consider a similar task consisting of moving a shadow. We know that shadows are caused by occluders blocking incoming light and therefore one way to move a shadow would be to move the light in the opposite direction. However, moving the light will also cause changes in the shading and shadows on all other surfaces in the scene, which is what we might not want at all since it will disrupt the design we created before. The situation becomes even more complicated if we want to work with environment maps and all-frequency shadows. The following are the reasons why editing of all-frequency shadows is still a complicated problem in production pipelines:

1. The artist cannot simply move the light, because there are many lights (texels) in the environment map.
2. There is no decoupling between lights and shadows in most renderers, because their relation is physically-based.
3. There are no shadow selection tools available for scenes with environment lighting.

Both global illumination and all-frequency shadows are complex visual aspects often present in computer-generated images. Their editing/design, as demonstrated in the examples above, often requires approaches that allow artists to break the inherent physical relations. In our work, we focus on such non-physical approaches in the context of existing pipelines. We strive to achieve three important goals: real-time feedback, adherence to existing pipelines, a wide range of editing operations (including physical and non-physical ones). Based on these observations, we formulate the thesis of this dissertation as follows:

The combination of non-physical editing and real-time feedback, implemented by efficient algorithms on GPUs and coupled with a wide range of editing operations, greatly improves the efficiency of production pipelines by facilitating the artistic design process with tools that allow for more creativity, expressiveness and easy transformation of high-level design decisions into images.

1.2 Our Approach

In order to demonstrate the viability of our solution, we focus on three important components of interactive design: editing tools, internal representation of performed edits and all-frequency effects. We research and propose a variety of editing tools and concepts for designing lighting in a global illumination environment with one-bounce interreflections. Our tools allow artists to adjust and edit effects such as color bleeding or light intensity falloff with distance. We build our interface on top of a real-time rendering engine and provide interactive feedback.

We follow up on this work with a robust and intuitive representation of performed edits. Such a representation allows us to easily integrate and build a global illumination relighting pipeline. We first design edits in a real-time renderer, then store them in a renderer-independent fashion and use in a high-quality offline renderer. Finally, we focus on important all-frequency visual aspects such as all-frequency shadows. We develop a similar precomputation-based pipeline, including a novel representation for shadows, hereby allowing artists to design shadows in real-time.

Our work attempts to address the problem of efficient design by providing tools and representations that can support both physical and non-physical operations. We build our pipelines in ways that not only support new edits, but also encourage users to experiment and develop new editing techniques. We do not modify existing, time-proven, workflows, but rather introduce algorithms that can be implemented using different interfaces.

In summary, our work makes the following major contributions:

- An efficient pipeline for lighting design with global illumination based on:
    - An efficient representation for edited indirect lighting.
    - A set of novel editing operations for indirect lighting.
    - Renderer-independence.
- An efficient pipeline for design of all-frequency shadows based on:
    - Decoupling of visibility from lighting.
    - A wide range of novel and interactive shadow editing operations.
    - A novel representation of all-frequency shadows based on visibility ratios.
    - A very efficient precomputation for easy integration into existing pipelines.

We believe that powerful, intuitive and real-time design applications are the next step toward next-generation computer graphics in movies and video games. Artists, whose creativity is negatively impacted by long rendering times, crave for better and faster tools, which would allow them to express and tweak their visions on a computer screen in a matter of seconds. We hope that the work described in this thesis will motivate other researchers to push the boundaries of real-time design even further and turn the design task from a tedious and cumbersome process into a creative, interactive and mainly enjoyable part of movie/game productions.

1.3 Overview

The rest of this thesis is organized as follows: In Chapter 2, we introduce fundamental concepts of computer graphics including radiometry, bidirectional reflectance functions (BRDFs), global illumination, etc. The reader should be familiar with these concepts (albeit detailed understanding is not necessary) as we refer to them throughout the rest of this dissertation. In Chapter 3 we review previous work carried out in the field of lighting, material and appearance design. We also review important rendering techniques used in high-quality renderers.

In Chapter 4, we introduce LightShop, our user interface for relighting with global illumination. The interface also presents our novel editing operations. Chapter 5 describes iCheat, our general representation of modifications to light transport and its role in the relighting pipeline. We show how a suitable representation can be used to achieve renderer-independence and how such an approach adheres to existing pipelines.

Chapter 6 describes our approach to all-frequency shadows. We develop efficient precomputation algorithms running on the GPU, show novel techniques for shadow selection and extraction and describe our editing tools. Finally, we conclude the thesis with results in Chapter 7 and propose future work in Chapter 8.

CHAPTER 2

BACKGROUND

This chapter introduces the fundamental concepts of computer graphics. We start off by describing basic radiometric quantities, reflectance functions and rendering equations. We conclude the chapter with higher-level concepts including methods for global illumination and relevant compression techniques such as wavelets. Information is this chapter will provide the groundwork for our design concepts developed later.

2.1 Radiometry

For our purposes we will describe light in terms of quantum units, photons. A photon is a light particle with associated direction, energy and wavelength λ. The speed of a photon depends on the refractive index of the medium. For air, the speed equals $c = 3 \times 10^8 ms^{-1}$. The energy q carried by a photon depends on its frequency f:

$$q = hf = \frac{hc}{\lambda}$$

where $h = 6.63 \times 10^{-34} J$ is Plank's Constant.

Radiant energy Q is the total energy of photons passing through a surface of a given area in a given period of time. It is measured in Joules (J) and can be computed by summing all energies of all photons in a given area.

Light arriving at a surface or emitted from a light source can be quantified in terms of its *radiant power (flux)*, which equals the amount of energy transferred per unit of time. Flux is denoted by Φ and is measured in Watts (W):

$$\Phi = \frac{dQ}{dt}$$

In order to relate energy to area, we use the quantity called *irradiance* (see Figure 2.1), which tells us how much energy is transferred per unit area. Irradiance does not take directions of traveling photons into account. It is measured in Watts per square meter and defined as:

$$E = \frac{d\Phi}{dA}$$

It is sometimes useful to differentiate between incoming and outgoing light. For light arriving at a surface, we use the aforementioned term *irradiance*, whereas for light leaving the surface, we use the term *radiosity (radiant exitance)*, denoted B.

More often than not, we are interested only in light transferred from/to certain directions. Imagine for example a light bulb with a paper cone attached to it. Such a light source emits photons in all directions, but the amount of energy transferred through the paper cone is surely different from the total energy transferred in all direction. In order to quantify this

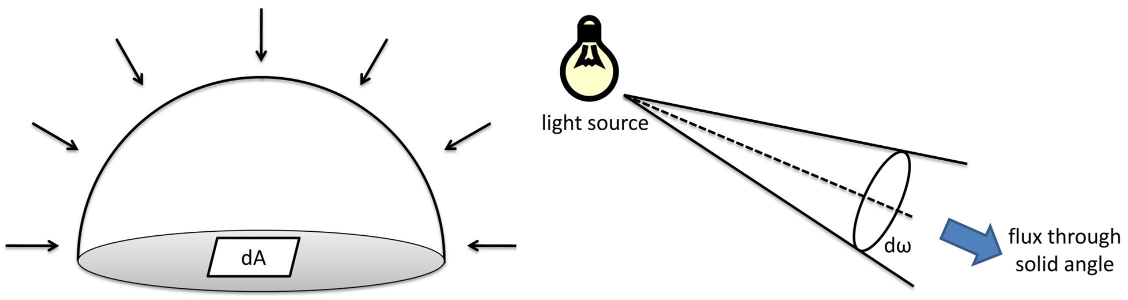


Figure 2.1: (Left) Irradiance as total energy arriving at a surface from all direction, per unit area. (Right) Radiant intensity is equal to flux flowing through a unit solid angle.

directional nature of light, we define *radiant intensity*, which equals to flux per unit solid angle (see Figure 2.1):

$$I = \frac{d\Phi}{d\omega}$$

where $d\omega$ is a solid angle in steradians.

The most important radiometric quantity is called *radiance*, which relates flux to unit solid angle and to unit projected area. It tells us that the amount of energy received by a surface depends on the direction from which the light arrives. This relationship can be expressed by the following equation:

$$L = \frac{d^2\Phi}{d\omega dA_p} = \frac{d^2\Phi}{d\omega \cos\theta dA}$$

where dA_p is the unit projected area and θ is the angle between the surface normal and the solid angle direction (see Figure 2.2). Radiance describes the effect we all experience every day when we go out. Around noon, when the sun is directly above us, the temperature

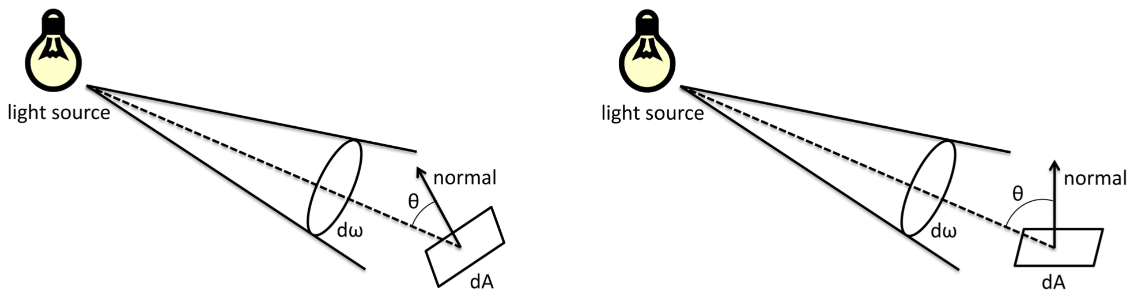


Figure 2.2: Radiance as flux per unit solid angle per unit projected area. (Left) The receiver is facing the light more head-on, thus receiving more light. (Right) The angle between the light direction and surface normal is greater, resulting is lower radiance received by the surface.

of the air is at its high, because the amount of energy transferred from the sun and hitting the surface of our planet is at its maximum. However, in the mornings and in the evenings, when the sun is near the horizon and light arrives from small grazing angles, the temperatures drop. Radiance (unlike irradiance or radiant exitance) takes this dependency into account.

Radiance can be used to express all other radiometric quantities. Irradiance at surface point x can be computed by integration over the visible hemisphere centered around the surface normal $\vec{N}$:

$$E(x) = \int_{\Omega_+(\vec{N})} L_i(x, \vec{\omega_i}) \cos\theta_i \; d\omega_i$$

where ω_i is the direction of incident radiance. Similarly, we can compute radiant intensity by integrating over the surface area A_x:

$$I(x, \vec{\omega_o}) = \int_{A_x} L_o(x, \vec{\omega_o}) \cos\theta_o \; dA$$

where the subscript o denotes outgoing directions.

In general, radiance does not vary over distance in media such as air or vacuum. The only exception to this rule is a *participating medium* such as fog or steam. Radiance is the quantity perceived by human eye and interpreted by a computer screen as color.

2.2 BRDF

BRDF stands for *bidirectional reflectance distribution function* and represents the formalism used to describe reflectance properties of a certain class of materials. The human eye only rarely perceives direct light (i.e. light traveling directly from a light source) and mostly deals with light reflected off objects around us. BRDFs can be used to describe reflectance properties of objects with the following characteristics:

- Light is reflected instantaneously, i.e. without any delay. *Phosphorescence* is the phenomenom that occurs when an object absorbs light and releases it with non-zero time delay.
- Light wavelength does not change after reflection. The shift in wavelength can be seen in minerals or in general, cold bodies and is referred to as *fluorescence*.
- Light arriving at surface location x is typically reflected off the same location $x' = x$. If these locations are different, the light must have traveled a non-zero distance inside the receiving object, i.e. we say that the light has *scattered* inside the object. Reflectance

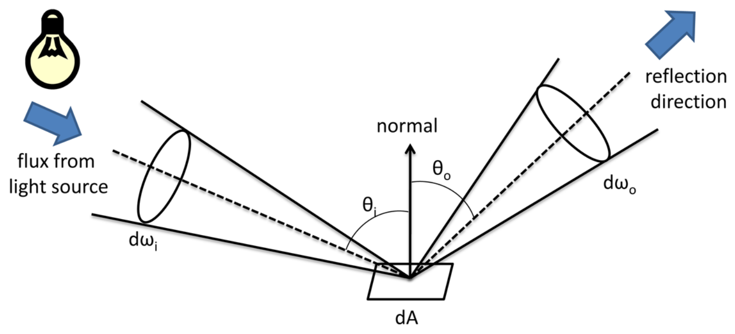


Figure 2.3: BRDF is the ratio of reflected radiance to incoming radiance per unit projected area.

functions used to describe surfaces with scattering properties are termed BSSRDFs (bidirectional surface scattering reflectance distribution functions).

For incoming light direction $\vec{\omega_i}$ and outgoing light direction $\vec{\omega_o}$, the bidirectional reflectance distribution function (BRDF) at location x is defined as:

$$f_r(x, \vec{\omega_i}, \vec{\omega_o}) = \frac{dL_o(x, \vec{\omega_o})}{L_i(x, \vec{\omega_i}) \cos\theta_i \; d\omega_i} = \frac{dL_o(x, \vec{\omega_o})}{dE(x, \vec{\omega_i})}$$

The value of a BRDF can be interpreted as the ratio of outgoing radiance to incoming radiance per unit projected area (see Figure 2.3).

Since the values of BRDFs have no upper bound, it is sometimes advantageous to define surface *reflectance (albedo)* $\rho(x)$:

$$\rho(x) = \frac{d\Phi_o(x)}{d\Phi_i(x)}$$

where $\Phi_o(x)$ and $\Phi_i(x)$ are outgoing and incoming radiant power (flux), respectively. Values of reflectance $\rho(x)$ fall into range $< 0, 1 >$, where $\rho(x) = 1$ represents no absorption by the surface (white color) and $\rho(x) = 0$ represents full absorption by the surface (black color).

2.2.1 Properties of BRDFs

Helmholtz reciprocity states that the value of a BRDF at point x does not change if we swap the incoming and outgoing directions:

$$f_r(x, \vec{\omega_i}, \vec{\omega_o}) = f_r(x, \vec{\omega_o}, \vec{\omega_i})$$

In other words, if a photon arriving at location x from direction $\vec{\omega_i}$ is reflected in direction $\vec{\omega_o}$, then a photon arriving at the same location from direction $\vec{\omega_o}$ will be reflected in direction $\vec{\omega_i}$ and vice versa. This property is of crucial importance for bidirectional rendering methods based on tracing of photon trajectories.

A BRDF is always positive

$$f_r(x, \vec{\omega_i}, \vec{\omega_o}) \geq 0$$

The law of energy conservation requires that the total amount of power reflected over all directions must be less than or equal to the total amount of power incident on the surface. The total incident power per unit surface area is the irradiance and the total power per unit surface area leaving the surface is defined as:

$$M(x) = \int_{\Omega_+(\vec{N})} L_o(x, \vec{\omega_o}) \cos\theta_o \ d\omega_o$$

From the definition of BRDF:

$$dL_o(x, \vec{\omega_o}) = L_i(x, \vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) \cos\theta_i \ d\omega_i$$

Then:

$$M(x) = \int_{\Omega_+(\vec{N})} \int_{\Omega_+(\vec{N})} L_i(x, \vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) \cos\theta_i \cos\theta_o \ d\omega_i \ d\omega_o$$

The constraint of energy conservation is then satisfied if:

$$\int_{\Omega_+(\vec{N})} f_r(x, \vec{\omega_i}, \vec{\omega_o}) \cos\theta_i \ d\omega_i \leq 1, \forall \vec{\omega_o} \in \Omega_+(\vec{N})$$

Anisotropy is a material property, which states that the reflectance properties depend also on the rotation of a surface about its normal (for example, velvet is a anisotropic material). *Anisotropic* BRDFs therefore depend on surface location x, incoming light direction $\vec{\omega_i}$ and outgoing light direction $\vec{\omega_o}$ (incoming light direction $\vec{\omega_i}$ can be expressed in terms of azimuth ϕ and elevation θ). In the case of *isotropic* BRDFs, the azimuth ϕ is not required.

2.2.2 The Rendering Equation

The Rendering Equation [Kaj86] is used to compute outgoing (reflected) radiance L_o at a surface location. Its complete form is given as:

$$L_o(x, \vec{\omega_o}) = L_e(x, \vec{\omega_o}) + \int_{\Omega_+(\vec{N})} L_i(x, \vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) \cos\theta_i \; d\omega_i$$

where $L_e(x, \vec{\omega_o})$ is *self-emission* from the surface. The rendering equation has a straightforward interpretation and tells us that reflected color (outgoing radiance) is computed by integration of BRDF-weighted lighting (incoming radiance $L_i(x, \vec{\omega_i})$) arriving at point x from all incoming directions $\vec{d\omega_i}$. Light emitted by the surface itself is then added to the reflected light, yielding total outgoing radiance $L_o(x, \vec{\omega_o})$.

2.2.3 BRDF Representation

BRDFs can be generally represented in two forms: tabulated and analytic. The tabulated representation is used for measured BRDFs, which are captured using real-world devices and describe complex materials such as velvet or silk. Uncompressed measured BRDFs require a significant storage space (due to their 4D nature and potentially high sampling resolution), but allow us to render images of real-world materials with great accuracy. Analytic BRDFs are empirical models of reflection, usually represented as functions. Their main advantage is practically zero storage space (since they can be evaluated directly from their analytic representation) and their main disadvantage is simplicity, which only allows for close-enough representation of a small class of real-world objects.

2.2.3.1 Diffuse reflection

Diffuse reflection is a mathematical abstraction and the most simple reflection model used in computer graphics. It tells us that the material reflects light uniformly in all outgoing directions (see Figure 2.4). It is also referred to as the *Lambertian* model. The outgoing radiance for diffuse surfaces can be computed as follows:

$$L_o(x, \vec{\omega_o}) = \int_{\Omega_+(\vec{N})} L_i(x, \vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) \cos\theta_i \; d\omega_i$$

using $f_r(x, \vec{\omega_i}, \vec{\omega_o}) = f_d(x)$ we obtain:

$$L_o(x, \vec{\omega_o}) = f_d(x) \int_{\Omega_+(\vec{N})} L_i(x, \vec{\omega_i}) \cos\theta_i \; d\omega_i = f_d(x) E(x)$$

The outgoing radiance for diffuse surface is therefore independent of the viewing angle and diffuse surface look the same no matter from where we are looking. Diffuse reflectance is also constant:

$$\rho_d(x) = \pi f_d(x)$$

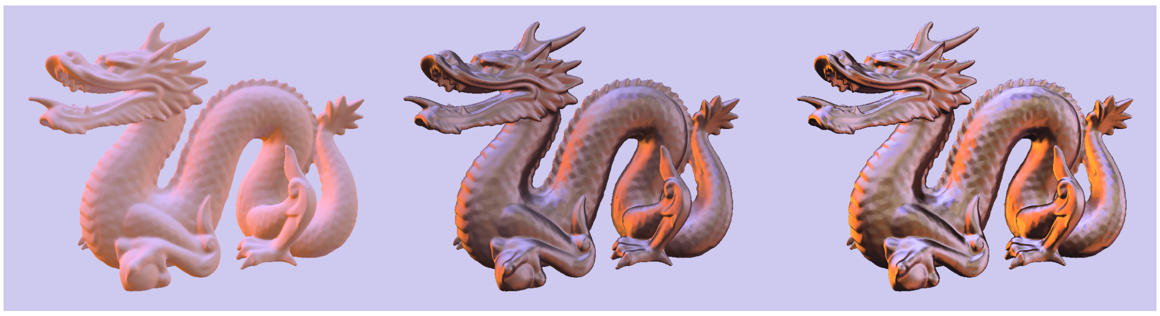


Figure 2.4: Different BRDF models. (Left) Diffuse (Middle) Blinn (Right) Anisotropic Ward. Images courtesy of Adrià Forés Herranz.

2.2.3.2 Specular reflection

The *Phong* reflection model enriches the diffuse model by adding a specular component. Its most suitable application is to shiny plastic surfaces (see Figure 2.4). The BRDF for Phong surfaces is defined as follows:

$$f_p(x, \vec{\omega_i}, \vec{\omega_o}) = \frac{\rho_d(x)}{\pi} + \frac{\rho_s(s+2)}{2\pi}(\vec{\omega_r} \cdot \vec{\omega_o})^s = k_d + k_s(\vec{\omega_r} \cdot \vec{\omega_o})^s$$

where k_d is the diffuse reflectance coefficient, k_s is the specular reflectance coefficient and s is surface *shininess*. Shininess is an appearance parameter, which defines the size of specular highlights. Phong reflections use an exact reflection direction $\vec{\omega_r}$, which can be computed using surface normal $\vec{N}$ as follows:

$$\vec{\omega_r} = 2(\vec{\omega_i} \cdot \vec{N})\vec{N} - \vec{\omega_i}$$

Another approach to the specular reflection is described by the *Blinn* model, which does not use the reflection vector at all. Blinn proposed to use the *half-angle* vector between incoming and outgoing directions:

$$\vec{H} = \frac{\vec{\omega_i} + \vec{\omega_o}}{||\vec{\omega_i} + \vec{\omega_o}||}$$

The Blinn BRDF is then computed using the half-angle vector and the normal $\vec{N}$:

$$f_b(x, \vec{\omega_i}, \vec{\omega_o}) = k_d + k_s(\vec{H} \cdot \vec{N})^{s'}$$

The differences between both models are visually almost indiscernible in most situations. However, when viewing from grazing angles, the Blinn model matches the reality a lot better [AHH08]. The shininess s' for the Blinn model is different from the shininess s for the Phong model and the following approximation holds:

$$(\vec{\omega_r} \cdot \vec{\omega_o})^s \approx (\vec{H} \cdot \vec{N})^{4s'}$$

2.3 Lighting

The computation of lighting in computer graphics amounts to evaluation of the rendering equation for each surface location. In general, we distinguish between *direct lighting* (i.e. light arriving at a surface location from directly visible light sources) and *indirect lighting* (i.e. light arriving at a surface location after bouncing off other surfaces), see Figure 2.5. The

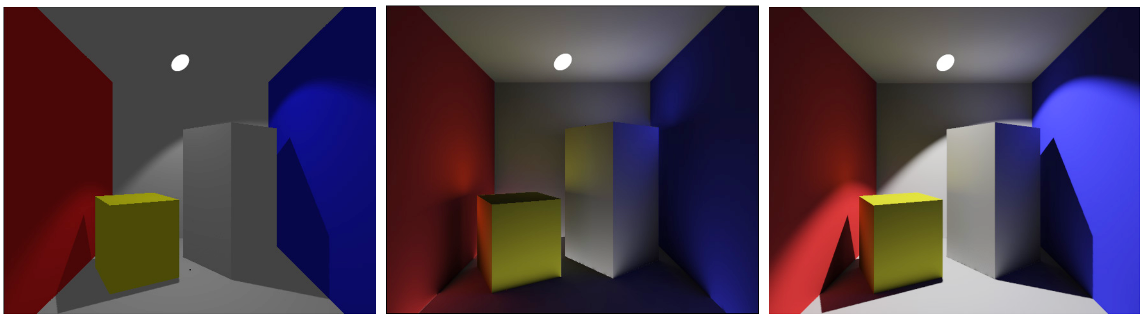


Figure 2.5: (Left) Direct illumination due to a spot light source. (Middle) Indirect illumination after one bounce. (Right) Combined direct and indirect illumination.

computation of indirect lighting is naturally more computationally expensive as it requires a simulation of different light paths.

A light source is an object that emits light and acts as a source of emitted radiance in our lighting simulations. There are several types of light sources in computer graphics, each serving a different purpose and posing different computational complexity. Before we proceed to descriptions of individual light source types, we will reformulate the rendering equation.

2.3.1 Area formulation of the rendering equation

The original rendering equation is formulated using integration over visible hemisphere. However, since computer graphics deals primarily with surfaces, it is advantageous to rewrite it using integration over infinitesimal areas. This will allow us to use simpler math later.

Looking at Figure 2.6, we see that solid angle $d\omega_i$ can be expressed as:

$$d\omega_i = \frac{\cos\theta^{'} dA^{'}}{||x' - x||^2}$$

where $\vec{\omega_i} = x' - x$ is the vector from location x to location x', also called the *light vector*. We are integrating over unit-area surface patches dA' and $\theta^{'}$ is the angle between the light vector and surface orientation (normal) at location x'. Plugging this into the rendering equation, we obtain:

$$\begin{aligned} L_o(x, \vec{\omega_o}) &= L_e(x, \vec{\omega_o}) + \int_{\Omega_+(\vec{N})} L_i(x, \vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) \cos\theta_i \; d\omega_i \\ &= L_e(x, \vec{\omega_o}) + \int_{A'} L_o(x', -\vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) V(x, x') \cos\theta_i \frac{\cos\theta^{'}}{||x' - x||^2} dA^{'} \\ &= L_e(x, \vec{\omega_o}) + \int_{A'} L_o(x', -\vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) V(x, x') G(x, x') dA^{'} \end{aligned}$$

where $G(x, x')$ is the *geometric term* and $V(x, x')$ is the *visibility function*. The geometric term describes the spatial relationship between the surface location being lit and the light source surface patch. The visibility function is a binary function that equals 1 if locations x and x' are mutually visible and 0 otherwise.

Using areas instead of hemispheres has many advantages. Assume that we are trying to compute incident radiance in a scene where geometry is represented as a polygon soup, i.e. a list of triangles, without any spatial data structure whatsoever. With hemispherical formulation, we would have to perform expensive computations for each infinitesimal solid

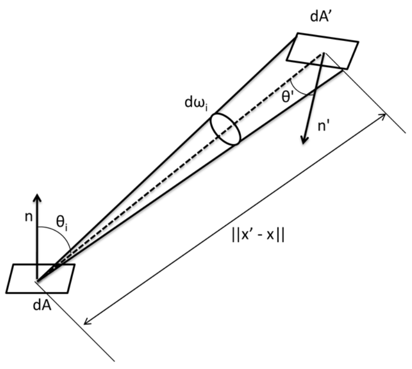


Figure 2.6: Relationship between differential solid angle $d\omega_i$ and differential area dA'.

angle $d\omega_i$. This could be accomplished by ray tracing (see Section 2.5.1) for example. The accuracy of this technique would depend on the number of rays traced. Such a computation would of course be very expensive.

On the other hand, with area formulation, we can simply iterate over all triangles and directly evaluate the geometric term. However, care must be taken to make sure that each triangle is small enough and can be considered to be an infinitesimal surface. Visibility information can again be established by ray tracing. The key difference is, however, that we do not trace rays for each direction $d\omega_i$, but only for directions that are guaranteed to intersect a surface.

2.3.2 Point light source

Point light sources emit light uniformly in all directions. They are the simplest model used in computer graphics and are defined by their spatial location and intensity I, see Figure 2.7.

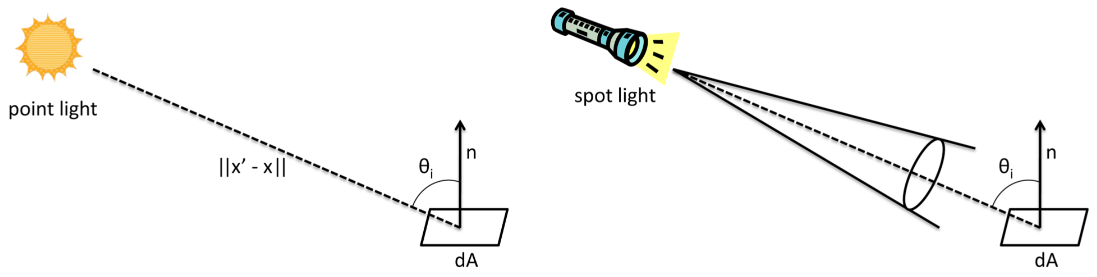


Figure 2.7: (Left) A point light source emits photons in all directions. (Right) A spot light emits photons only inside a cone-shaped volume.

Point light sources have zero area and irradiance from them is computed as follows:

$$E(x) = \int_{\Omega_+(\vec{N})} L_i(x, \vec{\omega_i}) \cos\theta_i \; d\omega_i = \frac{I\cos\theta_i}{||x' - x||^2}$$

where θ_i is the angle between surface normal at location x and the light vector. Assuming visible light source, this simplification allows us to compute outgoing radiance as:

$$\begin{aligned} L_o(x, \vec{\omega_o}) &= L_e(x, \vec{\omega_o}) + \int_{\Omega_+(\vec{N})} f_r(x, \vec{\omega_i}, \vec{\omega_o}) dE(x) \\ &= L_e(x, \vec{\omega_o}) + f_r(x, x' - x, \vec{\omega_o}) \frac{I\cos\theta_i}{||x' - x||^2} \end{aligned}$$

2.3.2.1 Spot light source

Spot light sources are a specialization of point light sources, which emit light in a cone. The formulas for irradiance and incident radiance are the same as for regular point light sources if the light vector $x' - x$ lies inside the cone. If not, the light contribution is zero. Besides this basic computation, some applications might apply light intensity falloff, i.e. decrease intensity with increasing angle between the cone central direction and light vector $x' - x$ (see Figure 2.7 for illustration).

2.3.3 Directional light source

Similar to point lights and spot lights, directional lights also have zero area. They emit light only in one direction and can be thought of as point lights infinitely far away (light emitted from point lights travels along nonparallel rays originating from one location; however, if we move the point light infinitely far away, the rays become parallel). A typical use case for directional lights is modeling of the sun, which is essentially a point light source very far away. Given light beam direction ω_d and light irradiance E_d, the rendering equation simplifies to:

$$L_o(x, \vec{\omega_o}) = L_e(x, \vec{\omega_o}) + f_r(x, -\omega_d, \omega_o) E_d \cos\theta_d$$

where θ_d is the angle between surface normal and light beam direction.

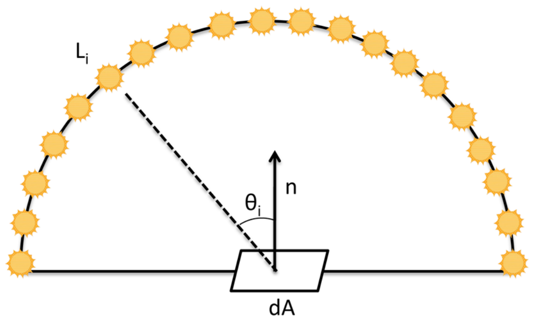


Figure 2.8: An environment light source can be visualized as a set of point light sources on the visible hemisphere around the surface normal. Each point light is considered to be infinitely far away.

2.3.4 Environment light source (Image-based lighting)

Environment maps are images of an environment that serve as light sources (see Figure 2.8). Pixel values in environment maps correspond to emitted radiance values for each incoming direction. A general assumption with environment maps is that the light source is infinitely far away, which makes incident radiance L_i be only a function of incoming direction $L_i = L_i(\vec{\omega_i})$. However, accurate evaluation of the rendering equation still requires integration over half of the pixels in the environment map, which makes environment maps computationally expensive. The rendering equation of image-based lighting is given as:

$$L_o(x, \vec{\omega_o}) = L_e(x, \vec{\omega_o}) + \int_{\Omega_+(\vec{N})} L_i(\vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) \cos\theta_i \; d\omega_i$$

There are four common representations of environment maps: latitude-longitude, sphere maps, cube maps and dual-paraboloid maps (see Figure 2.9 for examples of sphere maps

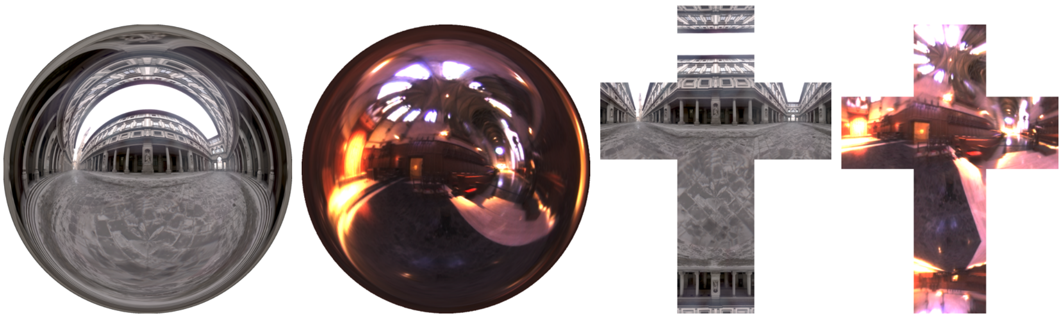


Figure 2.9: Examples of sphere maps and cube maps. Courtesy of Paul Debevec.

and cube maps). Latitude-longitude maps present the best visual representation for human beings, but suffer from non-linear sampling density around poles. Sphere maps allow us to use simple projection math to compute sampling coordinates, but suffer from artifacts due to magnified texels. Cube maps are the most GPU-friendly representation, because of their simple creation and sampling, but cause interpolation difficulties across cube map faces. This issue is addressed by dual-paraboloid maps.

2.4 Rendering

This section introduces the concepts of software and hardware rendering, their (dis)advantages and applicability to different rendering problems. We describe GPU architecture, shader programs and a few popular rendering packages.

2.4.1 Programmable Graphics Hardware

Graphics Processing Units (GPUs) have been around for several years. What first started as assembly-level programming, evolved into high-level C-like development of complex algorithms capable of running several orders of magnitude faster than traditional CPU programs. Furthermore, what was originally anticipated to become solely graphics-based hardware acceleration of slow CPU algorithms, turned into a general platform for parallel vector-based operations. In this section we describe the architecture and programming capabilities of GPUs.

2.4.1.1 Previous generation of graphics hardware

Programmable graphics devices are SIMD chips with high-performance vector units. Generally, data is fed to GPUs as a stream of per-vertex data structures (for example, a typical vertex structure would contain vertex position in object coordinate system, vertex normal and several sets of texture coordinates). At the beginning of the pipeline, the GPU distributes vertex data to several *vertex shader* units, which are hardware processors with dedicated instructions for vector operations.

A GPU contains several vertex shader units, thus allowing for parallel processing of several vertices. Vertex shader units are controlled by user-developed vertex programs, which process every vertex passed in from the application. Typical tasks that vertex programs perform are coordinate transformations, lighting computations or even primitive culling. In any case, every vertex program sets up data for further processing in the pipeline by filling in appropriate hardware registers.

After a vertex has been processed in a vertex shader unit, primitive assembly begins. Three vertices are connected to form a triangle and the triangle's area is scan-converted to fragments. Scan-conversion (also called rasterization) is a process that locates all fragments in a triangle's area and prepares them for further processing.

In the next step, every fragment is passed down the pipeline to a *fragment shader*. A fragment shader's only task is to compute a fragment's final color and decide whether or not it should be displayed on the screen. Fragment shaders are executed in fragment shader units and are developed by application developers. Again, there are several fragment shader units in a GPU (generally more than vertex shader units) that allow the device to achieve a high level of parallelism.

2.4.1.2 Latest generation of graphics hardware

Microsoft's Direct3D 10 and an OpenGL 2.0 extension introduced a significant difference to the graphics pipeline, which resulted in non-trivial hardware modifications and redesigns. In the previous generation, primitive assembly was a process executed after vertices have been processed in the vertex shader, but before fragments were shaded in the fragment shader. Given a number of vertex structures that were passed in to the vertex shader, the number of

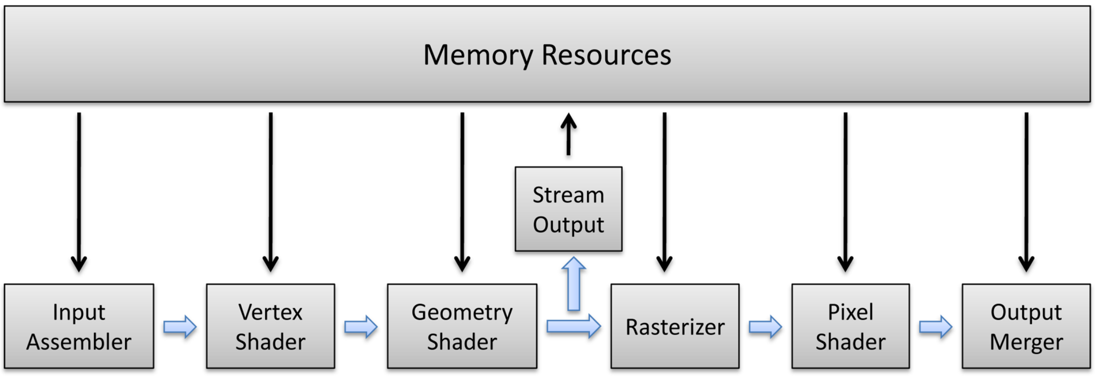


Figure 2.10: Direct3D 10 GPU pipeline.

assembled primitives (triangles) was given as well. Hence, the application developer had no control over the process of assembling primitives and was confined by limitations imposed by a particular hardware architecture.

In the latest generation, a *geometry shader* is a new kind of program that controls how primitives are assembled. Vertices processed in the vertex shader are passed down to a geometry shader residing in a geometry shader unit (also termed geometry processor). In this place, new triangles can be spawned. One might wonder if such an extension serves any important purposes and for the sake of completeness, we bring to attention applications such as subdivision surfaces. In a Direct3D 10 class hardware, depending on surface curvature, the geometry shader may decide to subdivide current surface triangles by inserting additional triangles and thus provide a higher level of visual detail.

Geometry shaders are again written by application developers and their output is a set of newly spawned triangles. Primitives emitted from the geometry shader are then rasterized into fragments and shaded in the fragment shader in the same way as it was done in the previous generation of graphics hardware. We refer the reader to Figure 2.10 for a block diagram of the Direct3D 10 pipeline.

The advent of geometry shaders increased the number of different types of vector units to three (vertex, geometry and fragment processors) and made the process of designing and manufacturing graphics chips more complicated than before. Furthermore, efficient utilization of all vector units became a lot harder to achieve — remember that some parts of the graphics pipeline might be stalled while waiting for previous stages to complete their execution (we used to differentiate between vertex-heavy and pixel-heavy situations, with the latter being more common). The problem is due to generally different running times of vertex and fragment shaders and subsequent stalls in the pipeline.

One solution was to unite all three processors into a single unit, with the same hardware architecture and the same instruction set (vertex and fragment processor used to have different hardware architectures and different instruction sets before) and devise a smart scheduling mechanism for forwarding data to units that are not utilized. This architecture is called *unified shader architecture* and is now commonly used across all major GPU vendors.

2.4.2 Software Rendering

Even though we first mentioned the concept of shader programs in relation to GPUs, shaders were introduced by Pixar in their RenderMan specification [Pix]. In software rendering (such as in an implementation of the RenderMan specification), shaders are used to describe how object materials interact with light. For example, a simple shader is able to compute outgoing radiance for a diffuse material illuminated by a point light source. Software shaders are more general and more robust than their hardware counterparts, because they allow callbacks into application code, such as callbacks into the raytracing engine (see Section 2.5.1). Hardware shaders, due to limited hardware capabilities and strong specialization, are more restricted.

Shaders are also widely used in modeling packages, such as Maya or 3D Studio MAX, to describe materials. These come with libraries of pre-made shaders for most common material types (such as diffuse, Blinn, etc.) and the artist applies them to objects in their scene. Users can create shader trees — hierarchical structures that allow them to combine multiple shaders and create more complex material types (see Figure 2.11). Shader trees can (and usually do) become quite complicated and their evaluation can present a significant burden for the rendering engine. Users are of course allowed to create their own shaders, either software or hardware-based. The former will be evaluated by the software renderer and the latter will be executed on GPUs.

Shaders have several advantages over fixed function pipeline architectures. In hardware rendering, a fixed function pipeline architecture is programmed by the application developer

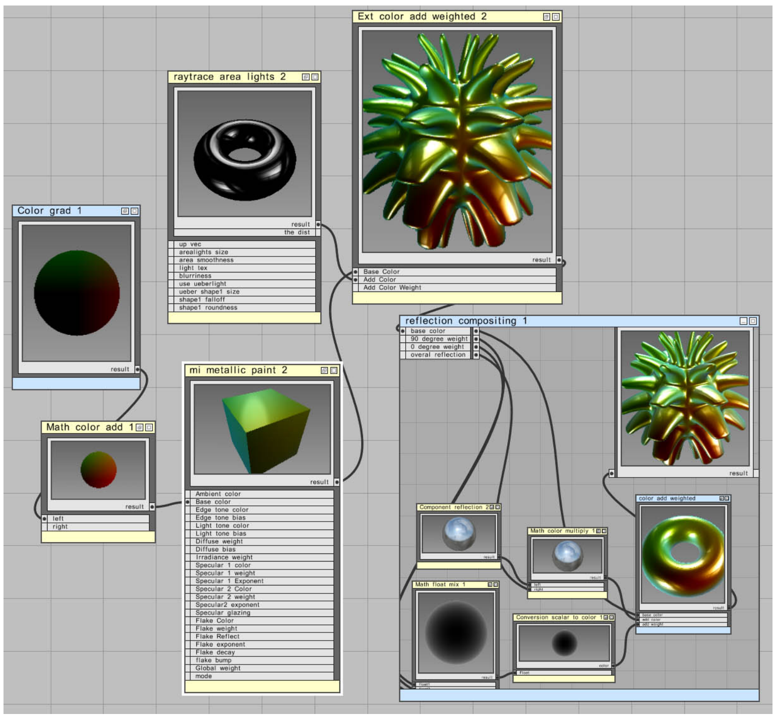


Figure 2.11: A moderately complex shading network in Mental Mill.

as a state machine. Each vertex and fragment goes through the same execution blocks, whose functions are hardwired into the hardware. These blocks are controlled by switching them on or off and/or by passing parameters. In software rendering, a fixed function pipeline architecture is usually a black box rendering engine, which can be parameterized by a configuration file.

The greatest advantage of shader-based architectures is their flexibility. In hardware, we can write a separate program for each execution block. In software, we can write plugins for the black-box renderer and completely change its execution model. Of course, giving full access to a system can sometimes be counter-productive. Even today, some parts of GPUs are not programmable (such as the merger stage, which performs blending). Similarly in software, the core of a ray tracer is often protected from modifications and can only be accessed through a dedicated API.

2.5 Global Illumination

Photo-realistic computer imagery requires physically-accurate simulation of light transport. The path of light emitted from a light source can change multiple times before it reaches the eye of the observer due to interactions of light and surfaces in the scene. This section describes three popular concepts used for simulation of global illumination effects.

In general, there is an infinite number of paths from the light source to the observer. Photons can bounce back and forth, they can be refracted, absorbed or scattered. Different global illumination algorithms deal with different global illumination effects, i.e. focus on specific photon trajectories. The most important effects due to global illumination are interreflections and secondary shadows.

2.5.1 Ray Tracing

Ray tracing describes a class of techniques, which compute global illumination by tracing rays along paths traveled by light. Ray tracing setup starts with shooting of rays through each pixel in the image plane (these rays are called *primary rays* as illustrated in Figure 2.13).

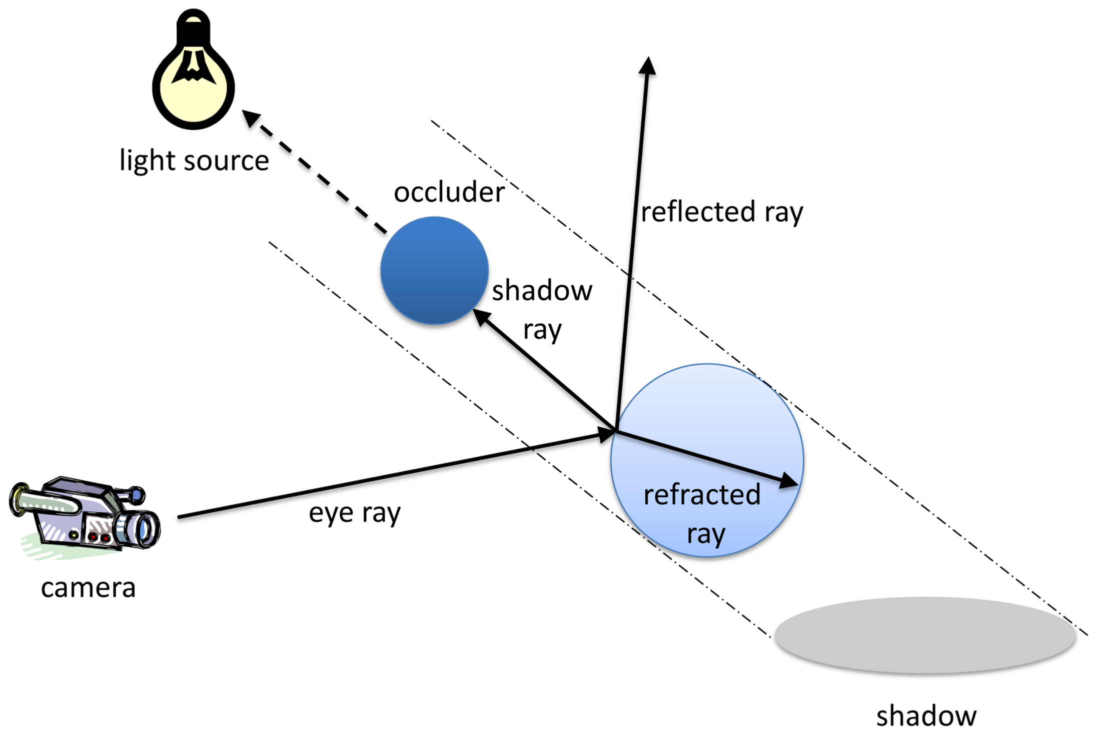


Figure 2.12: Types of rays used in ray tracing engines. Eye (primary) rays originate from the camera and look for first intersection with scene geometry, where new types of rays can be spawned. Shadow rays are traced toward light sources to establish visibility. For refractive objects, new refracted rays are spawned. Reflected rays are computed according to the law of reflection.

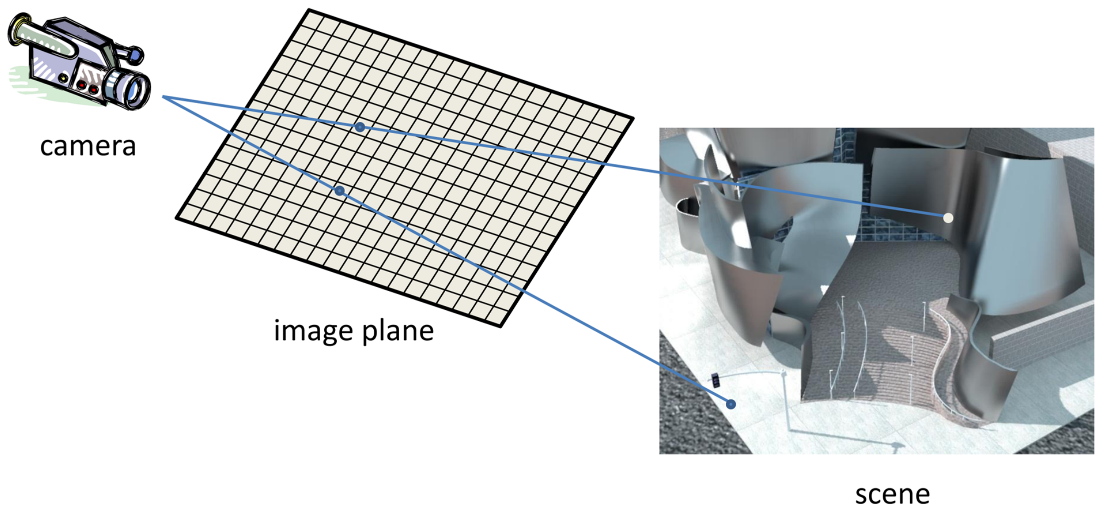


Figure 2.13: In ray tracing, pixel color is computed by shooting rays from the camera and finding intersections with scene geometry. Image courtesy of Jaroslav Křivánek.

Color of each pixel is computed by evaluating the rendering equation at scene locations hit by primary rays.

A brute force approach to ray tracing would shoot and trace as many rays as possible. For example, if a ray hits a surface with a fairly complex BRDF (first light bounce), a multitude of rays is required to accurately sample all outgoing directions (hemispherical sampling). If the scene is complex and multiple bounces are considered, the number of rays explodes exponentially. A great body of ray tracing research focuses on minimizing the number of rays required for noise-free rendering.

A commonly used alternative to hemispherical sampling relies on Monte Carlo sampling [DBB02]. The value of the rendering integral is estimated by evaluating the expected value of a random variable (termed the estimator). Generally, the sampling domain is the visible hemisphere and samples are generated according to some probability distribution function $p(\omega)$. The sampling itself requires evaluation of incident radiance $L_i(x, \omega)$ (according to formulation given in Equation 2.2.2), which leads to a recursive algorithm. Each step of the recursion corresponds to one bounce of light.

The problem with this simplistic formulation is that a light path will generate a non-zero radiance contribution if and only if it hits a light source in the scene. However, due to the random nature of sampling and the fact that light sources are generally very small, the majority of radiance evaluations will be zero and the resulting image will be mostly black. This problem can be addressed by a different sampling strategy, which also considers positions of light sources. *Shadow rays* are special rays sent to light sources from each location where incident radiance needs to be evaluated (see Figure 2.12).

It is a common practice to separate the computation of direct and indirect lighting when rendering a full global illumination solution. While shadow rays provide a good solution to the sampling problem for direct lighting, other techniques are necessary to obtain accurate results for indirect illumination. These techniques include BRDF-proportional importance sampling (to account for increased radiance contributions around lobes for non-diffuse BRDFs) and cosine-weighted sampling (to account for increased radiance contributions from directions closer to the surface normal) [DBB02].

The main disadvantage of ray tracing-based (sometimes referred to as path tracing as well) algorithms is their slow convergence. They typically require a huge number of rays to

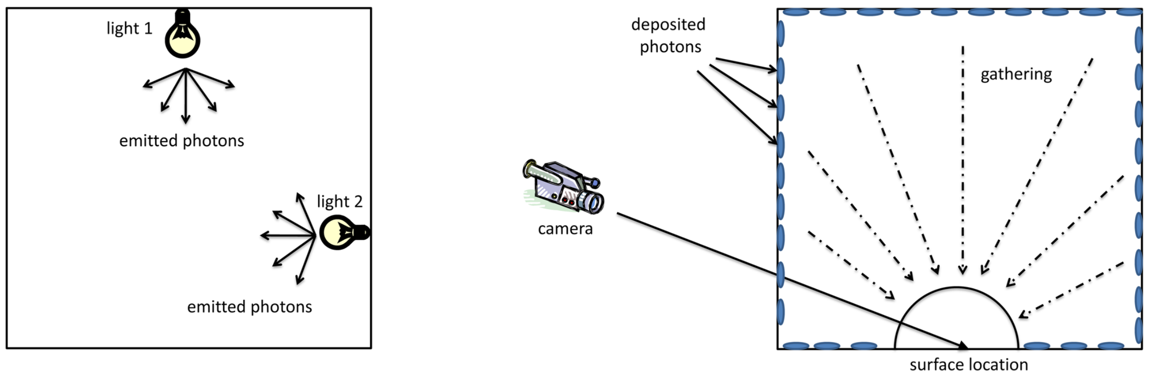


Figure 2.14: (Left) First pass of photon mapping. Photons are emitted from light sources, propagated through the scene and deposited on receiver surfaces. (Right) Second (gathering) pass. Instead of performing expensive hemispherical sampling, incident radiance is gathered from photons deposited in the first pass.

be traced in order to generate a noise-free image. For example, rendering of the image in Figure 2.13 took 30 hours and used 8 CPU cores.

2.5.2 Photon Mapping

Photon mapping is an example of a multi-pass bidirectional rendering method. Unlike the basic ray tracing algorithm, which only deals with light paths originating from the camera, bidirectional methods also use light paths originating from light sources. Both of these are computed in separate rendering passes and finally combined together.

The first pass of the photon mapping method involves emitting photons from light sources. Each photon carries flux information and travels along light paths generated according to scene information (i.e. photons are reflected, transmitted or absorbed). Photons can bounce multiple times to account for multiple light bounces and at the end of their path (determined by imposing a maximum path length or by the Russian roulette [DBB02]), the energy they carry is stored in a global data structure called the *photon map*. The photon map is a cache data structure (such as a kd-tree), designed for fast lookups based on photon positions. The first pass of the photon mapping method essentially traces light paths originating from light sources.

Once the photon map has been generated, the second pass of the rendering algorithm takes care of the actual image formation. The energy deposited by photons can be used to estimate the irradiance at any surface point (and subsequently outgoing radiance after multiplication with the BRDF), but this approach is not accurate since it blurs the radiance values. Instead, the second pass of the rendering algorithm relies most often on ray tracing. Ray tracing computes the direct illumination by the algorithm described in Section 2.5.1 and indirect illumination is computed by hemispherical sampling. However, in contrary to classical ray tracing, the sampling does not require recursion anymore, because the incident radiance term $L_i(x, \omega)$ is estimated by looking up values in the photon map.

Photon mapping can efficiently handle visual phenomena such as caustics, participating media or subsurface scattering. For detailed information about photon mapping methods, we refer the reader to the book [Jen01].

2.5.3 Radiosity Methods

Another approach to computing global illumination is based on computing transport of radiosity between finite elements (patches) in a scene. Recall that radiosity (or radiant exitance) is a radiometric quantity defined as the ratio of outgoing flux per unit area:

$$B = \frac{d\Phi_o}{dA}$$

Radiosity methods formulate a system of linear equations, which describes the equilibrium state of energy transfer. Solution of this system gives us values of radiosity of each patch. Given patch radiosity due to reflection B_i, patch radiosity due to self-emission B_{ei}, patch diffuse albedo ρ_i and form factors F_{ij}, the radiosity system of equations is given as:

$$B_i = B_{ei} + \rho_i \sum_j F_{ij} B_j$$

The classic method can then be described by the following four steps:

1. Discretization of input geometry into input patches i.
2. Computation of form factors F_{ij}.
3. Numerical solution of radiosity system of linear equations.
4. Rendering of scene surfaces.

Since radiosity methods compute light transport between patches, the quality of the solution will largely depend on their sizes. Care must be taken to have small enough patches to capture higher-frequency effects such as sharp shadows. On the other hand, too many small patches will result in long computation times.

The main problem of radiosity methods is the computation of form factors. Form factors are mathematical quantities that describe how much energy is transferred between two surfaces S_i and S_j. Their exact computation is given as:

$$F_{ij} = \frac{1}{A_i} \int_{S_i} \int_{S_j} \frac{\cos(R_{xy}, N_x)\cos(-R_{xy}, N_y)}{\pi r_{xy}^2} V(x, y)\ dA_x\ dA_y$$

where $V(x, y)$ is the visibility term between location x on surface S_i and surface location y on surface S_j and R_{xy} is the radius vector between locations x and y.

If we look back at the radiosity system of equations, we see that radiosity B_i is a sum of two contributions: the first contribution is the self-emitted radiosity and the second contribution is the fraction of irradiance that gets reflected. The form factor F_{ij} indicates what fraction of the irradiance on patch i originates at patch j.

Once form factors have been computed, the algorithm proceeds with numerical computation of the system of equations. The number of equations can reach hundreds of thousands for a complex scene, but since the system of equations is well-behaved, methods such as Jacobi or Gauss-Seidel usually converge very fast. Once patch radiosity values have been computed, they can be used to compute surface diffuse reflectance values and rendered using a GPU from any viewpoint.

2.5.4 Linearity of Light Transport

It can be seen from the previous three algorithms that computation of a solution to the global illumination problem is a recursive process. At each surface location, final outgoing radiance is a function of incident radiance. However, incident radiance from some direction ω_i is again reflected radiance from some other surface location in outgoing direction $-\omega_i$. It is sometimes very convenient to express this recursion using transport operators. The rendering equation can be written as

$$\begin{aligned} L_o(x, \vec{\omega_o}) &= L_e(x, \vec{\omega_o}) + \int_{\Omega_+(\vec{N})} L_i(x, \vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) \cos\theta_i \; d\omega_i \\ &= L_e(x, \vec{\omega_o}) + \int_{\Omega_+(\vec{N})} L_o(y, -\vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) \cos\theta_i \; d\omega_i \end{aligned}$$

where $y = r(x, \vec{\omega_i})$ is the first visible point from location x in direction $\vec{\omega_i}$. Also,

$$L = L_e + \mathcal{T} L \tag{2.1}$$

where $\mathcal{T}$ is the transport operator defined as:

$$(\mathcal{T}f)(x,\vec{\omega_o}) = \int_{\Omega_+(\vec{N})} f(r(x,\vec{\omega_i}), -\vec{\omega_i}) f_r(x,\vec{\omega_i},\vec{\omega_o}) \cos\theta_i \; d\omega_i$$

The problem with Equation 2.1 is that the unknown L is on both sides. The solution to this equation exists if and only if the operator T is invertible and is given by:

$$S = (\mathcal{I} - \mathcal{T})^{-1} = \sum_{i=0}^{\infty} \mathcal{T}^i = \mathcal{I} + \mathcal{T} + \mathcal{T}^2 + ...$$

where $\mathcal{I}$ is the identity operator. This expansion is called the *Neumann* series and its application to incident radiance has a physical interpretation of light transport:

$$L = L_e + \mathcal{T}\ L_e + \mathcal{T}^2\ L_e + ...$$

This equation expresses final outgoing radiance as the sum of self-emitted radiance L_e, reflected radiance after one bounce $\mathcal{T}\ L_e$, reflected radiance after two bounces $\mathcal{T}^2\ L_e$, etc.

The key observation here is that the operator $\mathcal{T}$ is *linear*. In the context of global illumination and lighting, this property allows us to express reflected radiance due to a set of light sources as a sum of reflected radiances computed individually for each light. As we shall see in Chapter 2.6.3, methods based on precomputed radiance transfer (PRT) rely on this property very heavily.

2.6 Data Representation

Data in computer graphics can be expressed and stored in a variety of different bases. The natural basis for data in computer graphics is the pixel basis, used for example for uncompressed diffuse textures. The pixel basis has, however, several disadvantages, the most prominent one being large storage requirements. On the other hand, other bases such as spherical harmonics or wavelets allow applications to significantly reduce storage requirements while preserving all necessary details. Other applications of non-pixel bases include fast computations of vector dot product, or non-linear compression (see later). This section describes the two aforementioned bases — spherical harmonics and wavelets and explains their role in relighting applications.

2.6.1 Spherical Harmonics

Spherical harmonics are orthonormal basis functions defined by the following equation:

$$Y_l^m(\theta,\phi) = K_l^m e^{im\phi} P_l^{|m|}(\cos\theta), l \in \mathcal{N}, -l \leq m \leq l$$

This equation is defined in the complex domain and l is the order of the spherical harmonic function, m is the band index of the spherical harmonic function and K_l^m is the normalizing term defined as

$$K_l^m = \sqrt{\frac{(2l+1)}{4\pi}\frac{(l-|m|)!}{(l+|m|)!}}$$

and $P_l^{|m|}$ is the associated Legendre polynomial defined as:

$$\begin{aligned} P_{l+1}^m(x) &= P_{l-1}^m(x) - (2l+1)\sqrt{1-x^2}P_l^{m-1}(x) \\ P_0^0 &= 1 \end{aligned}$$

Spherical harmonics are typically used in the real-valued version, since complex numbers incur additional computational overhead:

$$\begin{aligned} y_l^m(\theta,\phi) &= \begin{cases} \sqrt{2}Re[Y_l^m(\theta,\phi)] & m > 0 \\ \sqrt{2}Im[Y_l^m(\theta,\phi)] & m < 0 \\ Y_l^0(\theta,\phi) & m = 0 \end{cases} \\ &= \begin{cases} \sqrt{2}K_l^m \cos(m\phi)P_l^m(\cos\theta) & m > 0 \\ \sqrt{2}K_l^m \sin(m\phi)P_l^{-m}(\cos\theta) & m < 0 \\ K_l^0 P_l^0(\cos\theta) & m = 0 \end{cases} \end{aligned}$$

Spherical harmonics are basis functions for functions defined over a sphere. They have global support and hence are more useful for representation of low-frequency effects such as indirect or distant lighting [SKS02]. A significant disadvantage of spherical harmonics is the mathematical and computational complexity of spherical harmonics rotation.

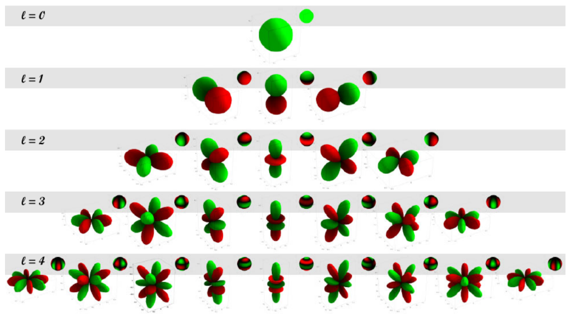


Figure 2.15: Spherical harmonics visualization courtesy of Robin Green.

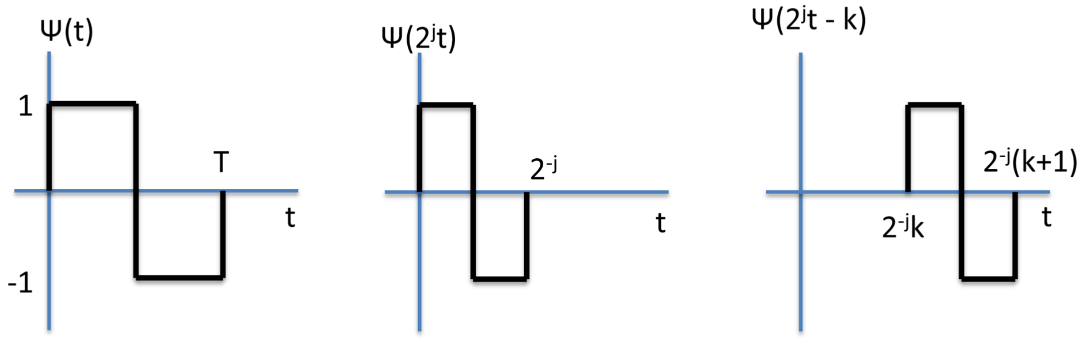


Figure 2.16: Haar mother wavelet function (Left) with period T is shifted and scaled to generate the Haar basis (Middle) and (Right).

2.6.2 Wavelets

Another disadvantage of spherical harmonics is their lack of localization in space. Spherical harmonics have global support, which makes them inefficient for handling of sharp peaks and edges. Wavelets are an example of basis functions with local support, which allows them to match the input as close as possible. Better matching, of course, minimizes the approximation error. Wavelets, as opposed to spherical harmonics, are localized in both space and frequency.

There are many different versions of wavelet basis, with the *Haar* basis being the most commonly used one. All wavelet basis functions are scaled and translated versions of the *mother wavelet* function $\Psi(t)$. The mother wavelet is defined as:

$$\Psi(t) = \begin{cases} 1 & \text{for} \quad 0 \leq t < 0.5 \\ -1 & \text{for} \quad 0.5 \leq t < 1 \\ 0 & \text{otherwise} \end{cases}$$

Haar basis functions $\Psi_{j,k}(t)$ are indexed by their *scale* j and *shift* k and generated as follows (see Figure 2.16):

$$\Psi_{j,k}(t) = \sqrt{2^j}\Psi(2^j t - k) \text{ for } j = 0, 1, ... \text{ and } k = 0, 1, ..., 2^j - 1$$

Each Haar basis function is normalized ($\sqrt{2^j}$ is the normalization coefficient) and any two Haar basis functions are orthogonal. The Haar basis can be easily generalized to a two-dimensional dataset, such as a matrix, by first applying the 1D version on matrix rows and then on matrix columns. This is referred to as the *standard decomposition.* Another way to apply the Haar basis in 2D is by alternating operations on rows and columns, always halving the size of the input by 2. This technique is referred to as *non-standard decomposition* [SDS94].

Algorithm 1 demonstrates a simple encoder for a discrete 1D spatial function. The input to the algorithm is a vector of real numbers (note, the vector's length is assumed to be a power of 2). The algorithm has $log_2 n$ passes where n is the length of the input vector. In each pass, two quantities are computed. First, averages of two adjacent vector elements are computed in the first for-loop and stored in a temporary vector. Second, the differences between two adjacent vector elements are computed and stored in the second half of the original vector. Finally, the temporary array is copied to the first half of the input vector. Each pass cuts the number of processed elements by half and the algorithm completes when n reaches 1.

Algorithm 1: 1D Haar in-place encoder.

```
Data: vector — length is a power of 2
Data: n = length(vector)
while $n >= 2$ do
    n = n / 2;
    for $i \in < 0, n-1 >$; $i$++ do
        tmpArray[i] = $\frac{\text{vector[2 * i] + vector[2 * i + 1]}}{\sqrt{2}}$;
    end
    for $i \in < n-1, 0 >$; $i$−− do
        vector[n + i] = $\frac{\text{vector[2 * i] - vector[2 * i + 1]}}{\sqrt{2}}$;
    end
    for $i \in < 0, n-1 >$; $i$++ do
        vector[i] = tmpArray[i];
    end
end
```

A sample decoder is demonstrated in Algorithm 2. The decoder also has $\log_2 n$ passes. Each pass reconstructs a part of the original input vector. The reconstruction works by iterating over wavelet coefficients in the inner for-loop. To reconstruct two values of the original vector, the algorithm fetches their average and difference and uses simple algebra to obtain the original values.

Algorithm 2: 1D Haar in-place decoder.

```
Data: vector — length is a power of 2
Data: n = length(vector)
levels = $log_2 n$;
for $i \in < 0, levels - 1 >$ do
    currentSize = $2^i$;
    for $c \in < 0, currentSize >; c = c + 2$ do
        x = vector[c / 2];
        y = vector[(c + currentSize) / 2];
        tmpArray[c] = $\frac{x+y}{\sqrt{2}}$;
        tmpArray[c + 1] = $\frac{x-y}{\sqrt{2}}$;
    end
    for $c \in < 0, currentSize >; c{+}{+}$ do
        vector[c] = tmpArray[c];
    end
end
```

2.6.3 Application to Light Transport

Basis functions (namely Haar wavelets in this section) are often used to represent and compress static information about light transport. As an example, in relighting applications, we tend to fix geometry and materials and allow the user to change lighting while providing real-time rendering feedback. Without loss of generality, recall the rendering equation without the emissive term and with explicitly included visibility term:

$$L_o(x, \vec{\omega_o}) = \int_{\Omega_+(\vec{N})} L(x, \vec{\omega_i}) f_r(x, \vec{\omega_i}, \vec{\omega_o}) V(x, \vec{\omega_i}) \cos\theta_i \; d\omega_i$$

Assume now that we fix the viewpoint (as it is often the case in relighting applications), making outgoing direction ω_o a function of location x. We can now define a transport operator:

$$T(x,\vec{\omega_i}) = f_r(x,\vec{\omega_i},\vec{\omega_o}(x))V(x,\vec{\omega_i})\cos\theta_i$$

The rendering equation becomes:

$$L_o(x) = \int_{\Omega_+(\vec{N})} L(x,\vec{\omega_i})T(x,\vec{\omega_i})\ d\omega_i = \int_{\Omega_+(\vec{N})} L(x,\vec{\omega_i})f_r(x,\vec{\omega_i},\vec{\omega_o}(x))V(x,\vec{\omega_i})\cos\theta_i\ d\omega_i$$

This formulation tells us that outgoing radiance at a given location and seen from a given viewpoint can be simply computed by integration of the product of incident lighting, visibility and BRDF.

In a relighting application (where the viewpoint is fixed), we usually precompute the transport operator in the pixel basis (by using ray tracing for example) and then project to a different basis (we consider Haar wavelets in the following). The evaluation of the rendering equation happens at runtime. The projection to a different basis is performed by the inner product with basis functions for both the transport operator and incident lighting:

$$\mathbf{T_i}(x) = \int_{\Omega_+(\vec{N})} T(x,\vec{\omega_i})\Psi_i(x,\vec{\omega_i})\ d\omega_i$$
$$\mathbf{L_j}(x) = \int_{\Omega_+(\vec{N})} L(x,\vec{\omega_i})\Psi_j(x,\vec{\omega_i})\ d\omega_i$$

where $\Psi_i(x,\vec{\omega_i})$ and $\Psi_j(x,\vec{\omega_i})$ are Haar wavelet basis functions.

We can now rewrite the rendering equation in terms of basis functions and obtain:

$$\begin{aligned}
L_o(x) &= \int_{\Omega_+(\vec{N})} T(x, \vec{\omega_i}) L(x, \vec{\omega_i}) \, d\omega_i \\
&= \int_{\Omega_+(\vec{N})} (\sum_i \mathbf{T_i}(x) \Psi_i(x, \vec{\omega_i}))(\sum_j \mathbf{L_j}(x) \Psi_j(x, \vec{\omega_i})) \, d\omega_i \\
&= \sum_i \sum_j \mathbf{T_i}(x) \mathbf{L_j}(x) \int_{\Omega_+(\vec{N})} \Psi_i(x, \vec{\omega_i}) \Psi_j(x, \vec{\omega_i}) \, d\omega_i \\
&= \sum_i \sum_j \mathbf{T_i}(x) \mathbf{L_j}(x) \delta_{ij} = \sum_i \mathbf{T_i}(x) \mathbf{L_i}(x) \\
&= \mathbf{T} \cdot \mathbf{L}
\end{aligned}$$

In other words, the integral simplifies to a dot product of two vectors, wavelet coefficient vectors of the static transport operator $\mathbf{T_i}(x)$ (precomputed prior to rendering) and lighting $\mathbf{L_j}(x)$. Relighting applications allow the user to change lighting dynamically and after each change, the lighting is projected to the Haar wavelet basis, yielding coefficients $\mathbf{L_j}(x)$. Both the projection and the dot product can be evaluated very efficiently on current graphics hardware. The transport operator is very often stored as a matrix with rows corresponding to locations x and columns corresponding to basis functions.

Material (BRDF) editing applications take an approach similar to relighting applications, but define the transport operator differently. Generally, they fix the geometry and lighting and allow the user to modify the BRDF at runtime.

CHAPTER 3

PREVIOUS WORK

Our work is related to a large body of previous and recent research carried out in the fields of lighting design, material and appearance design and global illumination. In this chapter, we review the most significant publications related to our research on a global level. We describe some techniques in more detail, giving the reader the proper context required for understanding the contributions in our work. In the following chapters, we discuss how our work improves on related work in the context of each individual contribution. At the end of this chapter, we summarize our design goals.

3.1 Rendering and Global Illumination

We start our review with popular approaches used for rendering with global illumination. As already mentioned in the previous chapter, global illumination methods rely most commonly on ray tracing or photon mapping.

Irradiance caching by Ward et al. [WRC88] is a common method for rendering of diffuse interreflections. Indirect illumination at any location in the scene is represented as irradiance (see Section 2.1), thus independent of viewer location. Irradiance is computed by hemispherical sampling, which is a rather costly operation. Irradiance caching takes advantage of the fact that spatial variation of irradiance is very slow and performs hemispherical sampling only for a sparse set of locations. Computed values are stored in a cache and used to compute all other irradiance values by interpolation. Irradiance at location p is interpolated as:

$$E_S(p) = \frac{\sum\limits_{i \in S}(E_i + (n_i \times n) \cdot \nabla_r E_i + (p - p_i) \cdot \nabla_t E_i) w_i(p)}{\sum\limits_{i \in S} w_i(p)} \tag{3.1}$$

where p_i and n_i are locations and normals corresponding to already-cached irradiance value E_i. Weights corresponding to cached irradiance value are computed as:

$$w_i(p) = (\frac{||p - p_i||}{R_i} + \sqrt{1 - n \cdot n_i})^{-1} \tag{3.2}$$

where R_i is the harmonic mean of distances to objects visible from p_i. Irradiance values are interpolated using samples from set S defined as:

$$S = \{i | w_i(p) > \frac{1}{a}\} \tag{3.3}$$

where a is defined by the user. $\nabla_r E_i$ and $\nabla_t E_i$ are rotational are translational gradients.

The main disadvantage of irradiance caching is that it only support diffuse interreflections. Radiance caching (Křivánek et al.[KGP05]) addresses this issue and caches the direction-aware radiance instead. Incoming radiance at any scene location is stored as a coefficient vector in spherical or hemispherical harmonics [GKP04], which reduces the rendering equation to a dot product of two coefficient vectors. Radiance interpolation is performed by direct interpolation of spherical harmonics coefficients.

The idea of compressed light transport is elaborated by Ng et al. [NRH03] who use 2D Haar wavelets to compress light transport from an environment map. They fix geometry and scene BRDFs, precompute a transport operator and apply non-linear compression by cutting off wavelet coefficients of lower magnitudes. A similar approach is pursued by Sloan [SKS02] who uses spherical harmonics instead of wavelets. Ng et al. [NRH03] show that wavelets yield higher-quality compression than spherical harmonics.

The wavelet formulation is later extended to support static glossy objects with varying viewpoint [NRH04]. This is accomplished by analyzing triple product integrals, involving visibility, lighting and BRDF. Interactive performance is achieved by a novel sub-linear rendering algorithm in the Haar basis. Sun et al. [SM06] extend this approach and consider multi-function product integrals that allow them to render dynamic glossy objects.

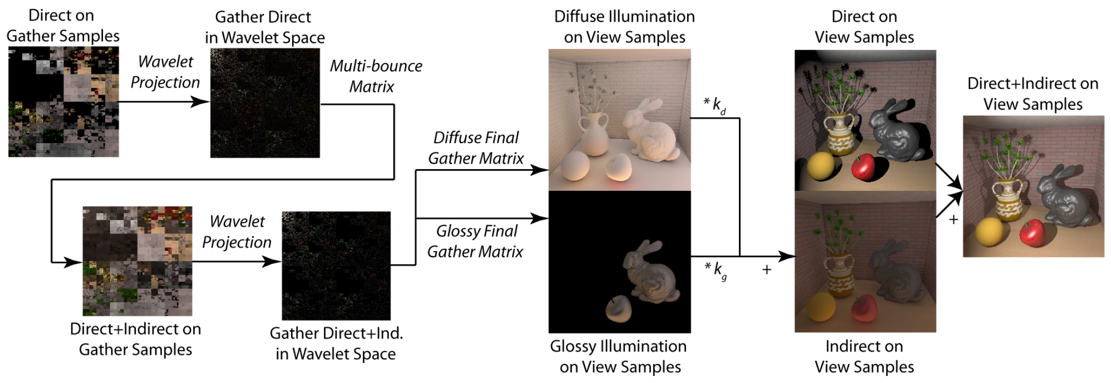
Finally, Sun et al. [SR09] show how affine transformations can be performed directly in the wavelet space. This allows for relighting using near-field illumination, which can be expressed by translating and scaling the source radiance in the environment map. Haar wavelets are used as the implementation domain, with additional applications in wavelet importance sampling and image processing.

Wavelet-based rendering approaches achieve data compression by projections on basis functions and cutting off values of low magnitudes. In other words, if incident radiance from a certain direction is negligible, its contribution is discarded. The Lightcuts framework [WFA05] introduces an alternative technique that deals with problems requiring up to hundreds of thousands of lights. The authors develop a sublinear rendering technique that creates clusters of lights based on similarity and uses one representative light to compute radiance contributions from all lights in the cluster. Total incident radiance for each shaded point is then estimated by computing cuts through a light tree. The cuts are computed according to a perceptual metric and are the equivalent to the nonlinear wavelet compression. The advantage of the Lightcuts approach is that their metric is perceptual, while wavelets discard lights based solely on the magnitude of their radiance contributions. Lights have also been extended to higher dimensions [WAB06], enabling effects such as motion blur or depth of field.

3.1.1 Direct to Indirect Transfer (DTI)

Hašan et al. [HPB06] present a method for real-time relighting with global illumination. This method is again based on wavelets and we review it in greater detail (see Figure 3.1), because it is the basis of our rendering pipeline described in Chapter 3.4.

The DTI method is based on precomputation of two light transport operators. The final gather operator describes first-bounce indirect light originating from a set of gather points (the terms gather points and gather samples are used interchangeably in this thesis) and ending at screen pixels. The multi-bounce operator describes light bounces originating and ending at gather points. Gather points are scene samples distributed on all scene surfaces.



Direct on Gather Samples
Wavelet Projection
Gather Direct in Wavelet Space
Multi-bounce Matrix
Diffuse Illumination on View Samples
Direct on View Samples
Direct+Indirect on View Samples
Diffuse Final Gather Matrix
Glossy Final Gather Matrix
$*k_d$
$*k_g$
+
+
Wavelet Projection
Direct+Indirect on Gather Samples
Gather Direct+Ind. in Wavelet Space
Glossy Illumination on View Samples
Indirect on View Samples

Figure 3.1: Direct-to-Indirect Transfter method. Image courtesy of Miloš Hašan [HPB06].

The lighting model therefore includes light paths originating from light sources, arriving at gather points, performing several bounces and finally arriving at screen pixels.

Both operators are stored as wavelet-encoded matrices, with rows corresponding to receivers and columns corresponding to sources of indirect illumination. In the multi-bounce matrix, both rows and columns correspond to gather points, while in the final gather matrix, rows correspond to screen pixels and columns correspond to gather points. Both matrices are encoded to wavelets by row-by-row projection and subsequent non-linear compression.

After precomputation, the system heavily relies on the GPU to perform the actual rendering. The rendering starts with computation of direct illumination on gather samples (implemented as a deferred rendering pass [AHH08]), followed by wavelet projection (again implemented in the hardware). The result is then multiplied by the multi-bounce matrix, yielding indirect illumination on gather points. Direct and indirect illumination on gather points are added together, projected to wavelets again and multiplied by the final gather

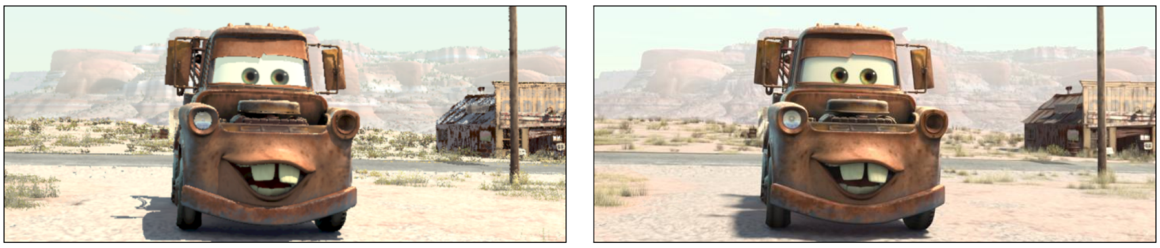


Figure 3.2: (Left) Image rendered by Lpics. (Right) Final render. Images courtesy of Fabio Pellacini [PVL05].

matrix. The multiplication yields indirect illumination for all screen pixels. Finally, direct illumination for all screen pixels is computed and added to the indirect illumination.

The main advantage of DTI is its simplicity. Both matrices can be thought of as linear operators acting on radiance values. GPU implementation allows for real-time previews with complete control over direct lighting.

3.2 Lighting Design and Relighting

Lighting design is the process of setting up light parameters. Due to the limited computational power and ever-growing complexity of scene geometries and materials, lighting design can become a cumbersome and tedious task on many occasions. A typical workflow is not interactive and consists of long periods of waiting for the rendering to complete.

This inefficiency was first addressed by Lpics (Pellacini et at.[PVL05]), during production of the movie Cars. Their system consists of surface and light shaders (see Renderman specification [Pix]), where surface shaders cache per-pixel parameters of the illumination model and light shaders are manually converted to a hardware shading language. At runtime, the GPU executes light shaders and evaluates lighting using the data in the cache. The system evaluates shading only for active lights, providing significant savings over an approach that evaluates all lights. Figure 3.2 compares images generated from Lpics and from a final renderer.

Barzel [Bar97] and Tabellion [TL04] describe common techniques used to achieve artistic effects in movie productions. One common approach relies on rendering into multiple layers, subsequent manual editing and final composition. Another approach relies on manual modifications of shader code. For example, shadows are often moved by introducing a secondary light with zero contribution to the final color of scene surfaces. Such a light is used to render only the shadow mask, while the primary light is used only for lighting. This technique is often referred to as *light linking* (see Figure 3.3).

The Lightspeed system [RKS07] is a robust lighting preview system that relies heavily on automation. The system automatically analyzes Renderman surface shaders and splits them into static and dynamic parts. The dynamic part is automatically converted to a GPU-friendly representation and used during the preview, while the static part is used by an offline renderer. Many other implementation techniques are used to achieve high performance — cache compression to fit complex scenes to the GPU memory, deep and indirect framebuffers to achieve good scalability when many lights are used and multi-pass rendering and computation graphs to support effects such as subsurface scattering.

Sun et a. [SZS08] develop a relighting system for rendering dynamic refractive objects. Their system is based on photon tracing inside a heterogenous volume and a final radiance-accumulating viewing pass. Their implementation runs entirely on the GPU and allows for relighting as well as dynamic objects. The main limitation is the voxel-based approach, which does not scale well with complex geometries with small-scale details. While the authors also present a simple painting interface for editing, this is not the primary focus of their work.

Wang et al. [WTL04] describe a precomputation-based approach to all-frequency relighting of glossy objects with a free camera. They factorize glossy BRDFs into a view-dependent and a light-dependent component and only use the light-dependent component in the precomputation. In [WTL06], the precomputation also supports multiple bounces of light. The view-dependent component is then used at runtime, to render the scene from the current viewpoint. The sampling rate in the view direction is lower than in the light direction, which effectively limits the upper frequencies in glossy reflections.

Dynamic BRDFs are introduced to interactive relighting in [SZC07]. Their approach supports interactive rendering and changes of BRDFs, lighting and the viewing direction. BRDFs are represented as tensors and factorized (similarly to [WTL04]) into multiple components for lighting, view direction and BRDF parameters. Precomputed transfer tensors are introduced to handle the non-linear relationship between outgoing radiance and scene BRDFs. The main advantage of this approach is the assumption of completely dynamic inputs (except for deformable geometry), but the main advantage, which might prevent its adoption in production environments, is huge precomputation times, which can reach more

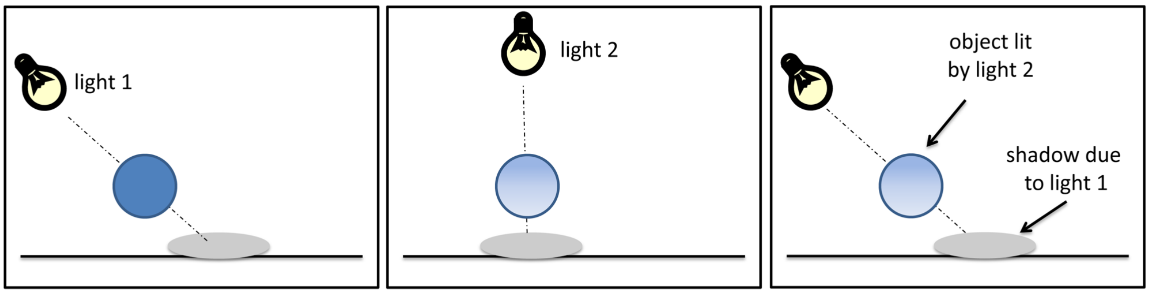


Figure 3.3: An example of light linking. (Left) Light 1 is used to render the shadow. (Middle) Light 2 is used to light the object of interest. (Right) Final composition is obtained by using the shadow from the first render and the shading results from the second render.

than 10 hours for complex scenes with complex materials. Another disadvantage is the assumption of a small linear basis for BRDFs.

Akerlund et al. [AUW07] and Cheslack-Postava et al. [CWA08] extend the Lightcuts approach and demonstrate is applicability to lighting and material design. They show how visibility and BRDFs can efficiently be represented as nonlinear piecewise constant functions (termed cuts) and evaluate the triple and double product integrals in this representation. They precompute visibility cuts and light cuts and evaluate the BRDF on the fly. In comparison to wavelets, the compression of visibility functions turns out to be slightly worse for cuts. However, the simplicity and efficient computation of cuts make them a viable alternative to wavelets in many applications.

Relighting, lighting design and material design solutions are often used across multiple studios and modeling/rendering packages. For this reason, they are very often required to conform to existing workflows, support multiple renderers and multiple user interfaces [RKS07].

3.2.1 Goal-based Design

Goal-based design approaches the lighting problem from the opposite angle. Instead of providing tools for direct manipulation of lights, goal-based design provides artists with tools for specifying target appearance. Optimization algorithms are then used to calculate light parameters which match the target design as closely as possible. This approach was first presented by Kawai [KPC93]. They provide a high-level design goals such as visual clarity. The user can also specify lighting constrains such as minimum lighting in certain areas. The system then solves for light source emissive values, surface reflectivity and spotlight directionality.

Schoeneman et al. [SDS93] allow the user to paint (or rather sketch) a target image and use least-squares minimization to compute intensities of pre-selected and fixed light sources. The advantage of this approach is its generality, because it does not rely on a particular rendering algorithm and its linearity. A disadvantage stems from the fact that the user has very little control over the optimization process.

Pellacini et al. [PBM07] extend the painting metaphor to higher dimensions. Artists are now allowed to paint color, shadows or highlights. The system not only adjusts light intensities, but can also add additional lights and/or modify parameters of existing lights. The nature of lighting makes this approach non-linear, but increases the applicability of the system tremendously. The authors were able to obtain quality results from a global illumination renderer, a direct illumination model, two nonphotorealistic (NPR) renderers as well as animation.

Okabe et al. [OMS07] show how painting can be used for lighting with environment maps. They design a system that allows users to paint using diffuse and specular brushes. The strokes are then used to synthesize the environment map using least-squares optimization. While the synthesis aspect of this work is certainly very interesting, the use of a low-frequency basis (spherical radial basis functions) limits its applicability to low-frequency environments.

In general, it can be argued that goal-based methods are very useful in practice, but still serve only as an intermediate solution to final design. The main reason is the lack of control over the optimization process. Typical workflows would use goal-based methods to quickly suggest a coarse-level design and then let artists tweak it to perfection. When it comes to user interfaces for goal-based design, painting has been used as the main metaphor, but it has been shown that it is the least-efficient one [KP09].

3.3 Material and Appearance Design

The main focus of material design is editing of the BRDF. As described in the previous chapter, BRDFs can be often very complex, thus difficult to evaluate and manipulate efficiently. Many BRDF-editing publications focus on providing real-time feedback while supplying artists with intuitive controls over BRDF parameters.

Ben-Artzi et al. [BOR06] provide a method for real-time editing of BRDFs under all-frequency lighting represented as environment maps. Similarly to relighting approaches, they take advantage of the linearity of light transport. They precompute a transport operator for a fixed lighting and fixed camera setting and compress it using wavelets. BRDFs are stored in a novel curve-based representation, which allows for efficient compression and manipulation. Another advantage of the curve-based representation is its user-friendliness.

In a global illumination setting, the linear nature of light transport (that is linearity with respect to BRDFs) vanishes due to multiple bounces and interreflections. A simple linear operator is not applicable anymore. Ben-Artzi et al. [BED08] propose to use a multi-bounce tensor of polynomial coefficients instead. They also take advantage of the low-frequency nature of indirect illumination and retain a high-quality representation only for the first bounce. Further bounces use lower-quality compression and often resort to diffuse terms only.

As described in Section 2.2, BSSRDFs are reflectance distribution functions used to describe phenomena such as subsurface scattering. Song et al. [STP09] describe a method, which allows for real-time editing of BSSRDFs. They represent subsurface scattering as a product of two scattering profiles — one for the incident and one for the outgoing surface location. The representation is further parameterized to support intuitive quantities such as albedo or scattering range. This work demonstrates that proper parametrization of even high-dimensional data can allow artists to edit appearance aspects of non-local nature.

Spatially-varying BRDFs (SVBRDFs) have been considered for real-time rendering only recently. Wang et al. [WRG09] introduce a novel SVBRDF representation based on spherical Gaussians. They also assume fixed geometry and present a novel visibility representation based on spherical signed distance functions. The combination of these and efficient evaluation on the GPU enables them to render dynamic SVBRDFs in real-time. This work, however, only considers relighting and does not present any methods for editing of SVBRDFs.

There are several other visual phenomena that often require artistic editing. Ritschel et al. [ROT09] present a system for real-time editing of reflections. They allow artists to specify reflection constraints by painting and dragging of two handles — one on the reflected object and one on the reflecting object. They propose a least-squares minimization algorithm to interpolate edited rays and run the whole system on the GPU. This system is an example, for which an explicit parametrization of the reflection phenomenon was not necessary.

3.3.1 Shadow Rendering and Design

Shadows are a very important aspect of visual fidelity in computer-generated images. In general, we distinguish between sharp, high-frequency, shadows and soft, all-frequency, shadows. Rendering of sharp shadows is a very well understood problem and multiple techniques exist for efficient GPU implementation. Typically, shadow mapping and stencil shadows are used [AHH08]. Editing/design of sharp shadows is traditionally accomplished by rendering the shadow to a separate layer (termed the shadow matte), editing the layer in an image processing application and finally compositing with the rest of the scene.

Pellacini et al. [PTG02] introduce a user interface for editing of shadows due to discrete light sources. They provide a variety of shadow manipulation tools (such as scale, translation, etc.), which indirectly modify light parameters. For example, a shadow translation results in light translation in the opposite direction. Application of these tools results in a different shadow matte, which is again rendered to a separate layer and composited with the scene.

In a related work, DeCoro et al. [DCF07] present an image-space approach to shadow editing. Instead of changing light parameters, they edit the shadow matte in image space. Typical examples of their edits include shadow blur or shadow inflation. The main limitation of their approach is its limitation to image space edits and missing support for selection or extraction of shadows.

While editing of sharp shadows is understood to some extent, the situation is different with soft shadows. Soft shadows appear due to area light sources and feature characteristic sharper edges in areas where the distance of the shadow caster and the shadow receiver is small and soft edges in areas where the distance is larger. Accurate and efficient rendering of soft shadows on the GPU is a nontrivial problem.

Zhou et al. [ZHL05] introduce the idea of precomputed shadow fields that describe shadowing effects of each object in a scene. They precompute object occlusion fields for scene entities and source radiance fields for light sources. At render time, both are combined according to the scene configuration. They support dynamic lighting and moving objects, but since object occlusion fields are precomputed, the objects themselves cannot be animated. The resulting datasets are compressed with wavelets, enabling all-frequency shadows.

Ren et al. [RWS06] present a smart technique for accumulating blocker visibility represented in the spherical harmonics basis. Instead of performing products of visibility vectors, they perform summation in the log space and finally exponentiate the result to obtain total occlusion. While this technique provides great insight, it is limited to low-frequency shadows. For performance reasons, occlusion is modeled by approximating blocker geometry with a set of spheres, which results in inaccuracies when complex geometry is present.

Soft shadows can often be approximated. Annen et al. [AMB07] show that the binary shadow test function used in traditional shadow mapping can be represented as a sum of weighted basis functions. Such a representation (the authors use the Fourier basis in their implementation) effectively enables shadow map prefiltering on the GPU and subsequent antialiasing of shadow edges. While this technique enables smoothing and blurring of shadow edges, it fails to sharpen them according to the distance between the shadow caster and the shadow receiver. An extension of this technique [ADM08] approximates soft shadows by computing the average blocker distance. This distance is then used to blur the shadow edge, yielding natural soft shadows. The average blocker distance is again computed by convolution.

3.4 Design Goals

The overview of the related work given in this chapter reveals that design aspects are often overlooked and considered to be secondary to the physical correctness of a particular rendering algorithm. One of the main goals of our work is to address this issue and develop appropriate techniques for design of complex visual aspects in images.

The overview also informs us about the importance of adhering to existing workflows and supporting multiple user interfaces. In our work, we do not focus on any particular user interface paradigm. We implement simple interfaces to showcase the editing power of our techniques, but focus primarily on representations, editing algorithms and efficient precomputation.

In the following chapters, we demonstrate two editing pipelines — one for lighting design with global illumination and one for all-frequency shadows. Global illumination and all-frequency shadows are complex visual aspects that require nontrivial algorithms to be rendered accurately and to the best of our knowledge, we are the first to present and publish (see Chapter 6.11 for the list of publications) efficient solutions that address their editing/design.

CHAPTER 4

LIGHTSHOP: INTERACTIVE LIGHT TRANSPORT EDITING FOR FLEXIBLE GLOBAL ILLUMINATION

In this chapter we present a novel user interface for lighting design with global illumination. The interface is dubbed LightShop and it is our first step toward a full relighting pipeline for global illumination. Chapter 4.7 will present the other parts of the pipeline — representation and rendering.

4.1 Introduction

The limited flexibility imposed by the physics of light transport has been a major hurdle in the use of global illumination in production rendering. To remove this hurdle, we present a novel interface that allows users to directly edit the light transport in a scene. Users can adjust the effects of indirect lighting cast by one object onto another or paint indirect illumination on surfaces. The interface makes fine-tuning of global illumination an intrinsic part of the lighting design process, thereby avoiding any post-processing work. In addition, all user actions are followed by an immediate rendering feedback crucial for users productivity.

Two things have traditionally hindered the use of global illumination in production rendering — long rendering times and limited flexibility imposed by the physics of light trans-

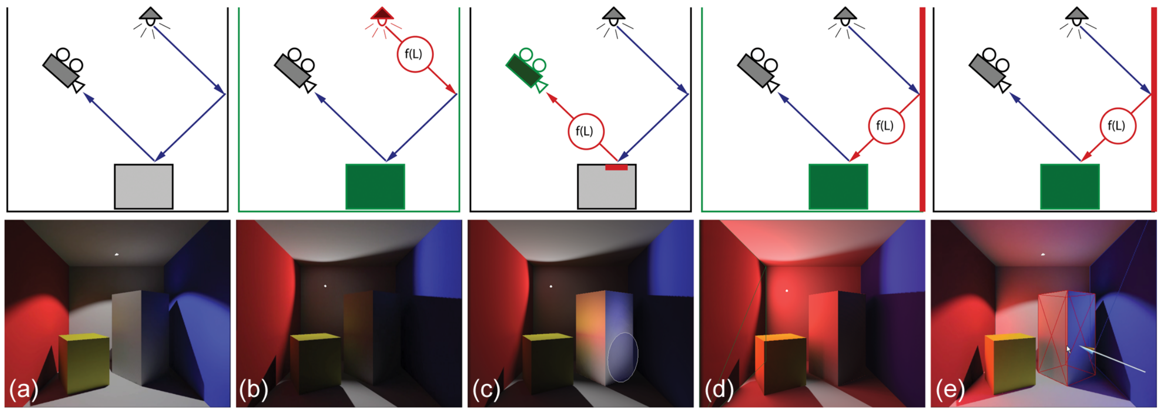


Figure 4.1: Bottom: Various lighting design tools offered by LightShop in action. Top: The location of the corresponding light transport edit operations on the light path. (a) the original image, (b) light source manipulation: the light source was placed on the back wall, (c) indirect illumination brush: indirect illumination was painted on the tall block, (d) indirect light source brush: the red wall casts more indirect light into the scene, (e) caster/receiver tool: the blue wall casts more light onto the tall block.

port. With the increased computer speed, rendering time is not the most serious issue anymore. However, global illumination is still not flexible enough; artists feel fettered by the physical laws used for computing global illumination since physical correctness in not the ultimate artistic goal. That is why post-processing is often applied to images rendered with global illumination.

There has been a long tradition in tweaking physics for the purposes of direct illumination (arbitrary falloffs, blockers, textures [Bar97, GH00], shadows disconnected from light sources [PTG02], etc.). Our objective is to apply similar ideas to the domain of global illumination by designing flexible methods for altering the physics of light transport and making these methods available to the user in an intuitive interface.

We introduce the concept of light transport editing as the key to promptly transforming high-level lighting design decisions involving global illumination into actual images. The light transport editing is brought to the users by our new interactive interface dubbed LightShop, where placement of light sources and artistic adjustments of the resulting global illumination can be performed as an inherent part of lighting design.

To free a creative user's mind from the trouble of understanding light transport and from tweaking dozens of low-level parameters of a renderer, our interface provides a set of easy-to-use controls for expressing design goals. The user can adjust the effects of indirect lighting cast by one object onto another, paint indirect light sources or the indirect illumination itself. Painting is facilitated by a new global illumination brush specifically designed to meet the frequency constraints of global illumination. The indirect lighting adjustments are expressed in terms of image operations traditionally used in post-production of global illumination renderings (color correction, change of brightness, contrast, etc.), which suit the low-frequency nature of indirect lighting.

The light transport modifications are all procedural and therefore manipulation of direct light sources later on does not invalidate them. All user actions are visualized in real-time, providing immediate visual feedback so important for high productivity. The result of a session is a lighting setup comprised of light source parameters and adjustments of indirect lighting, which, if rendered in high quality, provide an image that does not require further post-processing steps.

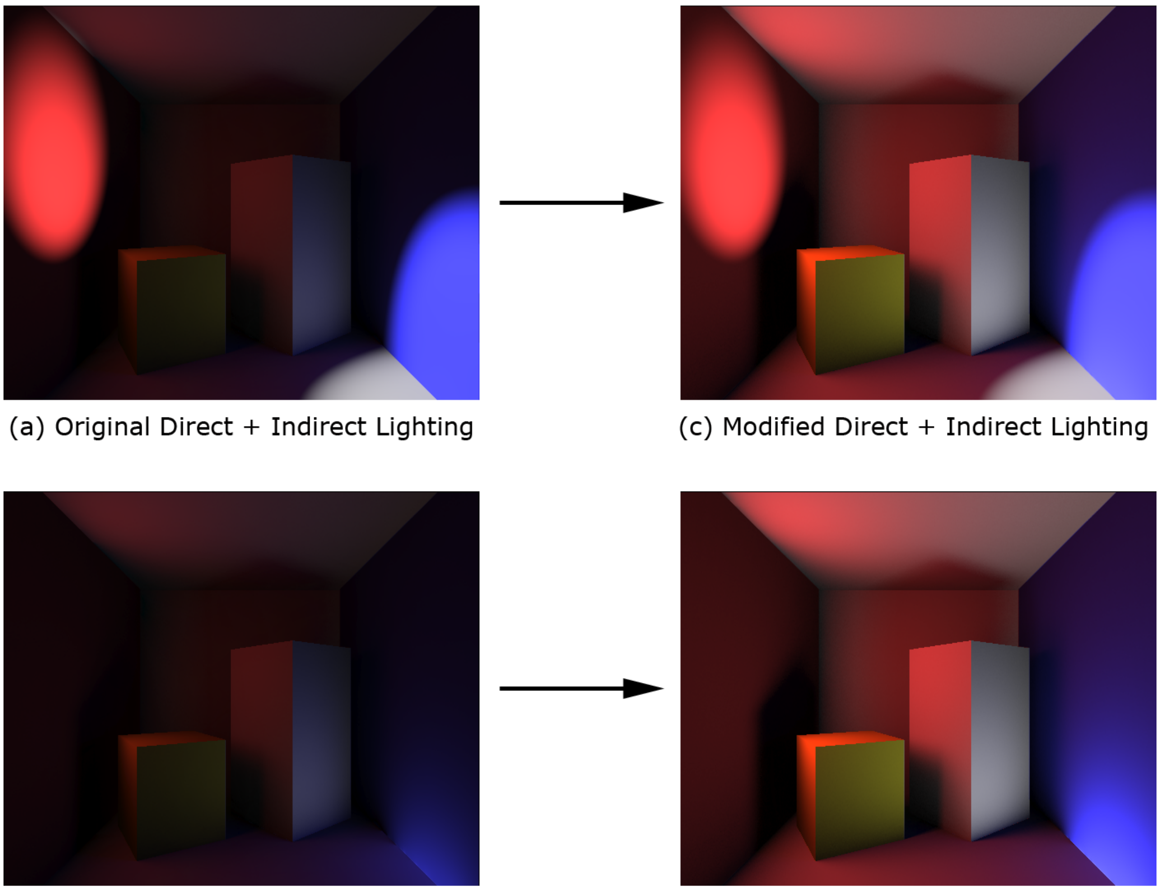
Our interface shows that the time-consuming workflow used in production environments (several passes of lighting design / rendering / post-processing) can be significantly shortened by providing the artist with appropriate tools. From the point of view of light transport, the edit operations modify light at various stages of its propagation through the scene. We show that only a few simple operations are expressive enough to cover most adjustments a user might want to apply to global illumination.

4.2 Related Work

Fine-tuning global illumination is often performed by rendering into layers, importing them into an image manipulation application and finalizing the work there. This is a cumbersome workflow since all the fine-tuning work has to be redone every time direct lighting changes and a new image is rendered. The novelty of our interface is in making the fine-tuning an intrinsic part of the lighting design.

Many of the effects we propose can be achieved by programming custom shaders [Chr03], which is a laborious and nonintuitive process. Our interface allows users to fine-tune global illumination without having to write a single line of shader code.

Relighting engines provide fast rendering feedback as the user manipulates light sources. Cinematic relighting engines usually allow changing arbitrary light source parameters, but support direct illumination only [GH00, PVL05]. Much research has gone into relighting



(a) Original Direct + Indirect Lighting

(c) Modified Direct + Indirect Lighting

(b) Original Indirect Lighting

(d) Modified Indirect Lighting

Figure 4.2: More LightShop editing. (a) and (b) demonstrate the effect of editing only the direct lighting. (c) and (d) demonstrate editing of the indirect component.

with global illumination [DSG91, NSD94, DKN95, TSH97, SKS02, NRH03]. To make the relighting problem linear and reduce its dimensionality, serious limitations are imposed on the light sources types and possible manipulations. Only recent work has allowed changing positions and directions of localized light sources in real-time [KAJ05, KTH06, HPB06]. Out of those, only the DTI method supports complex scenes and arbitrary light sources with reasonably short pre-computation time. Unlike our interface, none of the aforementioned relighting engines allow the user to edit global illumination as a part of the lighting design process.

Probably the most closely related relighting work is the tool described in [TL04], in which so called filter lights are used to modify indirect lighting. However, feedback is not immediate and slow global illumination computation has to be launched to update the display every time direct lighting changes.

We also depart from the goal-based paradigm described in Section 3.2.1 for multiple reasons. First, this approach does not deliver the flexibility sought by artists, since light transport is still limited by the physics and also by additional constraints imposed to make the problem tractable. Second, the optimization process is usually far from real-time and often converges to an undesirable local minimum.

4.3 Light Transport Editing Framework

Our light transport editing framework consists of applying operations on light at various stages of its propagation through the scene. Formally, each operation can be expressed as a function of a source radiance L_{src} that produces another radiance value, L_{dest}, arriving at the destination:

$$L_{dest} = f(L_{src})$$

Table 4.1 lists tools and segments of the light path, which we found to be the most useful for editing.

LightShop tool	*Source*	*Destination*	*Figure*
direct light manipulation	light source	surface	4.1(b)
indir. illumination brush	surface	camera	4.1(c)
indir. light source brush	surface	surface	4.1(d)
caster/receiver tool	surface	surface	4.1(e)

Table 4.1: LightShop tools.

While the table describes the localization of the source and the destination of an operation on the light path, our framework is also aware of their localization in space (i.e. where in the scene is the source and where is the destination). LightShop offers two strategies for the users to specify spatial localization of the operations: by painting onto surfaces and by selection on per-object basis.

We will show that with a few simple operations (change of contrast, brightness, saturation, hue shift, application of a tone mapping curve), our light transport editing framework is expressive enough to describe most conceivable adjustments of global illumination. The key to usability, however, lies in making light transport editing available to the user through an intuitive interface, as described in the next section.

4.4 Light Transport Editing Interface

This section describes the tools our interface provides for the user to edit light transport: the caster/receiver tool, the indirect light source brush, and the illumination brush. In addition to editing light transport, the user can place and manipulate sources of direct illumination.

4.4.1 Direct Light Source Placement

A typical lighting design session starts with a rough placement of direct light sources in the scene as shown in Figure 4.1(b). To simplify this task for the user, a light sources is placed on the surface under the mouse cursor, slightly offset along the surface normal. This is especially useful for interior scenes where most light sources are attached to the ceiling or the walls. Arbitrary manipulations with the light sources are allowed at any time later during the session. Full global illumination is immediately updated as the light sources are created or changed.

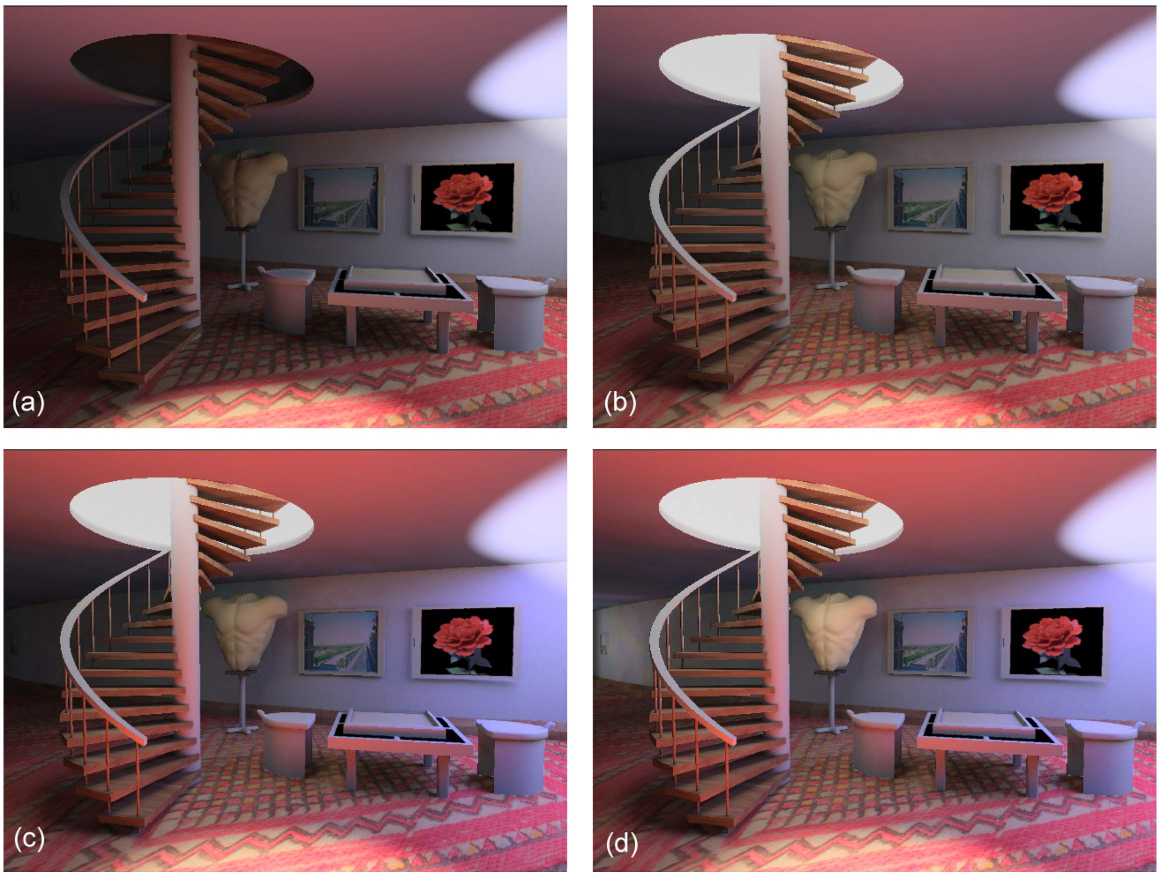


Figure 4.3: A lighting design session in LightShop. (a) The lighting design process starts by placing two spotlights. (b) Then, the intensity of the indirect light emitted by the cylindrical object above the staircase is increased. Notice the pronounced soft shadows under the staircase and under the table. (c) In the next step, we modify the contribution of light reflected from the floor by increasing its saturation and intensity. (d) Finally, the Illumination Brush is used to paint indirect illumination on the far end of the wall to the left and on the statue. The adjustments were intentionally exaggerated to better illustrate the achievable effects.

4.4.2 The Caster/Receiver Tool

This tool, illustrated in Figure 4.1(e), allows a user to fine-tune the indirect light transport between a selected pair of surfaces, often referred to as color bleeding. Any object in the scene is a caster of indirect light, and a receiver of indirect illumination. The user can select a caster/receiver pair and modify the illumination arriving at the receiver due to the caster. If either the caster or the receiver is unspecified, the whole scene is assumed instead.

With the caster/receiver tool, the user effectively applies an operation on the light incident at the receiver (i.e. the irradiance, $E(\mathbf{x})$) just before it is multiplied by the surface's diffuse texture (the reflectance, $\rho_d(\mathbf{x})$).

In theory, arbitrary operations could be applied. However, since we are dealing with indirect illumination, which is slowly varying by its nature, only monadic operations proved to be useful in practice. Currently, our interface supports the following ones:

- *Brightness.* Using this operation the user can increase or decrease the amount of indirect light the caster contributes to the receiver by applying a multiplicative factor. For example, such an operation would be useful when a large colored object causes an undesirably strong color cast on the receiver. Such an effect could be rectified by decreasing brightness.
- *Saturation.* It is very common that the saturation level of the color bled from a caster onto a receiver is too low or too high. This operation allows the user to adjust the saturation level.

- *Hue shift.* Using this operation, the user can change the shade of indirect light bled from the caster to the receiver's surface.
- *Global tone mapping curve.* Any non-linear curve can be applied to the intensity of the light arriving at the receiver. As an example, one can apply gamma curve to the indirect illumination corresponding to selected caster/receiver pairs separately. This feature is useful for simulating arbitrary distance falloff of indirect illumination, to replace the quadratic falloff that comes naturally from physically correct light transfer simulation. Using a concave curve would make the light intensity fall off more slowly with distance. Using a convex curve, on the other hand, would make the falloff faster.

The caster/receiver tool can be placed in our light transport editing framework as an operation applied on the light at the path segments originating at the caster and arriving at the receiver.

4.4.3 Indirect Light Source Brush

Every surface is a source of indirect light. The indirect light emitted by a surface depends on the amount of direct illumination it receives from light sources in the scene. One might want to boost or weaken the amount of indirect light naturally produced by a surface as a response to direct lighting. The *indirect light source brush*, illustrated in Figure 4.1(d), makes this possible in our interface.

The light source brush allows the user to modify indirect light emitted at any location in the scene. The user can choose to use an intuitive painting interface or apply indirect light modifications on a per-object basis (referred to as the caster tool). Other surfaces, illuminated by the one painted on, will react by changing the amount of incident illumination. If only one indirect bounce is used, painting indirect light source onto a surface will not alter the appearance of the surface itself—it will only be altered if multiple indirect bounces are used.

The practice has shown that the following two operations are the most useful ones for the light source brush:

- *Contrast.* This operation allows the user to adjust the amount of emitted indirect light by applying a multiplicative factor.
- *Brightness.* Using this operation the user can increase or decrease the amount of emitted indirect light by applying an additive factor. This is useful to brighten areas of the scene not lit by any direct light.

One can think of the indirect light source brush as a modifier of a surface's reflectivity applied solely when reflecting light onto another surface (as opposed to reflecting light into camera). However, it is more natural for an user to think of painting an indirect light source than to think of specifying two surface textures, one for direct lighting and the second for indirect illumination.

The placement of the indirect light source brush in our light transport editing framework is similar to the caster/receiver tool: it produces an operation applied on the light at the path segments originating from the painted indirect light source arriving at any other surface in the scene.

4.4.4 The Indirect Illumination Brush

Imagine that the lighting design for an interior scene has nearly been finished, but one corner is undesirably dark and should be brightened. The *indirect illumination brush*, illustrated in Figure 4.1(c), serves this very purpose of small final touch-ups. It allows the user to dodge or burn indirect illumination over the scene surfaces as desired. This feature is especially useful in combination with the adaptive brush size described in the next section. In our light transport editing framework, the illumination brush is applied at the last segment of an indirect illumination light path.

4.4.5 Adaptive Brush Size

It has been shown [WRC88] that indirect illumination changes slowly in open spaces, and can only change abruptly in areas of geometry variation. To make the painted edits of global illumination as plausible as possible, the interface can automatically modulate the brush size by the expected rate of change of indirect illumination at the painted location—large size is used where illumination is expected to change slowly and a small size is used where illumination can change abruptly. For example, this feature prevents the user from painting unnaturally abrupt illumination changes on a floor in the middle of a room, but allows adding small indirect illumination variations near corners. Our implementation modulates the brush size by the harmonic mean of distances to the scene geometry [WRC88] computed for each view sample in the preprocess.

All operations described in previous sections are applied as *procedural modifications* of the results produced by physically-based light transport simulation. For example, if the saturation of the color cast by a red cube is decreased, the same decrease in saturation will remain if the cube's material is later changed to navy blue. This feature is essential for smooth integration of light transport editing into the lighting design process.

4.5 Rendering

This section describes the rendering engine of LightShop. As already mentioned, the underlying rendering algorithm is based on the DTI method [HPB06]. The pseudocode in Algorithm 3 is the higher-level original rendering loop.

The light transport editing tools of our interface are implemented as follows:

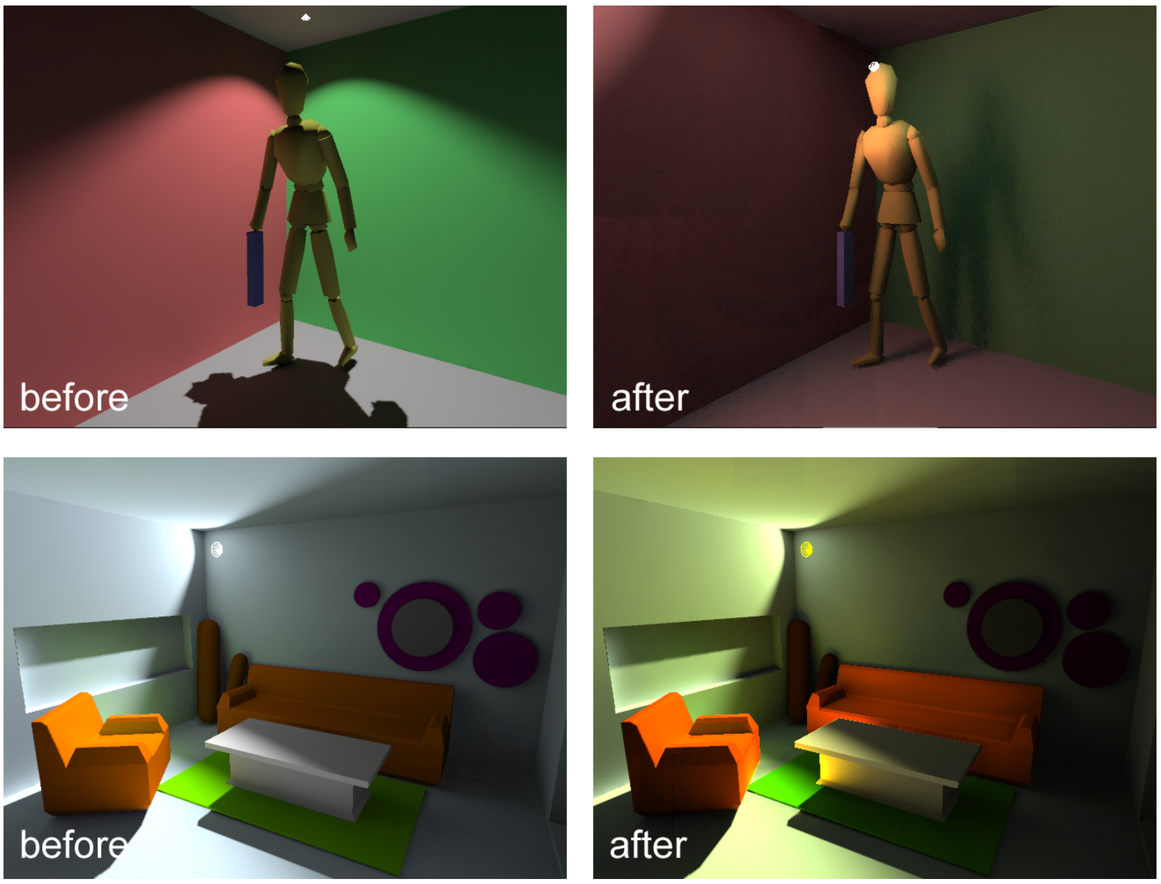


Figure 4.4: Results of lighting design sessions with LightShop for two scenes.

Caster/Receiver Tool. In order to restrict edits to light transferred from a caster to a receiver object, a selection must be applied. We accomplish this by enabling gather points on the caster object and disabling them everywhere else. Next, we multiply the vector of radiant exitance values on the gather samples with the transfer matrix, accumulating results only on the receiver surface (selected using a stencil). The specified edits (saturation, hue shift, etc.) are applied to the pixels of the receiver in a pixel shader, then multiplied by the receiver's diffuse texture, and added to the final image. This procedure is repeated for each edited caster/receiver pair and its pseudocode is shown in Algorithm 4.

Algorithm 3: LightShop render loop

```
Data: PixelBuffer — textures containing positions, normals and color for screen pixels
Data: GatherBuffer — textures containing positions, normals and color for gather
      points

while true do
    gatherDirect = deferredShading(GatherBuffer);
    gatherDirectHaar = encodeToWaveletsGPU(gatherDirect);

    viewIndirect = matrixVectorMultiply(gatherDirectHaar, finalGatherMatrix);
    viewDirect = deferredShading(PixelBuffer);

    toneMappingReinhardGlobal(viewDirect + viewIndirect);
end
```

Algorithm 4: Caster/Receiver tool in LightShop

```
Data: ActiveReceivers — set of active receivers
Data: PixelBuffer — textures containing positions, normals and color for screen pixels
Data: GatherBuffer — textures containing positions, normals and color for gather
      points

Clear(FrameBuffer);
foreach receiver ∈ ActiveReceivers do
    foreach caster ∈ receiver.ActiveCasters do
        casterPoints = selectGatherPoints(GatherBuffer, caster);
        casterDirect = deferredShading(casterPoints);
        casterDirectHaar = encodeToWaveletsGPU(casterDirect);

        enableStencil(receiver);
        receiverIndirectContrib = matrixVectorMultiply(casterDirectHaar,
        finalGatherMatrix);
        disableStencil(receiver);

        editedContrib = applyEdits(receiverIndirectContrib);
        additiveBlend(FrameBuffer, editedContrib);
    end
end
toneMappingReinhardGlobal(FrameBuffer);
```

Indirect Light Source Brush. The radiant exitance due to direct illumination at each gather sample is multiplied by a multiplicative indirect source factor and then an additive indirect source factor is added. Next, the vector of modified radiant exitance values on the gather samples is multiplied by the transfer matrix to compute indirect illumination on view samples. A separate value for both the multiplicative and the additive factor is kept for each gather sample (see Algorithm 5).

Algorithm 5: Light Source Brush

```
Data: GatherBuffer — textures containing positions, normals and color for gather
      points
gatherDirect = deferredShading(GatherBuffer);
foreach gatherPoint ∈ GatherBuffer do
    gatherDirect[gatherPoint] = gatherDirect[gatherPoint] * brushIntensity +
    brushEmissive;
    applyEdit(gatherDirect[gatherPoint]);
end
```

Indirect Illumination Brush. The indirect illumination at each view sample is multiplied by a *painted illumination factor*. This is applied after the color correction operations specified in the caster/receiver mode and the multiplication by the surface reflectance (see Algorithm 6).

Algorithm 6: Indirect Illumination Brush

```
Data: PixelBuffer — textures containing positions, normals and color for screen pixels
Data: GatherBuffer — textures containing positions, normals and color for gather
      points
while true do
    gatherDirect = deferredShading(GatherBuffer);
    gatherDirectHaar = encodeToWaveletsGPU(gatherDirect);
    viewIndirect = matrixVectorMultiply(gatherDirectHaar, finalGatherMatrix);
    foreach pixel ∈ PixelBuffer do
        viewIndirect[pixel] = viewIndirect[pixel] * brushIntensity;
        applyEdit(viewIndirect[pixel]);
    end
end
```

4.6 GPU Wavelet Encoding

We have already introduced the basic CPU algorithm for Haar wavelet encoding in Section 2.6.2. While the CPU implementation is very simple, its efficiency is not that great. Instead, we developed an efficient GPU implementation, which natively supports parallel execution.

We demonstrate our encoding algorithm on an example. Assume we have a 16-element input vector $A = (a_1, a_2, ..., a_{16})$. Without loss of generality, we omit the normalization term from Algorithm 1. CPU encoding will have $\log_2 n = \log_2 16 = 4$ passes and is demonstrated in Figure 4.5. In the notation used, $a_{x..y} = \sum_{i=x}^{i=y} a_i$.

Input: $A =$

$$(a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, a_{10}, a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16})$$

Pass 1: $W_{CPU}(A) =$

$$(a_1 + a_2, a_3 + a_4, a_5 + a_6, a_7 + a_8, a_9 + a_{10}, a_{11} + a_{12}, a_{13} + a_{14}, a_{15} + a_{16},$$
$$a_1 - a_2, a_3 - a_4, a_5 - a_6, a_7 - a_8, a_9 - a_{10}, a_{11} - a_{12}, a_{13} - a_{14}, a_{15} - a_{16})$$

Pass 2: $W_{CPU}(A) =$

$$(a_{1..4}, a_{5..8}, a_{9..12}, a_{13..16}, a_{1..2} - a_{3..4}, a_{5..6} - a_{7..8}, a_{9..10} - a_{11..12}, a_{13..14} - a_{15..16},$$
$$a_1 - a_2, a_3 - a_4, a_5 - a_6, a_7 - a_8, a_9 - a_{10}, a_{11} - a_{12}, a_{13} - a_{14}, a_{15} - a_{16})$$

Pass 3: $W_{CPU}(A) =$

$$(a_{1..8}, a_{9..16}, a_{1..4} - a_{5..8}, a_{9..12} - a_{13..16}, a_{1..2} - a_{3..4}, a_{5..6} - a_{7..8}, a_{9..10} - a_{11..12},$$
$$a_{13..14} - a_{15..16}, a_1 - a_2, a_3 - a_4, a_5 - a_6, a_7 - a_8, a_9 - a_{10}, a_{11} - a_{12}, a_{13} - a_{14},$$
$$a_{15} - a_{16})$$

Pass 4: $W_{CPU}(A) =$

$$(a_{1..16}, a_{1..8} - a_{9..16}, a_{1..4} - a_{5..8}, a_{9..12} - a_{13..16}, a_{1..2} - a_{3..4}, a_{5..6} - a_{7..8},$$
$$a_{9..10} - a_{11..12}, a_{13..14} - a_{15..16}, a_1 - a_2, a_3 - a_4, a_5 - a_6, a_7 - a_8, a_9 - a_{10},$$
$$a_{11} - a_{12}, a_{13} - a_{14}, a_{15} - a_{16}) \equiv (b_1, b_2, ..., b_{16})$$

Figure 4.5: 1D Haar encoding of a 16-element vector

Our GPU encoding algorithm takes the input vector as a 2D texture:

$$A = \begin{pmatrix} a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 \\ a_9 & a_{10} & a_{11} & a_{12} \\ a_{13} & a_{14} & a_{15} & a_{16} \end{pmatrix}$$

and encodes the vector using the pseudocode in Algorithm 7 (notice, $M = \sqrt{N}$, where N is the length of the input vector, 16 in our example).

Algorithm 7: 1D Haar GPU encoder

```
Data: MatrixA — input vector organized into an M × M array.
m = M;
while $m > 1$ do
    rowPassAllParallelGPU(MatrixA, m);
    m = m / 2;
end
m = M;
while $m > 1$ do
    columnPassFirstGPU(MatrixA, m);
    m = m / 2;
end
```

One disadvantage of the GPU algorithm is that it does not preserve ordering of the encoded coefficients. After completion, encoded matrix A contains the following values:

$$W_{GPU}(A) = \begin{pmatrix} a_{1..16} & a_{1..2} - a_{3..4} & a_1 - a_2 & a_3 - a_4 \\ a_{1..8} - a_{9..16} & a_{5..6} - a_{7..8} & a_5 - a_6 & a_7 - a_8 \\ a_{1..4} - a_{5..8} & a_{9..10} - a_{11..12} & a_9 - a_{10} & a_{11} - a_{12} \\ a_{9..12} - a_{13..16} & a_{13..14} - a_{15..16} & a_{13} - a_{14} & a_{15} - a_{16} \end{pmatrix}$$

If we linearize matrix $W_{GPU}(A)$ and compare it to the CPU-encoded vector $W_{CPU}(A)$, we can observe that they contain the same values, but in different order. We need to find a mapping from $W_{GPU}(a)$ to $W_{CPU}(A)$, which turns out to be straightforward — we first traverse the first column of $W_{GPU}(A)$ and then the remaining rows. Coefficients of $W_{CPU}(A)$ can be found in $W_{GPU}(A)$ as shown Figure 4.6.

$$W_{CPU}(A) = \begin{pmatrix} b_1 & b_2 & b_3 & b_4 \\ b_5 & b_6 & b_7 & b_8 \\ b_9 & b_{10} & b_{11} & b_{12} \\ b_{13} & b_{14} & a_{15} & b_{16} \end{pmatrix} \quad W_{GPU}(A) = \begin{pmatrix} b_1 & b_5 & b_9 & b_{10} \\ b_2 & b_6 & b_{11} & b_{12} \\ b_3 & b_7 & b_{13} & b_{14} \\ b_4 & b_8 & a_{15} & b_{16} \end{pmatrix}$$

Figure 4.6: Mapping between coefficients of the CPU and GPU Haar wavelet encoders.

4.6.1 Complexity Analysis

We first analyze the performance of the CPU algorithm described in Algorithm 1 in Section 2.6.2. We assume that with regard to complexity, all instructions (addition, multiplication, division and assignment) are equal. Encoding of N elements then takes:

$$T_{CPU}(N) = \sum_{i=1}^{\log_2 N} \left(2 + \frac{2^i}{2}(6+6+1)\right) = 2\log_2 N + 13(N-1) = \Theta(N + \log_2 N)$$

The GPU algorithm performs the computation in two steps. It processes each matrix row in $log_2\sqrt{N}$ passes and then the first column in $log_2\sqrt{N}$ passes. The actual computation for each matrix element is executed on the GPU in a pixel shader. The pixel shader contains two texture-fetch instructions and 6 arithmetic instructions.

Assuming that the GPU contains enough (i.e. N) pixel shader units, the total execution time of the GPU algorithm is:

$$T_{GPU}(N) = 8(\log_2 \sqrt{N} + \log_2 \sqrt{N}) = \Theta(\log_2 N)$$

The speedup of the GPU algorithm is the computed as:

$$S_{GPU}(N) = \frac{T_{CPU}(N)}{T_{GPU}(N)} = \Theta\left(\frac{N + \log_2 N}{\log_2 N}\right)$$

The work performed by the GPU algorithm is given as the sum of all instructions performed in all execution steps by all active pixel shader units:

$$
\begin{aligned}
W_{GPU}(N) = \underbrace{8\sqrt{N}\sum_{i=0}^{\log_2\sqrt{N}-1}\left(\frac{\sqrt{N}}{2}\right)^{i}}_{\text{rows}} + \underbrace{8\sum_{i=0}^{\log_2\sqrt{N}-1}\left(\frac{\sqrt{N}}{2}\right)^{i}}_{\text{first column}} &= 8(N+\sqrt{N})\sum_{i=0}^{\log_2\sqrt{N}-1}\left(\frac{1}{2}\right)^{i} \\
&= 8(N+\sqrt{N})2(1-\frac{1}{\sqrt{N}}) \\
&= 16\,(N-1) \\
&= \Theta\,(N)
\end{aligned}
$$

The cost of the GPU algorithm is:

$$C_{GPU}(N) = NT_{GPU}(N) = \Theta(N\log_2 N)$$

Since $C_{GPU}(N) \neq O(T_{CPU}(N))$, the GPU algorithm is not cost-optimal. However, it is work-optimal, since $W_{GPU}(N) = O(T_{CPU}(N)) = O(N)$. The algorithm is not cost-optimal, because not all pixel shader units are utilized during the execution. The number of utilized pixel shader units drops by half after each iteration. It is work-optimal, because we do not incur any additional overhead by parallelization. However, it is important to emphasize that it is optimal only in theory. In practice, it is very difficult to compare the cost of executing equivalent instructions on a GPU and on a CPU, since their hardware architectures are fundamentally different. We also assumed that texture reads on the GPU do not stall the pipeline and that we have enough pixel shader units. For large values of N, this might not be the case.

4.7 Conclusions

We introduced the light transport editing as the means of promptly transforming user's lighting design decisions involving global illumination into final rendered images. We implemented light transport editing in an interactive relighting interface featuring a set of easy-to-use tools, making light transport modification available to the user. We demonstrated that several very simple operations applied at various stages of light propagation through the scene provide enough expressive power for most conceivable adjustments of global illumination.

While we believe that our interface opens up many new possibilities for artists working with global illumination, one important issue needs to be addressed before deploying our solution in a production environment.

The problem of transferring the edits made in our interface to a production renderer is left unaddressed, mainly because a solution depends on the architecture of a particular renderer. A simple solution would be to render all the layers corresponding to the edited caster/receiver pairs, apply the edits and then blend the layers together. However, this strategy would scale poorly with the number of edits. Other possibility would be to export the edits in the form of indirect light shaders similar to [TL04]. Another option is applicable to renderers where surface shaders are responsible for invoking indirect illumination operators [Chr03]: the operators would be aware of the user edits and return the edited version of indirect illumination.

CHAPTER 5

ICHEAT: A REPRESENTATION FOR ARTISTIC CONTROL OF INDIRECT CINEMATIC LIGHTING

We concluded the previous chapter with a discussion about connecting LightShop to a production renderer. We observed that, while LightShop provides a very user-friendly interface, the way it handles edits of the light transport is not optimal. In this chapter, we further elaborate the concept of edit representation and present our solution to the problem. We conclude with the detailed description of the rendering algorithm and the whole global illumination editing pipeline.

5.1 Introduction

The procedural approach to editing of global illumination proposed in the previous chapter follows the long tradition of lighting "cheats" developed over the years in the computer graphics industry, e.g. arbitrary falloffs, fake blockers [Bar97] and shadows from tweaked positions [PTG02] to name a few. To fit production shader-based workflow, these adjustments are encoded procedurally in complex light shaders [Bar97, PVL05]. While this approach works well for direct lighting, the situation with global illumination is different.

Indirect illumination did not play an important role in cinematic lighting until recently, when an increase in its efficiency has made it viable for cinematic scenes [TL04, CFL06]. To this day though, artistic control of indirect illumination has been limited to simple adjustments and has not reached the flexibility common to direct lighting. An example of a common practice is the procedural modification of surface and light shaders to change the color of the emitted indirect light, as well as including and excluding objects from indirect transport [TL04]. While this works well for some adjustments, many effects cannot be expressed in simple manners just by altering shaders, in stark contrast with direct illumination where modifying shader code is comprehensive and efficient. Furthermore, for an intuitive and efficient lighting design, artists seek direct control over the *interaction* between materials, geometry and direct illumination, rather than their independent adjustments.

This chapter introduces a *representation* for artistic control of indirect illumination that is *comprehensive*, *efficient*, *intuitive* and not tailored to any software architecture or way of computing global illumination (i.e. *renderer-independent*). The key insight of our method is to encode indirect illumination edits as scale and offset values of the transport between points in the scene, and by keyframing them to support animation. Example edits made using our representation are shown in Figure 5.1. Rather than focusing on supporting specific effects, this representation is general in its support of any adjustment since artists can arbitrarily alter the transport between scene locations.

To make this efficient, we take advantage of the sparse, low-frequency nature of user edits to encode the adjustments efficiently in an approximate form that makes offline rendering practical while supporting interactive lighting design. Furthermore, since artists control transport directly, the representation is intuitive and naturally supports simple user interfaces, such as painting-based metaphors. Finally, our representation does not rely on procedurally altering shader code, which is renderer specific, but is defined by sampling a simple mathematical operator that can be implemented in any renderer and user interface.

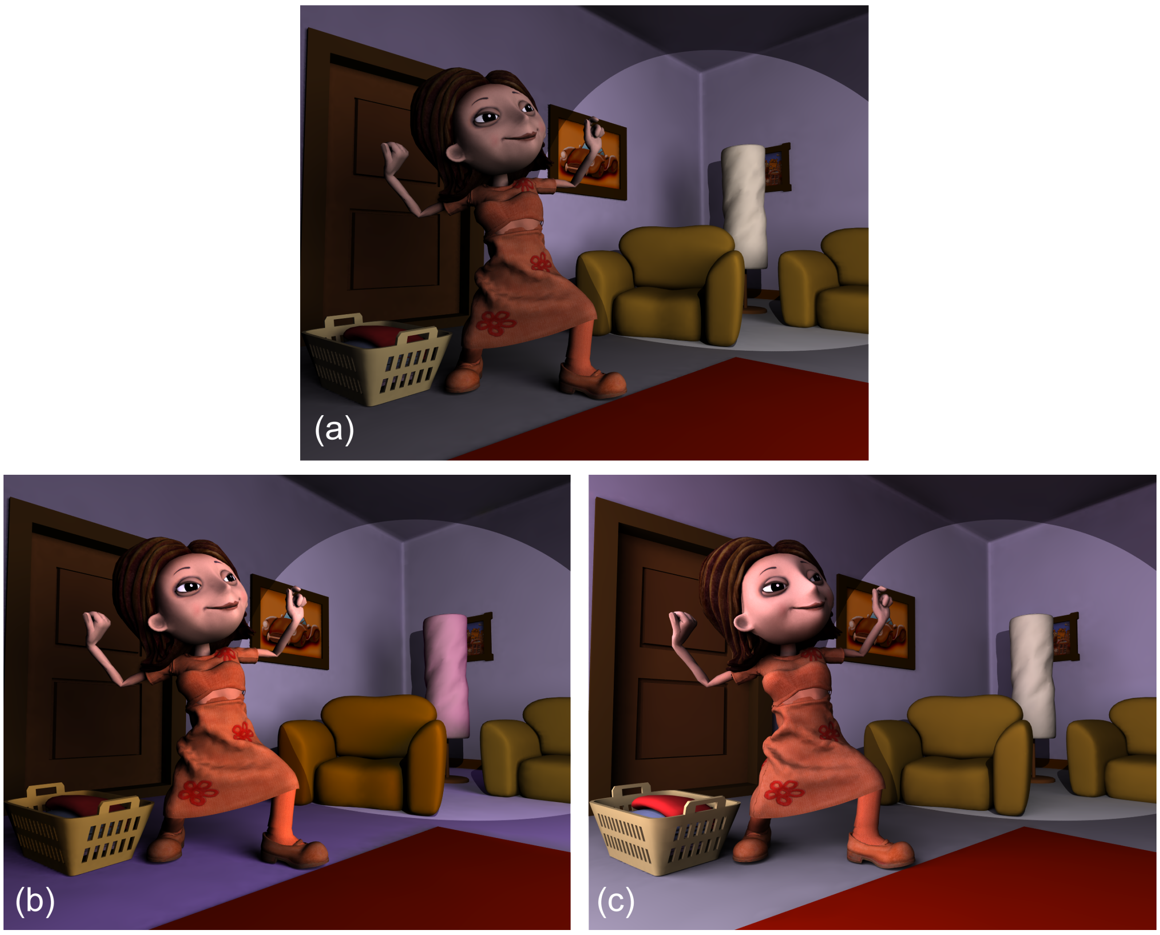


Figure 5.1: Artistically modified indirect illumination. (a) shows original unmodified global illumination. In (b) we directed more indirect light at the character, bringing her into focus. In (c) we modified hue/saturation of indirect lighting hitting the room and changed its directionality to make the character appear lit from her right-hand side.

The benefit of having a renderer-independent representation has been clear during the development of this project, where we have been able to integrate a real-time relighting engine to preview our edits, a user interface to interactively perform the adjustments by painting (LightShop), and an offline renderer for final high quality output. Furthermore, to achieve highest efficiency our two renderers used different algorithms for indirect lighting, making it simply impossible to support procedural alterations. This simplicity is in contrast with the complexity of matching light shaders between real-time and offline rendering in today's procedural shader-based pipelines, as demonstrated in the Lpics and Lightspeed systems [PVL05, RKS07].

The remainder of this chapter introduces our adjustment representation and demonstrates its benefits in the context of cinematic lighting design. While any number of bounces can be used to compute global illumination, the user only controls the last one, since this was found to be more controllable for artists and sufficient for cinematic purposes [TL04]. We make the following important contributions:

- we introduce a general and intuitive representation for artistic control of indirect illumination
- we demonstrate how arbitrary edits can be encoded in such formulation, including support for animated scenes
- we take advantage of the sparse, low frequency nature of user edits to encode them efficiently

- we derive a real-time rendering algorithm to preview edits, and show a simple painting interface to control them
- we demonstrate the feasibility of inclusion in a production pipeline by rendering all our final edits in an offline renderer that supports indirect illumination in a way similar to Renderman.

5.2 Related Work

In today's productions, two common approaches are used to artistically control indirect illumination. First, by rendering the scene in layers, adjustments to final colors can be performed with compositing tools. While these methods allow for global adjustment to be efficiently performed, adjustments that only affect parts of the scene remain too cumbersome with these methods. Our method supports any adjustment, whether local or global, in an intuitive and efficient manner. The practice most closely related to our own is the idea of controlling the computation of indirect illumination directly by procedurally altering shader code, exemplified in the production work in today's computer generated movies [TL04, CFL06]. Compared to this approach our method has significant benefits in generality, intuitiveness and practicality that we have already discussed in the introduction and we will further review in the following sections.

One of the advantages of our representation is that it can be easily supported in interactive relighting engines that simplify lighting design by interactively displaying the adjusted illumination during editing [PVL05, RKS07]. Of the various algorithms available, we have implemented a variation of [HPB06] since this algorithm was designed exactly to support indirect illumination for cinematic scenes and since our representation maps well to their relighting framework.

While our representation does not impose a specific user interface, one of its advantages is that is can support an intuitive painting metaphor to directly control indirect effects. We used our LightShop interface to generate the edits in this chapter and noticed that controlling complex adjustments is quite simple and efficient. The idea of using a painting metaphor is loosely related to goal-based lighting design, where artists specify how the final scene should look, often via painting, while an algorithm automatically sets scene parameters to best fit artists requirements [KPC93, SDS93, OMS07, PBM07]. We departed from this paradigm in our prototype since estimating parameters via optimization is quite slow in the case of indirect illumination and may not converge to a reasonable solution. The reader should notice, though, that our representation allows artists to choose whichever user interface they feel most comfortable with.

5.3 Representation

Our goal is to artistically adjust the indirect lighting for each frame of a cinematic sequence. To allow for maximum control, we specify these adjustments independently for each frame in the sequence. In the following paragraphs, we describe our method focusing on a single frame, and present its extensions to animated scenes later.

For the fixed camera position of the rendered frame, the indirect illumination B of a *view* point v can be written as

$$B(v) = \int_{g\in S} B(g)T(v,g)dg = \int_{g\in S} B(g)\left(\rho(v,g)G(v,g)V(v,g)\right)dg \tag{5.1}$$

where g is a *gather* point, S the set of all positions on object surfaces, $T(v,g)$ is the transport operator, $V(v,g)$ is visibility and $\rho(v,g)$ is the BRDF evaluated between v and g. The geometric term is computed as:

$$G(v,g) = -\frac{(N_v \cdot \overrightarrow{vg})(N_g \cdot \overrightarrow{vg})}{|\overrightarrow{vg}|^2}$$

Note that while ρ can be glossy in the last bounce, we assume that indirect effects only depend on the diffuse radiance of all other bounces as is common in cinematic lighting [TL04]. Current methods to control indirect illumination encode adjustments as procedural modifications of the various terms ρ, G, V, $B(g)$ [TL04]. While many effects can be represented this way, adjustments that depend on the interaction between these terms are cumbersome to capture and very hard to control for artists. Furthermore, since most of these adjustments are represented as shaders, it is often close to impossible to have multiple renderers compute the same image.

5.3.1 Edit Representation

We propose a representation where artists control indirect illumination by directly altering the transport coefficients $T(v, g)$. Formally, we define the edited transport T' by scaling and offsetting the original values

$$T'(v, g) = s(v, g)T(v, g) + o(v, g) \tag{5.2}$$

where s and o are scale and offset functions defined over the cartesian product of the scene locations $S \times S$. We support color adjustments by storing scale/offsets for each channel separately. This formulation can *represent any edit* an artist might want to achieve since any value of the transport coefficient can be obtained; we demonstrate several examples in the next section. Furthermore, users can control the values of the adjustments s and o directly and intuitively, without having to manipulate the individual components separately.

We sample the functions s and o over a set of view and gather samples v_i and g_j. In our implementation, we choose n view samples as the points visible through all pixels, and m gather samples uniformly distributed on the scene geometry as in [HPB06]. In this formulation, the functions s and o can be thought of as $n \times m$ scale and offset matrices $[S_{ij}]$ and $[O_{ij}]$

$$s(v,g) \approx [S_{ij}] \qquad o(v,g) \approx [O_{ij}] \tag{5.3}$$

We interpolate the matrix values to reconstruct scales and offsets for arbitrary view/gather pairs, as described in Section 5.5.3.

5.3.2 Efficient Encoding

High sampling density of scale and offset matrices is required to achieve high quality results. For our results, we use 360K view samples and 64K gather samples, making the direct storage of the matrices impractical. To make the storage tractable, we project each row of the matrices in 1D Haar wavelets and cull many of the less important wavelet coefficients

$$S^w \approx WS \qquad O^w \approx WO \tag{5.4}$$

where W is the wavelet transform matrix. Similarly to prior work on cinematic relighting [HPB06], we impose the wavelet basis by hierarchical clustering of the gather points to ensure spatial coherence after wavelet projection. In our experience, wavelets capture well the all-frequency structure of user edits. In our prototype we use 100 wavelet coefficients per row.

Furthermore, many edits can be represented as constant rows or columns of the matrices. These can be equivalently expressed as adjustments to $B(v_i)$ and $B(g_j)$ respectively. To take direct advantage of this, we additionally store scale and offset vectors $\{s_i^v, o_i^v\}$ and $\{s_j^g, o_j^g\}$ respectively for view and gather samples to obtain

$$B'(v_i) = s_i^v B(v_i) + o_i^v \qquad B'(g_i) = s_i^g B(g_i) + o_i^g \tag{5.5}$$

5.3.3 Discussion

The reader may wonder why we represent edits as scale and offset of the transport coefficients T, rather than simply storing the edited transport coefficients T'. In this respect, our representation has three main advantages. First, storing the transfer coefficients directly for a high quality offline rendering would be impractical, while user edits are significantly sparser and lower frequency. Second, offline and real-time rendering algorithms have different quality/efficiency tradeoffs when computing indirect illumination; our sampled representation can be integrated with these algorithms while preserving such tradeoffs. Third, and possibly most importantly, it is more intuitive to modify scale/offsets than the transport itself; for example to darken a wall, users can just scale the radiance by a constant, while still maintaining these beautiful realistic gradients so hard to achieve without indirect illumination.

5.3.4 Animation

We support animation assuming artists follow the same workflow typical of direct illumination, where adjustments are defined for a few keyframes in the sequence and interpolated in the remaining frames (see Figure 5.6). In our framework, we interpolate matrices and vectors between keyframes. This requires determining correspondence between matrix elements. We make this correspondence implicit in gather samples by choosing the same *parametric* surface location at each frame. View sample sets from the keyframes are reprojected to the current frame, too. We compute a matrix row for an arbitrary view point v in the current frame by performing spatial interpolation for two surrounding keyframes using their reprojected view sample positions (we take location and normal orientation into account when performing the lookup), and then temporally interpolate the two resulting rows. Thus, if a large area that was occluded in the key frame becomes visible in the in-between frame, artifacts may, indeed, appear.

5.4 Example Edits and Workflow

This section demonstrates the generality of our representation by performing several edits to a few scenes lit with indirect illumination. To generate all results in this chapter, we design the edits in an interactive relighting engine that supports our representation and then export them to an offline renderer for high quality imagery. This workflow takes direct advantage of the fact that our representation is renderer-independent and can be easily supported using different algorithms in any renderer, just like indirect illumination is. Furthermore, the use of an interactive tool for previewing lighting coupled with a high quality offline rendering for final output is typical of current cinematic lighting workflow [PVL05, RKS07], further proving the practicality of our representation. We present implementation details in the following section.

We control indirect illumination by first selecting view and gather samples (4D selection), using a painting interface, then applying arbitrary operations to the transport coefficients which are finally mapped to scale and offset values. While we do not focus on a specific workflow, we found this process intuitive and fast and thus we will present it in this chapter as an example of possible integration with cinematic production tools. Also, we remind the readers that the edits shown are just examples of what our representation can achieve, but we expect different artists and cinematic "styles" to require a variety of different effects. Our framework naturally supports any new effect, and has the substantial benefit that new edits can be implemented simply by altering the scale/offset coefficients and do not require any other change in the renderer and production assets (geometry, shaders, etc.).

5.4.1 Example Workflow

Indirect edits start by selecting view and gather samples using a painting interface. This essentially selects the block of the transport matrix at the intersection between the rows and columns corresponding to the selected view and gather samples respectively. Our prototype provides both painting and per-object selection mechanisms and supports smooth, non-binary selections. Arbitrary modifications are then applied to this block by projecting user edits to scale and offset coefficients and compressing them in the wavelet domain.

In our examples, we fixed the main camera for simplicity of explanation. However, our example workflow is more similar to a lighting design system in production. In the real-time renderer, the main camera is fixed (in a way similar to [PVL05]) and other viewports are provided for the user to navigate around the scene and work with surfaces not visible from the main camera.

5.4.2 Example edits

Figure 5.2 shows three samples edits, where we manipulate hue/saturation/value of reflected indirect light in (b), quadratic falloff coefficient in (c) and directly paint additive coefficients in (d). In the first example, we selected the blue wall as the casting object, the box and the cone as receiving objects and increased saturation. Notice that glossy objects are handled as easily as diffuse in this representation (see Figure 5.7 for more examples of glossy edits). In the second example, we changed the quadratic falloff coefficient of light reflected off the sphere and thus made more light arrive at the box and the cone. The third example is similar to the first, but uses painted selection (the floor) instead of per-object selection. Note that

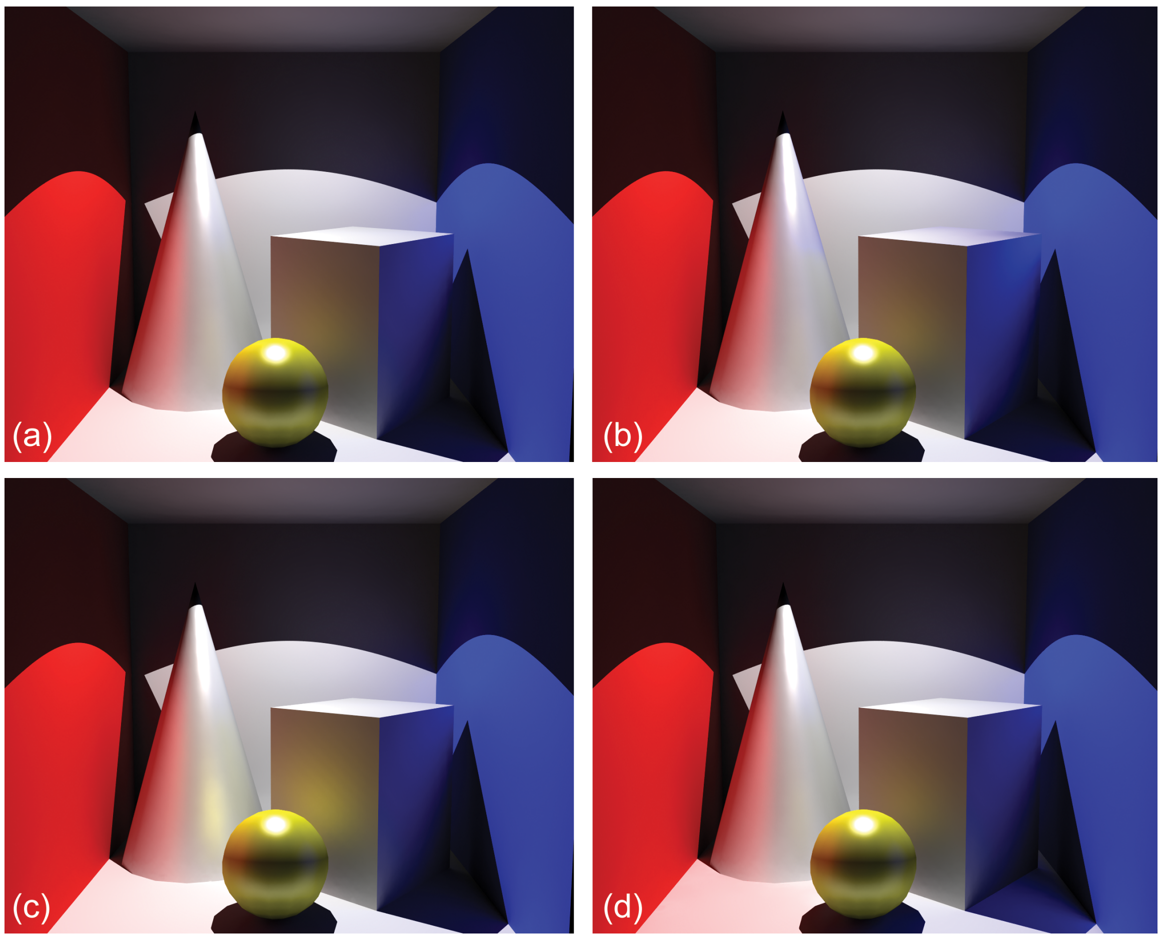


Figure 5.2: Different 4D edits were applied to original indirect illumination in image (a). We modified hue and saturation of indirect lighting reflected off the blue wall towards the glossy cone and the diffuse box in image (b). In image (c), we decreased the quadratic falloff coefficient from the glossy sphere towards the cone and the box. And finally, in image (d) we painted a 4D selection to emphasize color bleeding on the floor.

these are just a few example of possible edits. We also have successfully experimented with removing indirect shadows, or selectively increasing object albedos for indirect lighting, etc.

5.5 Implementation

We have implemented our representation in two applications: an interactive relighting engine to preview the edits coupled with a painting interface to perform them, and an offline renderer to compute final frames. For each, we have adapted a known algorithm to support our editing representation. All results were created by concurrent use of these two tools. Having a consistent representation for lighting cheats made it possible to have them interact efficiently. This section describes implementation details of the interactive relighting engine and the offline renderer. Once again, we remind the reader that these are just examples of how easy it is to adapt rendering algorithms to support our representation; we expect others to easily include our edits in their preferred systems.

5.5.1 Interactive Relighting Engine

Our interactive relighting algorithm is based on the direct-to-indirect transfer approach of [HPB06]. In their algorithm a transport matrix T is precomputed between a set of view samples v_i, chosen as the pixel centers, and a set of gather samples g_j, uniformly sampled in the scene (see Section 3.1.1 for additional details). The precomputed matrix is compressed by projecting each row in wavelet space and culling small coefficients to obtain an approximate sparse matrix T^w. For each change in the lighting, the direct illumination of gather samples is computed and multiplied column-wise by the sparse transfer matrix. In our implementation,

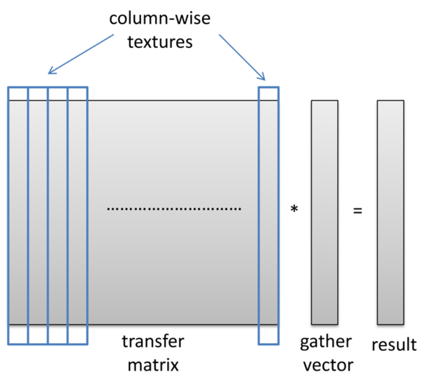


Figure 5.3: The transfer matrix is sliced up vertically and stored in textures. Each texture contains 4 wavelet coefficients. We used 100 wavelet coefficients in our examples, i.e. 25 textures to represent the whole matrix.

we also encode each row sparsely, but use row-wise instead of column-wise multiplication. To support indirect illumination cheats, scale and offset matrices are projected in the same wavelet basis as the transport operator and lossily compressed. To support animation, we simply keep in memory the transfer matrix and the scale/offset coefficients for each keyframe.

Figure 5.3 demonstrates the data layout for row-wise multiplication. We slice the transfer matrix vertically and store 4 wavelet coefficients per view sample in one texture. The gather vector is also stored as a texture and is encoded using Algorithm 7 in Section 4.6. Since this algorithm alters the order of wavelet coefficients, we need to remap them before multiplication. To accomplish this, we use index textures which encode the mapping from Figure 4.6. Since we need to store both u and v coordinates, we can only store coordinates for 2 wavelet coefficients per view sample in one texture. The pseudocode of the multiplication algorithm is shown in Algorithm 8 and the corresponding fragment shader in shown in Algorithm 9.

Algorithm 8: Row-wise matrix-vector multiplication

```
Data: TransferMatrix — diffuse or glossy gather-to-view matrix
passes = TransferMatrix.numberWaveletCoeffs / 4;
for $i \in < 0, passes - 1 >$ do
    ShaderManager.SetTexture("valueTexture", TransferMatrix.GetValueTexture(i);
    ShaderManager.SetTexture("indexTexture1", TransferMatrix.GetIndexTexture(2 *
    i);
    ShaderManager.SetTexture("indexTexture2", TransferMatrix.GetIndexTexture(2 *
    i + 1);
    DrawFullScreenQuad();
end
```

Algorithm 9: Row-wise matrix-vector multiplication shader

```
Data: uv — pixel location in UV space
Output: pixel color
begin
    float4 res = float4(0.0f, 0.0f, 0.0f, 0.0f);
    float4 value;
    value.rgba = tex2D(SamplerValue, uv);
    float4 coeff01;
    coeff01.rgba = tex2D(SamplerIndex1, uv);
    res += value.r * tex2D(SamplerVector, coeff01.rg) +
            value.g * tex2D(SamplerVector, coeff01.ba);
    float4 coeff23;
    coeff23.rgba = tex2D(SamplerIndex2, uv);
    res += value.b * tex2D(SamplerVector, coeff23.rg) +
            value.a * tex2D(SamplerVector, coeff23.ba);
    return res * fOneOverPi;
end
```

5.5.2 Edit Updates

As described above, the interactive renderer is unaware of how scale and offset coefficients are modified. During an editing session, it is the responsibility of the user interface code to update these coefficients interactively, as would normally happen with direct illumination light parameters. Once again, our simple representation makes this decoupling possible. We compute scale/offset coefficients every time the user commits an edit, every time direct lighting changes and every time a non-linear edit is applied (such as gamma edit, which is non-linear with respect to lighting values).

Sample edits in Figure 5.2 are implemented as follows. In Figure 5.2(c) we show editing of the quadratic falloff coefficient. This is an example of a geometry edit, i.e. an edit that alters the geometry term of the rendering equation. Our implementation of this edit consists of unrolling the matrix row from the wavelet space, then multiplication of matrix elements by a modification factor and finally projection back into wavelets. Then we divide the original and new matrix elements to obtain scales and offsets. Algorithm 10 shows the pseudocode of this operation.

Using a row-based encoding lets us access the matrix quickly since we only need rows corresponding to edited view samples. To gain further speed, we adaptively subsample the rows using irradiance caching [WRC88]. Wavelet triple product [NRH04] can further optimize the updates and is left for future work.

Algorithm 10: Computation of scale/offset coefficients for modification by an arbitrary edit

```
Data: ViewSamples — set of selected view samples, i.e. screen pixels
Data: TransferMatrix — glossy or diffuse gather-to-view transfer
foreach $s \in ViewSamples$ do
    encodedRow = TransferMatrix.GetRow(s);
    rawData = decodeHaarCPU(encodedRow);
    for $i \in < 0, rawData.Length - 1 >$ do
        modifiedData[i] = applyEdit(rawData[i].value);
    end
    modifiedRow = encodeHaarCPU(rawData);
    for $i \in < 0, modifiedData.Length - 1 >$ do
        if $rawData[i].value == 0$ then
            scaleData[i] = 0;
            offsetData[i] = modifiedData[i].value;
        else
            scaleData[i] = modifiedData[i].value / rawData[i].value;
            offsetData[i] = 0;
        end
    end
    encodedScales = encodeHaarCPU(scaleData);
    encodedOffsets = encodeHaarCPU(offsetData);
end
```

Figure 5.2(b) demonstrates editing of hue/saturation/value of indirect lighting computed between two sets of points. This is a non-geometry edit, as it doesn't require modification of the geometry term. In our implementation we first compute light transport between two sets of points using the underlying rendering algorithm [HPB06]. We store this information as a 2D texture (`tex1`). Then, we execute the user edits (i.e. change HSV in this example) and store the result as another texture (`tex2`). To obtain scales and offsets, we divide `tex2` by `tex1`. If division by zero occurs, offsets are set to values from `tex2` and scales are set to zero. Otherwise, scales are set to the result of the division and offsets are set to zero (see shader in Algorithm 11, where `tex1` is bound to texture sampler SamplerTexOrig and `tex2` is bound to texture sampler SamplerTexMod). Implementation of the edits in Figure 5.2(d) is the same, the images only differ in the selection mechanism used (per-object vs. painting).

Algorithm 11: Computation of scale/offset coefficients in a GPU shader

```
Data: uv — pixel location in UV space
Output: outScale — computed scale coefficient
Output: outOffset — computed offset coefficient
begin
    float4 fMod = tex2D(SamplerTexMod, uv);
    float4 fOrig = tex2D(SamplerTexOrig, uv);
    float4 scale = float4(1.0f, 1.0f, 1.0f, 1.0f);
    float4 offset = float4(0.0f, 0.0f, 0.0f, 0.0f);
    if fOrig.rgb <= epsilon then
        outScale.rgb = float3(0.0f, 0.0f, 0.0f);
        outOffset.rgb = fMod.rgb;
    else
        outScale.rgb = fMod.rgb / fOrig.rgb;
    end
end
```

It is important to realize that even though multiple edits can be applied over the same sets of gather/view points, the wavelet compression doesn't cause any major loss in quality for two reasons. First, we keep edits uncompressed while the user works with them and only project them into wavelets when they have been committed (i.e. finalized). Second, we only store modifications to the matrix, which are inherently of very low frequencies and therefore far less susceptible to cause artifacts.

5.5.3 Offline Renderer

We compute high quality images using an offline renderer based on final gathering accelerated by irradiance and radiance caching [WRC88, KGP05]. Multiple bounces are handled by using photon mapping. Our algorithm for indirect lighting is the same as used in most Renderman implementations, showing the feasibility of a deployment in production. While our realtime implementation uses point sampling, we interpolate the matrix rows for our final rendering. For primary rays hitting a surface at view point v, we locate the three nearest view samples (we use a kd-tree to locate nearest view and gather samples) and interpolate the wavelet coefficients in the corresponding matrix rows (see Figure 5.4). The interpolated coefficients are then decoded from the wavelet domain and retained for fast lookup in final gathering.

For each gather point, g, hit by a gather ray emitted from v, we let the renderer compute illumination as usual (e.g. by a photon map lookup or by evaluating direct illumination at g) and subsequently alter the computed illumination by the offset and scale factor looked up

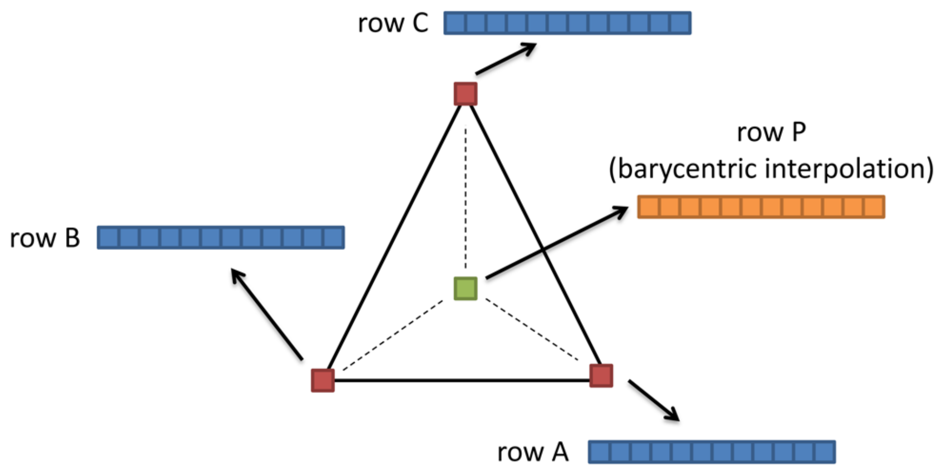


Figure 5.4: View sample interpolation. Scale/offset rows for a view sample are computed by barycentric interpolation from three nearest samples. An operation like this is useful when image resolution changes.

from the currently decoded matrix row. To index into the row, we simply locate the nearest gather sample to g. Nearest neighbor filtering is sufficient here, since the illumination is averaged in final gathering. Irradiance caching helps reduce the overhead since the matrix row interpolation and wavelet decoding is only performed when a new cache record is added. The process is outlined by the pseudocode in Algorithm 12.

Animation is performed by temporal edit interpolation as described in Section 5.3.4. Figure 5.5 demonstrates the process. We first interpolate scale/offset rows in the previous and next keyframes and then linearly between the keyframes.

Algorithm 12: Applying gather point scale/offset modifications in irradiance caching

```
Data: Rays — set of rays to be traced
Data: GatherScales — scaling coefficients for gather points
Data: GatherOffsets — offset coefficients for gather points
Output: outColor — computed pixel color

foreach ray ∈ Rays do
    Point p = trace(ray);
    sampleContrib = evalIllumination(p);

    gatherSample = kdTree.FindNearest(p);
    sampleContrib = sampleContrib * GatherScales[gatherSample.index] +
                    GatherOffsets[gatherSample.index];

    outColor += sampleContrib;
end
```

Scene	Edits size	T_{iCheat}	T_{normal}
Fig. 5.1	7MB	4:20	4:00
Fig. 5.2	7MB	0:55	0:40
Fig. 5.7	7MB / 9MB	2:40	2:30

Table 5.1: Size of the edits matrices and final renderer performance (in minutes:seconds) for the example scenes in the paper. T_{iCheat} gives the render time with the edits applied, T_{normal} without.

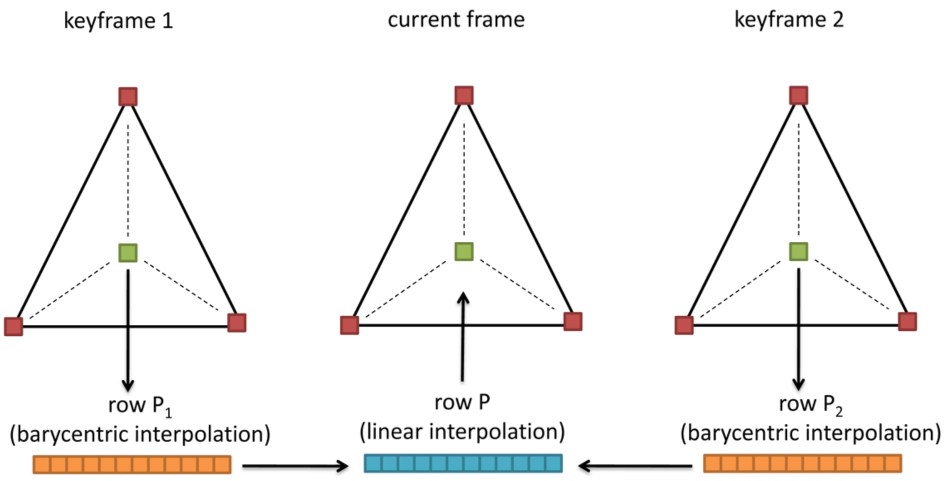


Figure 5.5: Temporal interpolation. We first interpolate scale/offset rows in nearest keyframes and then between keyframes.

5.6 Results

The performance results reported in this section were measured on a PC with Intel Core Duo 6850 processor, 4GB RAM, GeForce 8800 Ultra GPU running Windows XP. We rendered all our images at 640×480 resolution, with up to 36 samples per pixel in the high quality renderer.

The real-time renderer performs at frame rates around 150 FPS for 300k view samples, 64k gather samples, and image resolution of 640×480. The matrix updates are performed at 10 frames per second, a speed still largely sufficient for comfortable work.

The size of edit representation depends on the number of affected view samples and the number of coefficients stored per matrix row (which in turn depends on the nature of the

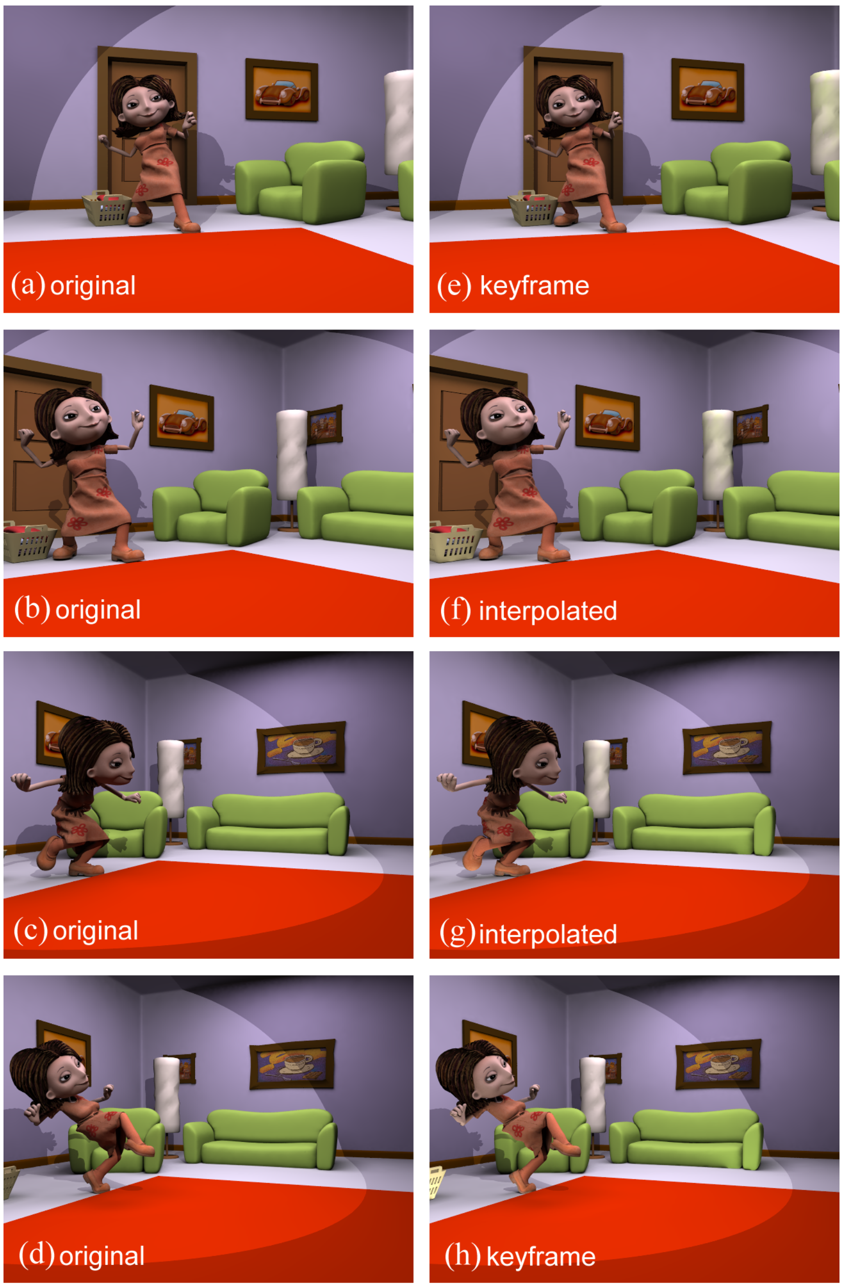


Figure 5.6: Keyframed edits. Original (first column) and modified (second column) frames from an animated sequence are shown. Frames (e) and (h) were selected as keyframes and lighting was designed for them. Frame (e) was left intact and in frame (h), indirect lighting arriving at the character, laundry basket and the couch was modified. Edits in images (f) and (g) were interpolated and show smooth transition of the edits between two keyframes.

edit). The maximum of 100 coefficients per row was sufficient for all the example edits shown in this chapter. Table 5.1 summarizes the matrix sizes for various scenes in this chapter.

Overhead implied by applying edits in final rendering is, for the most part, due to interpolating and decoding matrix rows and locating the nearest gather sample for each final gather ray. As such, it is proportional to the size of the edit matrices. In our tests, the overhead never exceeded one minute for a frame of 640×480 pixels. The rendering times are summarized in Table 5.1.

5.7 Conclusions

We have presented a representation for artistic control of indirect lighting, with application to computer cinematography. Rather than representing edits as procedural shader modifications, we numerically sample the edits using scale and offset coefficients and take advantage of their nature to efficiently compress and render the adjustments. This representation is general, easy to manipulate, and renderer-independent, allowing simple and effective workflow in production environments.

Our global illumination pipeline consists of three main components. First, a real-time global illumination algorithm with an editing interface allows artists to design edits to indirect lighting. The interface provides a real-time feedback and supports a variety of different editing operations. In fact, the number and variety of effects depends solely on user's cre-

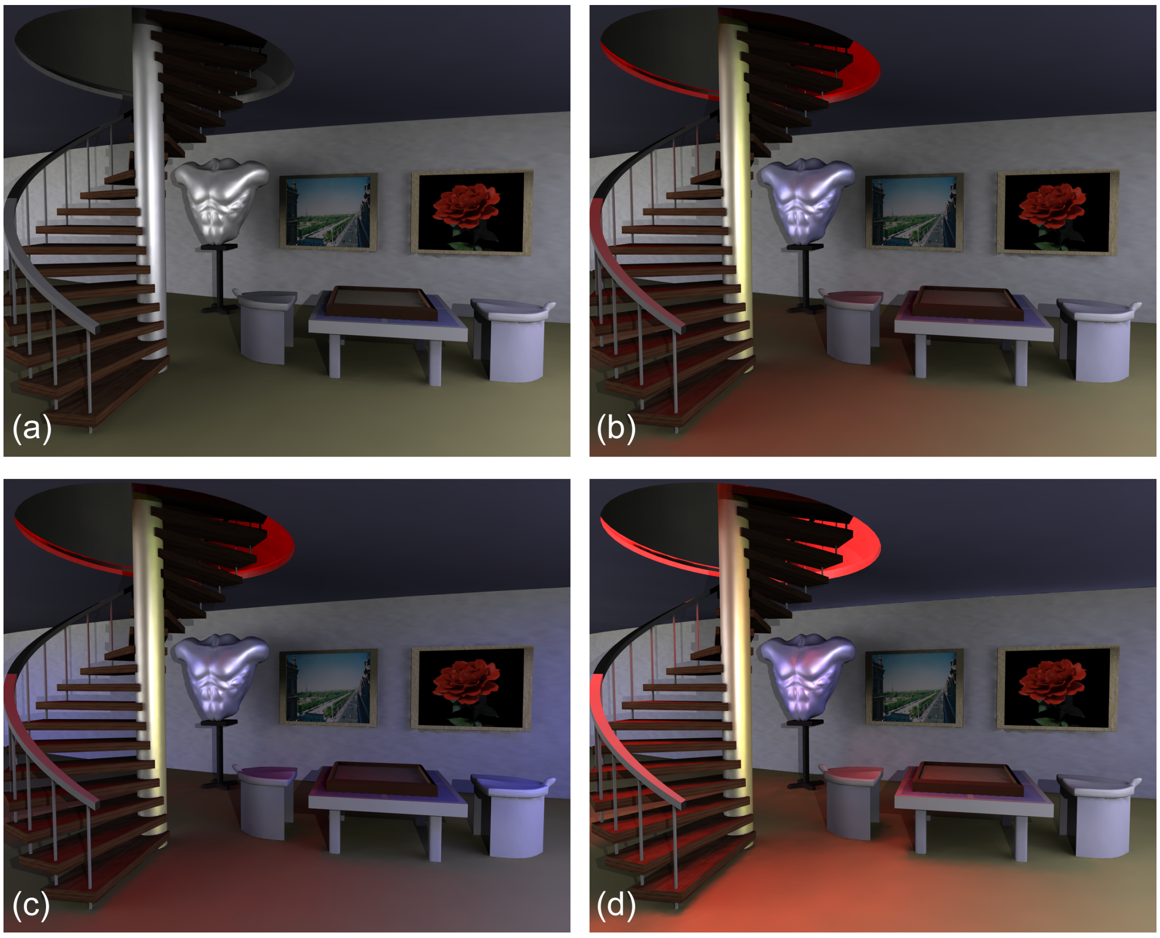


Figure 5.7: Examples of edits on glossy surfaces. (a) shows original rendering with global illumination and the remaining three images show our edits. In (b) we made the cylinder above the staircase emit red indirect light, changed indirect hue on the glossy staircase piece and on the glossy statue torso. In (c), we added a more bluish tone by reflecting more indirect light off the ceiling toward the floor and the wall. In (d) we modified hue of received indirect light.

ativity. Our representation is general enough to encompass any edit, which conforms to the gather-to-view light transport model.

Second, scale/offset representation is used to store the edits designed in the real-time user interface. Our representation is oblivious to the way edits are implemented and it is the job of the interface to perform them. In Section 5.5.3 we showed how common edits can be converted to the scale/offset representation. The same approach can be used by new edits.

And third, an offline high-quality renderer is used to obtain final images. We use an irradiance caching and radiance caching-based renderer. We render final images in significantly higher resolutions with antialiasing and use the scale/offset representation to apply edits to indirect lighting. The low-frequency nature of the scale/offset representation allows us to perform barycentric reprojection as well as temporal interpolation for animations.

CHAPTER 6

VISIBILITY EDITING FOR ALL-FREQUENCY SHADOW DESIGN

We have shown that the combination of a real-time designing interface, powerful representation and high-quality output can significantly boost cinematic production pipelines in terms of efficiency as well as quality. We demonstrated how this approach can be applied to indirect lighting. In this chapter we expand our focus and work with another important visual aspect — all-frequency shadows.

6.1 Introduction

The rendering equation tells us that the appearance of surfaces depends on three components — materials, lighting and scene geometry. In the context of the rendering equation, scene geometry is sufficiently represented by the visibility function (see Section 2.3.1). We again take a non-physical approach — we decouple visibility from lighting, develop an editing framework and finally render modified all-frequency shadows while keeping lighting intact.

We achieve real-time rendering by precomputing a transport operator as the product of lighting and BRDFs, similarly to PRT techniques, and compress it in the wavelet domain. Scene visibility is also precomputed and compressed in the wavelet domain. We generalize visibility from binary to colored (fractional three channel) to allow for more control. Visibility edits are performed by altering the precomputed visibility data. After editing, we render the final image by multiplying the visibility and the transport operator.

Simple edits to the visibility function allows artists to control shadow position, softness, color and gradient. To achieve more complex shadow edits, such as arbitrary projective transforms or applying arbitrary filters, we introduce a novel representation for shadows. We compute visibility ratios (VRs) by comparing two different renderings — the original rendering and a secondary rendering obtained by using modified visibility information.

All edits proposed in this work support animated environments and can be keyframed. When multiple shadows are visible, either from multiple bright spots in the environment map or due to multiple objects, designers might want to edit only a subset of all shadows. We facilitate this by introducing a sketch-based selection algorithm such that edits can be localized in the angular (lighting) and spatial domain.

Our work contains the following major contributions:

- Decoupling of visibility and lighting, which allows artists to interactively design all-frequency shadows independently of lighting.
- A shadow selection algorithm to localize shadow edits in the angular (lighting) and spatial domain.
- A shadow representation in terms of visibility ratios and fractional three-channel visibility representation for more artistic control.

- A wide range of shadow editing operations along with numerical parametrization for animation.

6.2 Related Work

Shadows are a very important topic in computer graphics and have been the subject of numerous research publications. However, most of the research has focused on rendering, rather than editing/designing shadows. To the best of our knowledge, ours is the first work to focus on editing/design of all-frequency shadows.

The work on user interfaces for shadow manipulation [PTG02] allows users to design sharp shadows, but only works by modifying parameters of discrete light sources. The closest to our work is an image-space algorithm for controlling four different parameters of shadows cast by point light sources [DCF07]. While their work focuses on rendering, we put emphasis on editing of visibility, shadow selection, shadow extraction and all-frequency environments. Furthermore, by editing visibility, we do not limit ourselves to image-space operations, but also support a variety of advanced edits (see Section 6.6).

Poulin and Fournier [PF92] present a simple user interface for shadow manipulation, but address neither shadow selection nor advanced edits such as recoloring. Poulin et al. [PRJ97] also demonstrate how light positions can be computed from shadows, an approach which only allows for global edits. [CGC03] describe a method for extracting shadows from one natural scene and inserting them into another, but do not handle any editing operations. Similarly, [FHD02] propose a technique for shadow removal from photographs, but again, do not handle any editing.

A whole body of work focuses on interactive relighting [RKS07], [SZC07], [WTL06], [SZS08], [HPB06], lighting design [KPC93], [PBM07], [TL04] and BRDF editing [BOR06], [CPK06], [BED08]. These and our work share the common aspect of interactive/real-time editing, but to the best of our knowledge, ours is the first one to focus on visibility editing with all-frequency lighting.

A very common and efficient way of compressing precomputed datasets uses projection to wavelet space. Haar wavelets have been used to render scenes with time-varying all-frequency illumination [NRH03]. Efficient evaluation of triple product wavelet integrals has enabled rendering of glossy objects with varying viewpoint [NRH04]. Further extensions have enabled rendering of dynamic glossy objects [SM06] and near-field illumination [SR09]. Our work utilizes wavelets to compress precomputed datasets too, but differs in the way we factor the components of the rendering equation.

In order to allow artists to achieve predetermined visual goals, researchers have devised techniques that do not rely on physically-correct rendering, but rather provide a wide variety of editing operations. Besides techniques for lighting design mentioned at the beginning of this section, algorithms exist to facilitate editing of reflections [ROT09], all-frequency lighting [OMS07] or subsurface scattering [STP09]. Our shadow editing operations complement these techniques and further increase the expressive power of digital artists.

6.3 Visibility Decoupling

One of the main goals of our work is to allow editing of shadows independently of lighting. Currently, two common approaches exist to solve this problem. The first one requires that artists render the scene into multiple layers, edit them separately using image processing techniques and use a compositing software for final blending. The second approach requires using of different light sources for rendering of shadows and different light sources for lighting [Bar97]. Our work proposes a technique, which allows artists to extract and edit shadows without using different light sources and makes shadow editing an inherent part of the design process (as opposed to being just a part of post-processing).

In order to decouple visibility from lighting, we proceed in a fashion similar to other PRT approaches. We start with the following formulation of the rendering equation:

$$L(x, \omega_o) = \int_{\Omega} L(\omega_i) f_r(\omega_i, \omega_o) V(x, \omega_i)(n_x \cdot \omega_i) d\omega_i$$

where $L(x, \omega_o)$ is computed outgoing radiance at point x in direction ω_o, $L(\omega_i)$ is incident lighting from an environment map in direction ω_i, f_r is the BRDF, $V(x, \omega_i)$ is binary visibility between point x and direction ω_i, n_x is the surface normal and $(n_x \cdot \omega_i)$ is the cosine term at point x.

Since our goal is to edit shadows by manipulating visibility, we group the non-visibility terms together and keep visibility separate:

$$L(x, \omega_o) = \int_\Omega V(x, \omega_i) T(x, \omega_i, \omega_o) d\omega_i$$

where $T(x, \omega_i, \omega_o) = L(\omega_i) f_r(\omega_i, \omega_o)(n_x \cdot \omega_i)$.

6.4 Framework

In order to maintain high-quality output and low storage requirements, we propose to use a wavelet-based framework for rendering. After projecting visibility to the wavelet basis, we obtain:

$$L(x, \omega_o) = \int_\Omega T(x, \omega_i, \omega_o) \sum_j v_j \Psi_j(x, \omega_i) d\omega_i = \sum_j v_j T_j(x, \omega_o)$$

where v_j are visibility coefficients in the wavelet basis and $T_j(x, \omega_o)$ are precomputed transport coefficients containing incident lighting and BRDF. Depending on further application requirements, the transport coefficients can be precomputed for two configurations.

We allow the designer to work in a fixed viewpoint setting with arbitrary BRDFs. Direction ω_o then becomes only a function of location x (image pixels in this configuration):

$$T_j(x) = \int_\Omega L(\omega_i) f_r(\omega_i, \omega_o(x))(n_x \cdot \omega_i) \Psi_j(x, \omega_i) d\omega_i$$

For free viewpoint scenarios, we limit the BRDF to be diffuse-only, hence eliminating outgoing direction ω_o from the equation completely:

$$T_j(x) = \int_\Omega L(\omega_i) f_r(\omega_i)(n_x \cdot \omega_i)\Psi_j(x, \omega_i) d\omega_i$$

Locations x in this configuration correspond to surface sampling points. We considered two options — per-vertex and per-texel sampling — and finally opted to use the latter one. While per-vertex sampling is very intuitive, scenes with high-frequency geometry and high-frequency shadows become undersampled and require an overwhelmingly large number of vertices. Per-texel sampling alleviates these problems by sampling the geometry according to surface UV coordinates, thus allowing for finer control and a simple level-of-detail mechanism based on texture resolution.

Choosing which configuration to use largely depends on the application of interest. While movie productions will benefit from the fixed viewpoint configuration (since lighting and shadow design is performed after the animation and camera paths have been fixed), real-time applications such as video games might prefer the free viewpoint scenario.

We represent both the transport operator and visibility information as coefficient matrices, with matrix columns corresponding to sampling directions ω_i and matrix rows corresponding to surface locations x. Since we use cube maps to represent environment lighting, we compute one visibility and one transport matrix for each cube map face visible from the positive side of the surface plane.

Visibility matrix $V_f(x)$ (cube map face f, location x) is precomputed by ray tracing as illustrated in Algorithm 13. For each sampling location (an image pixel or a surface texel), we shoot rays through cube map texels corresponding to the visible hemisphere around the normal and record visibility information.

Algorithm 13: Precomputation of the visibility matrix on the CPU.

```
Data: samples — pixel/texel samples
Data: matrix — visibility matrix
foreach $face \in \{FRONT, LEFT, RIGHT, TOP, BOTTOM, BACK\}$ do
    foreach $s \in samples$ do
        for $i \in < 0, cubeFaceSize - 1 >$ do
            for $j \in < 0, cubeFaceSize - 1 >$ do
                v = computeRayDirection(i, j, cubeFaceSize, face);
                if $dotProduct(v, s.Normal) > 0$ then
                    bool intersect = findNearest(s.Position, v);
                    if $intersect$ then
                        matrix[face].recordVisibility(s.index, i, j, 0);
                    end
                    else
                        matrix[face].recordVisibility(s.index, i, j, 1);
                    end
                end
                else
                    matrix[face].recordVisibility(s.index, i, j, 0);
                end
            end
        end
        matrix[face].encodeTo2DHaar(s.index);
        matrix[face].nonlinearCompression(s.index);
    end
end
```

Transport matrix $T_f(x)$ is precomputed by sampling the environment map and multiplying with the BRDF (see Algorithm 14). The structures of both algorithms are very similar, the only difference is the type of data stored in the matrix.

Algorithm 14: Precomputation of the transport matrix on the CPU.

```
Data: samples — pixel/texel samples
Data: matrix — visibility matrix
foreach face ∈ {FRONT, LEFT, RIGHT, TOP, BOTTOM, BACK} do
    foreach s ∈ samples do
        for $i \in < 0, cubeFaceSize - 1 >$ do
            for $j \in < 0, cubeFaceSize - 1 >$ do
                v = computeRayDirection(i, j, cubeFaceSize, face);
                cosineTerm = dotProduct(v, s.Normal);
                if cosineTerm > 0 then
                    brdf = evaluateBRDF(s.brdfType, v);
                    l = sampleEnvironmentMap(v);
                    matrix[face].recordTransport(s.index, i, j, l * brdf * cosineTerm);
                end
                else
                    matrix[face].recordTransport(s.index, i, j, 0);
                end
            end
        end
        matrix[face].encodeTo2DHaar(s.index);
        matrix[face].nonlinearCompression(s.index);
    end
end
```

Upon completing precomputation for each matrix row, we reduce the data size by employing non-linear wavelet compression. We compute 2D Haar wavelet coefficients, weigh them with areas of corresponding basis functions and discard the coefficients of lowest magnitudes. The compression yields wavelet matrices $\mathbf{V}_f(x)$ and $\mathbf{T}_f(x)$.

Given the transport and visibility matrices in wavelet space, we render the final image by computing dot products of corresponding matrix rows:

$$c_o(x) = \sum_f \mathbf{V}_f(x) \cdot \mathbf{T}_f(x)$$

where $c_o(x)$ is the color at location x and $\mathbf{V}_f(x) \cdot \mathbf{T}_f(x)$ is the dot product of matrix rows at location x for cube map face f. While it is possible to perform the computation on the GPU, we render using the CPU mainly due to the simplicity of implementation. For per-pixel sampling, we directly display the rendered image on the screen and for per-texel sampling, we store the result as per-object textures and rasterize the scene using the GPU.

6.5 2D vs. 1D Haar wavelets

In our previous work on lighting design with global illumination, 1D Haar wavelets provided sufficient compression for the lighting information. In the case of visibility, however, a different approach is necessary. Figure 6.2 demonstrates that even with 400 1D Haar wavelet coefficients, the quality of shadows decreases significantly. 2D Haar wavelets, on the other

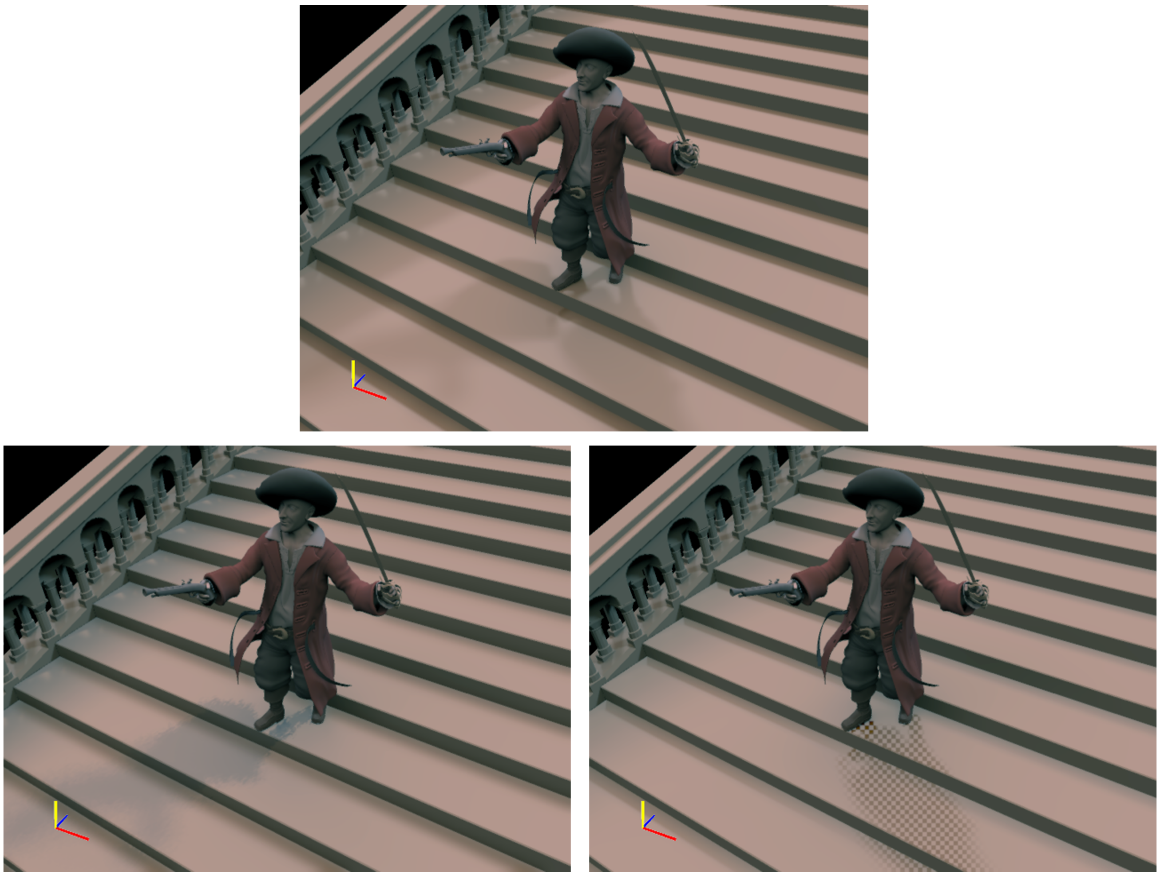


Figure 6.1: (Top) Original rendering. (Bottom Left) Gradient and recoloring applied to the shadow of the railing, artistic filter applied to the left shadow of the pirate, right shadow removed. (Bottom Right) Railing shadow removed, gradient removal applied to self-shadowing on the stairs, shadow pattern applied to the shadow of the pirate.

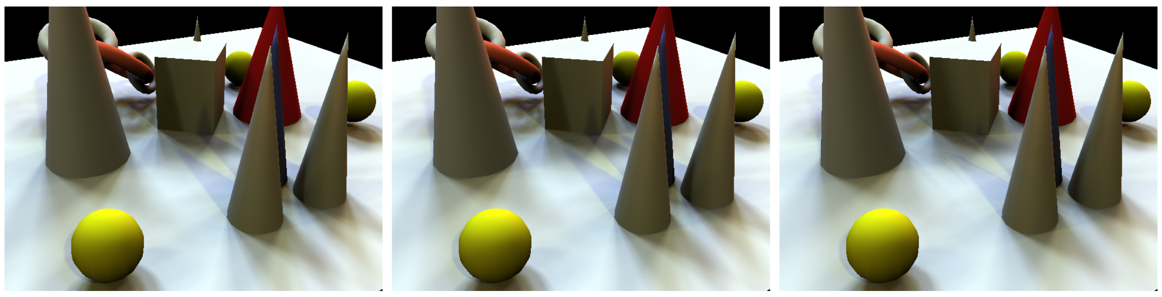


Figure 6.2: Comparing image quality. (Left) Reference ray traced image. (Middle) 2D Haar wavelets. (Right) 1D Haar wavelets. The quality of the shadows rendered using 2D Haar wavelets is comparable to the shadows in the reference image and clearly superior to the shadows in the image rendered using 1D Haar wavelets. We used 400 wavelet coefficients in each case and sampled the environment map with 128×128 rays for each face.

hand, are capable of preserving even the sharpest shadows in the image. The main reason for such significant difference in the quality of the shadows, is the frequency characteristics of the encoded information. While indirect lighting contains mainly low frequencies, visibility contains very sharp edges and therefore very high frequencies. 2D Haar wavelets are naturally more suitable for high frequencies than 1D Haar wavelets.

The increased quality comes at the cost of increased computational complexity of the encoding algorithm. Our implementation used *non-standard decomposition* as already introduced in Section 2.6.2 and described in detail in [SDS94]. Algorithm 15 illustrates the non-standard decomposition on the CPU.

The implementation on the GPU is similar to the 1D case and is shown in Algorithm 16.

Algorithm 15: 2D Haar encoder with non-standard decomposition.

```
Data: vector — length is a power of 4
Data: n = $\sqrt{\text{length(vector)}}$
while $n > 1$ do
    for $i \in < 0, n-1 >$ do
        onePassRow(vector, i, n);
    end
    for $j \in < 0, n-1 >$ do
        onePassColumn(vector, j, n);
    end
    n = n / 2;
end
```

Algorithm 16: 2D Haar GPU encoder

```
Data: MatrixA — input vector organized into an M × M array.
Data: n = M
while $n > 1$ do
    rowsParralelGPU(MatrixA, n);
    columnsParallelGPU(MatrixA, n);
    n = n / 2;
end
```

6.5.1 Complexity Analysis

To compare complexities of the CPU and the GPU algorithms, we make the same assumptions as in Section 4.6.1. First, the time complexity of the CPU algorithm is:

$$\begin{aligned} T_{CPU}(N) = \sum_{i=1}^{\log_2 n} \left(2 + 2(6+6+1)\frac{2^i}{2}2^i \right) &= 2\log_2 n + 13 \sum_{i=1}^{\log_2 n} 4^i \\ &= 2\log_2 n + \frac{52}{3}(n^2 - 1) \\ &= \log_2 N + \frac{52}{3}(N-1) = \Theta(\log_2 N + N) \end{aligned}$$

The time complexity of the GPU algorithm is:

$$T_{GPU}(N) = \log_2 \sqrt{N}(8+8+2) = \Theta(\log_2 N)$$

The speedup is the same as in the 1D case:

$$S_{GPU}(N) = \frac{T_{CPU}(N)}{T_{GPU}(N)} = \Theta\left(\frac{N + \log_2 N}{\log_2 N}\right)$$

Work done by the GPU algorithm is:

$$W_{GPU}(N) = 8\sqrt{N}\sqrt{N} \sum_{i=0}^{\log_2 \sqrt{N} - 1} \frac{1}{2^i 2^i} = 8N\left[1 + \frac{1}{3}\left(1 - \frac{4}{N}\right)\right] = \Theta(N)$$

The cost of the GPU algorithm is:

$$C_{GPU}(N) = NT_{GPU}(N) = \Theta(N \log_2 N)$$

The GPU algorithm is again not cost-optimal, since $\Theta(N \log_2 N) \neq O(\log_2 N + N)$. However, it is work-optimal, since $W_{GPU}(N) = O(T_{CPU}(N))$.

6.6 Direct Editing of the Visibility Function

The most straightforward way of editing shadows without affecting lighting is by editing of the visibility matrix. In general, we edit the whole visibility function, without performing selection in the angular domain. Refined selection mechanism will be described in a separate section.

Our approach generalizes the visibility function and stores three-channel fractional quantities instead of one binary quantity. This allows us to implement per-channel edits, such as recoloring. Out of the many other possibilities, we implemented four main visibility edits (see Figure 6.6).

Translation/Rotation Rotating and/or translating the visibility function is the main operation we use to change the shape of the shadow. Translating visibility by vector (v_x, v_y, v_z) results in a shadow transformation we would get if we moved the light in the opposite direction $(-v_x, -v_y, -v_z)$. However, translation by modifying visibility affects only the shadow, whereas translation by modifying lighting would affect the whole image. Preferably, this operation would be implemented by transformation in world coordinates, but we found it sufficient and faster to perform it by operating just on texels of the cube map (see Figure 6.5 for an example of editing on a non-planar surface).

Shadow Removal/Shadow Gradient Shadow removal allows us to remove a shadow from an image. We implement this operation by removing occluders so that all major contributing lights become visible. The shadow removal algorithm is described in detail in Section 6.7.3. Shadow gradient is a generalization, which allows the user to draw a gradient arrow over the shadow and remove it by interpolating its intensity:

$$\mathcal{G}(V_f(x)) = \mathrm{lerp}(V_f(x), \mathcal{L}(V_f(x)), \alpha)$$

where $\mathcal{L}$ refers to the shadow removal operator defined in Section 6.7.3 and α is the gradient value.

Shadow Blur Shadow blur is an operation, which softens the edges of a shadow, giving the illusion of a larger area light source. We implement it by applying a low-pass filter to the visibility function. Since the visibility function is in fact a 2D image of the environment, it can be easily seen that blurring it will blur the resulting shadow as well. Similarly to translation/rotation, we implement this operation in the cube map domain.

The structure of a general algorithm for performing edits of the visibility function is shown in Algorithm 17. In the visibility matrix, one row corresponds to either a view sample or a texel sample. The algorithm iterates over all selected samples, unpacks the visibility data from wavelets, performs the editing operation and finally projects the data back to wavelets.

Algorithm 17: Editing of the visibility function.

```
Data: samples — selected pixel/texel samples
Data: matrix — visibility matrix
foreach face ∈ {FRONT, LEFT, RIGHT, TOP, BOTTOM, BACK} do
    foreach s ∈ samples do
        rawData = matrix[face].inverse2DHaar(s.index);
        matrix[face].recordRow(s.index, applyTransform(rawData));

        matrix[face].encodeTo2DHaar(s.index);
        matrix[face].nonlinearCompression(s.index);
    end
end
```

The actual transformation of visibility is performed in the *applyTransform* function. As already mentioned, our implementation contains edits such as translation, rotation, blur, etc. The general algorithm applies the edits in the pixel domain, however, we need to emphasize that some edits can be performed directly in the wavelet domain as long as they can be expressed as affine transformations. The underlying mathematical theory and implementation can be found in [SR09].

6.7 Editing of Visibility Ratios

While direct editing of visibility already provides novel shadow editing operations, it doesn't allow users to perform advanced edits such as freeform transformations. To address this problem, we introduce a novel shadow representation based on *visibility ratios.*

6.7.1 Visibility Ratios (VRs)

Visibility ratios $v_r(x)$ are quantities computed for each sampling location by calculating the ratio of two colors — color of the original rendering $c_o(x)$ and color after shadow removal $c_r(x)$ (the secondary rendering created by the shadow removal operation). Visualized as an image, visibility ratios represent the shadow extracted from the original rendering.

$$v_r(x) = \frac{c_o(x)}{c_r(x)} = \frac{\sum_f V_f(x) \cdot T_f(x)}{\sum_f \mathcal{L}(V_f(x)) \cdot T_f(x)} \tag{6.1}$$

where $\mathcal{L}(V_f(x))$ is modified visibility after shadow removal. VRs will be equal to 1.0 for sampling locations where no shadow is present and less than 1.0 for sampling locations in shadow. As can be seen from the equation, a singularity occurs when the color after shadow removal equals zero. In practice, however, such situations do not pose any problems, because they only happen when the sampling location is either completely occluded (therefore invisible to the viewer) or when the environment map is completely black.

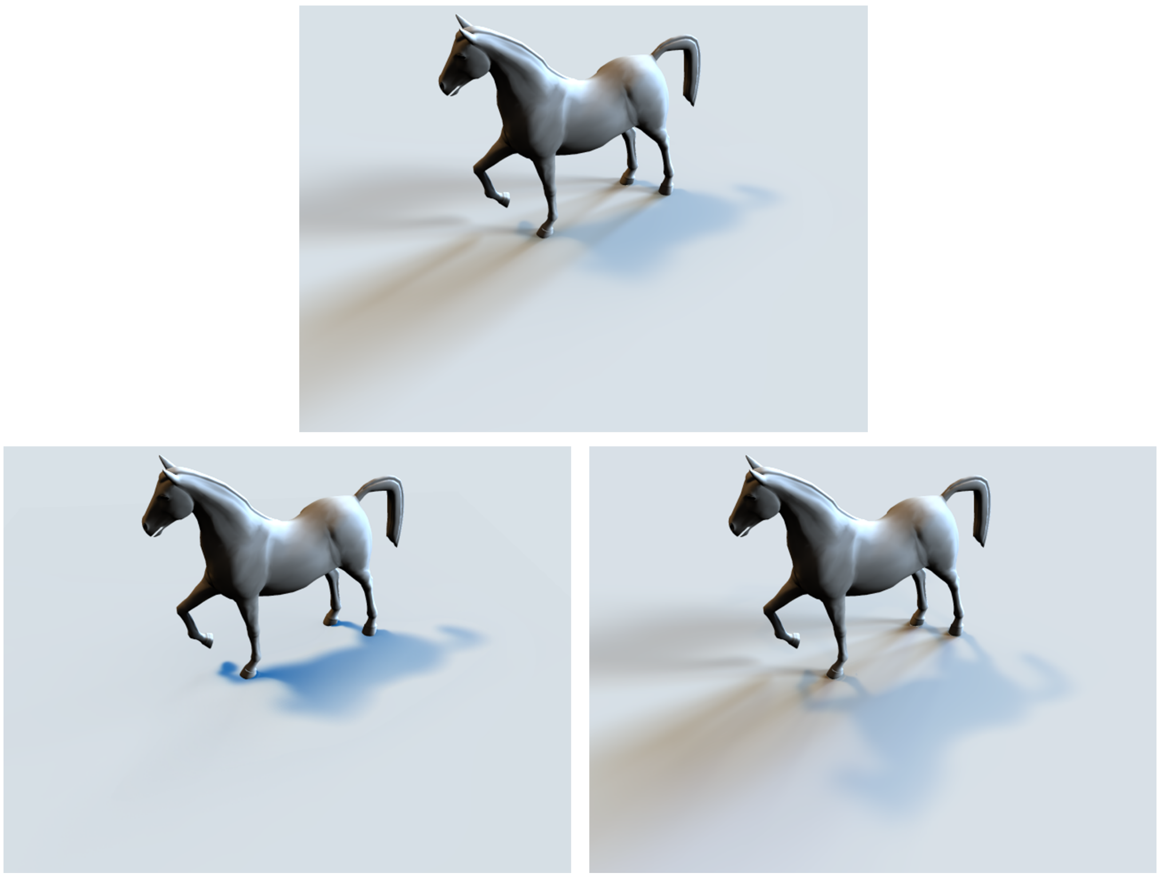


Figure 6.3: (Top) Original rendering. (Bottom Left) Removal of two shadows and gradient edit on the dark shadow. (Bottom Right) Translation of the dark shadow. Notice that only one shadow is affected, everything else stays intact.

Computation of visibility ratios requires two steps — (1) localization of major contributing lights responsible for shadows in the image and (2) shadow removal. Before dwelling into details of VR editing, we describe algorithms for both steps in the following sections.

6.7.2 Locating Major Contributing Lights

The goal of the light localization algorithm is to find lights, which are responsible for shadows in the image. We make the assumption that appearance of each shadow can be attributed to occluders blocking incoming light.

Our algorithm is based on accumulating visibility from shadowed areas and overlaying it with incoming light information. The process starts with the user painting over a shadowed area and identifying shadowed surface sampling locations (set $\mathbb{P}$). Since the sampling locations are in shadow, we know that corresponding visibility functions must be blocking the light (see Figure 6.7).

For each selected sampling location, we read its corresponding row from the visibility matrix (one sampling location corresponds to one matrix row), unproject the visibility back from wavelet space and accumulate its inverse into a buffer. Finally, we normalize the buffer, multiply it with the environment map and perform thresholding.

$$M_f = \text{threshold}(\frac{E_f}{|\mathbb{P}|}\sum_{x\in\mathbb{P}}(1.0 - V_f(x)))$$

where M_f is the located major light and E_f is the environment map. Values above the threshold then belong to the light responsible for the shadow, values below the threshold become 0.

6.7.3 Shadow Removal

We remove shadows by rendering the image with modified visibility information so that all occluded lights become visible. In our implementation, we first ask the user to paint a boundary around the shadow she wishes to edit/remove. Similarly to the light localization algorithm, the painting effectively selects shadowed surface sampling locations. Note that while the painted area for the light localization algorithm is very small, shadow removal requires that the user paint over the whole shadow.

Next, we read rows corresponding to painted locations from the visibility matrix, unproject them from wavelet space and modify. The modification removes occluding geometry by setting visibility values to 1 for incoming directions corresponding to the occluded light found by the light localization algorithm.

$$\mathcal{L}(V_f(x))[i,j] = \begin{cases} 1 & \text{if} \quad M_f[i,j] > 0 \\ V_f(x)[i,j] & \text{otherwise} \end{cases}$$

where $[i,j]$ indexes rows and columns of one cube map face. Finally, we render a secondary image $c_r(x)$ by multiplying the modified visibility matrix with the transport matrix.

It can be easily seen that this algorithm would only work for simple cases, for example scenes with one major contributing light and one object. However, it would fail for more complex scenarios, such as multiple overlapping shadows. To handle all other situations, we generalize the algorithm for three scenarios.

Multiple lights cause multiple shadows of an object to appear. Potentially, the shadows can overlap. In order to handle this situation, we allow the user to perform the selection process one light at a time. In each iteration, the algorithm locates one occluded light and extracts the shadows it causes. This process is repeated for all occluded lights. Figure 6.7 and Figure 6.8 demonstrate the algorithm for two scenes with two major contributing lights.

If silhouettes of multiple objects overlap when viewed from a light's point of view, their shadows overlap too. For such situations, we allow the user to partition the scene into separate *visibility groups*. We maintain a separate visibility matrix for each visibility group and have the shadow removal algorithm take an additional input — visibility group whose shadow we are interested in removing/editing. Multiple visibility matrices increases storage requirements, but pay off with increased robustness of the selection. Figure 6.10 demonstrates how we can split one shadow of one object into five separate shadows.

The third scenario deals with residual shadows, i.e. shadows caused by lights we don't consider to be occluded (termed residual lights). These lights are not responsible for the main shadow(s) in the image, but cause an additional, often barely visible, secondary shadow. In most cases, we are interested in editing/removing the main shadow, but we can use the same algorithms to remove/edit the residual shadow. After locating the major occluded light responsible for the main shadow, we invert the result to locate the residual light. Residual shadows are shown in Figure 6.9.

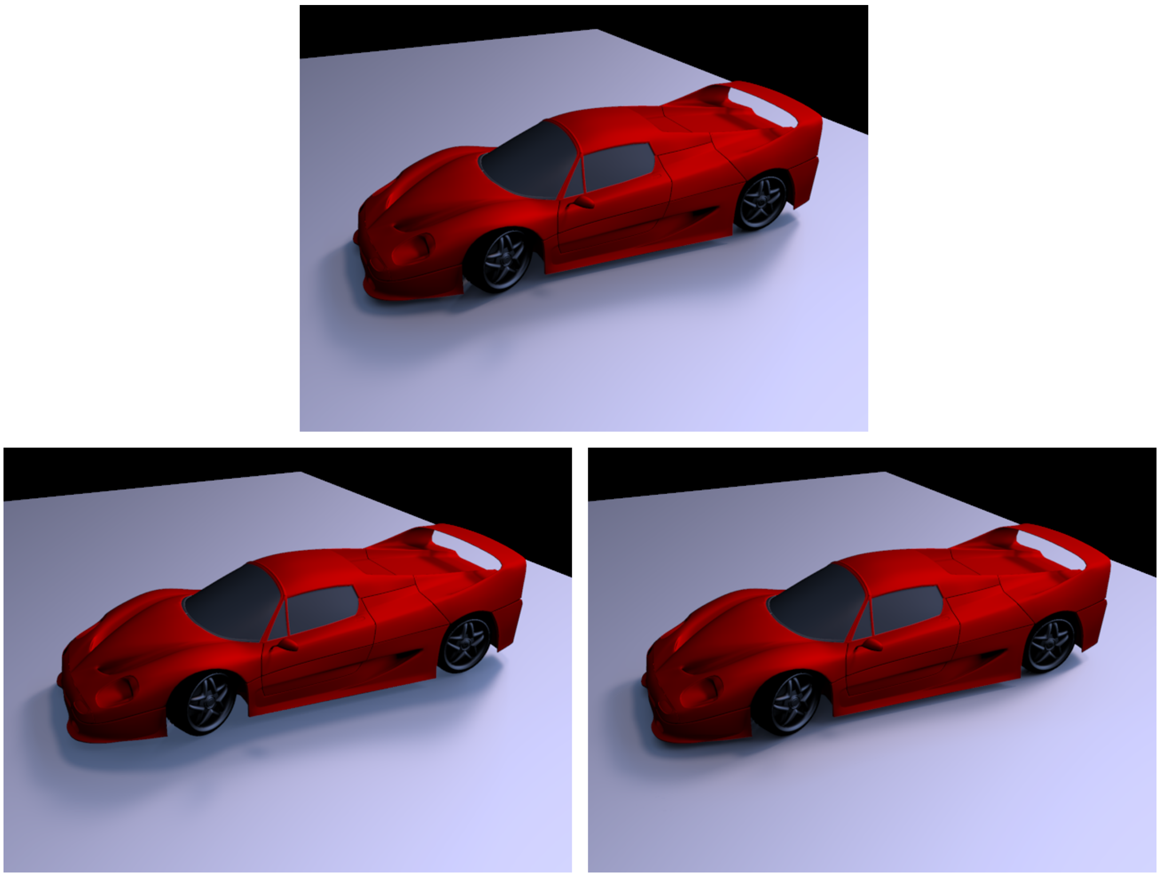


Figure 6.4: (Top) Original. (Bottom Left) The dark shadow of the car was translated toward the viewer. (Bottom Right) Both shadows were translated away from the viewer.

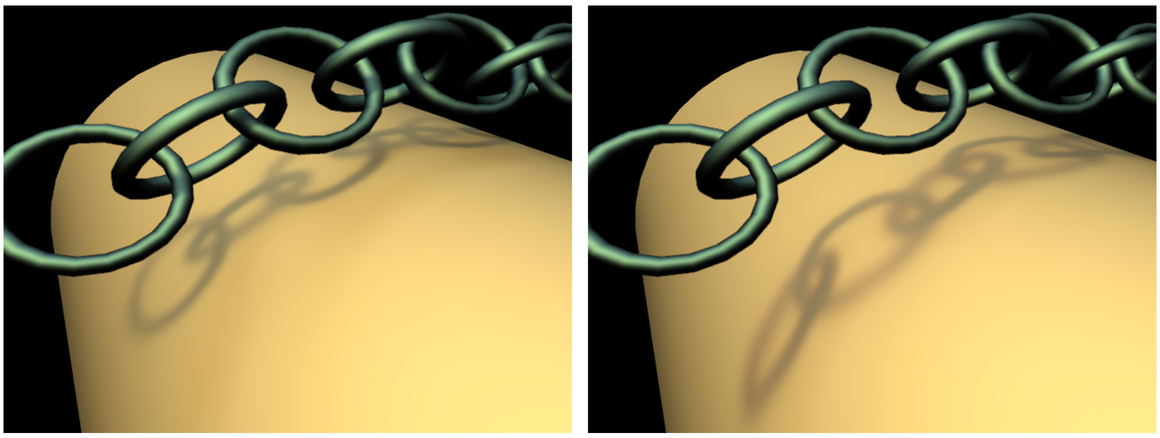


Figure 6.5: Shadows on a curved surface. (Left) Original. (Right) Shadows of the chain rings cast on the curved object were translated. Self-shadowing on the rings was removed. Notice that lighting stays intact.

6.7.4 Shadow Synthesis

Recall from Section 6.7 that visibility ratios are computed as:

$$v_r(x) = \frac{c_o(x)}{c_r(x)}$$

where $c_o(x)$ is the primary rendering and $c_r(x)$ is the secondary rendering, i.e. rendering obtained with the shadow removal operation described in the previous section.

Thus we can use visibility ratios to remove a shadow from the original (primary) image:

$$c_r(x) = \frac{c_o(x)}{v_r(x)}$$

Or in other words, we can use visibility ratios to synthesize a shadow in the secondary rendering:

$$c_o(x) = c_r(x)v_r(x)$$

Imagine now that the visibility ratios undergo a transformation (edit). This transformation yields new *target* visibility ratios $w_r(x)$ which we use to synthesize a new shadow:

$$c_m(x) = c_r(x)w_r(x) = \frac{c_o(x)}{v_r(x)}w_r(x) \qquad (6.2)$$

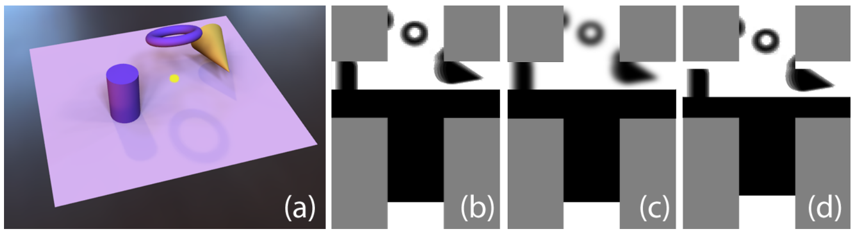


Figure 6.6: Visibility edits. (a) Sampling locations in yellow. (b) Original visibility. (c) Blur. (d) Translation.

where $c_m(x)$ is a new rendering with a modified shadow. Equation 6.2 is called the *shadow synthesis equation* and is the main equation we use for editing based on visibility ratios. It explains our synthesis algorithm in three steps: (1) removal of the old shadow and computation of visibility ratios (2) editing of visibility ratios (3) rendering of a new image.

6.7.5 VR editing

Editing of visibility ratios requires editing of a 2D array of 3-component values and is therefore considerably more efficient than editing of the entire visibility function. However, since VRs effectively discard the directional aspect of visibility, their expressive power is constrained. Therefore, the best use of VR edits is in combination with visibility editing operations from Section 6.6.

VR edits not only allow for very accurate approximation of existing edits, but also add new ones (see Figure 6.11). For example, blurring the visibility ratios results in blurring of the shadow (same as Shadow Blur). Similarly, Shadow Gradient can be implemented as a recoloring operation applied to visibility ratios. It can be easily seen that in both

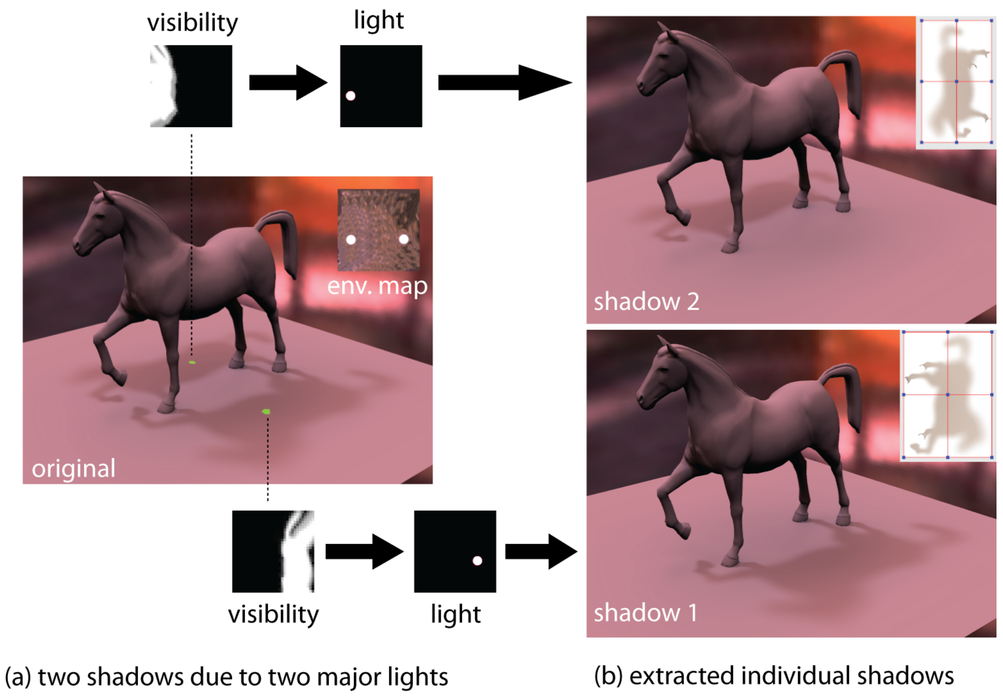

visibility
light
env. map
original
shadow 2
shadow 1
visibility
light
(a) two shadows due to two major lights
(b) extracted individual shadows

Figure 6.7: Two shadows are caused by two major lights in the environment map. The user paints over both shadows, effectively selecting both lights. The removal algorithm then extracts one shadow at a time.

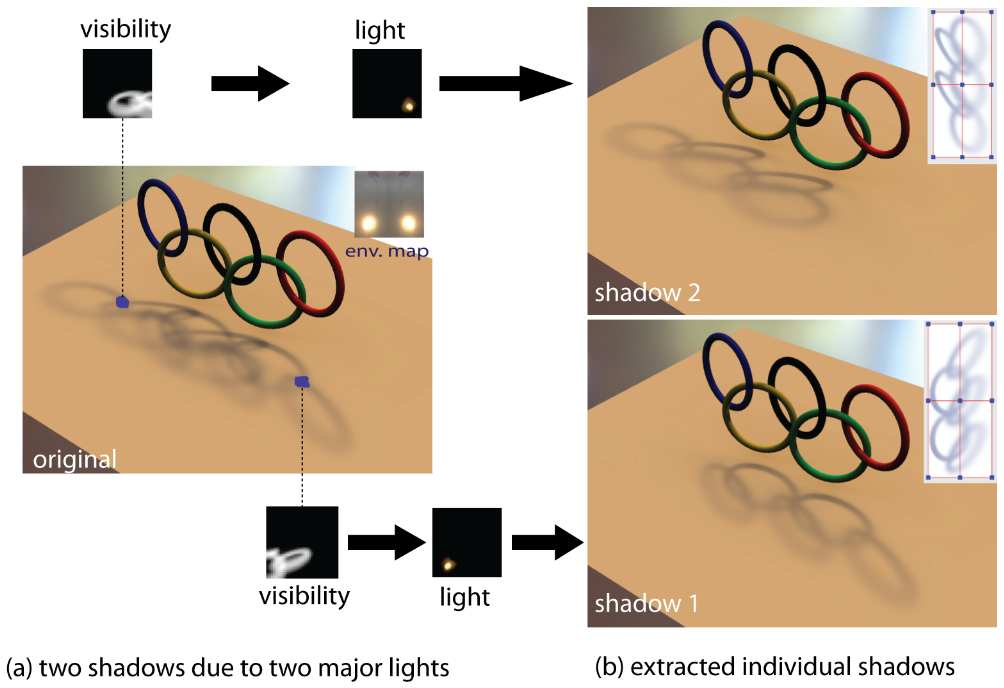

visibility
light
env. map
original
visibility
light
shadow 2
shadow 1
(a) two shadows due to two major lights
(b) extracted individual shadows

Figure 6.8: Another example of shadow extraction. Manual shadow separation in an image processing application would be very difficult in this case. Our algorithm handles this situation with ease.

cases, editing of visibility ratios is more efficient and allows the user to design both edits interactively.

Shadow Translation/Rotation can be approximated by applying a 2D image transformation to visibility ratios. For example, 2D projective transformation is very well suited for per-pixel sampling if the shadow is cast on a planar object. On the other hand, UV shifting is suitable for non-planar surfaces and per-texel sampling.

As an example, Figure 6.11 shows a lattice controller for manipulating shadow shape. The controller splits the shadow into quadrilateral regions and uses mean-value coordinates [HT04] to resample the shadow as the user manipulates the control points. Target VRs are computed using a filtered lookup:

$$w_r(x) = \text{filter}(v_r(\text{mvc}(x)))$$

where $\text{mvc}(x)$ returns mean-value coordinates for location x.

6.8 Rendering and Animation

Edits based on modifications of the visibility function do not require any changes to the rendering method described at the end of Section 6.4. For each sampling location, we simply compute a dot product of the transport row and visibility row. The remainder of this section therefore describes rendering changes for shadow editing with visibility ratios.

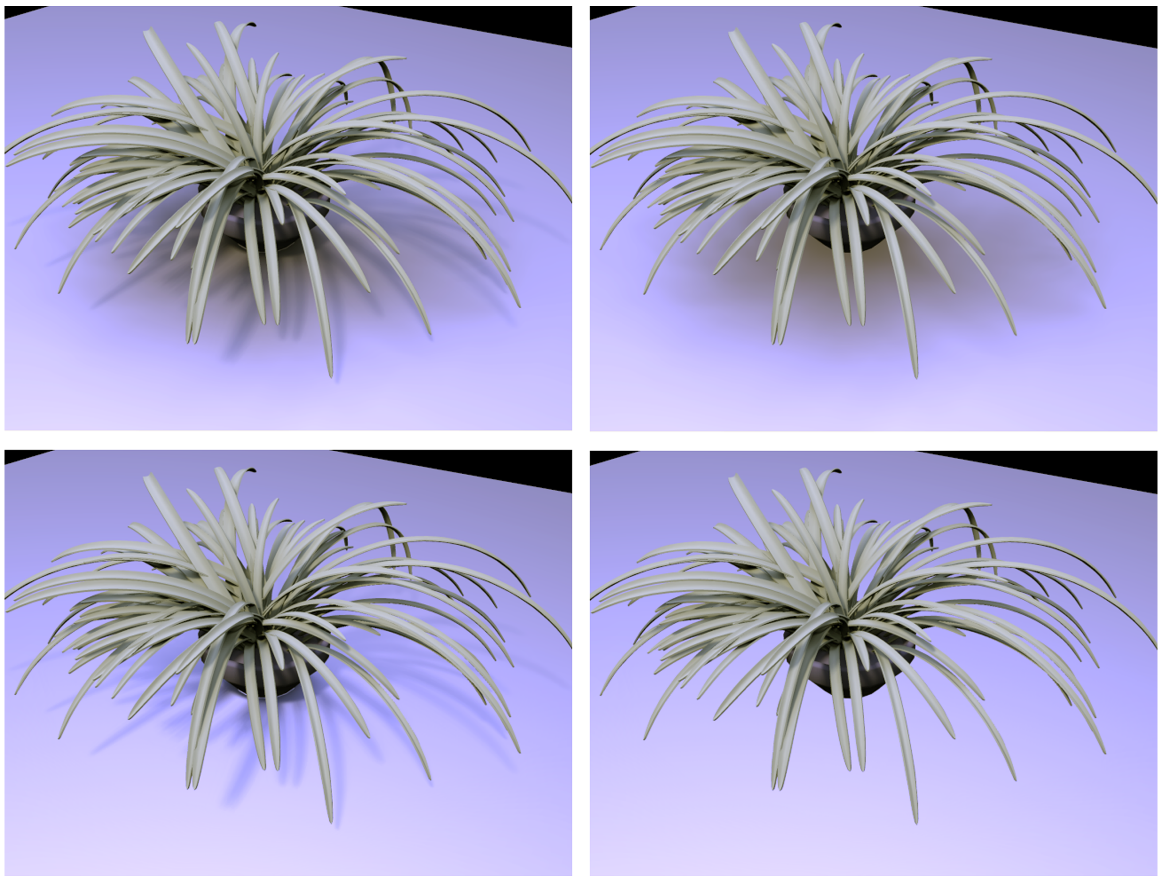


Figure 6.9: Shadow removal. (Top left) Original with main and residual shadows. (Top right) Main shadow removed. (Bottom left) Residual shadow removed and main shadow edited. (Bottom right) Main and residual shadows removed.

We implemented VR edits as GPU shaders, which take as input a 2D texture of visiblity ratios and perform edits according to user parameters (such as HSV parameters for a recoloring edit). The output is a modified texture containing new, target visibility ratios. Target visibility ratios are then used to synthesize a new shadow, as per Equation 6.2.

For per-texel sampling, we evaluate Equation 6.2 in texel space and the simply render the final image by projecting scene geometry on the screen. Since per-texel sampling uses only diffuse BRDFs, the user is allowed to freely navigate around the scene. Furthermore, per-texel sampling allows us to remove aliasing from the image by utilizing multisampling directly on the GPU.

With per-pixel sampling, each sampling location corresponds to one screen pixel and the camera is locked. While this restriction simplifies some parts of the rendering pipeline, it effectively prevents us from using GPU multisampling to remove edge aliasing. Therefore, we remove aliasing by applying a bilateral filter as follows:

1. Precompute visibility and transport matrices for desired image resolution $W \times H$.
2. Upsample $c(x)$ to resolution $2W \times 2H$ using a bilateral filter, where $c(x)$ refers to an image obtained by multiplication of the transport and visibility matrices.
3. Downsample the result back to resolution $W \times H$ using bilinear filtering.

For implementation details of bilateral upsampling, we refer the reader to [SGN07].

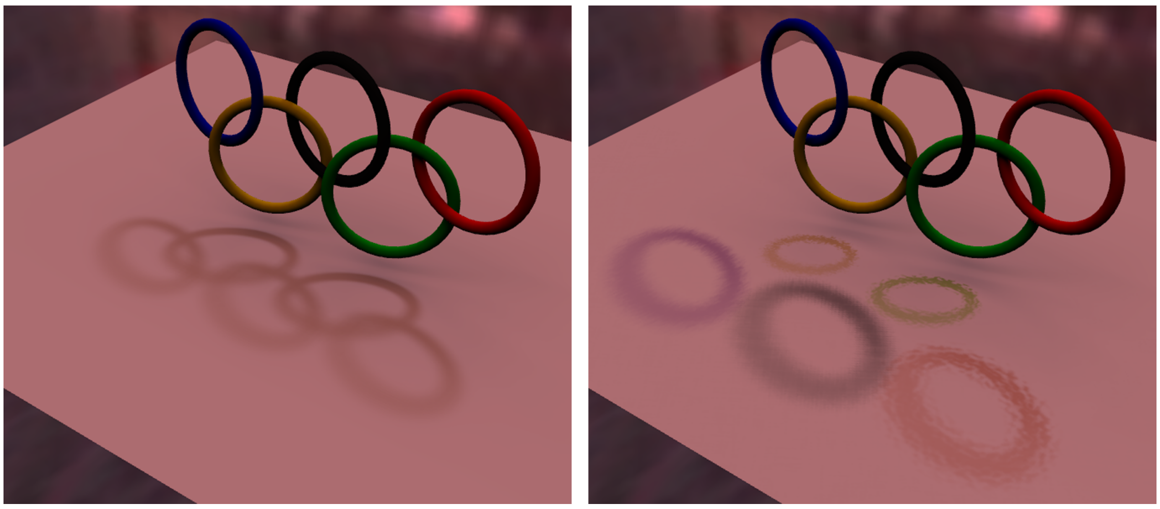


Figure 6.10: (Left) Overlapping shadows are caused by multiple objects and one major light. We precompute multiple visibility matrices, one for each visibility group, i.e. each olympic ring in this case. (Right) The algorithm splits the shadows and allows the user to edit them individually.

6.8.1 Animation

We support animation for both classes of edits by freezing edit parameters for a set of keyframes and interpolating them for all other frames. Each of our edits can be described by a set of numerical parameters — Shadow Blur is parameterized by the kernel size, Shadow Rotation is parameterized by the angle of rotation, etc.

For example, let $P(i)$ be the translation vector for the Shadow Translation edit for keyframe i and $P(k)$ be the translation vector for the same edit for keyframe k. For all other frames $j, i < j < k$, we compute the translation vector for Shadow Translation $P(j)$ as follows:

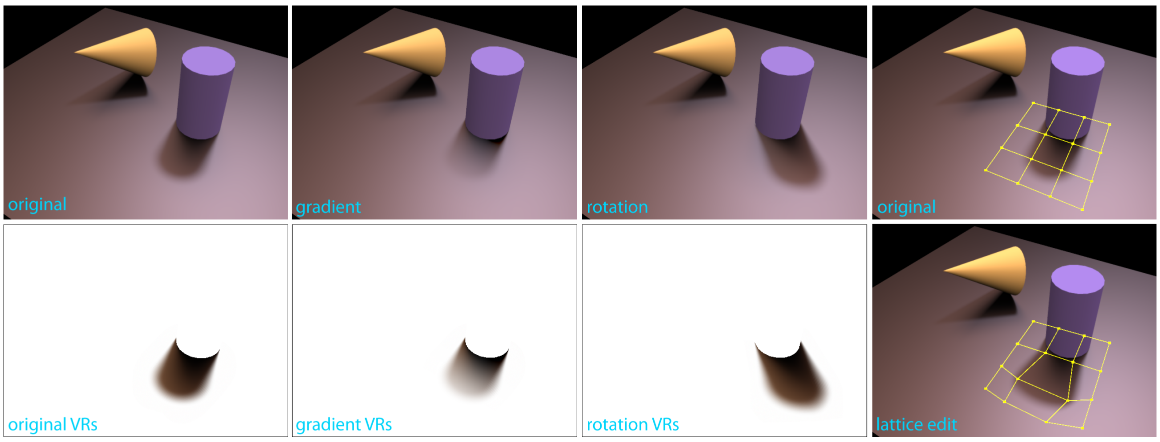


Figure 6.11: Examples of VR edits and the lattice controller.

$$P(j) = P(i)\frac{j-i}{k-i} + P(k)\frac{k-j}{k-i}$$

Computed translation vector $P(j)$ is then applied to either visibility function or VRs corresponding to frame j. Parameters for all other edits are computed in the same fashion. This interpolation scheme corresponds to linear interpolation, but can be easily generalized to curve-driven interpolation as seen in popular modeling applications.

6.9 Results and Performance

We present results for three main scenes with varying complexity and different BRDFs (see Figures 6.1, 6.3 and 6.4). All scenes were rendered using either one or two cube map faces of the environment map, sampling each face with 128x128 rays. We used commodity hardware, an Intel Core Duo E6850 processor, with 4GB of RAM and GeForce8800 Ultra. We used 200

Scene	**Samples**	**Triangles**	**CPU V**	**CPU T**	**GPU V**	**GPU T**
Rings	per-texel 589,824	8,192	140 min.	45 min.	27 min.	27 min.
Horse	per-pixel 307,200	16,416	64 min.	23 min.	16 min.	16 min.
Pirate	per-pixel 307,200	214,726	195 min.	23 min.	16 min.	16 min.
Car	per-pixel 307,200	184,560	146 min.	23 min.	16 min.	16 min.

Scene	**Raw Data**	**Compressed**
Rings	5.34 GB	1.1 GB
Horse	2.74 GB	172 MB
Pirate	2.68 GB	220 MB
Car	2.36 GB	617 MB

Table 6.1: Timings and data sizes for presented scenes. The key to making the precomputation of the visibility matrix efficient is in using the GPU to rasterize visibility (instead of ray tracing) and to perform the wavelet encoding. The transport matrix can be also precomputed on the GPU — by evaluating the BRDF and the cosine term in a shader.

wavelet coefficients in each case. The timings and required storage space are summarized in Figure 6.1.

In order to increase the performance of the precomputation, we implemented it on both the CPU and the GPU and compared. The CPU precomputation uses ray tracing to compute the visibility matrix (column CPU V in Table 6.1) and CPU-side evaluation of the BRDF and the cosine term for each direction in the cube map (column CPU T). Wavelet encoding is also performed on the CPU. Scene complexity and the number of samples are the most important factors affecting the performance of visibility precomputation. The precomputation time for the transport matrix depends only on the number of samples and BRDF complexity.

To precompute the visibility matrix on the GPU, we rasterize scene geometry into a cube map. To precompute the transport matrix, we render quads aligned with cube map faces and evaluate the BRDF and the cosine term in a pixel shader. We also perform the wavelet encoding on the GPU. The timings for GPU precomputation are summarized in columns GPU V and GPU T in Figure 6.1. While the precomputation on the GPU is significantly faster than on the CPU, equal timings for both matrices (GPU T and GPU V) imply that the GPU-CPU data transfer is the main bottleneck of our current implementation. Overall, GPU precomputation of the visibility matrix delivers a speedup of factor 4-5 for simple scenes and 9-12 for complex scenes. GPU precomputation of the transport matrix delivers a speedup of 1.5.

In all presented scenes, removal of all shadows took between 1 and 3 minutes and editing was interactive. We would like to point out that further reduction of the precomputation time can be achieved by parallelizing. Since precomputations for each texel/pixel are independent, we can use an arbitrary number of available GPUs/CPUs to speed up the process. We performed our tests on commodity hardware (manufactured in 2006) with only 2 cores. Current multi-core high-end hardware will increase the performance significantly. To conserve storage space, the user might opt to apply lossless compression to both matrices. For our data, the RAR algorithm performed the best (column Compressed in Figure 6.1) and achieved average compression ratio of 1:9.

6.10 Discussion

Even though we believe that our work presents a robust and simple approach to all-frequency shadow design, several important aspects need to be considered before deploying it in client applications.

The main limitation of our work is the upper frequency bound of shadows. Our examples illustrate shadows due to all-frequency lighting, for which our algorithms behave very well. However, in an extreme example of an environment map with a single non-zero texel, under-sampling problems arise. We believe that such situations are better handled with methods tailored specifically for high-frequency shadows, as opposed to an all-frequency method such as ours.

It is important to realize that thresholding used in the light localization algorithm might introduce artifacts if the threshold value is not set correctly. We found that the threshold values mainly depend on the environment map, but are in practice very easy to set. Other factors affecting the accuracy of the light localization algorithm include the resolution of the environment map and the number of wavelet coefficients used to compress the visibility matrix.

Another important aspect is the selection of a rendering technique. Our focus was on high-quality rendering and our implementation therefore relies on a precomputed approach. However, it can be easily seen that a real-time method such as [CK07] can be used as well. The only requirement is the ability to access explicit visibility information. Finally, our techniques are not limited to environment maps. Other types of all-frequency lights, such as area lights or near-field lights [SR09] can be used as well.

6.11 Conclusions

In the broader context, our approach to design of all-frequency shadows has demonstrated the importance of non-physical editing for an aspect present in almost all computer-generated images. Since we keep the lighting static, shadow deformation using the lattice controller, recoloring, blurring, etc. are all non-physical operations. We implement them by modifying visibility, which is an operation that has no physical equivalent (except for approximation by blocker objects as used in photography).

We have also achieved two other important goals — interactive editing and fast precomputation. Our novel representation, based on visibility ratios, has allowed us to edit all-frequency shadows in real-time, which was no possible before. By offloading the precomputation to the GPU, we have significantly reduced the precomputation time. This is very important in the context of the editing pipeline, because it allows artists to move on to the next stage very quickly (in general, it is desirable that the precomputation time be shorter than the time required to render the image in an offline renderer [RKS07]).

CHAPTER 7

CONCLUSION

This thesis presented a set of techniques for editing of two important aspects commonly present in computer-generated images — global illumination and all-frequency shadows. Our goal was to apply an approach that is not necessarily physically-correct and show that physical-correctness is not required for plausible and impressive designs.

We first tackled the problem of lighting design with global illumination. Our solution, which relies on editing of the light transport operator, enables the artists to interactively edit features such as color bleeding or light falloff. Our representation of performed edits is based on scale/offset vectors and matrices, supports new edits and is renderer-independent.

By providing a real-time editing interface built on top of the representation, we have demonstrated support for interactivity and immediate feedback — two very important qualities sought by artists. We have also created a variety of editing operations, many of them non-physical, which can find great use in practice. As an example, we created an operation that allows users to turn any object into a light source without negatively impacting the rendering performance (the additive mode of the Light Source Brush). While being completely non-physical, its applicability in lighting design applications is very high.

We followed up on this work by applying non-physical editing to all-frequency shadows. All-frequency shadows are inherently very difficult to edit and to the best of our knowledge, ours is the first work to address them in a complete fashion. We pursued an approach based on editing of visibility information. Editing of visibility is itself non-physical, but it gives us great great freedom when working with shadows. With demonstrated edits such as shadow translation or recoloring. To provide interactive feedback, we introduced a novel representation based on computation of visibility ratios, which dramatically decreases the computational requirements of editing.

Both of our solutions fit easily into existing production pipelines. In typical productions, lighting design is performed after the scene, animations and camera paths have been fixed, justifying the assumption of a fixed camera in our lighting solution. Due to the inherent coupling between lighting and shadows, shadow and lighting design are often performed simultaneously. Our work proposes to decouple these and perform shadow design after lighting has been fixed. This allows us to make the assumption of fixed lighting, precompute the transport and visibility matrices and perform shadow editing in real-time.

The goal of this thesis was to demonstrate how the combination of non-physical editing, real-time feedback and adherence to existing workflows can boost the creativity, expressive power and efficiency of the design process in computer-generated images. We have shown that non-physical editing can greatly expand the design space in both global illumination and all-frequency shadows and demonstrated edits, which have, until now, been extremely hard to achieve with existing techniques.

We provided interactive feedback for our editing operations, by offloading parts of the computations to the GPU and by providing a balance between the quality and the performance of our rendering engines. We have shown that by creating suitable representations for the edits designed by artists, we do not compromise the quality of the final output, but rather enable real-time preview for even very complex aspects, such as all-frequency shadows.

We believe that the greatest value of our work is in showing that while we already have many sophisticated and robust rendering algorithms, the quality of final images can be greatly improved by devoting more research to the design part. Artists will always benefit from new, less-restricting, approaches to design, from representations that are specifically tailored to editing (as opposed to rendering algorithms) and from real-time feedback.

To conclude this chapter, the following are the publications of our work:

- Juraj Obert, Fabio Pellacini and Sumanta Pattanaik
  Visibility Editing For All-Frequency Shadow Design
  Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering), Volume 29, Number 4, 2010

- Juraj Obert, Jaroslav Křivánek, Fabio Pellacini, Daniel Sýkora and Sumanta Pattanaik
  iCheat: A Representation for Artistic Control of Indirect Cinematic Lighting
  Computer Graphics Forum (Proceedings of Eurographics Symposium on Rendering), Volume 27, Number 4, 2008

- Juraj Obert, Jaroslav Křivánek, Daniel Sýkora and Sumanta Pattanaik
  Interactive Light Transport Editing for Flexible Global Illumination
  ACM SIGGRAPH 2007, Sketches and Applications, 2007

CHAPTER 8

FUTURE WORK

Our work has only scratched the surface of possibilities available for non-physical editing in the design process. Artists always look for new tools, interfaces and paradigms that will make their work more efficient and enjoyable.

Our global illumination pipeline can be extended to support other aspects of appearance design such as volumetric effects, caustics, refractions, etc. These effects are very complex and their editing, which is certainly non-trivial, will require sophisticated techniques that provide the three important components we consider to be necessary for applicability in real productions: real-time feedback, wide range of editing operations (including non-physical ones) and compatibility with existing pipelines.

We envision a next-generation production pipeline consisting of multiple stages, each dedicated to editing of certain aspects of the final image. We have already suggested such a separation for shadow design. Instead of designing shadows as a part of lighting, we decouple them and edit them independently. A similar approach can be pursued for highlights, secondary shadows, atmospheric effects, etc.

Editing in each stage will focus on a single aspect (or a small number of closely related aspects) and provide appropriate tools for the artists.

We are convinced that precomputation-based methods can be of great use in such a pipeline, as each stage can make the assumption that some part of the output from the previous stage will not change anymore (as its design was finalized in the previous stage). Currently, such assumptions are only made in the lighting stage, where the camera, as the output of the previous modeling/animation stage, is assumed to be fixed. Once lighting design has been finalized, the subsequent stages can assume it is fixed and use it in the precomputation (we indeed demonstrated this approach for shadow design).

GPUs and parallelization will continue to play very important roles in the precomputation algorithms. We have demonstrated how precomputation of the transport and visibility matrices can achieve significant speedups if the GPU is utilized. We believe that more research should be directed to the design and structure of precomputation algorithms as many of them currently do not map well to the graphics hardware and parallel architectures. GPUs have allowed us to massively parallelize rendering and by parallelizing the precomputation, we can increase the throughput and performance of our pipelines even further.

Certainly, user interfaces for editing are very important for artists. In our work, we have focused on providing algorithmic solutions, while not fixating to a particular user interface. Artists typically use multiple authoring packages and the choice of appropriate interfaces should be explored in the future. Current research has addressed editing interfaces only very briefly for direct lighting and materials, while more complex aspects have not been a focus of formal studies yet. Since editing of such complex aspects is still a relatively new topic, formal evaluations will only be possible after conducting more research.

We believe that effort should also be invested into investigating whether or not non-physical editing affects user perception of the information in the image. While our eyes are used to seeing objects lit and shadowed in certain ways in the real-world, we wonder whether users can, even subconsciously, notice subtle non-physical edits in rendered images and whether or not this changes how the images are perceived and interpreted. Significant results from such a study can provide more insights into the importance of physically-correct rendering in applications such as movies, video games, visual walkthroughs, etc.

To conclude, we believe that the combination of non-physical editing, real-time rendering feedback and adherence to existing workflows allows artists to become more efficient and enjoy their work a lot more. We believe that further investigation into non-physical editing of other visual aspects present in the images, such as highlights or reflections, will greatly benefit those who are finally responsible for creating beautiful images in computer graphics — the artists.

While we already know how to render images given input data, the process of creating the data fed to the renderers still requires major effort (since artists have to first create 3D models, then animate them, fix camera paths, paint textures, etc.) We believe that this effort can be minimized by providing artists with suitable tools, pipelines, algorithms and interfaces that will facilitate the creation process. By doing so, we not only make the design process very enjoyable, but mainly dramatically increase the quality of the final product.

LIST OF REFERENCES

[ADM08] Thomas Annen, Zhao Dong, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. "Real-time, all-frequency shadows in dynamic scenes." In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pp. 1–8, New York, NY, USA, 2008. ACM.

[AHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition.* A. K. Peters, Ltd., Natick, MA, USA, 2008.

[AMB07] Thomas Annen, Tom Mertens, Philippe Bekaert, Hans-Peter Seidel, and Jan Kautz. "Convolution Shadow Maps." In *Rendering Techniques 2007: Eurographics Symposium on Rendering*, pp. 51–60, Grenoble, France, 2007. Eurographics.

[AUW07] Oskar Akerlund, Mattias Unger, and Rui Wang. "Precomputed Visibility Cuts for Interactive Relighting with Dynamic BRDFs." *Computer Graphics and Applications, Pacific Conference on*, **0**:161–170, 2007.

[Bar97] Ronen Barzel. "Lighting Controls for Computer Cinematography." *Journal of Graphics Tools*, **2**(1):1–20, 1997.

[BED08] Aner Ben-Artzi, Kevin Egan, Frédo Durand, and Ravi Ramamoorthi. "A precomputed polynomial representation for interactive BRDF editing with global illumination." *ACM Trans. Graph.*, **27**(2):1–13, 2008.

[BOR06] Aner Ben-Artzi, Ryan Overbeck, and Ravi Ramamoorthi. "Real-time BRDF editing in complex lighting." In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pp. 945–954, New York, NY, USA, 2006. ACM.

[CFL06] Per Christensen, Julian Fong, David Laur, and Dana Batal. "Ray Tracing for the Movie Cars." In *Proc. of IEEE Symposium on Interactive Ray Tracing*, pp. 1–6, 2006.

[CGC03] Yung-Yu Chuang, Dan B Goldman, Brian Curless, David H. Salesin, and Richard Szeliski. "Shadow matting and compositing." In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pp. 494–500, New York, NY, USA, 2003. ACM.

[Chr03] Per Christensen. "Global Illumination and All That." In *ACM SIGGRAPH Course 9 (RenderMan, Theory and Practice)*, 2003.

[CK07] Mark Colbert and Jaroslav Křivánek. "Real-time shading with filtered importance sampling." In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, p. 71, New York, NY, USA, 2007. ACM.

[CPK06] Mark Colbert, Sumanta Pattanaik, and Jaroslav Krivanek. "BRDF-Shop: Creating Physically Correct Bidirectional Reflectance Distribution Functions." *IEEE Comput. Graph. Appl.*, **26**(1):30–36, 2006.

[CWA08] Ewen Cheslack-Postava, Rui Wang, Oskar Akerlund, and Fabio Pellacini. "Fast, realistic lighting and material design using nonlinear cut approximation." *ACM Trans. Graph.*, **27**(5):1–10, 2008.

[DBB02] Philip Dutre, Kavita Bala, and Philippe Bekaert. *Advanced Global Illumination.* A. K. Peters, Ltd., Natick, MA, USA, 2002.

[DCF07] Christopher DeCoro, Forrester Cole, Adam Finkelstein, and Szymon Rusinkiewicz. "Stylized shadows." In *NPAR '07: Proceedings of the 5th international symposium on Non-photorealistic animation and rendering*, pp. 77–83, New York, NY, USA, 2007. ACM.

[DKN95] Yoshinori Dobashi, Kazufumi Kaneda, Hideki Nakatani, and Hideo Yamashita. "Quick Rendering Method using Basis Functions for Interactive Lighting Design." *Comp. Graph. Forum*, **14**(3):229–240, 1995.

[DSG91] Julie O'B. Dorsey, François X. Sillion, and Donald P. Greenberg. "Design and Simulation of Opera Lighting and Projection Effects." In *SIGGRAPH'91 Proceedings*, pp. 41–50, 1991.

[FHD02] Graham D. Finlayson, Steven D. Hordley, Mark S. Drew, and England Nr Tj. "Removing Shadows from Images." In *ECCV 2002: European Conference on Computer Vision*, pp. 823–836, 2002.

[GH00] Reid Gershbein and Pat Hanrahan. "A fast relighting engine for interactive cinematic lighting design." In *SIGGRAPH'00 Proceedings*, pp. 353–358, 2000.

[GKP04] Pascal Gautron, Jaroslav Křivánek, Sumanta N. Pattanaik, and Kadi Bouatouch. "A Novel Hemispherical Basis for Accurate and Efficient Rendering." In *Rendering Techniques 2004, Eurographics Symposium on Rendering*, pp. 321–330, June 2004.

[HPB06] Miloš Hašan, Fabio Pellacini, and Kavita Bala. "Direct-to-indirect transfer for cinematic relighting." *ACM Trans. Graph.*, **25**(3):1089–1097, 2006.

[HT04] Kai Hormann and Marco Tarini. "A quadrilateral rendering primitive." In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pp. 7–14, New York, NY, USA, 2004. ACM.

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

[Kaj86] James T. Kajiya. "The Rendering Equation." In *SIGGRAPH'86 Proceedings*, pp. 143–150, 1986.

[KAJ05] Anders Wang Kristensen, Tomas Akenine-Möller, and Henrik Wann Jensen. "Precomputed local radiance transfer for real-time lighting design." *ACM Trans. Graph.*, **24**(3):1208–1215, 2005.

[KGP05] Jaroslav Křivánek, Pascal Gautron, Sumanta Pattanaik, and Kadi Bouatouch. "Radiance Caching for Efficient Global Illumination Computation." *IEEE Transactions on Visualization and Computer Graphics*, **11**(5), September/October 2005. Also available as Technical Report #1623, IRISA, http://graphics.cs.ucf.edu/RCache/index.php.

[KP09] William B. Kerr and Fabio Pellacini. "Toward evaluating lighting design interface paradigms for novice users." In *SIGGRAPH '09: ACM SIGGRAPH 2009 papers*, pp. 1–9, New York, NY, USA, 2009. ACM.

[KPC93] John K. Kawai, James S. Painter, and Michael F. Cohen. "Radioptimization: Goal based rendering." In *SIGGRAPH'93 Proceedings*, pp. 147–154, 1993.

[KTH06] Janne Kontkanen, Emmanuel Turquin, Nicolas Holzschuch, and François X. Sillion. "Wavelet Radiance Transport for Interactive Indirect Lighting." In *Proc. of Eurographics Symposium on Rendering*, pp. 161–171, 2006.

[NRH03] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. "All-frequency shadows using non-linear wavelet lighting approximation." *ACM Trans. Graph.*, **22**(3):376–381, 2003.

[NRH04] Ren Ng, Ravi Ramamoorthi, and Pat Hanrahan. "Triple product wavelet integrals for all-frequency relighting." *ACM Trans. Graph.*, **23**(3):477–487, 2004.

[NSD94] Jeffry S. Nimeroff, Eero Simoncelli, and Julie Dorsey. "Efficient Re-rendering of Naturally Illuminated Environments." In *Proc. of Eurographics Workshop on Rendering*, pp. 359–373, 1994.

[OMS07] Makoto Okabe, Yasuyuki Matsushita, Li Shen, and Takeo Igarashi. "Illumination brush: Interactive design of all-frequency lighting." In *Proc. Pacific Conference on Computer Graphics and Applications (2007), IEEE Computer Society*, pp. 171–180, 2007.

[PBM07] Fabio Pellacini, Frank Battaglia, R. Keith Morley, and Adam Finkelstein. "Lighting with paint." *ACM Trans. Graph.*, **26**(2), 2007.

[PF92] Pierre Poulin and Alain Fournier. "Lights from Highlights and Shadows." In *Proc. of Symposium on Interactive 3D Graphics*, pp. 31–38, 1992.

[Pix] Pixar. "The Renderman Interface.".

[PRJ97] Pierre Poulin, Karim Ratib, and Marco Jacques. "Sketching Shadows and Highlights to Position Lights." In *Proc. of Computer Graphics International*, pp. 56–63, 1997.

[PTG02] Fabio Pellacini, Parag Tole, and Donald P. Greenberg. "A User Interface for Interactive Cinematic Shadow Design." *ACM Trans. Graph.*, **21**(3):563–566, 2002.

[PVL05] Fabio Pellacini, Kiril Vidimče, Aaron Lefohn, Alex Mohr, Mark Leone, and John Warren. "Lpics: A hybrid hardware-accelerated relighting engine for computer cinematography." *ACM Trans. Graph.*, **24**(3):464–470, 2005.

[RKS07] Jonathan Ragan-Kelley, Charlie Kilpatrick, Brian W. Smith, Doug Epps, Paul Green, Christophe Hery, and Frédo Durand. "The lightspeed automatic interactive lighting preview system." *ACM Trans. Graph.*, **26**(3), 2007.

[ROT09] Tobias Ritschel, Makoto Okabe, Thorsten Thormählen, and Hans-Peter Seidel. "Interactive reflection editing." In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pp. 1–7, New York, NY, USA, 2009. ACM.

[RWS06] Zhong Ren, Rui Wang, John Snyder, Kun Zhou, Xinguo Liu, Bo Sun, Peter-Pike Sloan, Hujun Bao, Qunsheng Peng, and Baining Guo. "Real-time soft shadows in dynamic scenes using spherical harmonic exponentiation." *ACM Trans. Graph.*, **25**(3):977–986, 2006.

[SDS93] Chris Schoeneman, Julie Dorsey, Brian Smits, James Arvo, and Donald Greenberg. "Painting with Light." In *SIGGRAPH'93 Proceedings*, pp. 143–146, 1993.

[SDS94] Eric J. Stollnitz, Tony D. Derose, and David H. Salesin. "Wavelets for Computer Graphics: A Primer.", 1994.

[SGN07] Peter-Pike Sloan, Naga K. Govindaraju, Derek Nowrouzezahrai, and John Snyder. "Image-Based Proxy Accumulation for Real-Time Soft Global Illumination." In *PG '07: Proceedings of the 15th Pacific Conference on Computer Graphics and Applications*, pp. 97–105, Washington, DC, USA, 2007. IEEE Computer Society.

[SKS02] Peter-Pike J. Sloan, Jan Kautz, and John Snyder. "Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments." *ACM Trans. Graph.*, **21**(3):527–536, 2002.

[SM06] Weifeng Sun and Amar Mukherjee. "Generalized wavelet product integral for rendering dynamic glossy objects." In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pp. 955–966, New York, NY, USA, 2006. ACM.

[SR09] Bo Sun and Ravi Ramamoorthi. "Affine double- and triple-product wavelet integrals for rendering." *ACM Trans. Graph.*, **28**(2):1–17, 2009.

[STP09] Ying Song, Xin Tong, Fabio Pellacini, and Pieter Peers. "SubEdit: a representation for editing measured heterogeneous subsurface scattering." In *SIGGRAPH '09: ACM SIGGRAPH 2009 papers*, pp. 1–10, New York, NY, USA, 2009. ACM.

[SZC07] Xin Sun, Kun Zhou, Yanyun Chen, Stephen Lin, Jiaoying Shi, and Baining Guo. "Interactive relighting with dynamic BRDFs." In *SIGGRAPH '07: ACM SIGGRAPH 2007 papers*, p. 27, New York, NY, USA, 2007. ACM.

[SZS08] Xin Sun, Kun Zhou, Eric Stollnitz, Jiaoying Shi, and Baining Guo. "Interactive relighting of dynamic refractive objects." In *SIGGRAPH '08: ACM SIGGRAPH 2008 papers*, pp. 1–9, New York, NY, USA, 2008. ACM.

[TL04] Eric Tabellion and Arnauld Lamorlette. "An approximate global illumination system for computer generated films." In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pp. 469–476, New York, NY, USA, 2004. ACM.

[TSH97] Patrick C. Teo, Eero P. Simoncelli, and David J. Heeger. "Efficient Linear Re-Rendering for Interactive Lighting Design." Technical Report STAN-CS-TN-97-60, Stanford University, 1997.

[WAB06] Bruce Walter, Adam Arbree, Kavita Bala, and Donald P. Greenberg. "Multidimentsional Lightcuts." *ACM Trans. Graph.*, **25**(3), 2006.

[WFA05] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. "Lightcuts: a scalable approach to illumination." *ACM Trans. Graph.*, **24**(3):1098–1107, 2005.

[WRC88] Gregory J. Ward, Francis M. Rubinstein, and Robert D. Clear. "A ray tracing solution for diffuse interreflection." In *SIGGRAPH'88 Proceedings*, pp. 85–92, 1988.

[WRG09] Jiaping Wang, Peiran Ren, Minmin Gong, John Snyder, and Baining Guo. "All-frequency rendering of dynamic, spatially-varying reflectance." In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pp. 1–10, New York, NY, USA, 2009. ACM.

[WTL04] Rui Wang, John Tran, and David Luebke. "All-Frequency Relighting of Non-Diffuse Objects using Separable BRDF Approximation." In *Proc. of Eurographics Symposium on Rendering*, pp. 345–354, 2004.

[WTL06] Rui Wang, John Tran, and David Luebke. "All-frequency relighting of glossy objects." *ACM Trans. Graph.*, **25**(2):293–318, 2006.

[ZHL05] Kun Zhou, Yaohua Hu, Stephen Lin, Baining Guo, and Heung-Yeung Shum. "Precomputed shadow fields for dynamic scenes." In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pp. 1196–1201, New York, NY, USA, 2005. ACM.