High-quality Shading and Lighting for Hardware-accelerated Rendering

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Wolfgang Heidrich

Erlangen – 1999

Realistische Oberflächen- und Beleuchtungseffekte für die hardware-beschleunigte Bildsynthese

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Wolfgang Heidrich

Erlangen – 1999

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	8.2.1999
Tag der Promotion:	7.4.1999
Dekan:	Prof. Dr. G. Herold
Berichterstatter:	Prof. Dr. H.-P. Seidel
	Prof. Dr. W. Straßer

Revision 1.1

Abstract

With fast 3D graphics becoming more and more available even on low end platforms, the focus in developing new graphics hardware is beginning to shift towards higher quality rendering and additional functionality instead of simply higher performance implementations of the traditional graphics pipeline. On this search for improved quality it is important to identify a powerful set of orthogonal features to be implemented in hardware, which can then be flexibly combined to form new algorithms.

This dissertation introduces a set of new algorithms for high quality shading and lighting using computer graphics hardware. It is mostly concerned with algorithms for generating various local shading and lighting effects and for visualizing global illumination solutions. In particular, we discuss algorithms for shadows, bump mapping, alternative material models, mirror reflections and glossy reflections off curved surfaces, as well as more realistic lens systems and a complex model for light sources.

In the course of developing these algorithms, we identify building blocks that are important for future generations of graphics hardware. Some of these are established features of graphics hardware used in a new, unexpected way, some are experimental features not yet widely used, and some are completely new features that we propose. When introducing new functionality, we make sure that it is orthogonal to existing stages in the graphics pipeline.

Thus, the contribution of this dissertation is twofold: on the one hand, it introduces a set of new algorithms for realistic image synthesis using computer graphics hardware. These methods are capable of providing image qualities similar to those of simple ray-tracers, but provide interactive frame rates on contemporary graphics systems. On the other hand, this thesis identifies and introduces features and building blocks that are important for realistic shading and lighting, and thus contributes to the development of better graphics hardware in the future.

Acknowledgments

The work presented in this thesis would not have been possible without the encouragement and help offered by many people. In particular, I would like to thank my advisor, Professor Hans-Peter Seidel and the external reader Professor Wolfgang Straßer for their interest in the work and for their valuable comments. It was Professor Seidel who first awakened my interest in the field of computer graphics, and encouraged and supported my research in this area.

Furthermore, I owe thanks to all my colleagues of the graphics group in Erlangen. It was a lot of fun to work in this interesting and truly inspiring environment. I would especially like to thank the Erlangen rendering group consisting of Hartmut Schirmacher, Marc Stamminger, and in particular Philipp Slusallek for the many fruitful discussions we had. Furthermore, Rüdiger Westermann and Peter-Pike Sloan contributed ideas to the shadow map algorithm, and Michael F. Cohen contributed to the light filed based refraction algorithm.

Early versions of this thesis where proof-read by Katja Daubert, Michael McCool, Hartmut Schirmacher, and Philipp Slusallek. Finally, I owe thanks to several graduate and undergraduate students that helped with the implementation of some of the concepts and supporting tools. In alphabetical order, these are Stefan Brabec, Alexander Gering, Jan Kautz, Hendrik Kück, Hendrik Lensch, Martin Rubick, Detlev Schiron, and Christian Vogelgsang.

Parts of this work were funded by the German Research Council through the collaborative research centers #603 (Model-based Analysis and Synthesis of Complex Scenes and Sensor Data) and #182 (Multiprocessor- and Networkconfigurations).

Contents

List of Figures

List of Equations

Chapter 1

Introduction

> *Interactive graphics is a field whose time has come. Until recently it was an esoteric specialty involving expensive display hardware, substantial computer resources, and idiosyncratic software. In the last few years, however, it has benefited from the steady and sometimes even spectacular reduction in the hardware price/performance ratio (e.g., personal computers with their standard graphics terminals), and from the development of high-level, device-independent graphics packages that help make graphics programming rational and straightforward.*
>
> James Foley and Andries van Dam, Foreword to *Fundamentals of Interactive Computer Graphics*, 1982.

When this statement was written in the early 1980s, it characterized a situation where raster graphics displays were quickly replacing the previously used vector displays, and the first graphics standards like Core [GSPC79] and GKS [Encarnação80] were evolving. Although neither the IBM PC nor the Apple Macintosh had been introduced at that time, framebuffer hardware was getting cheaper, 2D line graphics was becoming available on relatively low-end systems, and graphical user interfaces started to appear. The time for interactive graphics had indeed come.

About 15 years later, in the mid-1990s, a similar situation arose for 3D graphics. Personal computers were starting to be able to render shaded, lit, and textured triangles at rates sufficient for practical applications. Starting with games, but soon extending to business and engineering applications, 3D graphics entered the mass market. Today, in 1999, hardly any computer is sold without substantial 3D graphics hardware.

This success was again triggered by an enormous reduction of the price/performance ratio on the one hand, and a *de-facto* standardization of programming interfaces (APIs) for graphics hardware on the other hand.

The reduction of the price/performance ratio is mostly due to two facts. Firstly, modern CPUs are fast enough to perform much of the geometry processing in software. As a consequence, dedicated geometry units are no longer required for low-end to mid-range systems. Secondly, memory costs have dropped significantly throughout the last couple of years. This makes it feasible to have significant amounts of dedicated framebuffer and texture memory tightly coupled to the rasterization subsystem.

On the software side, the *de-facto* standard OpenGL has widely replaced proprietary libraries such as Starbase (Hewlett Packard), Iris GL (Silicon Graphics), and XGL (Sun Microsystems). This allows programmers to develop graphics applications that run on a wide variety of platforms, and has reduced the development cost of these applications significantly. The topic of standards for 3D libraries is discussed in more detail in Section 1.2 below.

Until recently, the major concern in the development of new graphics hardware has been to increase the performance of the traditional rendering pipeline. Today, graphics accelerators with a performance of several million textured, lit triangles per second are within reach even for the low end. As a consequence, we see that the emphasis is beginning to shift away from higher performance towards higher quality and an increased feature set that allows for the use of hardware in a completely new class of graphics algorithms.

This dissertation introduces a set of new algorithms for high quality shading and lighting using computer graphics hardware. In the course of developing these algorithms, we identify building blocks that we deem important for future hardware generations. Some of these are established features of graphics systems that are used in new, innovative ways, some are experimental features that are not yet widely used, and some are completely new features that we propose.

Thus, the contribution of this dissertation is twofold: on the one hand, it introduces a set of new algorithms for realistic image synthesis using existing computer graphics hardware. These methods are capable of providing image qualities similar to those of simple ray-tracers, but provide interactive frame rates on contemporary graphics systems. On the other hand, this thesis also identifies and introduces features and building blocks that are important for realistic shading and lighting, and thus contributes to the development of better graphics hardware in the future.

1.1 Graphics Architectures

In order to discuss algorithms using graphics hardware and future extensions for this hardware, it is useful to have a model for abstracting from a specific implementation. The standardization process mentioned above has brought forward such an abstract model, the so-called *rendering pipeline*. The vast majority of graphics systems available today are based on this model, which will be described in more detail in Chapter 4. It has also been shown that the rendering pipeline is

flexible enough to for allow new functionality to be added without breaking existing applications.

In recent years there has also been a lot of research on alternative graphics architectures. The most important of these are:

Frameless Rendering: One of the drawbacks of the traditional interactive rendering systems is that changes become only visible after the whole scene has been rendered. This is particularly disturbing if the scene is too large to be rendered at interactive frame rates. The corresponding delay between user interaction and visual feedback results in a feeling of sickness (*motion sickness*), especially when used with immersive output devices such as head mounted displays.

Frameless rendering [Bishop94] tries to overcome this problem by displaying partial results of the user interaction. Randomly chosen pixels on the screen are updated using the most recent object- and eye positions. As a result, the image looks noisy or blurry directly after a strong motion, but converges to the true image when the motion stops. Since each user interaction is immediately visible through the update of a subset of the pixels, motion sickness is dramatically reduced. Initial studies seem to indicate that noisiness and blurriness of the intermediate images are not so disturbing to the human visual system.

The major disadvantage of frameless rendering is the loss of coherence. Where traditional graphics systems can use efficient scanline techniques to scan convert triangles, ray-casting is virtually the only way to update randomly chosen pixels. Thus, it is to be expected that more powerful hardware is required to achieve the same pixel fill rate than with traditional hardware. Also, frameless rendering seems inappropriate for tasks where fine detail has to be identified or tracked by the user. This detail will only be detectable when no motion has occurred for a certain amount of time.

To date, no hardware implementation of frameless rendering is known. A software simulation was used to evaluate the idea in [Bishop94].

Talisman: The Talisman project [Torborg96, Barkans97] is an initiative by Microsoft that is tailored towards the low-end market. It differs from the conventional rendering pipeline in two major ways. First of all, instead of traversing the scene database only once and rendering each polygon no matter where it is located on the screen, Talisman subdivides the framebuffer into *tiles*[1] of 32×32 pixels. Each tile has a list of the geometry visible in it. This list is generated as a preprocessing step to each frame. Due to this approach, special purpose framebuffer RAM is saved since the depth buffer and other non-visible channels such as the alpha channel can be shared across tiles. Considering the dropping prices for memory, it is questionable how significant this advantage is.

[1] We use the term “tiles” instead of the original *chunks* used in [Torborg96] in order to be consistent with other literature describing similar approaches.

The other major difference from traditional systems is the idea of reusing previously rendered image parts for new frames. To this end, the scene is subdivided into independent layers that can be depth sorted back to front. The layers are rendered independently and composed to form the final image. When the viewpoint changes or objects move in subsequent frames, some portion of the rendered layers can be re-used by applying 2D affine transformations to the image layers. The actual geometry only has to be rerendered if the error introduced by this approach exceeds a certain limit.

One difficulty in using the Talisman architecture is finding a good grouping of objects into layers, and to reliably predict the error introduced by warping the images instead of rerendering the geometry. Of course it is also important to avoid popping effects when switching between warped and rendered representations. Some progress has been made on these issues since the Talisman project was first presented [Lengyel97, Snyder98], but several problems remain open. One of them is the question, what a programming interface for this architecture should look like. None of the existing APIs seems suitable for this tiled rendering approach without significant changes that would break existing applications. At the moment, no implementation of the Talisman architecture is commercially available, although Microsoft and other companies seem to be working on one.

PixelFlow: Perhaps the most exciting alternative to the standard rendering pipeline architecture is the PixelFlow project [Molnar92, Eyles97], which is based on image composition. The scene database is evenly split among a set of rendering boards containing both a geometry processor and a rasterizer. In contrast to the Talisman project, this splitting can be arbitrary and is not bound to depth sorted layers. Each of these boards renders a full resolution image of its part of the scene database. Similar to the Talisman project, the framebuffer is split into tiles of size 128×128 pixels. For each pixel in such a tile there exists a separate pixel processor. These scan convert triangles in parallel in a SIMD fashion.

Each of the tiles is then composed with the corresponding tile of the previous rendering board and handed to the next board in the row. The advantage of this approach is that the bandwidth required between the rendering boards is independent of scene complexity, and that the whole system scales well by adding additional rendering boards.

The most distinctive features of PixelFlow are *deferred shading* and *procedural shading*. While the normal rendering pipeline only computes the lighting of polygons at their vertices, and then interpolates the color values, deferred shading interpolates normal vectors and other information, and then performs the shading and lighting on a per-pixel basis. Combined with procedural shaders, this allows for sophisticated, highly realistic surface renderings. Some of the effects described in this thesis, such as bump- or shadow mapping are, at least in principle, straightforward to implement on PixelFlow [Olano98].

There are, however, also downsides of the PixelFlow architecture. The deferred shading

approach increases the required bandwidth. Furthermore, due to the composition architecture and the tiling approach, latency is relatively high, so that motion sickness is an issue. Anti-aliasing and transparency are difficult with this architecture, since shading happens after the composition of the tiles from the different rendering boards. This means that information about pixels behind partly transparent objects is no longer available at the time of shading. Although there are ways of doing transparency and anti-aliasing, these methods increase latency and reduce performance.

The final disadvantage, which PixelFlow shares with the other two architectures described above, is the need to explicitly store a scene database on the graphics board. Not only does this increase the memory requirements on the rendering boards, but it also introduces a performance hit and increases latency for immediate mode rendering. Scenes that do not fit into the memory on the graphics subsystem cannot be rendered at all. Due to the tiled rendering approach, PixelFlow of course has similar problems to Talisman when it comes to programming interfaces.

Despite these issues, PixelFlow represents an exciting architecture which is likely to influence future graphics systems. At the time of this writing, several prototype implementations of the PixelFlow architecture have been built by the University of North Carolina at Chapel Hill and Hewlett Packard, but Hewlett Packard stopped the development of a commercial product based on this architecture.

This discussion shows that, while alternative architectures are being developed, systems based on the traditional rendering pipeline will dominate the available systems for some time to come. It is therefore reasonable to extend the rendering pipeline with new features to increase both realism and performance. The past has shown that this is possible without making incompatible changes, simply by adding new features at the appropriate stage of the pipeline. This flexibility of the rendering pipeline together with its wide availability is also the reason why we choose it as a basis for our discussions.

1.2 Programming Interfaces

Programming interfaces have played an important role in advancing interactive 3D computer graphics. Although we will try to be as independent of a specific API as possible in the remainder of this thesis, it seems appropriate to briefly mention some of the issues at this point.

The available APIs can be classified into three different categories. At the lowest level, there are the so-called *immediate mode* APIs, which act as a hardware abstraction layer, and often provide only a very thin layer on top of the hardware. The next higher level, the so-called *retained mode* or *scene graph* APIs, store the scene in the form of a directed acyclic graph (DAG). Finally,

on the highest level we find *large model* APIs that deal with freeform surfaces such as NURBS and subdivision surfaces. They perform polygon reduction and other optimizations such as view frustum culling and occlusion culling.

For the purposes of this dissertation, scene graph and large model APIs are not of importance since many of the issues we discuss here are directly related to the underlying hardware. In the area of immediate mode APIs, a considerable amount of standardization has taken place in recent years. In the workstation market and for professional applications on personal computers, proprietary APIs such as Starbase (Hewlett Packard), Iris GL (Silicon Graphics), and XGL (Sun Microsystems) have graduately been replaced by the *de-facto* standard OpenGL. At the same time, the Direct 3D immediate mode API has been introduced by Microsoft, primarily for game development.

Since both APIs are based on the assumption that the underlying hardware follows the concepts of the rendering pipeline (see Chapter 4), it is not surprising that the feature sets of the newest versions of these APIs are very similar. Differences in functionality are not so much a consequence of conceptual differences but of the differences in the perceived application domain.

This allows us to keep the discussion in the remaining chapters largely independent of a specific programming interface. Although the abstract system we use for our algorithms (Chapter 4), mostly follows the definition of OpenGL [Segal98], most of what is said is also true for other immediate mode APIs. OpenGL is only chosen due to its open structure and the fact that it is well specified and documented.

1.3 Chapter Overview

The remainder of this thesis is organized as follows. In Chapter 2 we briefly review the physical underpinnings of image synthesis. Chapter 3 then discusses relevant previous work for the topics covered in this dissertation, and Chapter 4 defines the features a graphics system should have in order to be used for the algorithms introduced in the remainder of this thesis.

We then discuss a series of effects that add to the realism of rendered images, and how to achieve them. We start in Chapter 5 with a description of techniques for using complex, physically-based reflection models for local illumination with graphics hardware. In particular, we discuss anisotropic reflections and shaders based on the Torrance-Sparrow illumination model.

This chapter is followed by a discussion of algorithms to add shadows in Chapter 6. These include projected geometry, shadow volumes, and shadow maps. We introduce a new shadow map algorithm based on the hardware features laid out in Chapter 4. Following this, we present a light field-based model for representing complex light sources in Chapter 7.

Reflections and refractions based on environment maps are the topic of Chapter 8. This in-

cludes the development of a new parameterization for this kind of map, as well as prefiltering techniques for simulating glossy reflections. These algorithms allow for the interactive visualization of precomputed global illumination solutions for non-diffuse, curved objects.

In Chapter 9 we then discuss bump maps and normal maps. We show how normal maps can be combined with the techniques from Chapters 5 and 8.

This discussion is followed by Chapter 10 on light field techniques for even more realistic reflections and refractions than the ones presented in Chapter 8. Image-based techniques are then also applied to render realistic lens systems in Chapter 11.

Finally, in Chapter 12 we conclude by summarizing the proposed extensions to the rendering pipeline and discussing their effect on future rendering systems.

Chapter 2

Radiometry and Photometry

> *The field of* image synthesis, *also called* rendering *is a field of transformation: it turns the rules of geometry and physics into pictures that mean something to people. To accomplish this feat, the person who writes the programs needs to understand and weave together a rich variety of knowledge from math, physics, art, psychology, physiology, and computer science. Thrown together, these disciplines seem hardly related. Arranged and orchestrated by the creator of image synthesis programs, they become part of a cohesive, dynamic whole. Like cooperative members of any complex group, these fields interact in our minds in rich and stimulating ways.*
>
> Andrew S. Glassner, Foreword to *Principles of Digital Image Synthesis*, 1995.

Much of the work in computer graphics is based on results from other fields of research. Although listing all these contributions would exceed the scope of this thesis, we will in the following review some of the basic results from physics and psychophysics that are elementary for the methods discussed in the remaining chapters. More detailed discussions of these foundations can be found in [Born93] and [Pedrotti93], while the specific applications for image synthesis are discussed in [Hanrahan93] and [Glassner95].

2.1 Radiometry

Light is a form of electromagnetic radiation. As such, it can be interpreted both as a wave containing electrical and magnetic components at different frequencies (wave optics), and as a flow of particles, called *photons*, carrying the energy in certain quanta (particle optics).

In the case of wave optics, the energy is carried by oscillating electrical and magnetic fields. The oscillation directions for the electrical and the magnetic field are perpendicular to each other and to the propagation direction of the light. Light that only consists of waves whose electrical fields (and thus all magnetic fields) are aligned, is called *linearly polarized*, or simply *polarized*.

In particle optics, the energy is carried in the form of photons moving at the speed of light. Each photon has a certain amount of energy, depending on its frequency.

Wave and particle optics are largely complementary in that one can explain physical phenomena the other cannot explain easily. In computer graphics, it is possible to abstract from both wave and particle optics, and describe phenomena purely based on geometrical considerations (*geometrical* or *ray optics*) most of the time. In order to explain the laws of geometrical optics, however, results from wave and particle optics are necessary. For a derivation of these laws and a detailed discussion of optics in general, see [Born93] or [Pedrotti93].

Radiometry is the science that deals with measurements of light and other forms of electromagnetic radiation. The most important quantities in this field are also required for understanding the principles of digital image synthesis. These are

Radiant Energy is the energy transported by light. It is denoted by Q, and measured in *joules* $[J = Ws = kg\ m^2/s^2]$. Radiant energy is a function of the number of photons and their frequencies.

Radiant Flux or **Radiant Power,** denoted as Φ, is the power (energy per unit time) of the radiation. It is measured in *watts* $[W]$.

Irradiance and **Radiant Exitance** are two forms of *flux density*. The irradiance $E = d\Phi/dA$ represents the radiant flux $d\Phi$ *arriving* at a surface of area dA, while the radiant exitance B, which is often also called *radiosity* in computer graphics, describes the flux per unit area *leaving* a surface. Both quantities are measured in $[W/m^2]$.

In wave optics, flux density is defined as the product of the electrical and the magnetic fields (see [Pedrotti93] for details), and is therefore proportional to the product of their amplitudes. Since the two fields induce each other, their amplitudes are linearly dependent. As a consequence, flux density is proportional to the square of either amplitude.

Radiance is the flux per projected unit area and solid angle arriving at or leaving a point on a surface: $L(\mathbf{x}, \vec{\omega}) = d^2\Phi/(\cos\theta\ d\omega\ dA)$, where θ is the angle between the direction ω and the surface normal. Thus, radiance is measured in $[W/m^2\ sr]$, where sr stands for *steradian*, the unit for solid angles.

The relationship between irradiance and incoming radiance is

$$E(\mathbf{x}) = \int_{\Omega(\vec{n})} L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i\ d\omega_i,$$

where $\Omega(\vec{n})$ represents the hemisphere of incoming directions around the surface normal $\vec{n}$. L_i, the *incoming radiance*, is the radiance arriving at the surface point $\mathbf{x}$. A similar equation holds for the relationship between exitance (radiosity) and the radiance L_o leaving the surface (*outgoing radiance*).

Radiance is a particularly important quantity in computer graphics, since it is constant along a ray in empty space. Thus it is the quantity implicitly used by almost all rendering systems including ray-tracers and interactive graphics systems.

Intensity. *Point light sources*, which assume that all the radiant energy is emitted from a single point in 3-space, are a common model for light sources in computer graphics. Unfortunately, radiance is not an appropriate quantity to specify the brightness of such a light source, since it has a singularity at the position of the point light.

The *intensity* I is a quantity that does not have this singularity, and can therefore be used for characterizing point lights. Intensity is defined as flux per solid angle ($I = d\Phi/d\omega$). Since a full sphere of directions has a solid angle of $4\pi \cdot sr$, an isotropic point light (a light source that emits the same amount of light in each direction) has intensity $I = \Phi/4\pi \cdot sr$.

Radiant Exposure is the integral of irradiance over time, and is measured in $[W\ s/m^2]$. Another way to understand exposure is to see it as radiant energy per unit area. The response of a piece of film is a function of the exposure it was subject to.

All of the above quantities can, and, in general will, additionally vary with the wavelength of light. For example, the *spectral radiant energy* Q_λ is given as $dQ/d\lambda$, and its units are consequently $[J/m]$, while *spectral radiance* $L_\lambda := dL/d\lambda$ is measured in $[W/m^3\ sr]$.

Although wavelength dependent effects can be prominent, most rendering systems, and, in particular, graphics accelerators, do not deal with spectral quantities due to the high computational and storage costs this would impose. For the same reason the discussion in this thesis will also largely ignore spectral effects.

2.2 Photometry

In contrast to radiometry, which is a physical discipline, photometry is the *psychophysical* field of measuring the visual sensation caused by electromagnetic radiation. The sensitivity of the human eye to light is a function of the wavelength. This function, which is called the *luminous efficiency* of the eye, is depicted in Figure 2.1. It shows the relative perceived brightness of light with the same power at different frequencies.

The curve shows that the human eye is most sensitive for light at a wavelength of approximately $555\ nm$. The sensitivity for red light is only about 40% of the sensitivity for green, while

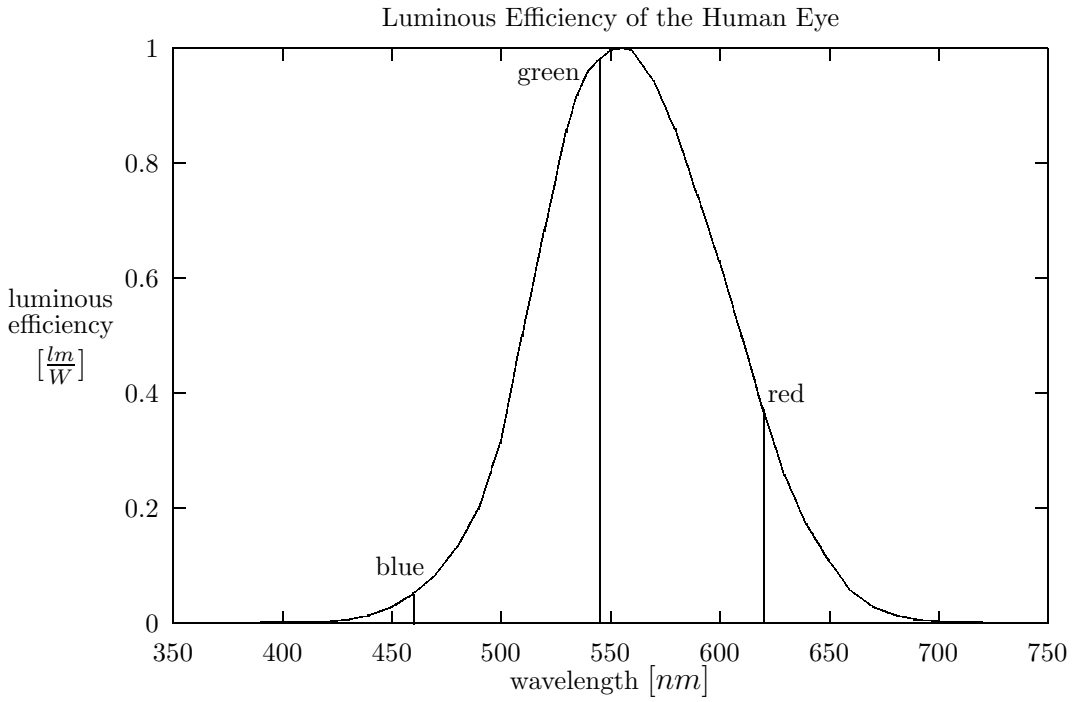

Figure 2.1: The luminous efficiency curve of the human eye. The visible spectrum lies between approximately $380\ nm$ (violet) and $770\ nm$ (red). The maximum is at $555\ nm$ (green).

for blue it is even lower (approximately 8%). The exact numbers, of course, depend on the frequencies assumed for red, green, and blue, which in turn depend on the specific application. For example for on-screen representations, the frequencies depend on the spectra of the phosphors used in the CRT, and on the calibration of the display.

For each radiometric quantity, photometry also provides a quantity where the different light frequencies are weighted by the luminous efficiency function. These quantities and their units are listed below:

Luminous Flux is the flux weighted by the luminous efficiency function, and is measured in *lumens* $[lm]$.

Illuminance and **Luminosity** are luminous flux densities, and correspond to irradiance and radiosity, respectively. They are measured in *Lux* $[lx := lm/m^2]$.

Luminance is luminous flux density per solid angle, and therefore corresponds to the photometric quantity radiance. It is measured in *Candela* $[cd := lm/m^2\ sr = lx/sr]$.

Luminous Intensity and **Luminous Exposure** are the photometric quantities corresponding to the intensity and the radiant exposure, respectively.

For a more detailed discussion of radiometric and photometric terms, refer to [Ashdown96], [Hanrahan93], and [Glassner95].

2.3 Bidirectional Reflection Distribution Functions

In order to compute the illumination in a scene, it is necessary to specify the optical properties of a material. This is usually done in the form of a *bidirectional reflection distribution function* (BRDF). It is defined as follows:

$$f_r(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o) := \frac{dL_o(\mathbf{x}, \vec{\omega}_o)}{dE(\mathbf{x}, \vec{\omega}_i)} = \frac{dL_o(\mathbf{x}, \vec{\omega}_o)}{L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i \, d\omega_i}. \tag{2.1}$$

The BRDF is the radiance L_o leaving a point $\mathbf{x}$ in direction $\vec{\omega}_o$ divided by the irradiance arriving from direction $\vec{\omega}_i$. Its unit is $[1/sr]$. The BRDF describes the *reflection* of light at a surface. Similarly, the *bidirectional transmission distribution function* (BTDF) can be defined for refraction and transmission. The combination of BRDF and BTDF is usually called *bidirectional scattering distribution function* (BSDF).

Although BRDFs are used very often in computer graphics, it is important to note that they cannot model all physical effects of light interacting with surfaces. The simplifying assumptions of BRDFs (and BTDFs) are:

- the reflected light has the same frequency as the incoming light. *Fluorescence* is not handled.
- light is reflected *instantaneously*. The energy is not stored and re-emitted later (*phosphorescence*).
- there are no participating media. That is, light travels in empty space, and if it hits a surface, it is reflected at the same point without being scattered within the object. This is the most restrictive assumption, since it means that atmospheric effects as well as certain materials such as skin cannot be treated adequately.

In general, the BRDF is a 6-dimensional function, because it depends on two surface parameters ($\mathbf{x}$) and two directions with two degrees of freedom each ($\vec{\omega}_i$ and $\vec{\omega}_o$). Often however, it is assumed that a surface has no *texture*, that is, the BRDF is constant across an object. This

reduces the dimensionality to four, and makes sampled BRDF representations smaller and easier to handle.

The dimensionality can be further reduced by one through the assumption of an *isotropic* material. These are materials whose BRDFs are invariant under a rotation around the normal vector. Let $\vec{\omega}_i = (\theta_i, \phi_i)$ and $\vec{\omega}_o = (\theta_o, \phi_o)$, where θ describes the angle between the normal and the respective ray (the elevation), and ϕ describes the rotation around the normal (the azimuth). Then the following equation holds for isotropic materials and arbitrary $\Delta\phi$:

$$f_r(\mathbf{x}, (\theta_i, \phi_i + \Delta\phi) \rightarrow (\theta_o, \phi_o + \Delta\phi)) = f_r(\mathbf{x}, (\theta_i, \phi_i) \rightarrow (\theta_o, \phi_o)).$$

All other materials are called *anisotropic*.

2.3.1 Reflectance and Transmittance

While the BRDF is an accurate and useful description of surface properties, it is sometimes inconvenient to use because it may have singularities. For example, consider the BRDF of a perfect mirror. According to Equation 2.1, $f_r(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o)$ goes to infinity if $\vec{\omega}_o$ is exactly the reflection of $\vec{\omega}_i$, and zero otherwise.

Another quantity to describe the reflection properties of materials is the reflectance ρ. It is defined as the ratio of reflected flux to incoming flux:

$$\rho := \frac{d\Phi_o}{d\Phi_i}. \tag{2.2}$$

From this definition it is obvious that ρ is unitless and bounded between 0 and 1. Unfortunately, the reflectance in general depends on the directional distribution of the incoming light, so that a conversion between BRDF and reflectance is not easily possible. In the important special case of a purely diffuse (Lambertian) reflection, however, f_r is a constant, and $\rho = \pi \cdot f_r$.

In analogy to the reflectance, the *transmittance* τ can be defined as the ratio of transmitted to received flux. The fraction of flux that is neither reflected nor transmitted, but absorbed, is called *absorptance* α. The sum of reflectance, transmittance, and absorptance is always one:

$$\rho + \tau + \alpha = 1.$$

2.3.2 Physical Reflection and Transmission Properties of Materials

Independent of the actual dimensionality of the BRDF, it has to obey certain physical laws. The first of all these laws is the conservation of energy: No more energy must be reflected than is received. This is guaranteed if the following equation holds (see [Lewis93] for a derivation).

$$\int_{\Omega(\vec{n})} f_r(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o) \cos\theta_o \, d\omega_o \leq 1 \quad \forall \vec{\omega}_i \in \Omega(\vec{n}). \tag{2.3}$$

The second physical law that a BRDF should obey is known as *Helmholtz reciprocity*. It states that, if a photon follows a certain path, another photon can follow the same path in the opposite direction. In the case of reflections, this means that

$$f_r(\mathbf{x}, \vec{\omega}_o \rightarrow \vec{\omega}_i) = f_r(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o). \tag{2.4}$$

For refraction this relation does not hold, since the differential diameter of a refracted ray is different from that of the original ray.

In addition to being physically valid, models for surface materials should also be *plausible* in the sense that they model the reflection characteristics of real surfaces. The basic principles underlying plausible BRDFs are reflection at a planar surface, Snell's law, and the Fresnel formulae. These principles will be reviewed in the following.

Consider a ray of light arriving from direction $\vec{l}$ at a perfectly smooth, planar surface between two materials with optical densities n_1 and n_2. This situation is depicted in Figure 2.2.

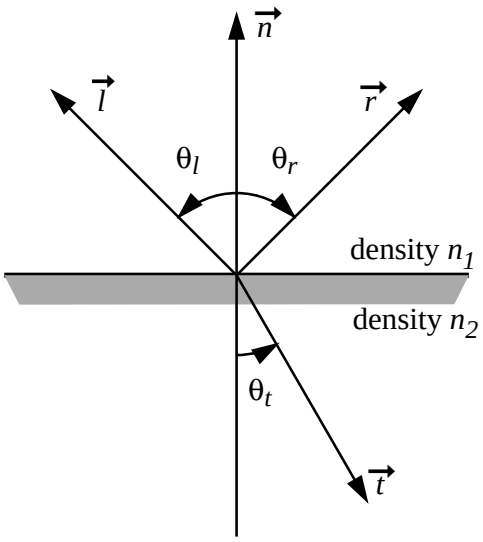

Figure 2.2: Reflection and refraction on a planar surface.

Since the surface is perfectly smooth, this ray will be split into exactly two new rays, one for the reflected, and one for the refracted part. The reflected ray is given via the relation

$$\theta_r = \theta_l, \tag{2.5}$$

while the refracted ray direction is given by Snell's law (see [Born93] for a derivation):

$$n_1 \sin\theta_l = n_2 \sin\theta_t. \tag{2.6}$$

The question is, how is the energy of the incoming ray split between the reflected and the refracted part, that is, what are the values for the reflectance and the transmittance. This depends on the polarization of the light. Let $r^\perp$ be the ratio of the reflected to the incoming amplitude of the electrical field perpendicular to the plane formed by the surface normal and the incoming light ray. Let $r^\parallel$ be the same ratio for an electrical field parallel to this plane, and let $t^\perp$ and $t^\parallel$ be the corresponding ratios for the transmitted amplitudes.

The Fresnel formulae specify these ratios in terms of the angles θ_l and θ_t for non-magnetic materials (permeability ≈ 1) without absorption ($\alpha = 0$):

$$r^\perp = \frac{n_1\cos\theta_l - n_2\cos\theta_t}{n_1\cos\theta_l + n_2\cos\theta_t}, \quad r^\parallel = \frac{n_2\cos\theta_l - n_1\cos\theta_t}{n_2\cos\theta_l + n_1\cos\theta_t} \tag{2.7}$$

$$t^\perp = \frac{2n_1\cos\theta_l}{n_1\cos\theta_l + n_2\cos\theta_t}, \quad t^\parallel = \frac{2n_1\cos\theta_l}{n_2\cos\theta_l + n_1\cos\theta_t}. \tag{2.8}$$

Since $\rho = d\Phi_o/d\Phi_i = dB/dE$, and because the flux density is proportional to the square of the amplitude of the electrical field (see Section 2.1), we need to square these ratios in order to get the reflectance and transmittance. For unpolarized light, which has random orientations of the electrical field, the reflectance for the perpendicular and the parallel components need to be averaged (see [Born93]):

$$\begin{aligned} \rho &= \frac{(r^\perp)^2 + (r^\parallel)^2}{2}, \quad \text{and} \\ \tau &= \frac{n_2\cos\theta_l}{n_1\cos\theta_t} \cdot \frac{(t^\perp)^2 + (t^\parallel)^2}{2}. \end{aligned} \tag{2.9}$$

Note that $\rho + \tau = 1$ due to the assumption of a non-absorbing material. Note also, that even if the incident light is unpolarized, the reflected light becomes polarized due to Equation 2.7. This fact is usually ignored in Computer Graphics. Figure 2.3 shows the reflectance for unpolarized light for the example of a surface between air ($n_1 \approx 1$) and glass ($n_2 \approx 1.5$).

As mentioned above, these formulae describe the reflection at a perfectly smooth, planar surface. In reality, however, surfaces are rough, and thus light is reflected and refracted in all

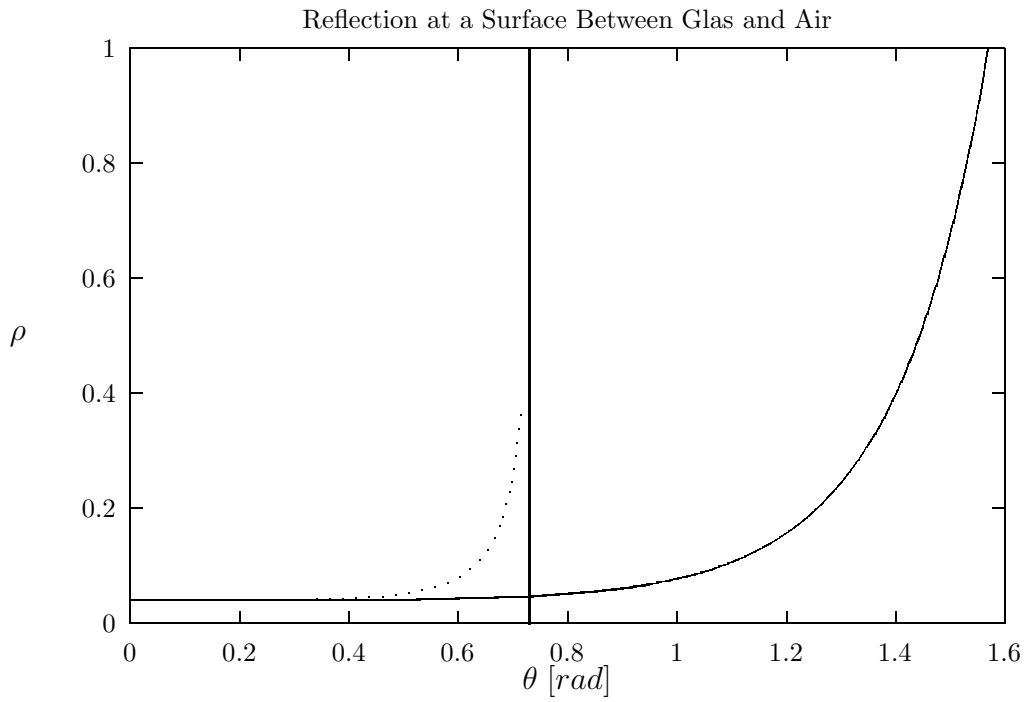

Figure 2.3: Reflectance ρ according to Fresnel for the interaction of unpolarized light with a surface between glass ($n = 1.5$) and air ($n \approx 1$). The solid curve is for light rays arriving from the air, while the dotted curve is for rays arriving from the glass side. Note the total reflection at an angle $\sin\theta = 1/1.5$ in the latter case.

directions, not only $\vec{r}$ and $\vec{t}$. In this case, the surface is assumed to consist of small planar regions, or *facets*, for which the above formulae can be applied. The reflection on such a rough surface then depends on the statistics of the orientations for these facets. This approach to describing the reflection on rough surfaces is called *micro facet theory*. In Chapter 3 we will review some of the related reflection models that have gained importance in the context of computer graphics.

2.4 Rendering Equation

Now that we have a description of the material properties of a surface, we can describe the illumination in a scene through an integral equation representing all the reflections in the scene. This equation is called *Rendering Equation*, and was first presented in [Kajiya86].

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \int_{\Omega(\vec{n})} f_r(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o) \cdot L_i(\mathbf{x}, \vec{\omega}_i) \cdot \cos(\vec{n}, \vec{\omega}_i) \quad d\omega_i. \tag{2.10}$$

The radiance L_o leaving a certain surface point $\mathbf{x}$ in direction ω_o is the radiance L_e emitted in this direction (if the surface is a light source), plus the reflected radiance. The latter is the contribution from the incoming radiance L_i at point $\mathbf{x}$ integrated over the hemisphere $\Omega(\vec{n})$. The contribution for each incoming direction is specified using the BRDF.

Due to the use of the BRDF, the rendering equation shares the limitations already mentioned in Section 2.3: the inability to handle participating media, fluorescence, and phosphorescence. In addition, the rendering equation assumes that the light is in a state of *equilibrium*, that is, changes in illumination happen relatively slowly (light sources and objects move much slower than the speed of light).

The rendering equation describes the inter-reflection of light in a scene (*global illumination*), and thus also accounts for indirect illumination in a scene. In contrast to this, graphics hardware only accounts for light directly arriving from a finite number of point-, directional-, or spot lights (*local illumination*). For this situation, Equation 2.10 simplifies to

$$L_o(\mathbf{x}, \vec{\omega}_o) = L_e(\mathbf{x}, \vec{\omega}_o) + \sum_{j=1}^{n} f_r(\mathbf{x}, \vec{\omega}_i \rightarrow \vec{\omega}_o) \cdot g(\mathbf{x}) \cdot I_j(\mathbf{x}, \vec{\omega}_i) \cdot \cos(\vec{n}, \vec{\omega}_i), \tag{2.11}$$

where I_j is the intensity of the j^{th} light source, and $g \cdot I_j$ is the incoming radiance at $\mathbf{x}$ due to that light source. For point and spot lights, the geometry term $g = 1sr/r^2$ represents the quadratic falloff of intensity with the distance r of the light source from the surface point. For directional light sources, such a quadratic falloff does not exist, and $g = 1sr/m^2$. Note that this assumes that each light source is visible from point $\mathbf{x}$. Shadows can be handled by incorporating a separate visibility term into g, which is either 0 (for shadowed points) or 1 (for illuminated points).

One of the topics of this thesis is to overcome the restrictions of local illumination, and to allow for the interactive visualization of global illumination solutions for non-diffuse scenes (see Chapters 8 and 10). These solutions need to be generated in a preprocessing stage that solves Equation 2.10.

Chapter 3

Related Work

Before we present our own algorithms for realistic shading and lighting, we give an overview of relevant previous work in this chapter. We start with a discussion of reflection models for describing the local interaction of light with surfaces. We then review several multi-pass techniques for achieving realism using graphics hardware, and finally give an overview of light field techniques, which form the basis for some of our own methods.

3.1 Reflection Models

While it is in principle possible to use measured BRDF data for image synthesis, this is in practice often not feasible due to the large amount of data required to faithfully represent these high-dimensional functions. *Reflection models* or *lighting models* attempt to describe classes of BRDFs in terms of simple formulae using only a few parameters to customize the model. These parameters can either be adjusted manually by a user or programmer, or they can be chosen automatically to best fit measured BRDF data.

In the following, we review a number of models that will be used throughout the thesis, in particular in the chapter on local illumination (Chapter 5), as well as for the prefiltering of environment maps in Chapter 8. The same models are also applied to normal-mapped surfaces (Chapter 9).

The choice of a specific reflection model is a trade-off between performance and physical accuracy. Since the lighting model is evaluated quite often during the synthesis of an image, its computational cost has a strong impact on the total rendering time. Nonetheless, for realistic lighting effects, a model should follow basic physical principles, as laid out in Section 2.3.2.

For the following discussion we use the geometric entities depicted in Figure 3.1. Let $\vec{n}$ be the surface normal, $\vec{v}$ the viewing direction, $\vec{l}$ the light direction, $\vec{h} := (\vec{v}+\vec{l})/|\vec{v}+\vec{l}|$ the halfway vector between viewing and light direction, and $\vec{r}_l := 2 < \vec{l}, \vec{n} > \vec{n} - \vec{l}$ the reflection of the light

vector at the surface normal. The tangent direction for any microscopic features that may cause anisotropy is denoted $\vec{t}$. All these vectors are assumed to be normalized.

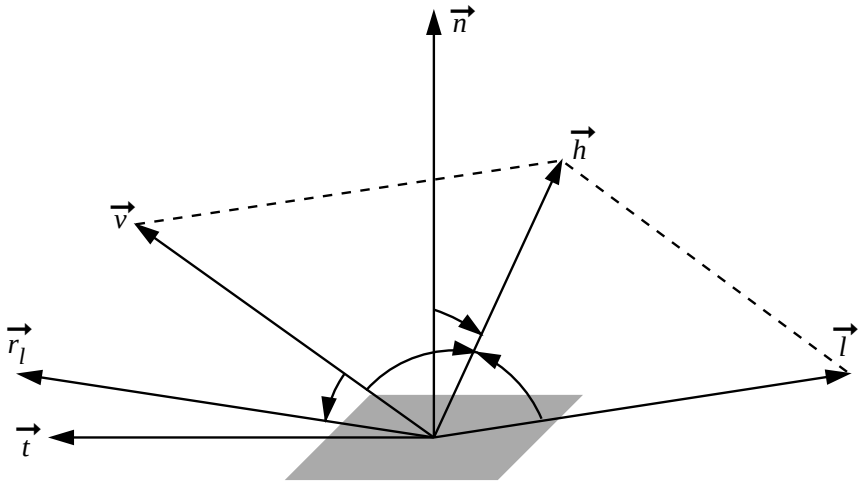

Figure 3.1: Geometric entities for the reflection models in this section.

3.1.1 Ambient and Diffuse Lighting

The most simplistic model is that of *ambient illumination*. It assumes that light is arriving uniformly from all directions. The ambient model is not physically valid by any means, but it can be used to approximate the indirect illumination in a scene. Due to the uniform distribution of the incoming light, the ambient light reflected off a surface is simply a constant times the ambient illumination, which is described by L_a, the radiance in the scene averaged over all points and directions:

$$L_o(\mathbf{x}, \vec{v}) = k_a \cdot L_a. \tag{3.1}$$

Many materials, especially in architectural scenes, appear roughly equally bright from all viewing directions. This is known as *diffuse* or *Lambertian* reflection. The BRDF of a Lambertian surface is a constant

$$f_r(\mathbf{x}, \vec{l} \rightarrow \vec{v}) = k_d. \tag{3.2}$$

Note that, according to Equation 2.3, the reflection coefficient k_d should be less than $1/\pi$ for reasons of energy conservation! In many rendering systems, and especially hardware-based

systems, k_d is a value between 0 and 1. This means that the intensity of the light source is implicitly multiplied by a factor of π!

3.1.2 Models by Phong and Blinn-Phong

The Phong lighting model [Bui-Tuong75] was one of the first models in computer graphics to account for specular reflections. For efficient computation, but without a physical justification, the model uses powers of the cosine between reflected light vector and viewing vector for the scattering of light on the surface:

$$L_o(\mathbf{x}, \vec{v}) = k_s \cdot \cos(\vec{r}_l, \vec{v})^n \cdot L_i = k_s \cdot < \vec{r}_l, \vec{v} >^n \cdot L_i. \tag{3.3}$$

In this equation and in the following, all cosine values are implicitly clamped to the range $[0 \dots 1]$. This is not made explicit in order not to complicate the formulae. By comparison with Equation 2.11, it is clear that the corresponding BRDF is

$$f_r(\mathbf{x}, \vec{l} \to \vec{v}) = k_s \frac{\cos(\vec{r}_l, \vec{v})^n}{\cos(\vec{l}, \vec{n})} = k_s \frac{< \vec{r}_l, \vec{v} >^n}{< \vec{l}, \vec{n} >}. \tag{3.4}$$

As pointed out in [Lewis93], this BRDF does not conserve energy due to the denominator $< \vec{l}, \vec{n} >$, which may become arbitrarily small. The Helmholtz reciprocity is also violated, since the equation is not symmetric in $\vec{l}$ and $\vec{v}$. [Lewis93] therefore proposed to use the term $k_s < \vec{r}_l, \vec{v} >^n$ directly as the BRDF, which fixes both problems if an additional scaling factor of $(n + 2)/2\pi$ is introduced for energy conservation (Equation 2.3). This factor can either be merged with the reflection coefficient k_s, or, more conveniently, be added as a separate term, which then allows for $k_s \in [0 \dots 1]$. This modified Phong model has come to be known as the *Cosine Lobe Model*.

A slightly modified version of the Phong model, known as the Blinn-Phong model was discussed in [Blinn76]. Instead of the powered cosine between reflected light and viewing direction, it uses powers of the cosine between the halfway vector $\vec{h}$ and the surface normal:

$$f_r(\mathbf{x}, \vec{l} \to \vec{v}) = k_s \frac{< \vec{h}, \vec{n} >^n}{< \vec{l}, \vec{n} >}. \tag{3.5}$$

This models a surface consisting of many small, randomly distributed micro facets that are perfect mirrors. The halfway vector is the normal of the facets contributing to the reflection for given viewing-, and light directions. The cosine power is a simple distribution function, describing how likely a certain micro facet orientation is.

Like the Phong model, the Blinn-Phong model itself is not physically valid, but with the same modifications described above, Helmholtz reciprocity and energy conservation can be guaranteed.

Both the Phong and the Blinn-Phong model are very popular in interactive computer graphics due to their low computational cost and simplicity. They are usually combined with a diffuse and an ambient term as described in Section 3.1.1. In Chapter 5 we present algorithms for replacing these simple models in hardware-based renderings with one of the physically-based models discussed below.

3.1.3 Generalized Cosine Lobe Model

As the name suggests, the generalized cosine lobe model [Lafortune97] is a generalization of Lewis' modifications to the Phong model. The term $k_s < \vec{r}_l, \vec{v} >^n$ can be written in matrix notation as $k_s[\vec{l}^T \cdot (2 \cdot \vec{n} \cdot \vec{n}^T - \mathbf{I}) \cdot \vec{v}]^n$. The generalized cosine lobe model now allows for an arbitrary symmetrical matrix $\mathbf{M}$ to be used instead of the Householder Matrix $2\vec{n}\vec{n}^T - \mathbf{I}$:

$$f_r(\mathbf{x}, \vec{l} \rightarrow \vec{v}) = k_s[\vec{l}^T \cdot \mathbf{M} \cdot \vec{v}]^n. \tag{3.6}$$

Let $\mathbf{Q}^T\mathbf{D}\mathbf{Q}$ be the singular value decomposition of M. Then, $\mathbf{Q}$ can be interpreted as a new coordinate system, into which the vectors $\vec{l}$ and $\vec{v}$ are transformed. In this new coordinate system, Equation 3.5 reduces to a weighted dot product:

$$f_r(\mathbf{x}, \vec{l} \rightarrow \vec{v}) = k_s(D_x\vec{l}x\vec{v}_x + D_y\vec{l}_y\vec{v}_y + D_z\vec{l}_z\vec{v}_z)^n. \tag{3.7}$$

The advantage of this model is that it is well suited to fit real BRDF data obtained through measurements [Lafortune97]. In contrast to the original cosine lobe model, the generalized form is also capable of describing anisotropic BRDFs by choosing $D_x \neq D_y$.

3.1.4 Torrance-Sparrow Model

The illumination model by Torrance and Sparrow [Torrance66, Torrance67] is one of the most important physically-based models for the interreflection of light at rough surfaces (the variation by Cook and Torrance [Cook81] is also quite widespread for spectral renderings). It is given as

$$f_r(\mathbf{x}, \vec{l} \rightarrow \vec{v}) = \frac{F \cdot G \cdot D}{\pi \cdot < \vec{n}, \vec{l} > \cdot < \vec{n}, \vec{v} >}, \tag{3.8}$$

where F, the Fresnel term, is the reflectance ρ from Equation 2.9. It is usually given in the form

$$F = \frac{(g-c)^2}{2(g+c)^2}\left[1 + \frac{(c(g+c)-1)^2}{(c(g-c)+1)^2}\right], \tag{3.9}$$

with $c =< \vec{h}, \vec{v} >$ and $g^2 = n^2 + c^2 - 1$. Equivalence to Equation 2.9 can be shown using trigonometric identities.

The term D in Equation 3.8 is the distribution function for the micro-facets. Multiple choices have been proposed for this term by different authors, including [Beckmann63]. The function given in [Torrance67] assumes a Gaussian distribution of the angle between normal and halfway vector:

$$D = e^{(k \cdot \sphericalangle(\vec{n},\vec{h}))^2}, \tag{3.10}$$

where k is the standard deviation of the surface angles, and is a surface property. Many other researchers have since used a Gaussian distribution of the *surface heights*, which also results in a Gaussian distribution of *surface slopes* [Smith67, He91]. Let σ be the RMS deviation of the surface height. Then the RMS deviation of surface slopes is proportional to σ/τ, where τ, a parameter of the model, is a measure for the distance of two surface peaks [He91].

Finally, the term G describes the geometrical attenuation caused by the self-shadowing and masking of the micro-facets. Under the assumption of symmetric, v-shaped groves, G is given as

$$G = \min\left\{1, \frac{2 < \vec{n}, \vec{h} >< \vec{n}, \vec{v} >}{< \vec{h}, \vec{v} >}, \frac{2 < \vec{n}, \vec{h} >< \vec{n}, \vec{l} >}{< \vec{h}, \vec{v} >}\right\}. \tag{3.11}$$

This model for shadowing and masking was later improved by [Smith67] for the specific case of a Gaussian height distribution function D (erfc denotes the error function complement):

$$\begin{aligned} G &= S(< \vec{n}, \vec{v} >) \cdot S(< \vec{n}, \vec{l} >) \\ \text{where} \quad S(x) &= \frac{1 - \frac{1}{2}\mathrm{erfc}(\frac{\tau \cot x}{2\sigma})}{\Lambda(\cot x) + 1}, \\ \text{and} \quad \Lambda(\cot x) &= \frac{1}{2}\left(\frac{2}{\sqrt{\pi}} \cdot \frac{\sigma}{\tau \cot x} - \mathrm{erfc}(\frac{\tau \cot x}{2\sigma})\right). \end{aligned} \tag{3.12}$$

Several variations of this model have been proposed, which will not be discussed in detail here. In particular, [Schlick93] has proposed a model in which all terms are approximated by

much simpler rational polynomial formulae to improve performance. Moreover, it is possible to account for anisotropic reflections by changing the micro facet distribution function D [Kajiya85, Poulin90, Schlick93]. An overview of the variations to the Torrance-Sparrow model can be found in [Hall89].

[He91] introduced an even more comprehensive model, which is also capable of simulating polarization effects. This model is too complex for our purposes, and will not be considered in this thesis. The Torrance-Sparrow model and its variants, however, can be used with computer graphics hardware by applying the techniques from Chapter 5.

3.1.5 Anisotropic Model by Banks

A very simple anisotropic model has been presented by Banks [Banks94]. The model assumes that anisotropy is caused by long, thin features, such as scratches or fibers. Seen from relatively far away, these features cannot be resolved individually, but their orientation changes the directional distribution of the reflected light.

This suggests that anisotropic surfaces following this model can be illuminated using a model for the illumination of lines in 3D. The fundamental difficulty in the illumination of 2D manifolds in 3D is that every point on a curve has a unique tangent, but an infinite number of normal vectors. Every vector that is perpendicular to the tangent vector of the curve is a potential candidate for use in the illumination calculation.

For reasons described in [Banks94], and [Stalling97], the vector $\vec{n}'$ selected from this multitude of potential normal vectors should be the projection of the light vector $\vec{l}$ into the normal plane, as depicted in Figure 3.2.

Applied to anisotropic materials, this means that the projection of the light vector $\vec{l}$ into the normal plane of the tangent vector $\vec{t}$ is used as the shading normal $\vec{n}'$. At this point, any of the isotropic reflection models described above can be used, but usually the Phong model is chosen.

Like the Torrance-Sparrow Model, the anisotropic model by Banks can be used for hardware-based renderings with the techniques from Chapter 5.

3.2 Hardware and Multi-Pass Techniques

Many researchers have in the past developed methods for improving the realism of hardware accelerated graphics. Most of these techniques fall into the category of *multi-pass rendering*, which means that they require the hardware to render the geometry multiple times with different rendering attributes (textures, colors, light sources and so forth). The resulting images or image parts are then combined using alpha blending or the accumulation buffer. The ones that are most directly related to our work are listed below:

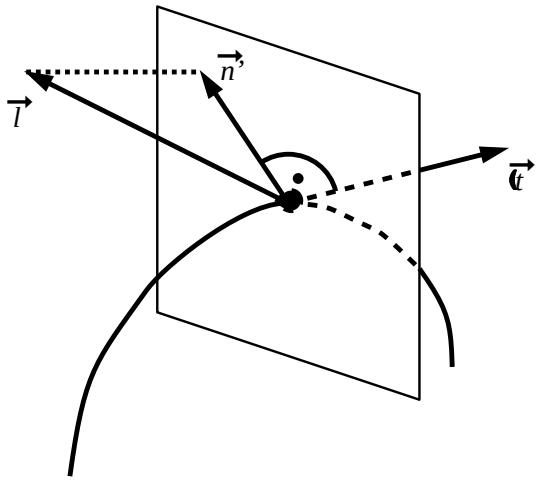

Figure 3.2: In order to find the shading normal $\vec{n}'$, the light vector $\vec{l}$ is projected into the normal plane.

Reflections off planar mirrors can be rendered using simple affine transformations on the geometry [Diefenbach94, Diefenbach96]. The reflector is first rendered once into the stencil buffer. Then the mirrored geometry is rendered at each pixel covered by the reflector. The method can be iterated to achieve multiple reflections. In Chapter 10 we describe light field-based techniques for curved reflectors that are a generalization of this approach.

Mirror reflections off curved surfaces are usually approximated with the help of *environment maps* [Blinn76]. Current graphics hardware supports *spherical environment maps* (see Chapter 8), which require re-generation of the map for every change of viewing direction relative to the environment. In Chapter 8 we propose a different parameterization for environment maps, and introduce methods for applying them to other materials than perfect mirrors.

Shadows can be rendered in many different ways. There are several algorithms that only work in very special settings, such as shadows cast on a single large polygon. General solutions for rendering of shadows are *shadow volumes* [Crow77, Diefenbach96] and *shadow maps* [Williams78, Segal92], which are directly supported by some hardware [Akeley93, Montrym97]. Shadow algorithms will be discussed in detail in Chapter 6, and a new algorithm for implementing shadow maps in hardware will be introduced.

Bump Maps [Blinn78] describe variations of the surface normal across a polygonal object. While some researchers have designed dedicated hardware for this task (see, for example [Peercy97, Miller98a, Ernst98]), others, like [McReynolds98], have developed multi-pass techniques to achieve bump mapping. Chapter 9 deals with bump- and normal mapping.

Realistic camera effects, especially depth-of-field and motion blur can be achieved using an accumulation buffer [Haeberli90]. A new light field based approach that is capable of simulating even more effects is described in Chapter 11.

This is only a small portion of the available multi-pass techniques; for a general overview see [McReynolds98]. Some of the techniques mentioned above will be discussed in more detail in one of the remaining chapters.

3.2.1 Visualization of Global Illumination Solutions

Since the hardware itself is only capable of computing the local illumination of a point on a surface, there has recently been a lot of interest in developing methods for using the graphics hardware to visualize global illumination solutions, or to accelerate the computation of these solutions with the help of graphics hardware. In our work in Chapters 8 and 10, we assume that global illumination solutions have been acquired in a precomputation phase, so that the task is reduced to visualizing this solution in real time.

A common technique is to use Gouraud shading and texture mapping to render radiosity solutions of diffuse environments from any perspective [Cohen93, Myskowski94, Sillion94, Bastos97]. Another approach for diffuse scenes is *Instant Radiosity* [Keller97], which computes a radiosity solution with a combination of a Quasi Monte Carlo photon tracer and hardware lighting with a large number (≈ 200) of light sources.

Algorithms for walkthroughs of scenes containing specular surfaces have also been developed. [Stürzlinger97] uses high-end graphics hardware to quickly display the solution of a photon map algorithm. In [Stamminger95] and [Walter97], OpenGL light sources are placed at virtual positions to simulate the indirect illumination reflected by glossy surfaces.

3.3 Light Fields

Some of the techniques developed in this dissertation apply the concept of *light fields* to hardware-accelerated techniques. A light field [Levoy96] is a 5-dimensional function describing the radiance at every point in space in each direction. It is closely related to the *plenoptic function* introduced in [Adelson91], which in addition to location and orientation also describes the wavelength dependency of light. Thus, the plenoptic function is a *spectral light field* in the notation from Section 2.1.

Since radiance does not change along a ray in empty space, the dimensionality of the light field in empty space can be reduced by one, if an appropriate parameterization is found. The so-called two-plane parameterization fulfills this requirement. It represents a ray via its intersection

points with two parallel planes. Since each of these points is characterized by two parameters in the plane, this results in a 4-dimensional function that can be densely sampled through a regular grid on each plane (see Figure 3.3).

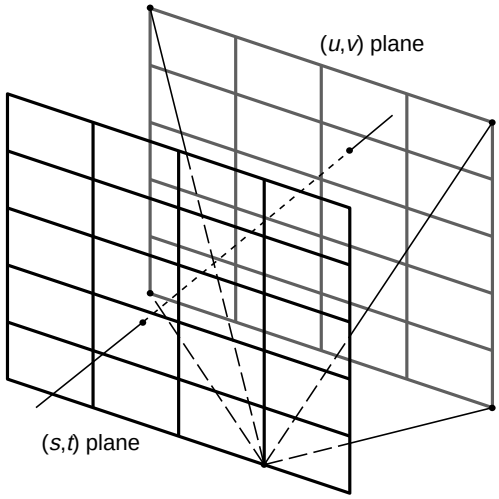

Figure 3.3: A light field is a 2-dimensional array of images taken from a regular grid of eye points on the (s,t)-plane through a window on the (u,v)-plane. The two planes are parallel, and the window is the same for all eye points.

One useful property of the two-plane parameterization is that all the rays passing through a single point on the (s,t)-plane form a perspective image of the scene, with the (s,t) point being the center of projection. Thus, a light field can be considered a 2-dimensional array of perspective projections with eye points regularly spaced on the (s,t)-plane. Other properties of this parameterization have been discussed in detail in [Gu97].

Since we assume that the sampling is dense, the radiance along an arbitrary ray passing through the two planes can be interpolated from the known radiance values in nearby grid points. Each such ray passes through one of the grid cells on the (s,t)-plane and one on the (u,v)-plane. These are bounded by four grid points on the respective plane, and the radiance from any of the (u,v)-points to any of the (s,t)-points is stored in the data structure. This makes for a total of 16 radiance values, from which the radiance along the ray can be interpolated quadri-linearly. As shown in [Gortler96, Sloan97], this algorithm can be considerably sped up by the use of texture mapping hardware.

Other parameterizations for the light field have been proposed [Camahort98, Tsang98] in order to achieve a better sampling uniformity. These parameterizations will, however, not be used here since they are not well suited for hardware acceleration.

3.3.1 Lumigraphs: Light Fields with Additional Geometry

The quadri-linear interpolation in the light field data works well as long as the resolution of the light field is high. For low resolutions, the interpolation only yields a sharp image for objects in the (u, v)-plane. The further away points are from this plane, the more blurred they appear in the interpolated image.

The Lumigraph [Gortler96] extends the concept of a light field by adding some geometric information that helps compensating for this problem. A coarse polygon mesh is stored together with the images. The mesh is used to first find the approximate depth of the object along the ray to be reconstructed, and then this depth is used to correct the weights for the interpolation.

In [Heidrich99c] a similar, but purely sampling-based approach is taken. Instead of a polygon mesh, the depth of each pixel in the light field is stored. This information is then used to refine the light field with warped images until the rendering quality is satisfactory. This decouples the more expensive depth correction from the efficient quadri-linear interpolation, and thus can be used to achieve higher frame rates.

Chapter 4

Rendering Pipeline

In this chapter, we lay out the fundamental features of a graphics system on which we rely for the algorithms in the following chapters. Since this abstract system is largely identical with version 1.2 of the OpenGL API including the imaging subset, we only give a coarse overview of the structure, and refer to [Segal98] for the details. However, we also use some newer extensions which have not yet found their way into any API standard. These features will be discussed in more detail.

Most contemporary graphics hardware and APIs are variations of the traditional 3D rendering pipeline [Foley90]. The key stages of this pipeline are depicted in Figure 4.1.

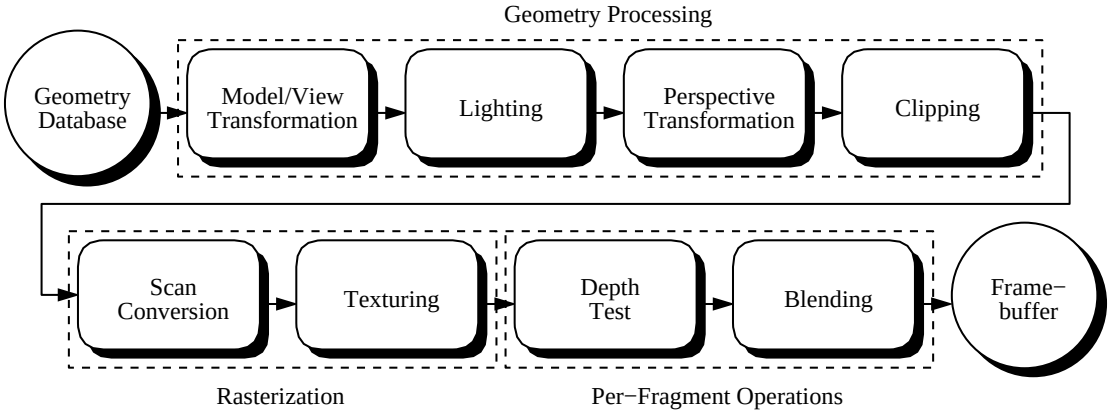

Figure 4.1: The traditional rendering pipeline.

These operations can be categorized into three groups: *geometry processing* operations are responsible for geometrical transformations and lighting, during *rasterization*, polygons and other geometrical primitives are scan-converted and textured, and finally the *per-fragment operations* perform depth, stencil and alpha tests, as well as blending operations.

Since the graphics hardware usually relies on fast, dedicated memory for textures and the framebuffer, there also needs to be a group of operations for transferring images between main memory, framebuffer and texture RAM. These are called *pixel transfer operations*. In addition to simply transferring the pixel data, this group also includes more complex operations such as color matrix transformations, color lookup tables, and convolutions. These operations are applied to the pixel data as it is transferred between the different kinds of memory.

Implementations of this basic pipeline structure can range from pure software implementations [Paul94], over hardware support for rasterization and per-fragment operations (see, for example [Kilgard97]) to systems implemented completely in hardware, such as the ones described in [Akeley93] and [Montrym97]. In the latter case, there is often one subsystem, called the *geometry engine*, which performs both geometry processing and pixel transfer operations, and another subsystem, called the *rasterizer* or *raster manager*, which performs rasterization and per-fragment operations. One advantage of this separation is that the geometry engine is often programmable for increased flexibility, while the raster manager is hardwired for optimal performance. Moreover, the geometry engine requires floating point operations, while the raster manager can be implemented with fixed point arithmetic only. In the following we briefly summarize the four groups of operations.

4.1 Geometry Processing

As stated above, the major task of the geometry processing operations are geometric transformations. Geometric primitives are specified in terms of a set of vertices, each of which has an associated location, normal, and color, as well as texture coordinates. Points and vectors are specified in homogeneous coordinates $[x, y, z, w]^T$. The vertex locations are transformed from the object coordinate system into the viewing coordinate system via an affine model/view transformation, specified as a homogeneous 4×4 matrix stack. The matrix stack is used instead of a single matrix in order to support hierarchical modeling. This is also true for all other transformations in the pipeline. The normal vectors, which are required for example for the lighting calculations, are transformed by the inverse transpose of the same affine transformation.

The next step in the pipeline are the lighting calculations. The types of light sources supported by the hardware are point lights, directional lights and spot lights. A falloff for point and spot lights can be specified in terms of three coefficients a, b, and c defining the attenuation factor $1/(a \cdot r^2 + b \cdot r + c)$ for the intensity, where r is the distance of the vertex from the light source. According to Equation 2.11, the characteristics of a point light are given by $a = 1$ and $b = c = 0$, while the attenuation characteristics of small area light sources can be approximated with $b, c \neq 0$. The positions and directions of these light sources are stored in viewing coordinates, so that they can be directly used with the transformed vertex coordinates and normal vectors.

As a reflection model, current hardware uses either the Phong or the Blinn-Phong model[1] (see Section 3.1). The user can choose whether the material is implicitly generated from the vertex color, or explicitly specified in terms of ambient, diffuse and specular reflection coefficients. The result of the lighting computation is stored as the new vertex color.

After lighting, a projective transformation (again specified as a 4×4 matrix stack) and a perspective division are applied in order to transform the viewing frustum into the unit cube. The reason why the lighting has to be performed before this step is that perspective projection is a non-affine transformation destroying the correspondence between normals and surface points. After this transformation, all geometric primitives are clipped against the unit cube and larger primitives are tessellated into triangles.

Like vertices and normals, texture coordinates are specified as 4D homogeneous vectors $[s, t, r, q]^T$. They can either be specified explicitly, or computed automatically. In the case of automatic generation, each component of the texture coordinate vector can be computed as the distance of the vertex from an arbitrary plane in object coordinates or viewing coordinates. Furthermore, a mode is available for computing s and t as the coordinates pointing into a spherical environment map [Haeberli93], see Chapter 8 for details. The specified or automatically generated texture coordinate vector can then be transformed using a third homogeneous matrix stack.

4.2 Rasterization

After geometry processing, each geometric primitive is defined in terms of vertices and associated colors and texture coordinates. These primitives are then scan-converted and yield *fragments*, that is, preliminary pixels with interpolated depth, color and texture coordinates. While the interpolation of texture coordinates is perspectively correct, colors are usually only interpolated linearly along scan lines (*Gouraud shading*).

The texture coordinates are then used to look up texture values in a 1D, 2D, 3D, or 4D texture.[2] Except for the case of 4D textures, all texture coordinate are divided by q before the lookup in order to allow for projective texturing. For the reconstruction of the texture values, the hardware should support nearest neighbor and linear sampling, as well as mip-mapping [Williams83]. The color resulting from the texture lookup is then combined with the fragment color according to one of several blending modes, including addition and multiplication of the two colors (see [Segal98] for a complete list).

[1] The OpenGL specification [Segal98] mandates the use of the Blinn-Phong model.

[2] While 3D textures have become a standard feature on modern graphics hardware, 4D textures are only available on a few systems. Moreover, high texture memory requirements often prohibit the use of 4D textures even where they would theoretically be useful. For these reasons we will only use them in one of our algorithms (Chapter 10).

4.2.1 Multiple Textures

Some of the algorithms presented here will have improved performance if the hardware supports multiple textures in a single rendering pass [SGI97]. In this case, each vertex has multiple texture coordinate vectors, one for each texture. Each of these can be specified explicitly, or it can be generated automatically as described in Section 4.1. Moreover, there is also a separate texture matrix stack for each of the textures.

The results of the texture lookup are combined with the fragment color as described above. The resulting color then acts as a fragment color for the next texture. Blending modes can be specified for each texture separately. This situation is depicted in Figure 4.2.

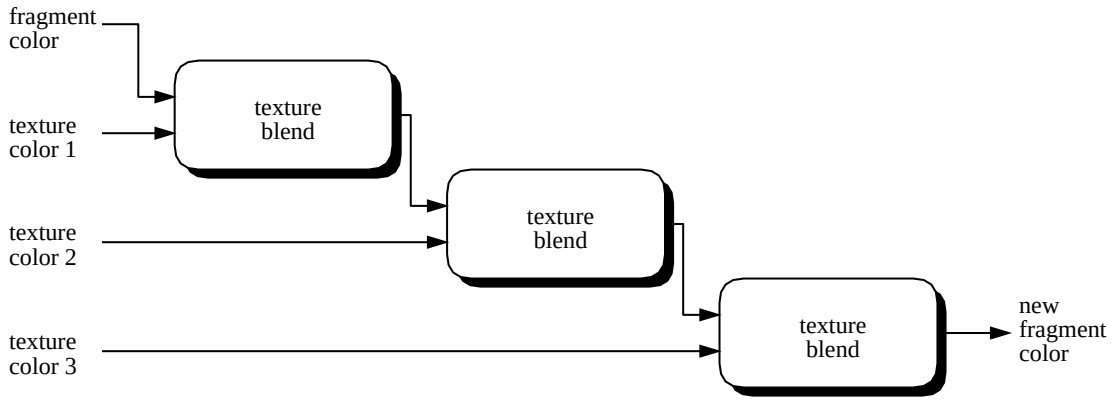

Figure 4.2: With multiple textures, the results from the blending operation act as a fragment color for the next texture.

4.3 Per-Fragment Operations

After texturing, the fragments have to pass a number of tests before being written to the framebuffer. These tests include an *alpha test*, which allows a fragment to be accepted or rejected based on a comparison of its alpha channel with a reference value, a *stencil test*, which is based of a comparison between a reference value and the value of the stencil buffer at the pixel location corresponding to the fragment, and finally a *depth test* between the fragment z value and the depth buffer. These tests are described in [Neider93] and [Segal98], and will not be explained in detail here.

Colors of fragments passing all these tests are then combined with the current contents of the framebuffer and stored there as new pixel values. In the easiest case, the fragment color simply replaces the previous contents of the framebuffer. However, several arithmetic and logic operations between the different buffers are also possible. Again, these are described in [Segal98].

4.4 Framebuffer and Pixel Transfer Operations

Systems with 3D graphics accelerators have three different memory subsystems, which are, at least logically, separated: the main memory of the CPU, framebuffer RAM for storing color, depth, and stencil buffers, and finally the texture RAM. This separation makes it necessary to transfer pixel data, that is, images, between the different memory subsystems. The possible paths for this transfer are depicted in Figure 4.3.

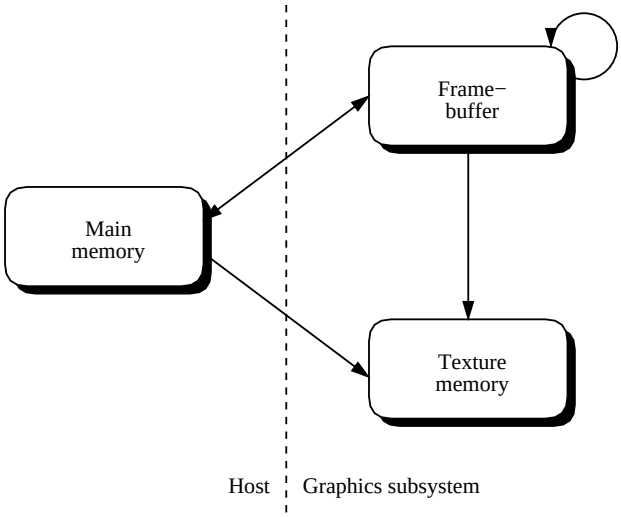

Figure 4.3: Possible paths for transferring pixel data.

Note that it is not possible to transfer pixel data out of the texture RAM (except by actual texture mapping), while parts of the framebuffer can be copied to other regions of the framebuffer. Moreover, the framebuffer can be scaled and added to the *accumulation buffer* [Haeberli90], a separate, deep buffer in the framebuffer memory. This allows one to compute weighted sums of previously rendered images.

In addition to simply moving image data around, pixel transfer operations also include certain transformations on the colors, such as histograms, convolutions, color lookup tables and color matrices. In OpenGL, these operations have been introduced with the so-called *imaging subset* formally defined for version 1.2 of the API.

Of the many features of this subset, we only use color matrices and color lookup tables. A color matrix is a 4×4 matrix that can be applied to any RGBα pixel group during pixel transfer.

During a pixel transfer, separate color lookup tables for each of the four components can be specified, which are applied before and/or after the color matrix. These allow for non-linear transformations of the color components. Scaling and biasing of the components is also possible at each of these stages. For a detailed discussion of these features and the whole imaging subset,

refer to [Segal98].

An additional feature, known as *pixel textures* [SGI96, Hansen97] is available for transfers between main memory and framebuffer, as well as within the framebuffer itself, but not for transfers to the texture memory. This stage interprets the color components R, G, B, and α of each individual pixel as texture coordinates s, t, r, and q, respectively. The result from the texture lookup is then used as the new fragment color. A schematic overview of the pixel transfer operations is given in Figure 4.4.

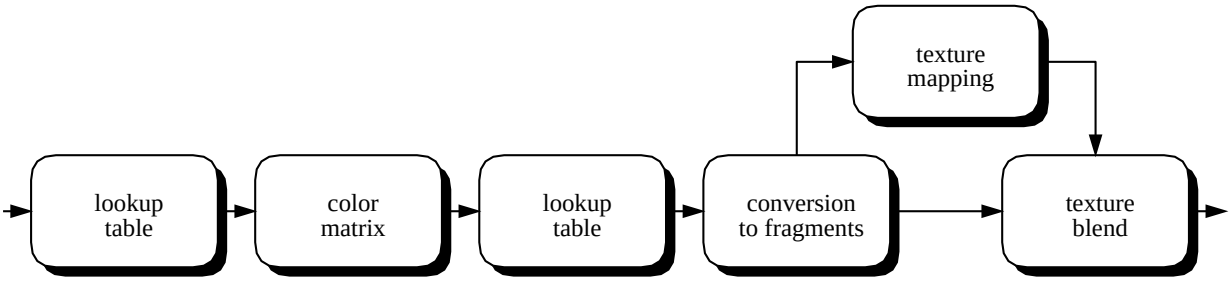

Figure 4.4: Per-fragment operations including imaging subset and pixel textures. Only the operations required in this thesis are shown.

It is important to note that in current implementations of pixel textures, R, G, B, and α are directly used as texture coordinates s, t, r, and q. A division by q does not take place. This means that projective texturing is not possible with pixel textures. As we will show in Chapters 6 and 9, this is a severe limitation that we propose to remove.

4.5 Summary

The combination of features from the pixel transfer operations and the elements of the rendering pipeline (geometry processing, rasterization and per-fragment operations) lays out an abstract system for which we will be designing our algorithms in the following.

This system mostly follows OpenGL and the imaging subset, but also contains some extensions which are not part of the current OpenGL standard. These extensions are 4D textures, pixel textures and the use of multiple textures in a single rendering pass.

Chapter 5

Local Illumination with Alternative Reflection Models

> *To model static* Jell-O® *we employ a new synthesis technique wherein attributes are added one a time using abstract object-oriented classes we call* ingredients. Ingredient *attributes are combined during a preprocessing pass to accumulate the desired set of material properties (consistency, taste, torsional strength, flame resistance, refractive index, etc.). We use the RLS orthogonal basis (raspberry, lime, strawberry), from which any type of* Jell-O® *can be synthesized.*
>
> Paul S. Heckbert, *Ray Tracing* Jell-O® *Brand Gelatin*, Computer Graphics (SIGGRAPH '87 Proceedings), pp 73–74, 1987.

Except for graphics systems supporting procedural shading such as, for example, [Olano98], current implementations of hardware lighting are restricted to the Phong or Blinn-Phong reflection models. It has been known for a long time that these violate the laws of Physics [Blinn77, Lewis93]. Many other, physically more plausible models have been proposed, but have so far only been used in software rendering systems. The most important of these models have been reviewed in Section 3.1.

This chapter introduces multi-pass techniques for using these models for local illumination on contemporary graphics hardware. These methods can be combined with the algorithms for shadows discussed in the next chapter, as well as the realistic light sources from Chapter 7. After presenting the techniques, we will then discuss several hardware extensions that could be used to more directly support the proposed methods in future hardware.

Rather than replacing the standard Phong model by another single, fixed model, we seek a method that allows us to utilize a wide variety of different models so that the most appropriate model can be chosen for each application.

To achieve this flexibility without introducing procedural shading, a sample-based representation of the BRDF seems most promising. However, a faithful sampling of 3D isotropic or 4D anisotropic BRDFs requires too much storage to be useful on contemporary graphics hardware. Wavelets [Lalonde97a, Lalonde97b] or spherical harmonics [Sillion89, Sillion91] could be used to store this data more compactly, but these representations do not easily lend themselves to hardware implementations, since they do not permit efficient point evaluation.

5.1 Isotropic Models

We propose a different approach. It turns out that Torrance-Sparrow style models, as well as many other models used in computer graphics, can be factored into independent components that only depend on one or two angles. Consider the Torrance-Sparrow model as formulated in Equation 3.8

$$f_r(\mathbf{x}, \vec{l} \rightarrow \vec{v}) = \frac{F \cdot G \cdot D}{\pi \cdot < \vec{n}, \vec{l} > \cdot < \vec{n}, \vec{v} >}, \tag{5.1}$$

The geometry is depicted in Figure 5.1:

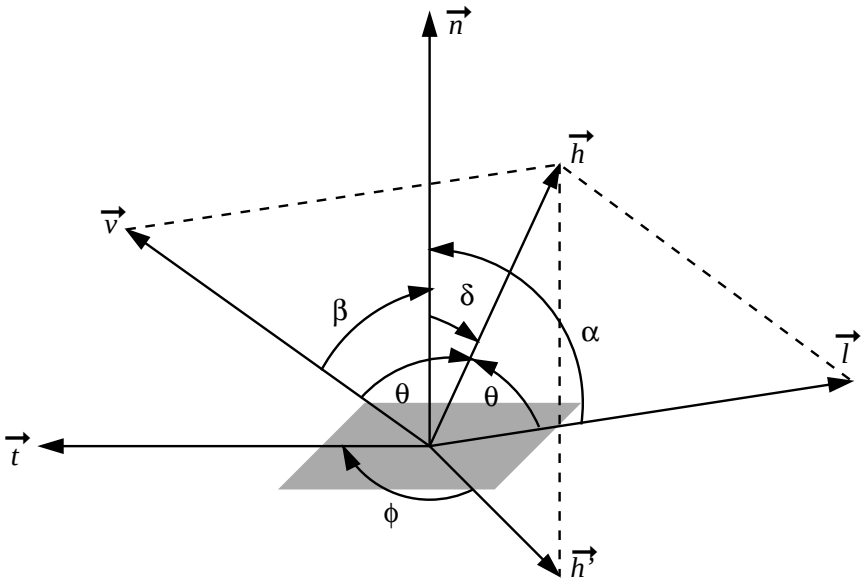

Figure 5.1: Geometric entities for the reflection models in this chapter.

For a fixed index of refraction, the Fresnel term F, which was given in Equation 3.9, only depends on the angle θ between the light direction $\vec{l}$ and the micro facet normal $\vec{h}$, which is the halfway vector between $\vec{l}$ and $\vec{v}$. Thus, the Fresnel term can be seen as a univariate function $F(\cos\theta)$.

The micro facet distribution function D, which defines the percentage of facets oriented in direction $\vec{h}$, depends on the angle δ between $\vec{h}$ and the surface normal $\vec{n}$, as well as a roughness parameter. This is true for all widely used choices of distribution functions, including the Gaussian angle distribution (Equation 3.10), the Gaussian height distribution, and the distribution by [Beckmann63]. Since the roughness is generally assumed to be constant for a given surface, this is again a univariate function $D(\cos\delta)$.

Finally, when using the geometry term G proposed by [Smith67] (Equation 3.12), which describes the shadowing and masking of light for surfaces with a Gaussian micro facet distribution, this term is a bivariate function $G(\cos\alpha, \cos\beta)$.

The contribution of a single point- or directional light source with intensity I_i to the outgoing radiance of the surface is given as $L_o = f_r(\mathbf{x}, \vec{l} \rightarrow \vec{v}) \cos\alpha \cdot g \cdot I_i$ (see Equation 2.11). If the material is assumed to be constant over a surface, the term $f_r(\mathbf{x}, \vec{l} \rightarrow \vec{v}) \cos\alpha$ can be split into two bivariate parts $F(\cos\theta) \cdot D(\cos\delta)$ and $G(\cos\alpha, \cos\beta)/(\pi \cdot \cos\beta)$, which are then stored in two independent 2-dimensional lookup tables.

2D texture mapping is used to implement the lookup process. If all vectors are normalized, the texture coordinates are simple dot products between the surface normal, the viewing and light directions, and the micro facet normal. These vectors and their dot products can be computed in software and assigned as texture coordinates to each vertex of the object. At the same time, the term $g \cdot I_i$ can be specified as a per-vertex color and multiplied with the results from texturing using alpha blending.

For orthographic cameras and directional lights (that is, when light and viewing vectors are assumed constant), the interpolation of these texture coordinates across a polygon corresponds to a linear interpolation of the normal without renormalization. Since the reflection model itself is highly nonlinear, this is much better than simple Gouraud shading, but not as good as evaluating the illumination in every pixel (Phong shading). This interpolation of normals without renormalization is commonly known as *fast Phong shading*.

This method for looking up the illumination in two separate 2-dimensional textures requires either a single rendering pass with two simultaneous textures, or two separate rendering passes with one texture each in order to render specular reflections on an object. If two passes are used, their results are multiplied using alpha blending. A third rendering pass with hardware lighting (or a third simultaneous texture) can be applied for adding a diffuse term, if necessary.

If the light and viewing directions are assumed to be constant, that is, if a directional light and an orthographic camera are assumed, the computation of the texture coordinates can even be

done in hardware. To this end, light and viewing direction as well as the halfway vector between them are used as row vectors in the texture matrix for the two textures:

$$\begin{bmatrix} 0 & 0 & 0 & \cos\theta \\ h_x & h_y & h_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \cos\delta \\ 0 \\ 1 \end{bmatrix} \tag{5.2}$$

$$\begin{bmatrix} l_x & l_y & l_z & 0 \\ v_x & v_y & v_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} n_x \\ n_y \\ n_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha \\ \cos\beta \\ 0 \\ 1 \end{bmatrix} \tag{5.3}$$

Figure 5.2 shows a torus rendered with several different roughness settings using this technique. The assumption of an orthographic camera for lighting purposes is quite common in hardware-accelerated rendering, since it saves the normalization of the viewing vector for each vertex. APIs like OpenGL have a separate mode for applications where this simplification cannot be used, and the viewing direction has to be computed for every vertex. This case is called a *local viewer*.

We would like to note that the use of textures for representing the lighting model introduces an approximation error: while the term $F \cdot D$ is bounded by the interval $[0, 1]$, the second term $G/(\pi \cdot \cos\beta)$ exhibits a singularity for grazing viewing directions ($\cos\beta \to 0$). Since graphics hardware typically uses a fixed-point representation of textures, the texture values are clamped to the range $[0, 1]$. When these clamped values are used for the illumination process, areas around the grazing angles can be rendered too dark, especially if the surface is very shiny. This artifact can be alleviated by dividing the values stored in the texture by a constant which is later multiplied back onto the final result.

The same methods can be applied to all kinds of variations of the Torrance-Sparrow model, using different distribution functions and geometry terms, or the approximations proposed in [Schlick93]. With varying numbers of terms and rendering passes, it is also possible to find similar factorizations for many other models. For example, the Phong, Blinn-Phong and Cosine Lobe models (Section 3.1.2) can all be rendered in a single pass with a single texture, which can even already account for an ambient and a diffuse term in addition to the specular one.

Last but not least, the sampling-based approach also allows for the use of measured or simulated terms. For example, in [Brockelmann66], the shadowing and masking of surfaces with a Gaussian height distribution was simulated. The results could be directly applied as a geometry term with our approach.

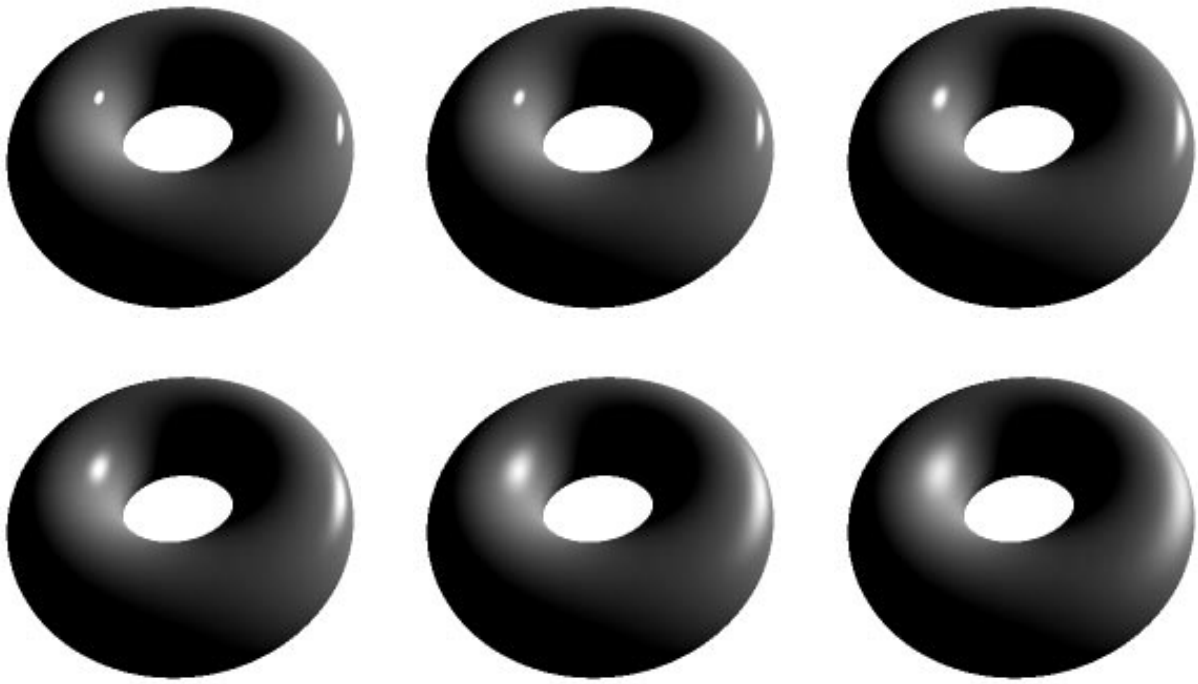

Figure 5.2: A torus rendered with the proposed hardware multi-pass method using the Torrance-Sparrow reflection model (Gaussian height distribution and geometry term by [Smith67]) and different settings for the surface roughness. For these images, the torus was tessellated into 200×200 polygons.

5.2 Anisotropy

Although the treatment of anisotropic materials is somewhat harder, similar factorization techniques can be applied here. For anisotropic models, both the micro facet distribution function and the geometrical attenuation factor may also depend on the angle ϕ between the facet normal and a reference direction in the tangent plane. This reference direction is given in the form of a tangent vector $\vec{t}$ (see Figure 5.1).

For example, the elliptical Gaussian model [Ward92] introduces an anisotropic facet distribution function specified as the product of two independent Gaussian functions, one in the direction of $\vec{t}$, and one in the direction of the binormal $\vec{n} \times \vec{t}$. This makes D a bivariate function in the angles δ and ϕ. Consequently, the texture coordinates can be computed in software in much the same way as described above for isotropic materials. This also holds for many other anisotropic models in computer graphics literature.

Since anisotropic models depend on both a normal and a tangent that varies per vertex, the texture coordinates cannot be generated with the help of a texture matrix, even if light and view-

ing directions are assumed to be constant. This is due to the fact that the anisotropic term can usually not be factored into a term that only depends on the surface normal, and one that only depends on the tangent.

One exception to this rule is the model by Banks [Banks94], which is mentioned here despite the fact that it is an *ad-hoc* model which is not based on physical considerations. The algorithm outlined in the following has been published in [Heidrich98b] and [Heidrich98c]. Banks defines the reflection off an anisotropic surface as

$$L_o = \cos\alpha \cdot (k_d < \vec{n'}, \vec{l} > + k_s < \vec{n'}, \vec{h} >^{1/r}) \cdot L_i, \tag{5.4}$$

where $\vec{n'}$ is the projection of the light vector $\vec{l}$ into the plane perpendicular to the tangent vector $\vec{t}$ (see Figure 3.2). This vector is then used as a shading normal for a Blinn-Phong lighting model with diffuse and specular coefficients k_d and k_s, and surface roughness r. In [Zöckler96] and [Stalling97], it has been pointed out that this Phong term is really only a function of the two angles between the tangent and the light direction, as well as the tangent and the viewing direction. This fact has been used for the illumination of lines in [Stalling97].

Applied to anisotropic reflection models, this means that this Phong term can be looked up from a 2-dimensional texture, if the tangent $\vec{t}$ is specified as a texture coordinate, and the texture matrix is set up as in Equation 5.3. The additional term $\cos\alpha$ in Equation 5.4 is computed by hardware lighting with a directional light source and a purely diffuse material, so that the Banks model can be rendered with one texture and one pass per light source. Figure 5.3 shows a disk and a sphere rendered with this reflection model.

5.3 Hardware Extensions for Alternative Lighting Models

The techniques described in the previous two sections have the disadvantage that the texture coordinates need to be computed in software if point lights or a local viewer are desired. The consequence is, that, depending on the number of polygons and the number of rendering passes, the CPU is typically the bottleneck of the method. For example, on a SGI O2 (195 MHz R10k), the implementation for the Torrance-Sparrow model achieves frame rates of about 20 Hz with a tessellation of 72×72 for the torus in Figure 5.2. This number is almost independent of the resolution, but drops immediately to about 8 frames, if a tessellation of 200×200 polygons is used.

These numbers indicate that some additional hardware support for these methods would be useful. In the following, we first propose a very moderate extension of the graphics pipeline, which allows for the generation of the texture coordinates in hardware. Then, we discuss the

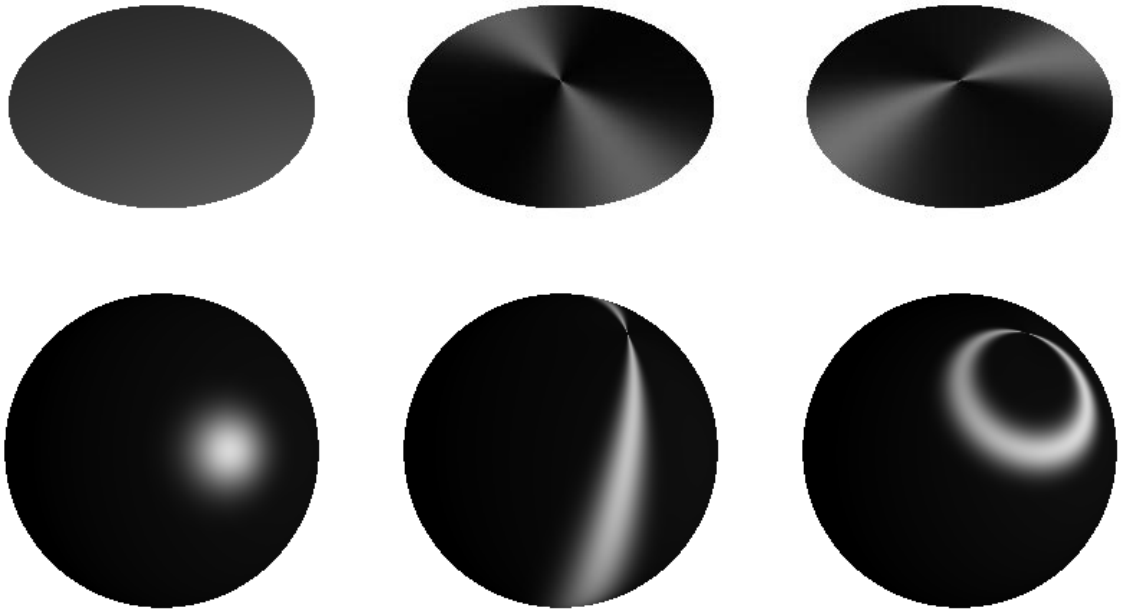

Figure 5.3: Disk and sphere illuminated with isotropic reflection (left), anisotropic reflection with circular features (center), and radial features (right).

opportunities for more ambitious changes, replacing the traditional Phong illumination model by a sampling-based system building on top of the previously discussed methods.

5.3.1 New Modes for Texture Coordinate Generation

The reason why the texture coordinate generation is so expensive when it is performed in software is that all the required vectors (light and viewing direction, as well as surface and micro facet normal) have to be transformed into a common coordinate system before the dot products can be evaluated. To this end, it is either necessary to transform the eye point and the light position (or direction, in the case of a parallel light) into object space, or the surface point and normal into eye space. Both transformations require access to the current model/view matrix, and need to be performed in software (since this is where the results are required). After the transformation, the light and viewing vectors as well as their halfway vector can be computed, and then the dot products for the texture coordinates can be calculated.

However, all four vectors are also required for evaluating the regular Blinn-Phong illumination model in hardware. Since light positions and directions are stored in eye coordinates (Section 4.1), the hardware uses the model/view stack to transform the surface point and normal

into the same space. This means that, by computing the texture coordinates in software, as described above, the application is forced to perform operations the hardware is already capable of doing.

Moreover, the automatic texture generation mechanism of the hardware is located *after* these transformations in the rendering pipeline (see Section 4.1 and [Segal98]). This leads us to the following proposal: in order to directly support a wide variety of sampling-based reflection models, a series of new modes for automatic texture coordinate generation should be added. More specifically, it should be possible to let the hardware compute any of the dot products between the following four normalized vectors in eye space: light direction, viewing direction, the halfway vector between the two, and finally the surface normal.

This proposal introduces a total of six new modes, five of which require an additional parameter, namely the index of the OpenGL light source for which the light direction and the halfway vector are to be computed. All modes introduce only very little computational overhead in addition to what is already done for Phong lighting.

With these modifications, the techniques for isotropic models from Section 5.1 can be implemented very efficiently. Likely, other reflection models developed in the future will also only depend on these six parameters. The only condition for applying the same techniques to another model is that it should be possible to factorize the model into terms that only depend on one or two of these angles. Otherwise, 3-dimensional, or even 4-dimensional textures need to be applied, which in principle is possible, but might not be feasible due to the amount of texture memory required.

As pointed out in Section 5.1, anisotropic models also require an additional tangent vector to be specified per vertex. To support them in hardware, another modification is therefore necessary: instead of only being able to specify a normal per vertex, it should also be possible to have a per-vertex tangent, which is transformed using the hardware model/view stack. This, combined with another four texture generation modes for computing the dot product of the tangent with any of the other vectors, would then allow for the use of a wide variety of anisotropic reflection models. The cost of introducing such an additional tangent is moderate: the bandwidth from the CPU to the graphics board is increased by four floating point values per vertex (in addition to the eight required for point and normal, as well as the bandwidth for colors and texture coordinates), and an additional vector-matrix multiplication is required (in addition to the two already needed for transforming point and normal).

Some anisotropic models also require the binormal $\vec{b} := \vec{n} \times \vec{t}$. Although it could prove useful to also add that vector to the rendering pipeline, this is not strictly necessary, since the angle between $\vec{b}$ and any other vector can be expressed as a function of the angles between that vector and $\vec{n}$ and $\vec{t}$. Through reparameterization, this function can be rolled into the texture maps representing the factors of the reflection model.

5.3.2 A Flexible Per-Vertex Lighting Model

The introduction of a tangent vector and the additional modes for texture coordinate generation allow for the efficient use of complex lighting models for a single light source (if multiple simultaneous textures are supported by the hardware). With more than one light source, the contribution from each light has to be rendered in a separate pass. However, since the material of an object remains the same for all light sources, the textures for each pass are the same. Also, the normal and viewing direction, as well as the tangent for anisotropic materials, remain the same in each pass. Only the light direction $\vec{l}$ and the halfway vector $\vec{h}$ change for each light source.

To exploit this fact, and also to reduce the cost for geometry processing and rasterization, we propose a new sampling-based lighting model for hardware rendering, which is designed to replace the existing Phong or Blinn-Phong lighting. A schematic overview of this model is depicted in Figure 5.4.

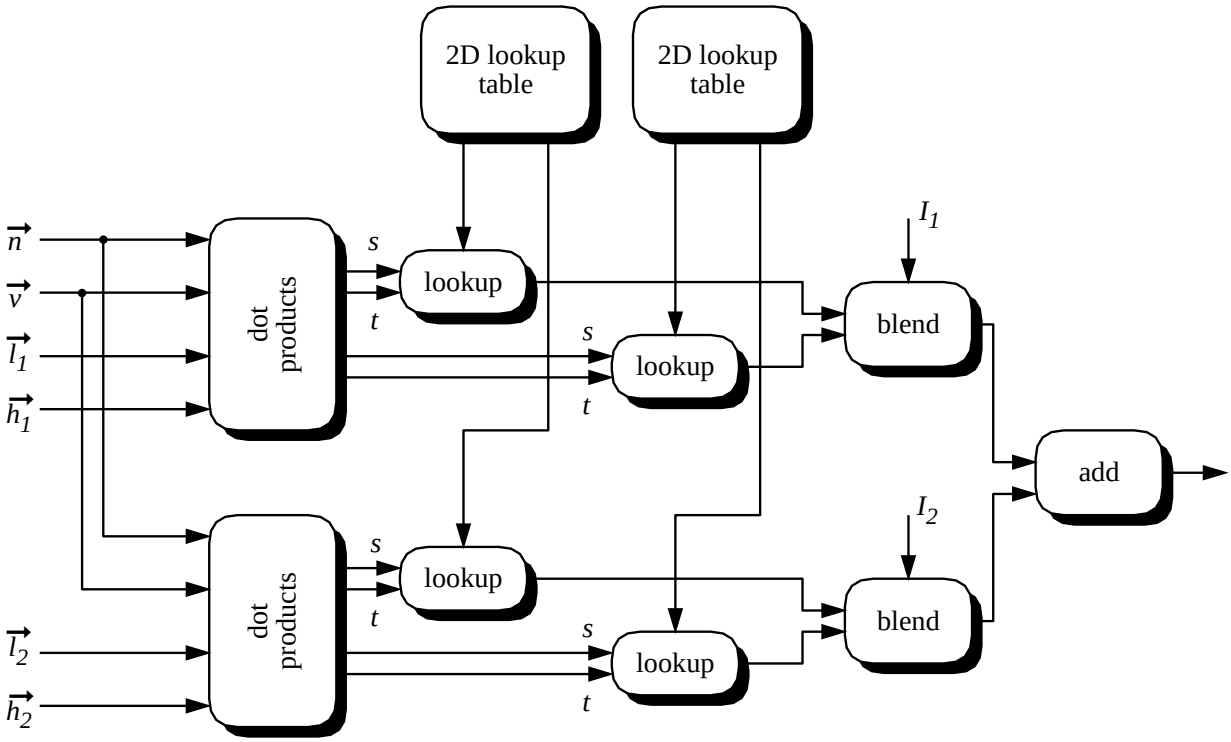

Figure 5.4: A sampling based lighting model with two light sources.

As in the previous discussion, material properties of an object are specified in terms of a number of 2-dimensional lookup tables (two in Figure 5.4). Moreover, as described in the previous section, the indices for this lookup can be any of the dot products of the vectors $\vec{n}$, $\vec{v}$, $\vec{l}_i$ and $\vec{h}_i$, and, in the case of anisotropy, $\vec{t}$. The results from the table lookups for a single light source are combined using a set of blending operations, most notably addition and multiplication. During

this blending operation, the result should also be multiplied with the intensity of the respective light source divided by the square distance (this is also an operation that already has to be performed in current hardware). Finally, the illumination contributions from all light sources are summed up to yield the final vertex color.

As mentioned in Section 5.1, the Blinn-Phong model can be implemented using a single 2-dimensional lookup table. The required dot products for each light source are $< \vec{n}, \vec{h}_i >$ for the specular, and $< \vec{n}, \vec{l}_i >$ for the diffuse part. Thus, all vectors depicted in Figure 5.4 are required. The computation of the cosine power in traditional hardware implementations is replaced by a 2D lookup. The lookup is certainly not much slower than computing the power; in some implementations it might even improve the performance. It is even possible that some hardware already uses 1D tables for implementing the cosine power. Thus, with the proposed model, Blinn-Phong lighting is possible at approximately the same computational cost as in traditional implementations.

With other, more complicated reflection models that require more than one lookup table, the cost increases only moderately. Each additional term requires the computation of two dot products for the coordinates s and t, one table lookup, and a blending operation. These costs are typically smaller than the cost for normalizing the vectors $\vec{v}$, $\vec{l}_i$ and $\vec{h}_i$ (the normal and the tangent do not need to be normalized if a rigid body transformation is used and the vectors are normalized in object coordinates), so that the total illumination cost using two tables will be significantly less than double the cost for a single table. The only disadvantage is that very frequent changes of the material, which require loading different lookup tables, will cause an increased bandwidth.

5.4 Discussion

In this chapter we have introduced a new, sampling based approach for local illumination with graphics hardware. Using contemporary graphics systems, this method achieves high frame rates for a few moderately tessellated objects.

Since the implementation on contemporary hardware requires the software to compute per-vertex texture coordinates for each frame, and possibly for multiple passes per frame, the performance for a large number of highly tessellated objects is not sufficient for real-time applications. To overcome this problem, we have proposed two levels of hardware extensions for direct support of our sampling-based lighting model.

The first, easier to implement version defines a set of new texture coordinate generation modes, as well as a per-vertex tangent vector to be transformed with the current model/view matrix. This variant would already significantly lower the CPU load, since computations that are performed in contemporary hardware anyway, can efficiently be reused to generate the texture coordinates.

The second version, which requires more fundamental changes to the pipeline, replaces the traditional Phong model with a flexible sample-based model. We have argued that the computational cost for our model is not significantly higher than the cost of the Phong model (although more gates will be required on the chip in order to implement the lookup tables). This second version has the advantage of being able to compute the illumination from several light sources in one pass.

Chapter 6

Shadows

After discussing models for local illumination in the previous chapter, we now turn to global effects. In this chapter we deal with algorithms for generating shadows in hardware-based renderings.

Shadows are probably the visually most important global effect. This fact has resulted in a lot of research on how to generate them in hardware-based systems. Thus, interactive shadows are in principle a solved problem. However, current graphics hardware rarely directly supports shadows, and, as a consequence, fewer applications than one might expect actually use the developed methods.

We first review the most important of the existing methods, and then describe an implementation of shadow maps based on the hardware features listed in Chapter 4. This implementation is efficient, and does not require any specific shadow mapping hardware beyond what is already available on most graphics boards, even in the low end. Therefore, it integrates nicely with the traditional rendering pipeline.

6.1 Projected Geometry

Besides precomputed shadow textures, projected geometry [Blinn88] is currently the most often used implementation of shadows in interactive applications. This method assumes that a set of small occluders casts shadows onto a few large, flat objects. Since this assumption does not hold for most scenes, applications using this approach typically do not render all the shadows in a scene, but only some of the visually most important. In particular, shadows of concave objects onto themselves are typically not handled.

The method works as follows. Given a point or directional light source, the shadowed region of a planar receiver can be computed by projecting the geometry of the occluder onto the receiver (see Figure 6.1). The projection is achieved by applying a specific model/view transformation to

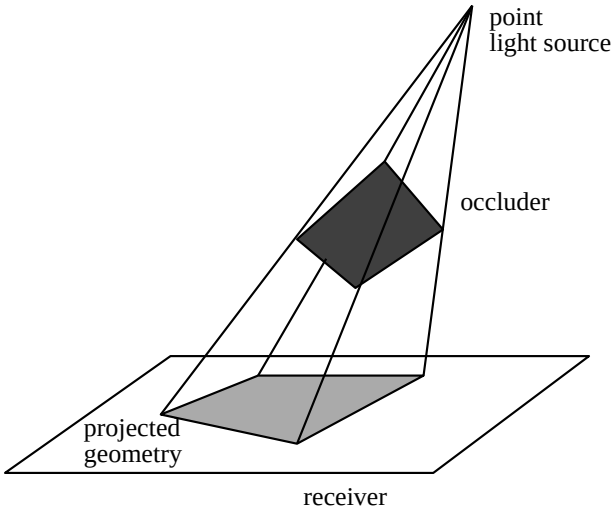

Figure 6.1: One way of implementing shadows cast onto large, planar objects is to project the geometry onto the surface.

the original surface geometry. This transformation is given as follows:

$$\mathbf{S} = \mathbf{V} \cdot \mathbf{P} \cdot \mathbf{M}, \tag{6.1}$$

where $\mathbf{V}$ is the viewing transformation, $\mathbf{M}$ is the modeling transformation, and $\mathbf{P}$ is the projection matrix that would be used for rendering the scene from the light source position with the receiver as an image plane.

The occluder is then rendered twice, once with the regular model/view matrix $\mathbf{V} \cdot \mathbf{M}$ and the regular colors and textures, and once with the matrix $\mathbf{S}$ and the color of the shadow.

All kinds of variations on this basic algorithm are possible. For example, if parts of the projected geometry fall outside the receiver, a stencil buffer can be used to clamp it to the receiving polygon. Also, instead of simply drawing the shadow in black or dark gray, alpha blending can be used to modulate the brightness of the receiver in the shadow regions. Although this is not actually physically accurate, it looks more realistic, since the texture of the underlying surface remains visible.

As pointed out above, projected geometry is only appropriate for a small number of large, planar receiving objects. As the number of receivers grows, the method quickly becomes infeasible due to the large number of required rendering passes. Thus, this method can hardly be called a general solution to the shadowing problem; rather it simply adds shadows as a special effect in areas where they have the most visual impact.

6.2 Shadow Volumes

The second implementation of shadows are shadow volumes [Crow77], which relies on a polyhedral representation of the spatial region that a given object shadows (see Figure 6.2).

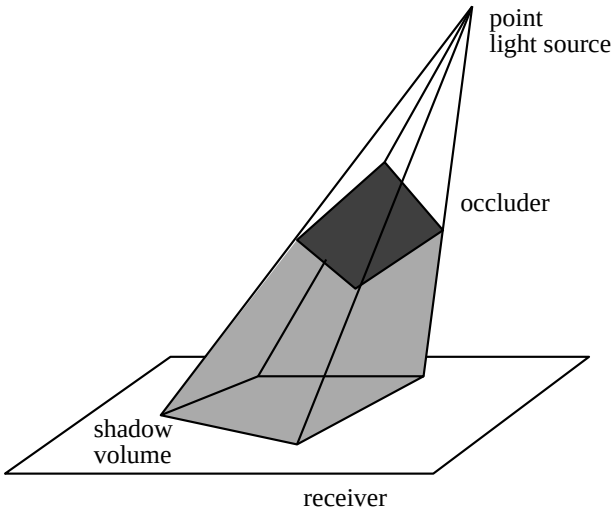

Figure 6.2: The shadow volume approach uses a polyhedral representation of the spatial region shadowed by an occluder.

This polyhedron is generated in a preprocessing step by projecting each silhouette edge of the object away from the light source. With graphics hardware and a z-buffer, the shadowing algorithm then works as follows [Brotman84, Diefenbach94, Diefenbach96]: First, the geometry is rendered without the contribution of the point light casting the shadow, and the stencil buffer is cleared. Then, without clearing the z-buffer, the front-facing polygons of all shadow volumes are rendered without actually drawing to the color buffers. Each time a pixel of a shadow volume passes the depth test, the corresponding entry in the stencil buffer is incremented. Then, the backfacing regions of the shadow volume are rendered in a similar fashion, but this time, the stencil buffer entry is decremented. Afterwards, the stencil buffer is zero for lit regions, and larger than zero for shadowed regions. A final conditional rendering pass adds the illumination for the lit regions.

Many details have to be solved to make this algorithm work in practice. For example, if the eye point lies inside a shadow volume, the meaning of the stencil bits is inverted. Even worse, if the near plane of the perspective projection penetrates one of the shadow volume boundaries, there are some areas on the screen where the meaning of the stencil bits is inverted, and some for which it is not.

In [Diefenbach94] and [Diefenbach96], this algorithm was used for rendering the complete set of shadows of a rather complex scene. However, the rendering times were far from interactive.

Although today's graphics hardware is faster than the one used by Diefenbach, shadow volumes are typically still too costly to be applied to a complete scene. Another issue is the size of the data structures required for the shadow volumes, which can exceed several hundred megabytes [Diefenbach96]. The use of simplified geometry for generating the shadow volumes can help to reduce these problems. Nonetheless, most interactive applications and games only apply shadow volumes to a subset of the scene geometry, much in the same way projected geometry is used. Even then, regeneration of the shadow volumes for moving light sources is a costly operation.

6.3 Shadow Maps

In contrast to the analytic shadow volume approach, shadow maps [Williams78] are a sampling-based method. First, the scene is rendered from the position of the light source, using a virtual image plane (see Figure 6.3). The depth image stored in the z-buffer is then used to test whether a point is in shadow or not.

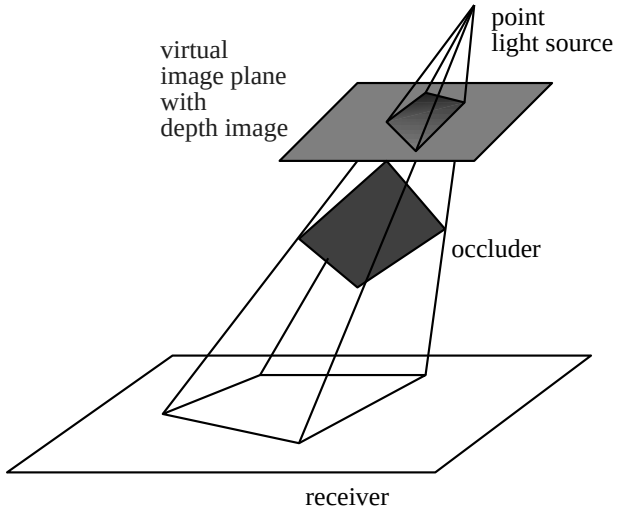

Figure 6.3: Shadow maps use the z-buffer of an image of the scene rendered from the light source.

To this end, each fragment as seen from the camera needs to be projected onto the depth image of the light source. If the distance of the fragment to the light source is equal to the depth stored for the respective pixel, then the fragment is lit. If the fragment is further away, is is in shadow.

A hardware multi-pass implementation of this principle has been proposed in [Segal92]. The first step is the acquisition of the shadow map by rendering the scene from the light source position. For walkthroughs, this is a preprocessing step, for dynamic scenes it needs to be performed each frame. Then, for each frame, the scene is rendered without the illumination contribution

from the light source. In a second rendering pass, the shadow map is specified as a projective texture, and a specific hardware extension is used to map each pixel into the local coordinate space of the light source and perform the depth comparison. Pixels passing this depth test are marked in the stencil buffer. Finally, as in Section 6.1, the illumination contribution of the light source is added to the lit regions by a third rendering pass.

The advantage of the shadow map algorithm is that it is a general method for computing all shadows in the scene, and that it is very fast, since the representation of the shadows is independent of the scene complexity. On the down side, there are artifacts due to the discrete sampling and the quantization of the depth. One benefit of the shadow map algorithm is that the rendering quality scales with the available hardware. The method could be implemented on fairly low end systems, but for high end systems a higher resolution or deeper z-buffer could be chosen, so that the quality increases with the available texture memory. Unfortunately, the necessary hardware extensions to perform the depth comparison on a per-fragment basis are currently only available on two high-end systems, the RealityEngine [Akeley93] and the InfiniteReality [Montrym97].

6.3.1 Shadow Maps Using the Alpha Test

Instead of relying on a dedicated shadow map extension, it is also possible to use projective textures and the alpha test. Basically, this method is similar to the method described in [Segal92], but it efficiently takes advantage of automatic texture coordinate generation and the alpha test to generate shadow masks on a per-pixel basis. This method takes one rendering pass more than required with the appropriate hardware extension.

In contrast to traditional shadow maps, which use the contents of a z-buffer for the depth comparison, we use a depth map with a *linear* mapping of the z values in light source coordinates. This allows us to compute the depth values via automatic texture coordinate generation instead of a per-pixel division. Moreover, this choice improves the quality of the depth comparison, because the depth range is sampled uniformly, while a z-buffer represents close points with higher accuracy than far points.

As before, the entire scene is rendered from the light source position in a first pass. Automatic texture coordinate generation is used to set the texture coordinate of each vertex to the depth as seen from the light source, and a 1-dimensional texture is used to define a linear mapping of this depth to alpha values. Since the alpha values are restricted to the range $[0 \ldots 1]$, near and far planes have to be selected, whose depths are then mapped to alpha values 0 and 1, respectively. The result of this is an image in which the red, green, and blue channels have arbitrary values, but the alpha channel stores the depth information of the scene as seen from the light source. This image can later be used as a texture.

For all object points visible from the camera, the shadow map algorithm now requires a comparison of the point's depth with respect to the light source with the corresponding depth

value from the shadow map. The first of these two values can be obtained by applying the same 1-dimensional texture that was used for generating the shadow map. The second value is obtained simply by using the shadow map as a projective texture. In order to compare the two values, we can subtract them from each other, and compare the result to zero.

With multi-texturing, this comparison can be implemented in a single rendering pass. Both the 1-dimensional texture and the shadow map are specified as simultaneous textures, and the texture blending function is used to implement the difference. The resulting α value is 0 at each fragment that is lit by the light source, and > 0 for fragments that are shadowed. Then, an alpha test is employed to compare the results to zero. Pixels passing the alpha test are marked in the stencil buffer, so that the lit regions can then be rendered in a final rendering pass.

Without support for multi-texturing, the same algorithm is much more expensive. First, two separate passes are required for applying the texture maps, and alpha blending is used for the difference. Now, the framebuffer contains an α value of 0 at each pixel that is lit by the light source, and > 0 for shadowed pixels. In the next step it is then necessary to set α to 1 for all the shadowed pixels. This will allow us to render the lit geometry, and simply multiply each fragment by $1 - \alpha$ of the corresponding pixel in the framebuffer (the value of $1 - \alpha$ would be 0 for shadowed and 1 for lit regions). In order to do this, we have to copy the framebuffer onto itself, thereby scaling α by 2^n, where n is the number of bits in the α channel. This ensures that $1/2^n$, the smallest value > 0, will be mapped to 1. Due to the automatic clamping to the interval $[0 \ldots 1]$, all larger values will also be mapped to 1, while zero values remain zero. In addition to requiring an expensive framebuffer copy, this algorithm also needs an alpha channel in the framebuffer ("destination alpha"), which might not be available on some systems.

Figure 6.4 shows an engine block where the shadow regions have been determined using this approach. Since the scene is rendered at least three times for every frame (four times if the light source or any of the objects move), the rendering times for this method strongly depend on the complexity of the visible geometry in every frame, but not at all on the complexity of the geometry casting the shadows. Scenes of moderate complexity can be rendered at high frame rates even on low end systems. The images in Figure 6.4 are actually the results of texture-based volume rendering using 3D texturing hardware (see [Westermann98] for the details of the illumination process). The frame rates for this data set were roughly 15 Hz for an Octane MXE.

6.4 Discussion

In this chapter, we have reviewed algorithms for generating shadows of point and directional light sources with graphics hardware, and presented an efficient way of implementing shadow maps through a linear alpha coding of the depth values. All the methods described in this chapter can be extended to soft shadows cast by area light sources. This involves a Monte Carlo sampling

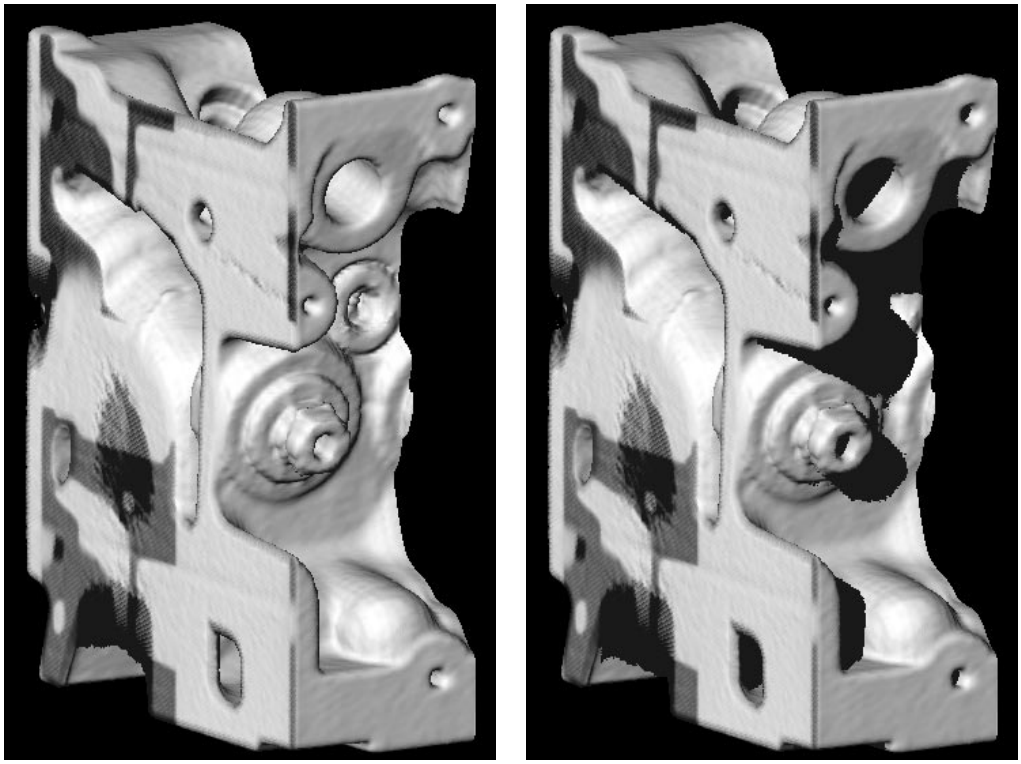

Figure 6.4: An engine block generated from a volume data set with and without shadows. The shadows have been computed with our algorithm for alpha-coded shadow maps. The Phong reflection model is used for the unshadowed parts.

of the light source surface by randomly placing a number of point lights on it. The contribution from each of the point lights can then be rendered with any of the described methods, and the resulting images are blended together using an accumulation buffer.

It is possible to combine any of the shadow algorithms with the local illumination models from the previous chapter. Since the determination of the shadowed pixels and the actual illumination of the lit pixels happen in separate rendering passes, any reflection model can be applied.

An alternative algorithm for implementing shadow maps, based on pixel textures, has been proposed in [Heidrich99e]. It also codes the depth information into the alpha channel, but uses pixel textures to perform the depth comparison. Even without multiple texture support, this approach only uses a total of 3 rendering passes for the geometry, but it requires copying the framebuffer twice. Besides the fact that pixel textures are required, which are not available on most platforms today, this method also has a few other drawbacks, which is why it is not explained in full detail here: since pixel textures do not currently support projective textures, the algorithm is restricted to orthographic cameras and parallel light sources. A variation of that algorithm, recently proposed by Sloan [Sloan99], eliminates one of the framebuffer copies and

works for projective images, but not for point lights. If projective pixel textures were available, point lights would also work, both for the original algorithm published in [Heidrich99e], and for the variation by Sloan. Another application of projective pixel textures is given in Chapter 9.

The dynamic range of the alpha channel used for the depth comparison in our algorithm is critical. With a 12 bit alpha channel, the quantization artifacts were reasonable, which is also demonstrated by Figure 6.4. Nonetheless, a deeper alpha buffer and deeper texture formats would be useful to further improve the rendering quality. In future graphics hardware, it would be appropriate to have special framebuffer configurations in which the alpha channel is particularly deep. For example, instead of splitting 48 bits per pixel into 12 bits per channel, it would, for our algorithm, be better to have only 8 bits each for red, green, and blue, but 24 bits for alpha.

For the future, we believe that the sampling-based shadow map algorithm has advantages over the analytic methods for hardware implementations. This is because the analytic methods suffer from the dependency on the scene complexity, while a sampling-based representation is independent of scene complexity in terms of storage requirements and computational cost. Moreover, shadow maps allow for improving the quality, simply by adding more texture memory. This shift towards sampling-based methods has a precedent in the area of visibility algorithms. For interactive applications, today the z-buffer is used almost exclusively for hidden surface removal. Analytic methods are mostly applied when a device- and resolution independent output is seeked, for example for printing.

Chapter 7

Complex Light Sources

After introducing shadows as the first global lighting effect, we now turn to an extended model for light sources based on light fields. Complex light sources can greatly contribute to the realism of computer generated images, and are thus interesting for a variety of applications. However, in order to correctly simulate the indirect light bouncing off internal parts of a lamp, it has been necessary to apply expensive global illumination algorithms.

As a consequence, it has so far not been possible to use realistic light sources in interactive applications. For offline techniques such as ray-tracing, complex light source geometry unnecessarily slows down the rendering, since reflections and refractions between internal parts of the light source have to be recomputed for every instance of a light source geometry in every frame.

In this chapter we propose a method for applying realistic light sources in interactive image synthesis. For a given lamp geometry and luminary, the outgoing light field is precomputed using standard global illumination methods, and stored away in a light field data structure. Later the light field can be used to illuminate a given scene while abstracting from the original lamp geometry. We call a light source stored and used in this fashion a “canned light source”.

Instead of simulating the light field with global illumination methods, canned light sources could also be measured [Ashdown93], or even provided by lamp manufacturers in much the same way farfield information is provided today. Thus, a database of luminaries and lamps stored as canned light sources becomes possible.

Our method also speeds up the rendering process by factoring out the computation of the internal reflections and refractions of the light source into a separate preprocessing step. As a consequence, realistic light sources can be efficiently used for interactive rendering, and for other applications where a complete global illumination solution would be too expensive. In this chapter, we will focus on the application of canned light sources in hardware accelerated rendering. A discussion of techniques for ray-tracing has been previously published by the author in [Heidrich98a].

7.1 Simulating and Measuring Light Sources

As mentioned above, we store the canned light source in a light field data structure. From this data structure radiance values can be extracted efficiently. Light Fields can be generated easily using existing rendering systems or real world images. Since they are merely a two dimensional array of projective images, they lend themselves to efficient implementations using computer graphics hardware (see Section 7.2.2).

The left side of Figure 7.1 shows the geometry of a single light slab. Every grid point corresponds to a sample location. As in [Gortler96] the (u, v) plane is the plane close to the object, in our case a lamp, whereas the (s, t) plane is further away, potentially at infinity.

Placing the (s, t) plane at infinity offers the advantage of a clear separation between spatial sampling on the (u, v) plane, and directional sampling on the (s, t) plane. In this situation, all rays through a single sample on the (u, v) plane, that is, the projective image through that point, represent the farfield of the light source. The higher the resolution of the (u, v) plane, the more nearfield information is added to the light field.

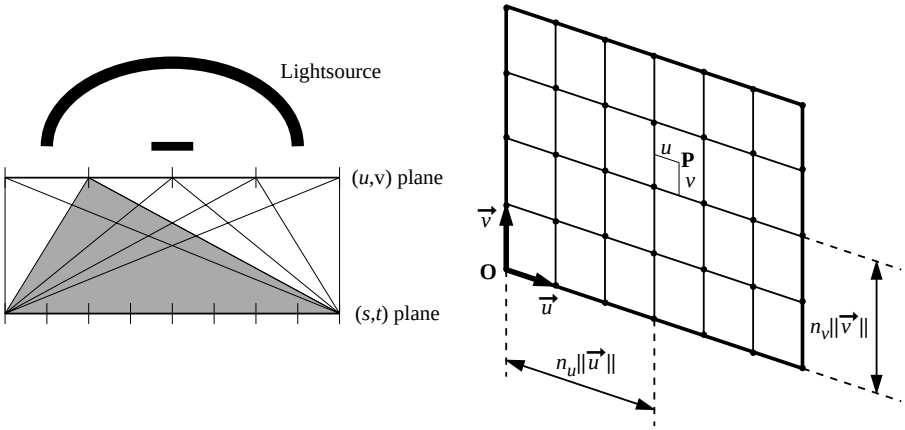

Figure 7.1: Geometry of a single light slab within a light field.

The resolutions used for spatial and directional information of course depend on the application domain of the specific light source. For example, for car headlights the nearfield information is usually negligible, yet it becomes quite important for lamps used in interior design. However, for almost all applications the distance from the light will be relatively large compared to the dimensions of the lamp. Therefore, the spatial resolution will usually be much smaller than the directional resolution, and thus the resolution of the (u, v) plane will be significantly lower than on the (s, t) plane.

If the (s, t) plane is *not* placed at infinity, it is not possible to clearly distinguish between spatial and directional resolution. Nonetheless, we can still view the light slab as an array of projective images with centers of projection on the (u, v) plane.

Since a light slab representing a canned light source is merely a 2D array of projective images, we can create it by first generating a global illumination solution for the lamp geometry, and then rendering it from multiple points of view. For the global illumination step, we can use any kind of algorithm, such as Radiosity, Monte-Carlo ray-tracing, Photon Maps, or composite methods [Slusallek98]. Alternatively, canned light sources could be generated by resampling measured data [Ashdown93] much in the same way as described in [Gortler96]. Finally, it is also possible to generate non-physical light sources for special effects.

7.2 Reconstruction of Illumination from Light Fields

Given one or more canned light sources, we can use them to illuminate a scene by reconstructing the radiance emitted by the light source at every point and in every direction. As in [Gortler96] and [Levoy96], we use bilinear or quadrilinear interpolation between the samples stored in the light slabs. For a ray-tracer, we could simply obtain a large number of radiance samples from the light field. However, the challenge is to obtain a good reconstruction with the smallest number of samples possible. On the other hand, it is important not to miss any narrow radiance peaks, because this would lead to artifacts in the reconstructed illumination.

7.2.1 High-quality Reference Solutions

In order to describe the illumination in a scene caused by a canned light source, we view a light slab as a collection of $N_u \times N_v \times N_s \times N_t$ independent little area light sources ("micro lights"), each corresponding to one of the light slab's 4D grid cells. Here, N_u, N_v, N_s, and N_t are the resolutions of the slab in each parametric direction. Each of the micro lights causes a radiance of

$$L_o^{n_u,n_v,n_s,n_t}(\mathbf{x},\omega_o) = \int_{\Omega(\vec{n})} f_r(\mathbf{x},\omega_i \rightarrow \omega_o) L_i^{n_u,n_v,n_s,n_t}(\mathbf{x},\omega_i)\cos\theta \quad d\omega_i \tag{7.1}$$

to be reflected off any given surface point $\mathbf{x}$. The total reflected radiance caused by the canned light source is then the sum of the contributions of every micro light. $f_r(\mathbf{x},\omega_i \rightarrow \omega_o)$ is the BRDF of the surface, θ is the angle between the surface normal and the direction ω_i, and $L_i^{n_u,n_v,n_s,n_t}(\mathbf{x},\omega_i)$ is the radiance emitted from the micro light towards $\mathbf{x}$ along direction ω_i.

Since $L_i^{n_u,n_v,n_s,n_t}(\mathbf{x},\omega_i)$ is obtained by quadrilinear interpolation of 16 radiance values, Equation 7.1 can be rewritten in a more suitable form. Consider the geometry of a light slab as shown on the right side of Figure 7.1. Every point $\mathbf{P}$ on the (u,v) plane can be written as

$$\mathbf{P} = \mathbf{O} + (n_u + u)\vec{u} + (n_v + v)\vec{v},$$

where the vectors $\vec{u}$ and $\vec{v}$ determine the dimensions of a 2D grid cell, and $u, v \in [0, 1]$ describe the coordinates of $\mathbf{P}$ within the cell. In analogy we can describe a point $\mathbf{P}'$ on the (s, t) plane, where we assume that the vectors $\vec{s}$ and $\vec{t}$ are parallel to $\vec{u}$ and $\vec{v}$, respectively.

For each point $\mathbf{x}$ and each micro light (n_u, n_v, n_s, n_t), we can now determine the region on the (u, v) plane that is *visible* from point $\mathbf{x}$. This is achieved by projecting the 2D cell (n_s, n_t) onto the (u, v) plane, using $\mathbf{x}$ as the center of projection (see Figure 7.2). The resulting rectangle is clipped against the 2D cell (n_u, n_v).

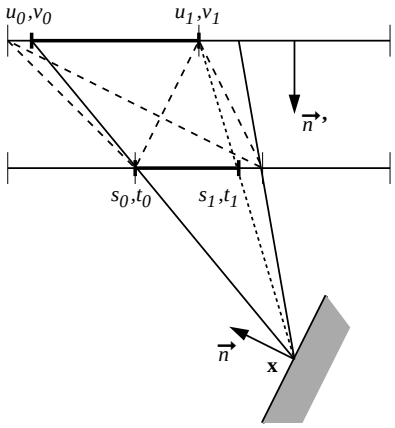

Figure 7.2: Clipping a light field grid cell. $u_{\{0,1\}}$, $v_{\{0,1\}}$, $s_{\{0,1\}}$, and $t_{\{0,1\}}$ describe the portion of the cell that is visible from point $\mathbf{x}$. In this area, every ray $r(u, v, s, t)$ passing through $\mathbf{x}$ is characterized by $u = (1 - x)u_0 + xu_1$, $v = (1 - y)v_0 + yv_1$, $s = (1 - x)s_0 + xs_1$, and $t = (1 - y)t_0 + yt_1$ for some $x, y \in [0, 1]$. The dashed lines indicate points and directions for which the radiance values are stored in the light field.

This yields boundaries $u_{\{0,1\}}$, $v_{\{0,1\}}$, $s_{\{0,1\}}$, and $t_{\{0,1\}}$ relative to the 4D cell boundaries of the micro light. In this area, every ray $r(u, v, s, t)$ passing through $\mathbf{x}$ is characterized by $u = (1 - x)u_0 + xu_1$, $v = (1 - y)v_0 + yv_1$, $s = (1 - x)s_0 + xs_1$, and $t = (1 - y)t_0 + yt_1$ for some $x, y \in [0, 1]$. The values x and y can be used for the quadrilinear interpolation of the radiance emitted towards point $\mathbf{x}$ by inserting the formulas for u, v, s, and t into the following equation:

$$\begin{aligned} L_i^{n_u,n_v,n_s,n_t}(x, y) =& (1 - u) \cdot (1 - v) \cdot (1 - s) \cdot (1 - t) \cdot L^{0000} + \\ & (1 - u) \cdot (1 - v) \cdot (1 - s) \cdot t \cdot L^{0001} + \\ & \cdots + \\ & u \cdot v \cdot s \cdot t \cdot L^{1111}. \end{aligned} \tag{7.2}$$

$L^{0000}, L^{0001}, \ldots, L^{1111}$ are the 16 radiance values at the corners of the 4D grid cell. These are contained in the light field. With this result, we can now rewrite Equation 7.1 as follows:

$$L_o^{n_u,n_v,n_s,n_t}(\mathbf{x},\omega_o) = \int_A f_r(\mathbf{x},(\mathbf{P}-\mathbf{x})\rightarrow\omega_o) L_i^{n_u,n_v,n_s,n_t}(\mathbf{x},\mathbf{P}\rightarrow\mathbf{x}) \frac{\cos\theta\cos\theta'}{r^2} d\mathbf{P} \qquad (7.3)$$
$$= A\cdot\int_0^1\int_0^1 f_r(\mathbf{x},(\mathbf{P}-\mathbf{x})\rightarrow\omega_o) L_i^{n_u,n_v,n_s,n_t}(x,y) \frac{\cos\theta\cos\theta'}{r^2} dxdy,$$

where $A=(u_1-u_0)||\vec{u}||\cdot(v_1-v_0)||\vec{v}||$ is the size of the visible region on the (u,v) plane, and $r=||\mathbf{P}-\mathbf{x}||$ is the distance of the two points $\mathbf{P}$ and $\mathbf{x}$. Note that the vectors $\vec{s}$ and $\vec{t}$ do not occur in this equation. That is, the exact position and parameterization of the (s,t) plane is only required for clipping. The use of an (s,t) plane at infinity is transparent in Equation 7.3.

While an analytic solution of this integral exists, it is too complicated for practical use. However, the integrand in Equation 7.3 is quite smooth, so that Monte Carlo integration performs well, and only a few samples are required to obtain good results.

In order to compute the complete illumination in $\mathbf{x}$, we have to sum up the contribution of all micro lights. Although there are $N_u\times N_v\times N_s\times N_t$ micro lights, only $O(\max(N_u,N_s)\cdot\max(N_v,N_t))$ have a non-zero visible area from any point $\mathbf{x}$. These visible areas are easily determined since 2D grids on the two light field planes are aligned. Thus, $u_{\{0,1\}}$ and $s_{\{0,1\}}$ only depend on the grid *column* (n_u,n_s), while $v_{\{0,1\}}$ and $t_{\{0,1\}}$ only depend on the grid *row* (n_v,n_t).

The result of this method is that we know how many samples are at least required for faithfully computing the illumination in a given point $\mathbf{x}$: at least one sample is necessary for each visible micro light. Figure 7.3 shows an image generated via ray casting, using a canned light source resembling a slide projector. The focal plane of this slide projector, which is defined through the (s,t)-plane of the canned light source, is approximately located at the projection screen. There, the image is very sharp, while it is somewhat blurred on the wall behind the screen. The chair in front of the screen is so far away from the focal plane that no fine detail is visible at all – all that can be seen is that the chair is bright. This image was generated with the Vision rendering system [Slusallek95, Slusallek98], using some speed-ups of the above method, which are described elsewhere [Heidrich98a]. The image, at a resolution of 600×600, took about 30 minutes to render on a 195 MHz R10k. This is a very good result given that any other global illumination method would have serious problems rendering this effect at all.

7.2.2 Hardware Reconstruction

In addition to ray-tracing, it is also possible to use computer graphics hardware for reconstructing the illumination from a canned light source. Our method can be combined with one of the algorithms for generating shadows, as discussed in Chapter 6.

For hardware-based rendering, we assume that the BRDF $f_r(\mathbf{x},(\mathbf{P}-\mathbf{x})\rightarrow\omega_o)$ used in Equation 7.3 corresponds to the Phong model, which is the hardware lighting model. We numerically

Figure 7.3: A ray-traced image with a canned light source resembling a slide projector.

integrate Equation 7.3 by evaluating the integrand at the grid points on the (u, v) plane. For these points, the quadrilinear interpolation reduces to a bilinear interpolation on the (s, t) plane, which can be done in hardware.

[Segal92] describes how projective textures can be used to simulate light sources such as high-quality spotlights and slide projectors. Our method is an extension of this approach, which also accounts for the quadratic falloff and the cosine terms in Equation 7.3. To see how this works, we rewrite the integrand from this equation in the following form:

$$f_r(\mathbf{x}, (\mathbf{P} - \mathbf{x}) \rightarrow \omega_o) \cos\theta \cdot A \frac{\cos\theta'}{r^2} \cdot L_i^{n_u, n_v, n_s, n_t}(x, y) \tag{7.4}$$

The first factor of this formula is given by the Phong material and the surface normal $\vec{n}$ (see Figure 7.2). The second factor is the illumination from a spot light with intensity A, located at $\mathbf{P}$ and pointing towards $\vec{n}'$ with a cutoff angle of $90°$, a spot exponent of 1, and quadratic falloff. Finally, the third factor is the texture value looked up from the slice of the light slab corresponding to point $\mathbf{P}$ on the (u, v) plane.

In other words, the integrand of Equation 7.4 can be evaluated by combining projective textures as in [Segal92] with the illumination from a spotlight using the standard hardware lighting model. The results from texturing and lighting are then multiplied together using texture blending.

The integral from Equation 7.3 is approximated by the sum of the contributions from all

$N_u \times N_v$ grid points on the (u, v) plane. The illumination from each grid point has to be rendered in a separate pass. We compute the sum by adding multiple images, each rendered with a different spotlight position and projective texture. To this end we use either alpha blending or, if alpha is already used for other purposes, an accumulation buffer [Haeberli90].

It should be noted that it is also possible to use spotlight positions other than the (u, v) grid points, if the corresponding projective texture is previously generated by bilinearly interpolating the images of the adjacent grid points. Although hardware can again be used for this task, it is questionable whether the additional cost is worth while.

The top row of Figure 7.4 shows two images rendered with the proposed method and a canned light source of resolution $8 \times 8 \times 64 \times 64$. The canned light source represents the simulated light field of a point source located in the focal point of a parabolic reflector. A grid of 3×3 polygons is located in front of the reflector, as depicted in the center of Figure 7.4. The scene, which in this case consists of 3 large polygons subdivided into smaller rectangles, had to be rendered $8 \times 8 = 64$ times, once for each u and v grid point. For a 400×400 image, the program runs at about 8 fps. on an SGI O2, 18 fps. on an Octane MXE and > 20 fps. on an Onyx2 BaseReality. In this example, pixel fill rate is the limiting factor on both platforms. In general, the total rendering time strongly depends on the complexity of the scene and the number of pixels rendered in each pass. An implementation for more complex geometries should therefore employ some sort of view frustum culling that restricts the rendered geometry for each pass to the actual frustum covered by the projective texture. For comparison, the bottom left image in Figure 7.4 shows the Utah teapot rendered directly as cubic Bézier surfaces (OpenGL evaluators). The Onyx2 BaseReality achieved a frame rate of 1 Hz for these two images, although the resolution of the canned light source was the same as for the other scene. The bottom right image shows a scene similar to the ray-traced image from Figure 7.3, which can be rendered at $10 - 12$ frames per second on an SGI Octane.

7.2.3 Other Material Models and Shadows

With this technique, a canned light source is represented as a 2-dimensional array of projective textures. Each individual texture defines a spot light as described in [Segal92], and is modulated with the result from local illumination using hardware lighting. This is what is often called a *light map* in the PC graphics world. The above derivations show that canned light sources can be implemented as an array of these light maps.

Instead of using the Phong illumination model performed by the hardware, it is also possible to apply one of the reflection models from Chapter 5. Then, however, more passes are required for each projective texture. For example, for the Torrance-Sparrow model, we first require three rendering passes that compute the specular and diffuse reflection components for a isotropic point light, as described in Section 5.1. The result is then multiplied in a fourth pass by the

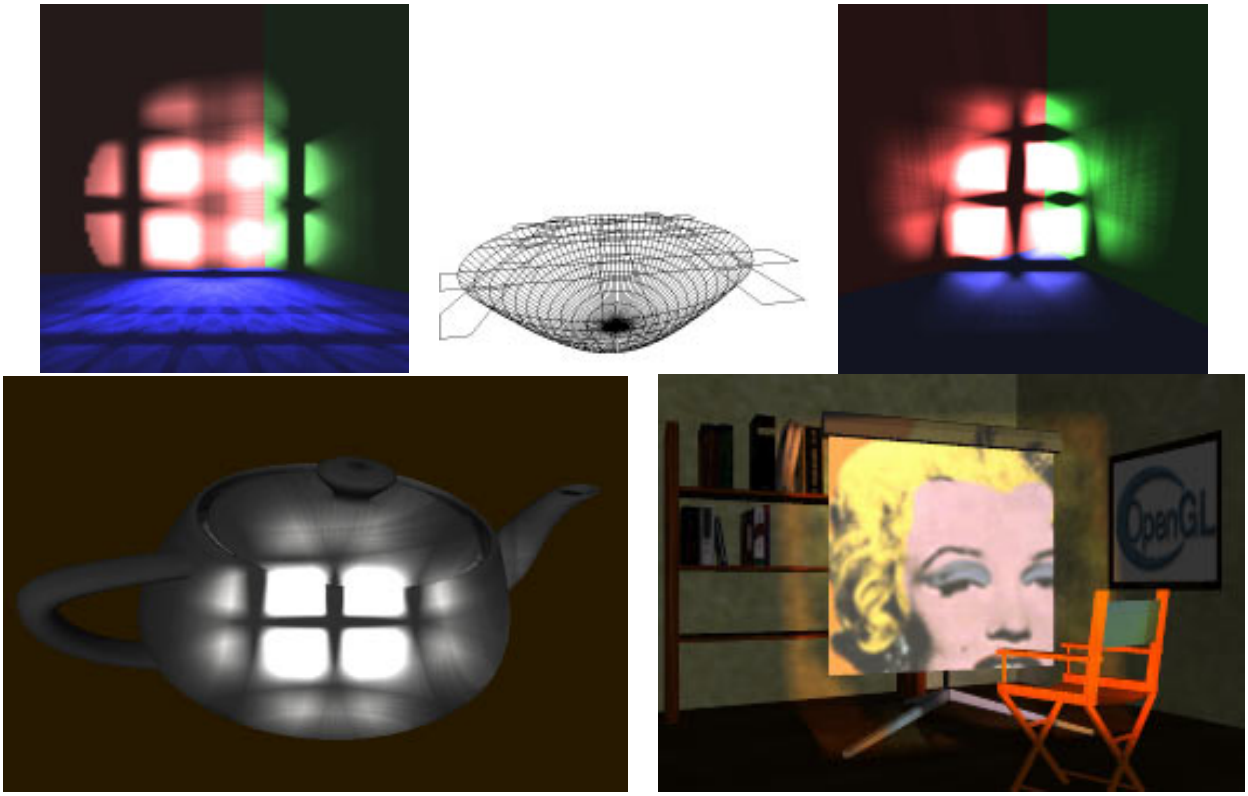

Figure 7.4: Images rendered with a canned light source and hardware texture mapping. Top row, center: light source geometry from which the canned light source for three of the images was generated through simulation. Bottom row, right: OpenGL rendering of a scene similar to Figure 7.3.

result of the projective texturing. The contributions of all projective textures are accumulated using an accumulation buffer. Thus, canned light sources with a Torrance-Sparrow reflection model require a total of $N_u \times N_v \times 4$ passes. With multi-texturing, or the hardware modifications proposed in Section 5.3.2, this can again be done in $N_u \times N_v$ passes, as above.

The combination with shadow mapping techniques as discussed in Chapter 6 is also possible, again through the use of additional rendering passes. As in Chapter 6, the shadowed pixels have to be masked in a separate first pass, and the illumination must only be applied to the lit pixels.

7.3 Discussion

In this chapter we have introduced the notion of a canned light source, which is a precomputed, discretely sampled version of the light field emitted by a complex light source or lamp. This information is stored away in a light field data structure. Later, the precomputed information

can be used to illuminate a scene while abstracting from the lamp geometry. Thus, canned light sources act as blackboxes representing complex light sources.

Canned light sources can be used to create databases of light sources based on simulation or measurement. Ideally, this information could be directly distributed by the lamp manufacturers in much the same way farfield information is provided today.

The precomputed or measured information can be used as a realistic light source for hardware-accelerated rendering. In order to achieve interactive frame rates on contemporary hardware, the (u, v) resolution of the light field has, however, to be relatively small, since a total of $n_u \times n_v$ rendering passes are required. Nonetheless, canned light sources offer an efficient, in many cases interactive, way of visualizing the illumination from complex luminaries. For use in highly interactive applications, such as games, the method is currently too slow. In applications involving interior design, or the development of new light sources, this method could be an interesting way to provide a fast previewing mechanism.

Chapter 8

Environment Mapping Techniques for Reflections and Refractions

With the results from the previous three chapters, it is now possible to render objects with a wide variety of different materials, using arbitrary light sources and shadowing effects. Although the light sources can, to some degree, already be the result of a global illumination solution as mentioned in Chapter 7, this is only part of the complete global illumination problem, which involves the interreflection of light in the complete scene. In this chapter, we now examine environment mapping techniques for interactively visualizing such global illumination solutions. We do *not* use the hardware to compute these solutions directly, but we employ it to generate interactive walkthroughs of static scenes with non-diffuse, curved objects using precomputed global illumination solutions in the form of environment maps.

The basic idea of environment maps is striking [Blinn76]: if a reflecting object is small compared to its distance from the environment, the incoming illumination on the surface really only depends on the direction of the reflected ray. Its origin, that is the actual position on the surface, can be neglected. Therefore, the incoming illumination at the object can be precomputed and stored in a 2-dimensional texture map.

If the parameterization for this texture map is cleverly chosen, the illumination for reflections off the surface can be looked up very efficiently. Of course, the assumption of a small object compared to the environment often does not hold, but environment maps are a good compromise between rendering quality and the need to store the full 4-dimensional radiance field on the surface (see Chapter 10).

Both offline [Hanrahan90, Pixar89] and interactive, hardware-based renderers [Segal98] have used this approach to simulate mirror reflections, often with amazing results.

In this chapter, we first discuss the issue of parameterizations for environment mapping. It turns out that the parameterizations used today are either not appropriate for use from arbitrary

viewing directions, or pose difficulties for hardware implementations. We then develop a new parameterization that is both view independent and easy to implement on current and future hardware. Parts of these ideas have been published in [Heidrich98d]. Following this discussion, we present techniques for using environment maps for global illumination visualization. This includes a more flexible reflection model for use with view independent environment maps as well as prefiltering for glossy reflection [Heidrich99d].

8.1 Parameterizations for Environment Maps

Given the above description of environment maps, one would think that it should be possible to use a single map for all viewing positions and directions. After all, the environment map is supposed to contain information about illumination from *all* directions. Thus, it should be possible to modify the lookup process in order to extract the correct information for all possible points of view.

In reality, however, this is not quite true. The parameterization used in most of today's graphics hardware exhibits a singularity as well as areas of extremely poor sampling. As a consequence, this form of environment map cannot be used for any viewing direction except the one for which it was originally generated.

The parameterization used most commonly in computer graphics hardware today is the *spherical environment map* [Haeberli93]. It is based on the analogy of a small, perfectly mirroring metal ball centered around the object. The reason why this ball should be made out of metal instead of, for example, glass, can be found in the Fresnel formulae (Equation 2.7). Whereas the reflectivity of materials with relatively small indices of refraction, such as glass, varies strongly with the direction of the incoming light, it is almost constant for materials with a high index of refraction, which is typical for metals. The image that an orthographic camera sees when looking at such a ball from a certain viewing direction is the environment map. An example environment map from the center of a colored cube is shown in Figure 8.1.

The sampling rate of spherical maps reaches its maximum for directions opposing the viewing direction (that is, objects behind the viewer), and goes towards zero for directions close to the viewing direction. Moreover, there is a singularity in the viewing direction, since all points where the viewing vector is tangential to the sphere show the same point of the environment.

With these properties, it is clear that this parameterization is not suitable for viewing directions other than the original one. Maps using this parameterization have to be regenerated for each change of the view point, even if the environment is otherwise static.

The major reason why spherical maps are used anyway is that the lookup can be computed efficiently with simple operations in hardware (see Figure 8.2 for the geometry): for each vertex compute the reflection vector $\vec{r}$ of the per-vertex viewing direction $\vec{v}$. A spherical environment

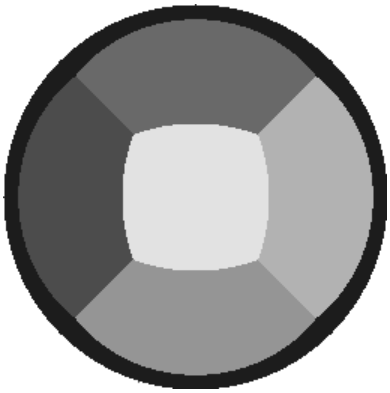

Figure 8.1: A spherical environment map from the center of a colored cube. Note the bad sampling of the cube face directly in front of the observer (black).

map which has been generated for an orthographic camera pointing into direction $\vec{v}_o$, stores the corresponding radiance information for this direction at the point where the reflective sphere has the normal $\vec{h} := (\vec{v}_o + \vec{r})/||\vec{v}_o + \vec{r}||$. If $\vec{v}_o$ is the negative z-axis in viewing coordinates, then the 2D texture coordinates are simply the x and y components of the normalized halfway vector $\vec{h}$. For environment mapping on a per-vertex basis, these texture coordinates are automatically computed by the automatic texture coordinate generation mechanism. For orthographic cameras, the halfway vector simplifies to the surface normal.

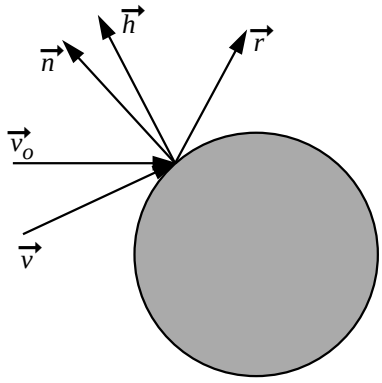

Figure 8.2: The lookup process in a spherical environment map.

Another parameterization is the *latitude-longitude map* [Pixar89]. Here, the s, and t parameters of the texture map are interpreted as the latitude and longitude with respect to a certain viewing direction. Apart from the fact that these maps are severely oversampled around the poles, the lookup process involves the computation of inverse trigonometric functions, which is inappropriate for hardware implementations.

Finally, *cubical environment maps* [Greene86, Voorhies94] consist of six independent per-

spective images from the center of a cube through each of its faces. The sampling of these maps is fairly good, as the sampling rates for all directions differ by a factor of $3\sqrt{3} \approx 5.2$. However, although hardware implementations of this parameterization have been proposed [Voorhies94], these are not available at the moment.

The reason for this is probably that the handling of six independent textures poses problems, and that anti-aliasing across the image borders is difficult. The lookup process within each of the six images in a cubical map is inexpensive. However, the difficulty is to decide which of the six images to use for the lookup. This requires several conditional jumps, and interpolation of texture coordinates is difficult for polygons containing vertices in more than one image. Because of these problems, cubical environment maps are difficult and expensive to implement in hardware, although they are quite widespread in software renderers (e.g. [Pixar89]).

Many interactive systems initially obtain the illumination as a cubical environment map, and then resample this information into a spherical environment map. There are two ways this can be done. The first is to rerender the cubical map for every frame, so that the cube is always aligned with the current viewing direction. Of course this is slow if the environment contains complex geometry. The other method is to generate the cubical map only once, and then recompute the mapping from the cubical to the spherical map for each frame. This, however, makes the resampling step more expensive, and can lead to numerical problems around the singularity.

In both cases, the resampling can be performed as a multi-pass algorithm in hardware, using morphing and texture mapping. However, the bandwidth imposed onto the graphics system by this method is quite large: the six textures from the cubical representation have to be loaded into texture memory, and the resulting image has to be transferred from the framebuffer into texture RAM, which is a slow operation on most hardware.

8.2 A View-independent Parameterization

The parameterization we use is based on an analogy similar to the one used to describe spherical environment maps. Assume that the reflecting object lies at the origin, and that the viewing direction is along the negative z axis. The image seen by an orthographic camera when looking at a metallic, reflecting paraboloid

$$f(x, y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2), \quad x^2 + y^2 \leq 1, \tag{8.1}$$

contains the information about the hemisphere facing towards the viewer. The complete environment is stored in two separate textures, each containing the information of one hemisphere. The geometry is depicted in Figure 8.3.

This parameterization has recently been introduced in [Nayar97] and [Nayar98] in a different context. Nayar actually built a lens and camera system that is capable of capturing this sort of

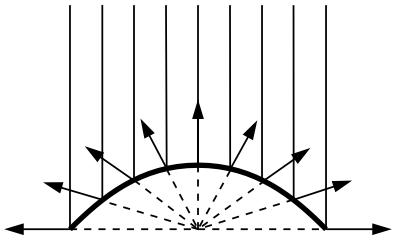

Figure 8.3: The rays of an orthographic camera reflected off a paraboloid sample a complete hemisphere of directions.

image from the real world. Besides ray-tracing and resampling of cubical environment maps, this is actually one way of acquiring maps in the proposed format. Since two of these cameras can be attached back to back [Nayar98], it is possible to create full $360°$ images of real world scenes.

The geometry described above has some interesting properties. Firstly, the reflected rays in each point of the paraboloid all originate from a single point, the focal point of the paraboloid, which is also the origin of the coordinate system (see dashed lines in Figure 8.3). This means that the resulting image can indeed be used as an environment map for an object in the origin. Spherical environment maps do not have this property; the metal spheres used there have to be assumed small.

Secondly, the sampling rate of a parabolic map varies by a factor of 4 over the complete image. This can be shown easily through the following considerations. Firstly, a point on the paraboloid is given as $\vec{f} = (x, y, \frac{1}{2} - \frac{1}{2}(x^2 + y^2))^T$. This is also the vector from the origin to the point on the paraboloid, and the reflection vector $\vec{r}$ for a ray arriving in this point from the viewing direction (see Figure 8.3). If the viewing ray has a differential area dA (which corresponds to the pixel area), then the reflected ray also has this area. The solid angle covered by the pixel is thus the projection of dA onto the unit sphere:

$$\omega(x, y) = \frac{dA}{||\vec{f}(x, y)||^2} \cdot sr. \tag{8.2}$$

Since all pixels have the same area dA, and $\vec{f}(0, 0) = 1/2$, the sampling rate for $x = 0$ and $y = 0$, that is, for reflection rays $\vec{r} = (0, 0, 1)^T$ pointing back into the direction of the viewer, is $\omega_r = 4sr/m^2 \cdot dA$. Thus, the change in sampling rate over the hemisphere can be expressed as

$$\frac{\omega(x, y)}{\omega_r} = \frac{1}{4||\vec{f}(x, y)||^2} = \frac{1}{4(x^2 + y^2 + (\frac{1}{2} - \frac{1}{2}(x^2 + y^2))^2)} \tag{8.3}$$

Pixels in the outer regions of the map cover only $1/4$ of the solid angle covered by center pixels. This means that directions perpendicular to the viewing direction are sampled at a higher

rate than directions parallel to the viewing direction. Depending on how we select mip-map levels, the factor of 4 in the sampling rate corresponds to one or two levels difference, which is quite acceptable. In particular this is somewhat better than the sampling of cubical environment maps. The sampling rates for different parameterizations are compared in Figure 8.4.

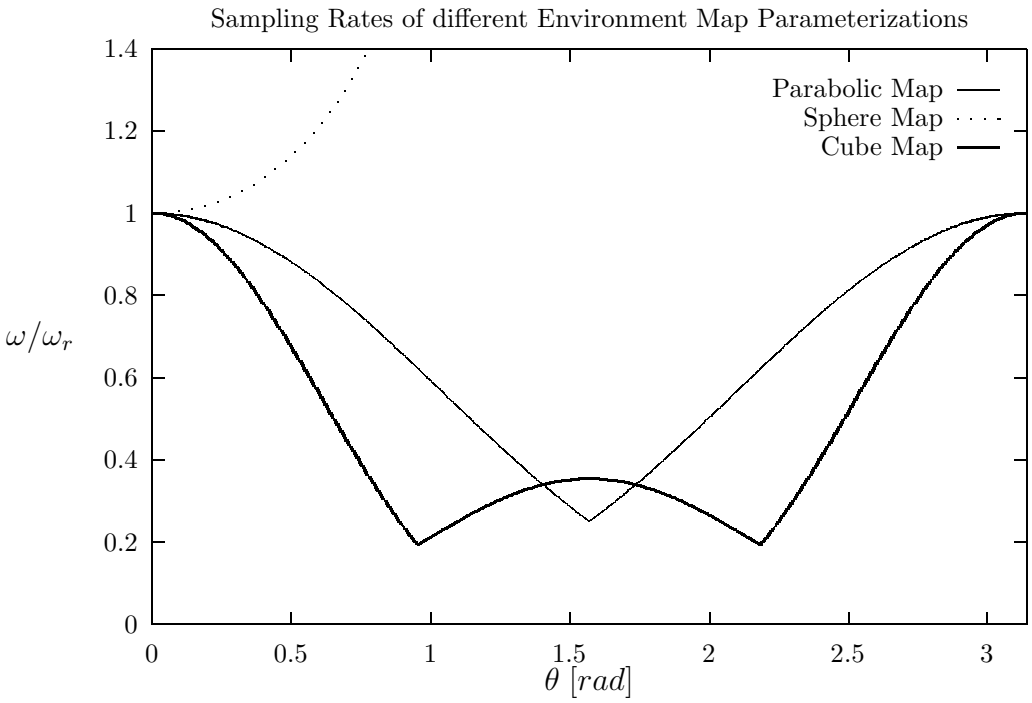

Figure 8.4: The change of solid angle ω/ω_r covered by a single pixel versus the angle θ between the negative z-axis and the reflected ray. ω_r is the solid angle covered by pixels showing objects directly behind the viewer. The parabolic parameterization proposed here gives the most uniform sampling.

Figure 8.5 shows the two images comprising a parabolic environment map for the simple scene used in Figure 8.1. The left image represents the hemisphere facing towards the camera, while the right image represents the hemisphere facing away from it.

8.2.1 Lookups from Arbitrary Viewing Positions

In the following, we describe the math behind the lookup process of a reflection value for an arbitrary viewing position and -direction. We assume that environment maps are specified relative to a coordinate system in which the reflecting object lies at the origin, and the map is generated

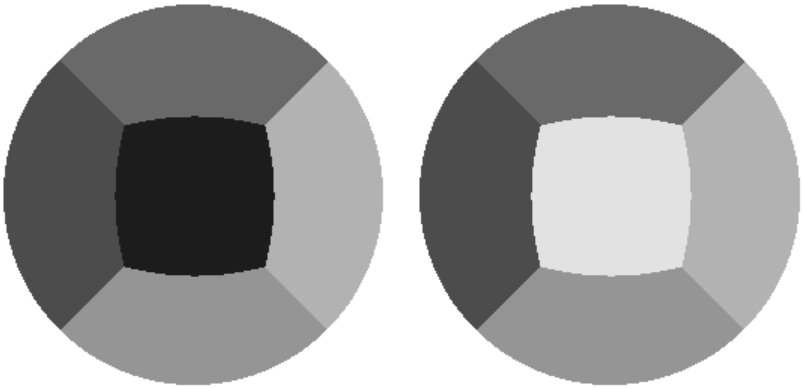

Figure 8.5: The two textures comprising an environment map for an object in the center of a colored cube.

for a viewing direction (i.e. vector from the object point to the eye point) of $\vec{d_o} = (0, 0, 1)^T$. It is not necessary that this coordinate system represents the object space of the reflecting object, although this would be an obvious choice. However, it is important that the transformation between this space and eye space is a rigid body transformation, as this means that vectors do not have to be normalized after transformation. To simplify the notation, we will in the following use the term "object space" for this space.

In the following, $\vec{v}_e$ denotes the normalized vector from the point on the surface to the eye point in eye space, while the vector $\vec{n}_e = (n_{e,x}, n_{e,y}, n_{e,z})^T$ is the normal of the surface point in eye space. Furthermore, the (affine) model/view matrix is given as $\mathbf{M}$. This means, that the normal in eye space $\vec{n}_e$ is really the transformation $\mathbf{M}^{-T} \cdot \vec{n}_o$ of some normal vector in object space. If $\mathbf{M}$ is a rigid body transformation, and $\vec{n}_o$ is normalized, then so is $\vec{n}_e$. The reflection vector in eye space is then given as

$$\vec{r}_e = 2 < \vec{n}_e, \vec{v}_e > \vec{n}_e - \vec{v}_e. \tag{8.4}$$

Transforming this vector with the inverse of $\mathbf{M}$ yields the reflection vector in object space:

$$\vec{r}_o = \mathbf{M}^{-1} \cdot \vec{r}_e. \tag{8.5}$$

The illumination for this vector in object space is stored somewhere in one of the two images. More specifically, if the z component of this vector is positive, the vector is facing towards the viewer, and thus the value is in the first texture image, otherwise it can be found in the second. Let us, for the moment, consider the first case.

$\vec{r}_o$ is the reflection of the constant vector $\vec{d_o} = (0, 0, 1)^T$ at some point (x, y, z) on the paraboloid:

$$\vec{r_o} = 2 < \vec{n}, \vec{d_o} > \vec{n} - \vec{d_o}, \tag{8.6}$$

where $\vec{n}$ is the normal at that point of the paraboloid. Due to the formula of the paraboloid from Equation 8.1, this normal happens to be

$$\vec{n} = \frac{1}{\sqrt{x^2 + y^2 + 1}} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \frac{1}{z} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \tag{8.7}$$

The simplicity of this formula is the major reason that the parabolic parameterization can be easily implemented in hardware. It means that an unambiguous representative for the normal direction can be computed by dividing a (not necessarily normalized) normal vector by its z component, which can be implemented as a perspective division. Another way to disambiguate the normal direction would be to normalize the vector, which involves the more expensive computation of an inverse square root. This is the approach taken by spherical maps. The combination of Equations 8.6 and 8.7 yields

$$\vec{d_o} + \vec{r_o} = 2 < \vec{n}, \vec{v} > \vec{n} = \begin{pmatrix} k \cdot x \\ k \cdot y \\ k \end{pmatrix}. \tag{8.8}$$

for some value k.

In summary, this means that x and y, which can be directly mapped to texture coordinates, can be computed by calculating the reflection vector in eye space (Equation 8.4), transforming it back into object space (Equation 8.5), adding it to the (constant) vector $\vec{d_o}$ (Equation 8.8), and finally dividing by the z component of the resulting vector.

The second case, where the z component of the reflection vector in object space is negative, can be handled similarly, except that $-\vec{d}$ has to be used in Equation 8.8, and that the resulting values are $-x$ and $-y$.

8.2.2 Implementation Using Graphics Hardware

An interesting observation in the above equations is that almost all the required operations are linear. There are two exceptions. The first is the calculation of the reflection vector in eye space (Equation 8.4), which is quadratic in the components of the normal vector $\vec{n}_e$. The second exception is the division at the end, which can, however, be implemented as a perspective divide.

Given the reflection vector $\vec{r}_e$ in eye coordinates, the transformations for the frontfacing part of the environment can be written in homogeneous coordinates as follows:

$$\begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} = \mathbf{P} \cdot \mathbf{S} \cdot (\mathbf{M}_l)^{-1} \cdot \begin{bmatrix} r_{e,x} \\ r_{e,y} \\ r_{e,z} \\ 1 \end{bmatrix}, \tag{8.9}$$

where

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{8.10}$$

is a projective transformation that divides by the z component,

$$\mathbf{S} = \begin{bmatrix} -1 & 0 & 0 & d_{o,x} \\ 0 & -1 & 0 & d_{o,y} \\ 0 & 0 & -1 & d_{o,z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8.11}$$

computes $\vec{d}_o - \vec{r}_o$, and $\mathbf{M}_l$ is the linear part of the affine transformation $\mathbf{M}$. Another matrix is required for mapping x and y into the interval $[0, 1]$ for the use as texture coordinates:

$$\begin{bmatrix} s \\ t \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \\ 1 \end{bmatrix} \tag{8.12}$$

Similar transformations can be derived for the backfacing parts of the environment. These matrices can be used as texture matrices, if $\vec{r}_e$ is specified as the initial texture coordinate for the vertex. Note that $\vec{r}_e$ changes from vertex to vertex, while the matrices remain constant.

Due to non-linearity, the reflection vector $\vec{r}_e$ has to be computed in software. This step corresponds to the automatic generation of texture coordinates for spherical environment maps on standard graphics hardware (see Section 4.1). Actually, this process could be further simplified by assuming that the vector $\vec{v}$ from the eye to the object point is constant. This is true if the object is far away from the camera, compared to its size, or if the camera is orthographic. Otherwise, the assumption breaks down, which is particularly noticeable on flat objects (planar objects would receive a constant color!).

What remains to be done is to combine frontfacing and backfacing regions of the environment into a single image. Using multiple textures, this can be achieved in a single rendering pass. To this end, the backfacing part of the environment map is specified as an RGB texture with the appropriate matrix for the backfacing hemisphere. Then, the frontfacing part is specified as a second texture in RGBα format. The alpha is used to mark pixels inside the circle $x^2 + y^2 \leq 1$ with an alpha value of 1, pixels outside the circle with an alpha value of 0. The blending between the two maps is set up in such a way that the colors of the two textures are blended together using the alpha channel of the frontfacing map.

The important point here is that backfacing vectors $\vec{r}_o$ will result in texture coordinates $x^2 + y^2 > 1$ while the matrix for the frontfacing part is active. These regions fall outside the circular region marked in the frontfacing map, and are thus covered by the backfacing environment map.

Alpha values between 0 and 1 can be used to blend between the two maps if seams are visible due to inconsistent data for the two hemispheres (e.g. if the two hemispheres are recovered from photographs). This would require the pixels outside the circle $x^2 + y^2 \leq 1$ to be valid as in Figure 8.6. These maps have been generated by extending the domain of the paraboloid (Equation 8.1) to $[-1, 1]^2$.

Figure 8.6: Left/Center: two textures comprising the environment map for an object in the center of an office scene. Right: A resampled image from [Haeberli93]. The original image was taken with a $180°$ fisheye lens showing one hemisphere. Then a cubical map was generated by replicating the image for the second hemisphere. From this cubical map, we resampled the map for our parameterization. The map for the second hemisphere would be identical to this one.

If the hardware does not support multiple simultaneous textures, the parameterization can still be applied using a multi-pass method and alpha testing. As above, the pixels of the frontfacing map are marked in the alpha channel. In pseudo-code, the algorithm then works as follows:

```
renderingOfParabolicEnvironmentMaps()
{
  set up the alpha test so that only fragments
      with a source alpha of 1 are rendered

  load the frontfacing part of the environment as a texture
  load the matrix for the frontfacing part on the texture stack
  draw object with r_o as texture coordinate

  load the backfacing part of the environment as a texture
  load the matrix for the backfacing part on the texture stack
  draw object with r_o as texture coordinate
}
```

8.2.3 Mip-map Level Generation

Anti-aliasing of parabolic environment maps can be performed using any of the known prefiltering algorithms, such as mip-mapping [Williams83] or summed area tables [Crow84]. For correct prefiltering, the front- and backfacing maps need to contain valid information for the whole domain $[-1, 1]^2$, as in Figure 8.6.

The next level in the mip-map hierarchy is then generated by computing a weighted sum for each 2×2 block of texels. The weight for each texel is proportional to the solid angle it covers (Equation 8.2). The sum of these solid angles is the solid angle covered by the texel in the new mip-map level, and is used as a weight for the next iteration step. The generation of summed area tables is similar; the solid angles of the pixels are again used as weights.

Mip-mapping is, of course, based on isotropic filtering, and therefore produces errors for grazing viewing angles. Summed area tables are better, but also produce artifacts when the direction of anisotropy does not align with one of the coordinate axes of the texture. These problems, are, however typical for these methods, and in no way specific to environment maps. With the above method, the texture is correctly anti-aliased within the limits of the chosen method, since each texel in the map is correctly anti-aliased, and each pixel on the object is textured by exactly one hemispherical map.

8.3 Visualizing Global Illumination with Environment Maps

Once an environment map is given in the parabolic parameterization, it can be used to add a mirror reflection term to an object. Figure 8.7 shows a reflective sphere and torus viewed from

different angles with the environment maps from Figure 8.6.

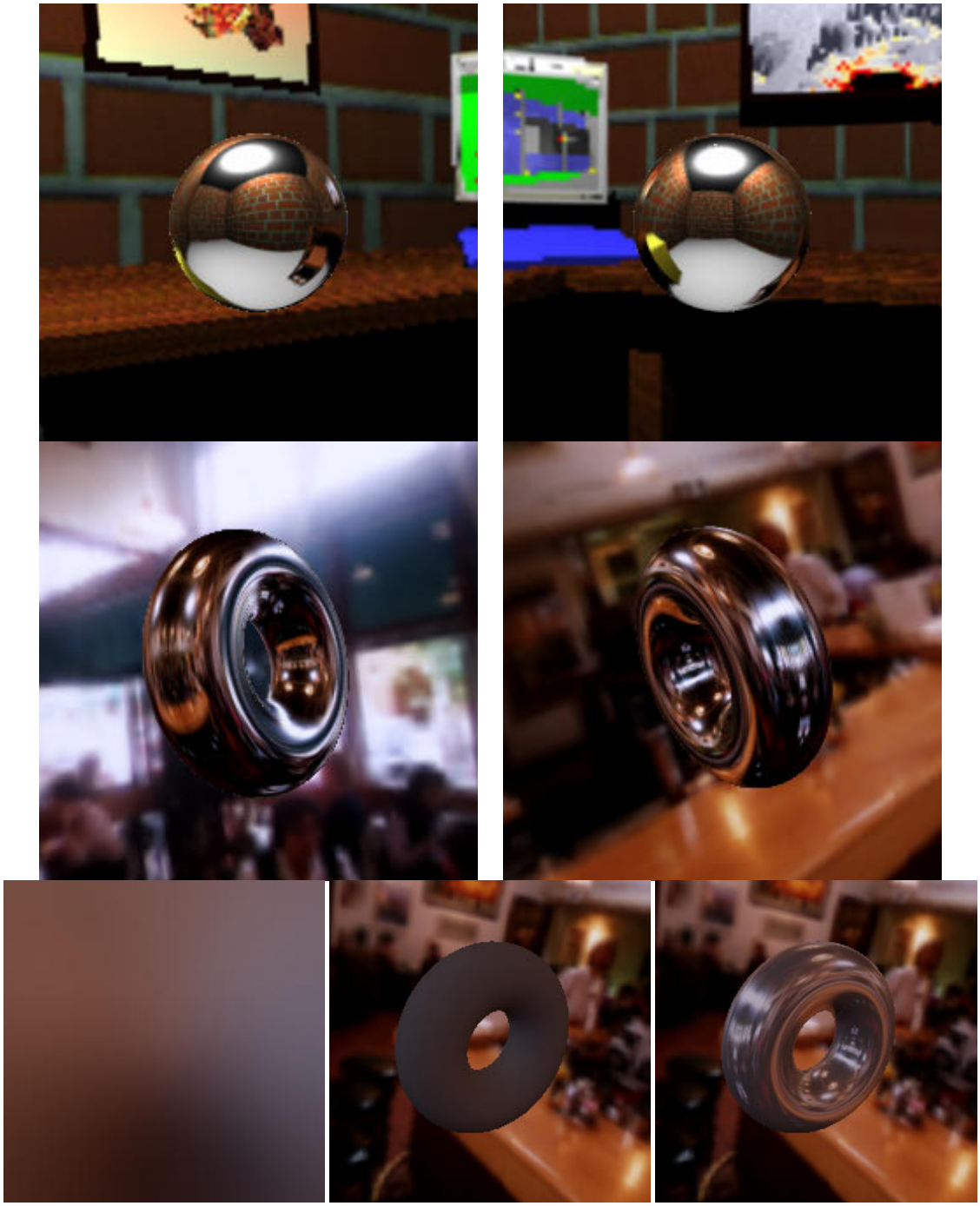

Figure 8.7: Top and center rows: the environment maps from Figure 8.6 applied to a sphere and a torus, and seen from different viewpoints. Bottom left: diffusely prefiltered environment map of the cafe scene. Bottom center: diffusely illuminated torus. Bottom right: same torus illuminated with both a diffuse and a mirror term.

Using multi-pass rendering and alpha blending, this mirror reflection term can be added to local illumination terms that are generated using hardware lighting or the methods from Chapter 5. It is also possible to add a diffuse global illumination term through the use of a precomputed texture. For the generation of such a texture, there exist two methods. In the first approach, a global illumination algorithm such as Radiosity [Sillion94] is used to compute the diffuse global illumination of every surface point.

The second approach is purely image-based, and was proposed by [Greene86]. The environment map used for the mirror term contains information about the incoming radiance $L_i(\mathbf{x}, \vec{l})$, where $\mathbf{x}$ is the point for which the environment map is valid, and $\vec{l}$ the direction of the incoming light. According to Equation 2.10 the outgoing radiance for a diffuse BRDF is then:

$$L_o(\mathbf{x}, \vec{n}) = k_d \cdot \int_{\Omega(\vec{n})} L_i(\mathbf{x}, \vec{l}) \cdot \cos(\vec{n}, \vec{l}) d\omega(\vec{l}). \tag{8.13}$$

Due to the constant BRDF of diffuse surfaces, L_o is only a function of the surface normal $\vec{n}$ and the illumination L_i stored in the environment map, but not of the outgoing direction $\vec{v}$. Thus, it is possible to precompute a map which contains the diffuse illumination for all possible surface normals. For this map, like for the mirror map, the parameterization from Section 8.2 can be used. The only difference is that diffusely prefiltered maps are always referenced via the normal of a vertex in environment map space, instead of via the reflection vector. Thus, diffuse texturing is performed by using the diffuse environment map, a texture matrix that corresponds to $\mathbf{P}$ in Equation 8.10, and initial texture coordinates that correspond to the normal. Figure 8.7 shows such a prefiltered map, a torus with diffuse illumination only as well as a torus with diffuse and mirror illumination.

8.3.1 Generalized Mirror Reflections using a Fresnel Term

A regular environment map without prefiltering describes the incoming illumination at a point in space. If this information is directly used as the outgoing illumination, as is described above, and as it is state of the art for interactive applications, only metallic surfaces can be modeled. This is because for metallic surfaces (surfaces with a high index of refraction) the Fresnel term (Equation 2.7) is almost one, independent of the angle between light direction and surface normal. Thus, for a perfectly smooth (i.e. mirroring) surface, incoming light is reflected in the mirror direction with a constant reflectance.

For non-metallic materials (materials with a small index of refraction), however, the reflectance strongly depends on the angle of the incoming light. Mirror reflections on these materials should be weighted by the Fresnel term for the angle between the normal and the reflected viewing direction $\vec{r}_v$, which is, of course, the same as the angle between normal and viewing direction $\vec{v}$.

Similar to the techniques for local illumination presented in Section 5, the Fresnel term $F(\cos\theta)$ for the mirror direction $\vec{r}_v$ can be stored in a 1-dimensional texture map, and rendered to the framebuffer's alpha channel in a separate rendering pass. The mirror part is then multiplied with this Fresnel term in a second pass, and a third pass is used to add the diffuse part. This yields an outgoing radiance of $L_o = F \cdot L_m + L_d$, where L_m is the contribution of the mirror term, while L_d is the contribution due to diffuse reflections.

In addition to simply adding the diffuse part to the Fresnel-weighted mirror reflection, we can also use the Fresnel term for blending between diffuse and specular: $L_o = F \cdot L_m + (1 - F)L_d$. This allows us to simulate diffuse surfaces with a transparent coating: the mirror term describes the reflection off the coating. Only light not reflected by the coating hits the underlying surface and is there reflected diffusely.

Figure 8.8 shows images generated using these two approaches. In the top row, the Fresnel-weighted mirror term is shown for indices of refraction of 1.5, 5, and 200. In the center row, a diffuse term is added, and in the bottom row, mirror and diffuse terms are blended using the Fresnel term. Note that for low indices of refraction, the object is only specular for grazing viewing angles, while for a high indices of refraction we get the metal-like reflection known Figure 8.7.

8.3.2 Glossy Prefiltering of Environment Maps

So far, we are able to use environment maps for generating a mirror term as well as diffuse illumination, the latter through prefiltering of a given environment map [Greene86]. We would now like to extend the concept of environment maps to glossy reflections, also based on a prefiltering technique. Voorhies et al. [Voorhies94] used a similar approach to implement Phong shading for directional light sources.

These two ideas can be combined to precompute an environment map containing the glossy reflection of an object with a Phong material. With this concept, effects similar to the ones presented by Debevec [Debevec98] are possible in real time. As shown in [Lewis93], the Phong BRDF is given by

$$f_r(\mathbf{x}, \vec{l} \rightarrow \vec{v}) = k_s \cdot \frac{<\vec{r}_l, \vec{v}>^{1/r}}{\cos\alpha} = k_s \cdot \frac{<\vec{r}_v, \vec{l}>^{1/r}}{\cos\alpha}, \tag{8.14}$$

where $\vec{r}_l$, and $\vec{r}_v$ are the reflected light- and viewing directions, respectively, and $\cos\alpha =< \vec{n}, \vec{l} >$ as in Chapter 5. Thus, the specular global illumination using the Phong model is

$$L_o(\mathbf{x}, \vec{r}_v) = k_s \cdot \int_{\Omega(\vec{n})} <\vec{r}_v, \vec{l}>^{1/r} \; L_i(\mathbf{x}, \vec{l}) \; d\omega(\vec{l}), \tag{8.15}$$

for some roughness value r. This is only a function of the reflection vector $\vec{r}_v$ and the environment map containing the incoming radiance $L_i(\mathbf{x}, \vec{l})$. As for diffuse illumination, it is therefore

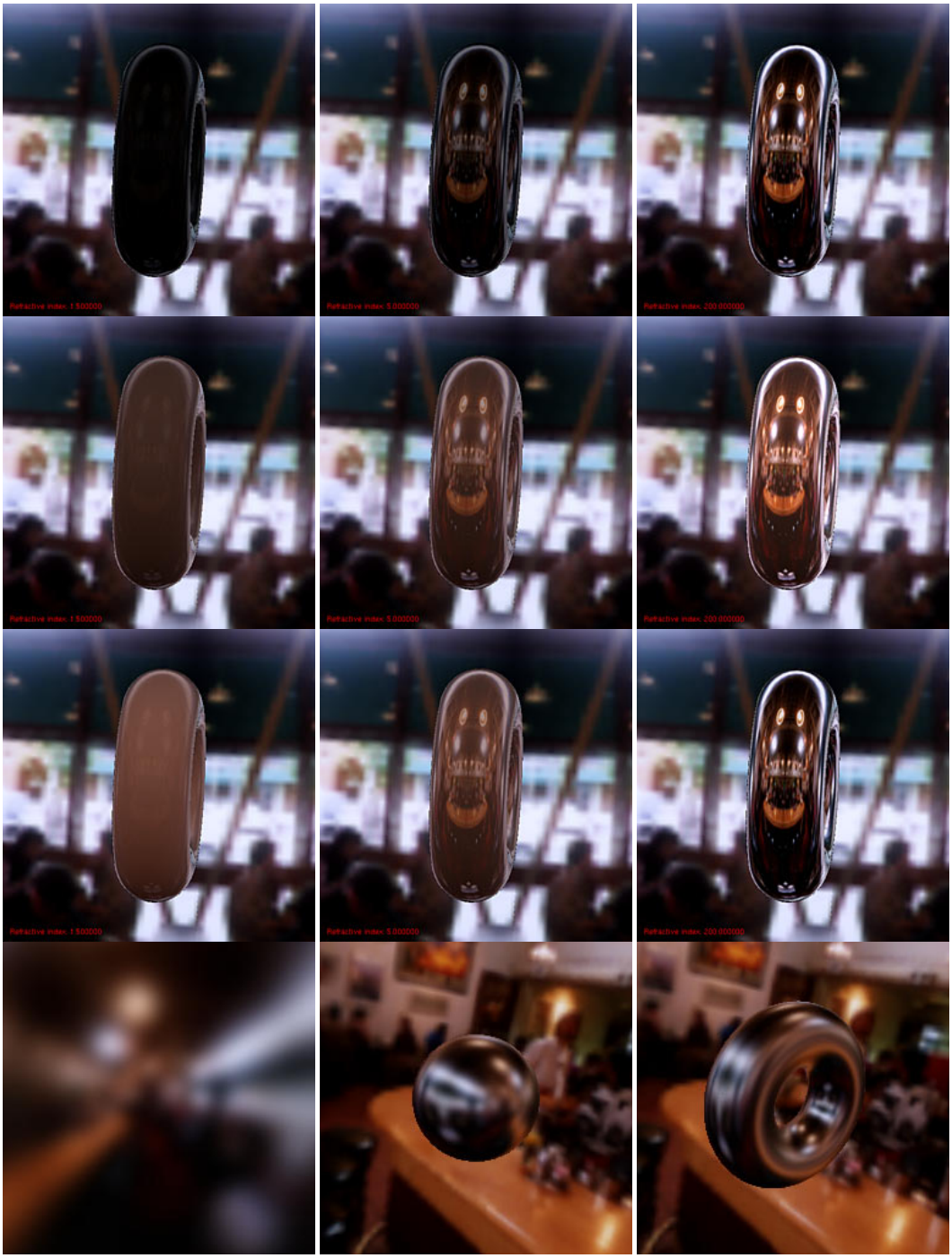

Figure 8.8: Top row: Fresnel weighted mirror term. Second row: Fresnel weighted mirror term plus diffuse illumination. Third row: Fresnel blending between mirror and diffuse term. The indices of refraction are (from left to right) 1.5, 5, and 200. Bottom row: a prefiltered version of the map with a roughness of 0.01, and application of this map to a reflective sphere and torus.

possible to take a map containing $L_i(\mathbf{x}, \vec{l})$, and generate a filtered map containing the outgoing radiance $L_o(\mathbf{x}, \vec{r}_v)$ for a glossy Phong material.

Figure 8.8 shows such a map generated from the original cafe environment in Figure 8.6, as well as a glossy sphere and torus textured with this map.

The use of a Phong model for the prefiltering is somewhat unsatisfactory, since this is not a physically valid model, as pointed out in Section 3.1.2. However, it is the only model that only depends on the angle between reflected light direction and viewing direction, and thus the only one that can be prefiltered in this way. Prefiltering of other models would cause the dimensionality of the map to increase. Even a Fresnel weighting along the lines of Section 8.3.1 is only possible with approximations. The exact Fresnel term for the glossy reflection cannot be used, since this term would have to appear inside the integral of Equation 8.15. However, for glossy surfaces with a low roughness, the Fresnel term can be assumed constant over the whole specular peak (which is very narrow in this case). Then the Fresnel term can be moved out of the integral, and the same technique as for mirror reflections applies.

If the original environment map is given in a high-dynamic range format such as [Larson97, Debevec97], then this prefiltering technique allows for effects similar to the ones described in [Debevec98]. Despite the use of the Phong model, and although reflections of an object onto itself cannot be modeled by environment maps, the renderings are quite convincing, considering that these images can be rendered at interactive frame rates on contemporary low end workstations such as an SGI O2.

8.3.3 Refraction and Transmission

All the techniques described above are also applicable for transmission and refraction on thin surfaces. A thin surface is one for which the thin lens approximation holds (see Chapter 11 and [Born93]). This means the surface is assumed infinitely thin, so that the entry and exit points of each ray coincide. This approximation does not hold for solid objects like glass balls, but it can be used for windows or spectacle lenses (see Figure 8.9).

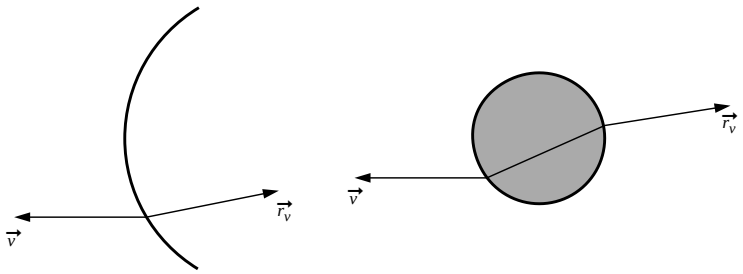

Figure 8.9: Refractions in infinitely thin surfaces can be modeled with environment maps (left), but not refractions in thick objects (right).

Figure 8.10: Rendering frosted glass with prefiltered environment maps.

The only difference that has to be considered for applying environment maps to such a refractive or transmissive object is that the refracted or transmitted viewing ray is used to index the environment maps, instead of the reflected viewing direction. This vector can also be computed in software. Figure 8.10 shows two images with a transmissive polygon that has been texture mapped with a prefiltered Phong environment map to simulate a frosted glass effect.

The restriction to thin surfaces is necessary for this algorithm, because for surfaces with a finite thickness, each ray is refracted twice, once when it enters the object, and once when it leaves. The final ray direction after these two refractions depends on the exact point where the ray leaves the object, which, in turn, depends both on the normal at the point of entrance, and the viewing direction. Thus, a 2-dimensional lookup table is insufficient to characterize the refraction of light in thick objects. A technique based on higher-dimensional lookup tables is discussed in Chapter 10.

8.4 Discussion

In this chapter, we have introduced a novel parameterization for environment maps, which makes it possible to use a single map consisting of two textures for all viewing points and -directions. This allows us to generate walkthroughs of static scenes with reflecting objects without the need to recompute environment maps for each frame. Furthermore, we have shown how environment maps can be used to render global illumination effects based on a mirror, a diffuse, and a glossy term, and how to appropriately weight the mirror and glossy terms in order to simulate non-metallic objects.

All the techniques proposed here work in real time on contemporary graphics hardware (15-20 fps. on SGI O2, 20-25 fps. on SGI Octane MXE and Onyx2 BaseReality for image resolutions of 1280×1024). All textures were mip-mapped for these measurements. Since multiple simultaneous textures are not supported by these platforms, the timings include a total of 5 rendering

passes for images like the ones in Figure 8.8.

Although the methods work well on current hardware, there are some simple modifications to the graphics hardware, which would both further improve the performance and simplify the implementation of our environment map parameterization. Firstly, since the texture coordinates have to be generated in software, it is not possible to use display lists. Secondly, all vectors required to compute the reflected vector $\vec{r}_e$ are already available further down the pipeline (that is, when automatic texture coordinate generation takes place), but are not easily available in software. This situation is a lot like the one in Chapter 5.

For example, the normal vector in a vertex is typically only known in object space, but is required in eye space for the computation of the reflection vector. In order to determine $\vec{n}_e$, the normal has to be transformed by software, although for lighting calculations the hardware later performs this operation anyway.

The solution we propose is again to add a new texture generation mode, which computes the reflection vector $\vec{r}_e$ in eye space based on the available information ($\vec{v}_e$ and $\vec{n}_e$), and assigns its components to the s, t, and r texture coordinates.[1]

Interestingly, the texture coordinates generated for spherical environment maps are the x and y components of the halfway vector between the reflection vector $\vec{r}_e$ and the negative viewing direction $(0, 0, 1)^T$ [Segal98]. Thus, current hardware implementations essentially already require the computation of $\vec{r}_e$ for environment mapping. Our texture generation mode is even cheaper, since the reflection vector is automatically normalized (if normal and viewing direction are), while the halfway vector for spherical maps requires an additional normalization step.

[1] Shortly before preparing the final version of this thesis, Mark Kilgard [Kilgard99] implemented this extension for both the Mesa software library and the nVIDIA Riva TNT/TNT2 drivers based on our suggestion [Heidrich98d].

Chapter 9

Bump- and Normal Mapping

Bump maps have become a popular approach for adding visual complexity to a scene, without increasing the geometric complexity. They have been used in software rendering systems for quite a while [Blinn78], but hardware implementations have only occurred very recently, and so far, no general agreement has been reached on how exactly bump mapping should be integrated into the rendering pipeline. Some of the proposed techniques are described in [Schilling96], [Peercy97], [Miller98a], and [Ernst98].

Bump mapping, as originally formulated in [Blinn78], perturbs the normal of the surface according to a given height field $B(s,t)$ which describes a slight movement of the surface point for each location (s,t) in the parameter domain. The height field defines the spatial position for the moved point $\mathbf{P}'$ as an offset along the surface normal $\vec{n}$ (for simplicity, we assume unit length normals in the following, the general formulae can be found, for example in [Blinn78]):

$$\mathbf{P}' = \mathbf{P} + B(s,t) \cdot \vec{n}. \tag{9.1}$$

The perturbed normal of the displaced surface is given as the cross product of the two tangent vectors in $\mathbf{P}'$:

$$\vec{n}' = \frac{d\mathbf{P}'}{ds} \times \frac{d\mathbf{P}'}{dt} \approx \vec{n} + \underbrace{\left(\vec{n} \times \frac{d\mathbf{P}}{ds}\right) \cdot \frac{dB(s,t)}{dt} + \left(\frac{d\mathbf{P}}{dt} \times \vec{n}\right) \cdot \frac{dB(s,t)}{ds}}_{\vec{d}} \tag{9.2}$$

This formula includes an approximation which assumes that the bump height is small compared to the dimensions of the surface. The details of this approximation, and the derivation of the formula, can be found in [Blinn78].

The difficulty for implementing bump maps in hardware is the computation of the tangent vectors $\vec{t}_s := \vec{n} \times (d\mathbf{P}/ds)$ and $\vec{t}_t := (d\mathbf{P}/dt) \times \vec{n}$, which normally has to be performed for each

pixel. Alternatively, these vectors could be interpolated across polygons, but even this is quite expensive, since it requires a normalization step per pixel. The partial derivatives of the bump map itself, on the other hand, can be precomputed and stored in a texture.

Several ways have been proposed to simplify the computation of $\vec{t}_s$ and $\vec{t}_t$ by making additional assumptions, or to use a completely different, simpler coordinate system. For example, [Schilling96] proposes to build the local coordinate frame in each point using a global reference direction $\vec{m}$, while [Peercy97], among other approaches, suggests to precompute the normal in object space for each pixel, and store *it* in a texture map. This is what we call a *normal map* in the following.

Normal maps have the advantage that the expensive operations (computing the local surface normal by transforming the bump into the local coordinate frame) have already been performed in a preprocessing stage. All that remains to be done is to use the precomputed normals for lighting each pixel. As we will show in the following, this allows us to use normal maps with a fairly standard rendering pipeline that does not have explicit support for bump mapping. Another advantage of normal maps is that recently methods have shown up for measuring them directly [Rushmeier97], or for generating them as a by-product of mesh simplification [Cohen98].

The major disadvantage of normal maps is that the precomputation introduces a surface dependency, since the geometry of the underlying surface is required to compute the object space normals. This means that a different texture is required for each object, while bump maps can be shared across different objects.

In the following, we first describe how normal maps can be lit according to the Blinn-Phong illumination model using a set of hardware imaging operations (see Section 4.4). Afterwards, we discuss how the techniques for other local illumination models from Chapter 5, as well as the environment mapping techniques from Chapter 8 can be used together with normal maps. This part relies on the presence of the pixel texture extension in addition to the imaging operations.

The methods described here also assume a non-local viewer and directional light sources. The artifacts introduced by these assumptions are usually barely noticeable for surfaces with bump maps, because the additional detail hides much of the approximation error.

9.1 Local Blinn-Phong Illumination

For implementing ambient, diffuse, and Phong lighting for normal maps, a small subset of the hardware imaging operations (see Section 4.4) is required. These are a 4×4 color matrix followed by a color lookup table during the transfer of textures to the texture memory.

With these two mechanisms, the illumination for a given color coded normal map can be computed in two rendering passes. For the first pass, a color matrix is specified, which maps the per-pixel normal from object space into eye space and then computes the ambient and diffuse

components of the illumination:

$$\mathbf{L} \cdot \mathbf{D} \cdot \mathbf{M} \cdot \vec{n} = (R, G, B)^T, \tag{9.3}$$

where

$$\mathbf{L} = \begin{bmatrix} k_{d,R} \cdot I_R & 0 & 0 & k_{a,R} \cdot I_{a,R} \\ k_{d,G} \cdot I_G & 0 & 0 & k_{a,R} \cdot I_{a,G} \\ k_{d,B} \cdot I_B & 0 & 0 & k_{a,R} \cdot I_{a,B} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{D} = \begin{bmatrix} l_x & l_y & l_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{9.4}$$

$\mathbf{M}$ is the model/view matrix, I is the intensity of the directional light, I_a that of the ambient light, and k_a and k_d are the ambient and diffuse reflection coefficients of the material. The matrix $\mathbf{D}$ computes the dot product between the normal $\vec{n}$ and the light direction $\vec{l}$. When the normal image is now loaded into texture RAM, the lighting computations are performed. Afterwards, the loaded, lit texture is applied to the object using texture mapping. A similar second rendering pass draws the specular part. This time, however, the matrix $\mathbf{D}$ computes the dot product between normal and the halfway vector $\vec{h}$, and the matrix $\mathbf{L}$ is replaced by a lookup table containing the function $p_i(x) = k_{s,i} \cdot I_i \cdot x^{1/r}$ for each color component $i \in \{R, G, B\}$.

The top of Figure 9.1 shows two images in which one polygon is rendered with this technique. On the left side, a simple exponential wave function is used as a normal map. The normal map for the image on the right side was measured from a piece of wallpaper with the approach presented in [Rushmeier97]. The bottom of the figure shows a sphere and a torus with a normal map generated from a Perlin noise function [Perlin89, Ebert94].

9.1.1 Anti-aliasing

An interesting note is that the graphics hardware as defined in Chapter 4 does not allow for proper mip-mapping of these lit normal maps. The only way to specify a mip-map hierarchy is to provide a hierarchy of normal maps at different resolutions, which are then transformed (lit) separately through the use of pixel transfer operations.

However, this does not provide a correct hierarchy of mip-map levels. The filtering for the downsampled versions of the texture should occur *after the lighting*, that is, after the non-linear operations implemented by the color lookup table. This is true not only for the above algorithm, but for *any* non-linear pixel transfer operation (which is what lookup tables are for). These applications mandate that the hardware should be able to downsample a given texture itself in order to generate a valid mip-map hierarchy.

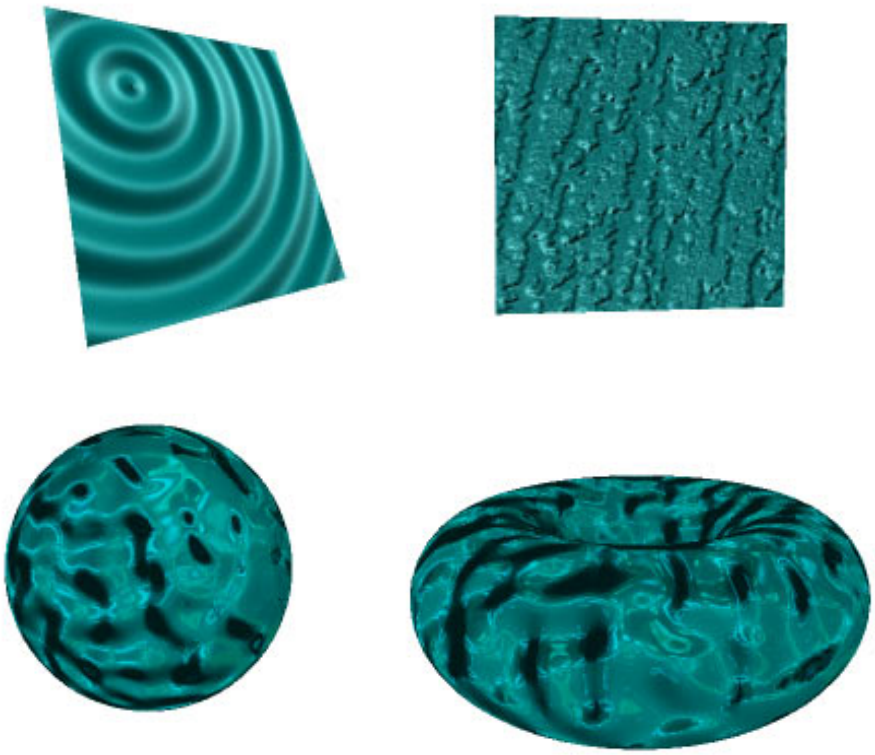

Figure 9.1: Top: two Phong-lit normal maps. The right one has been measured from a piece of wallpaper using the approach presented in [Rushmeier97]. Bottom: sphere and torus with normal maps generated from a Perlin noise function.

For our specific application, such a down-sampling would compute a dynamic transition from bump maps to 2-dimensional slices of a 4-dimensional BRDF, much in the spirit of [Cabral87].

The problem that filtering should occur *after* the illumination process instead of beforehand, has also been noted by Schilling [Schilling97]. He proposes an additional roughness pyramid that is used in a modified Blinn-Phong reflection model. His approach has the advantage that it also works for environment mapping, which ours does not. On the other hand, our proposal would yield the correct mip-map hierarchy, while Schilling's method is still only an approximation.

9.2 Other Reflection Models

In order to apply the reflection models from Chapter 5 to normal mapped surfaces, the indices into the lookup tables storing the materials have to be computed on a per-pixel basis. This can be achieved using pixel textures. First, the object is rendered with the normal map as a texture. The resulting image is then read into main memory, yielding an image containing the normal for

each pixel. Then, the color matrix is set up to compute the texture coordinates for the reflection models, in exactly the same way the texture matrix was used in Section 5.1. Since we assume a non-local viewer and directional light sources, it is not necessary to compute any part of the texture coordinates in software.

This leads to the following algorithm for applying alternative lighting models:

```
normalMappingWithAlternativeReflectionModels()
{
  set up stenciling to mark each rendered pixel
  load the normal map as a texture
  render the object

  copy the framebuffer to main memory

  set up stenciling so that only pixels marked
      in the stencil buffer are affected
  for each texture used by the reflection model
  {
    load the texture
    set up the texture matrix to compute the dot products
        used as texture coordinates
    set up blending as required by the reflection model
    copy the normal image from main memory into the framebuffer
  }
}
```

9.3 Environment Mapping

Similarly, environment maps can be applied to normal-mapped surfaces with the help of pixel textures. As pointed out in Chapter 8, both the spherical and the parabolic parameterizations use the halfway vector between the reflected viewing direction and a reference direction as an index into the environment map. For orthographic cameras this vector simplifies to the surface normal.

Thus, if the normal map contains unit length normals, these can be directly used as texture coordinates for a spherical map, while normal maps in which the z component is normalized to 1 can be used for parabolic maps. As in the previous section, the object is first rendered with the normal map, the resulting image is read back to main memory, and then it is written back to the framebuffer with activated pixel textures.

Unfortunately, since pixel textures currently do not support projective textures, the division by z, which in Chapter 8 was used to compute the texture coordinates for arbitrary viewing directions, cannot be performed with pixel textures. This means that even if parabolic environment maps are used, they can only be applied for one viewing direction, which, of course, defeats the point of introducing parabolic environment maps in the first place. This shows again how useful support for projective texturing with pixel textures would be.

The images in Figure 9.2 have been generated using the pixel texture extension and a single, view-dependent environment map.

Figure 9.2: Combination of environment mapping and normal mapping. Left: environment map only. Right: local Phong illumination plus environment map.

9.4 Discussion

While the techniques discussed in this chapter can help to achieve realistic renderings on contemporary hardware, the future clearly belongs to dedicated bump mapping hardware, since it allows the re-use of bump maps for more than one object. Moreover, since bump mapping does not rely on pixel transfer functions to compute the illumination, the bandwidth between CPU and graphics subsystem would be reduced. However, there are some lessons to be learned from the algorithms in this chapter:

It is clear that applying environment maps to normal- or bump mapped surfaces requires two successive texturing stages. Bump mapping hardware which wants to support environment maps will have to take this fact into account. The similarity between the algorithms in Sections 9.2 and 9.3 shows that the same mechanisms used for environment mapping can also be applied to support physically based material models, if the hardware is designed carefully. In particular, this means that the second texturing stage should have the flexibility described in Section 5.3.2 with respect to texture coordinate generation. If the hardware performs per-fragment lighting, which bump mapping hardware has to do anyway, then all the required vectors need to be interpolated

across polygons, and are therefore available for each fragment. All that remains to be done is to introduce a per-pixel texture coordinate generation mechanism, which computes the required dot products.

Chapter 10

Light Field-based Reflections and Refractions

In Chapter 8 we discussed how environment maps can be used to render reflections and refractions on non-diffuse, curved objects. Like all techniques based on environment maps, these methods break down for large objects reflecting other nearby geometry. In this chapter we explore a number of light field-based techniques that could help to overcome these restrictions. In particular, we share some thoughts on the method by [Miller98b], and argue that, for increased efficiency, the light field representation should be decoupled from the surface geometry. The discussion leads to issues for future research.

A commonly used technique for rendering mirror reflections on planar objects is given in [Diefenbach94] and [Diefenbach96]: with a simple affine model/view matrix, the scene is mirrored at the planar reflector. This mirrored scene is rendered at every pixel where the reflector is visible in the current view. This is typically achieved in two rendering passes. First, the original scene is rendered, and all pixels of the planar reflector are marked in the stencil buffer. Then, the model/view matrix is modified to accommodate for the reflection. The scene is now rendered again, but only pixels marked in the stencil buffer are set. If the stencil buffer has more than one bit, it is also possible to realize multiple reflections [Diefenbach96].

A similar effect can be achieved using texture mapping. Instead of mirroring the scene, the eye point $\mathbf{P}$ is mirrored, yielding a reflected eye point $\mathbf{P}'$, as depicted on the left side of Figure 10.1. Rendering the scene from this eye point with the reflector as an image plane yields the texture image to be applied to the reflector as seen from the eye. Note that this approach has two major disadvantages relative to the one from Diefenbach. Firstly, the rendered image from the first pass needs to be transferred from the framebuffer into texture memory, which requires additional bandwidth, and secondly, the texturing represents a resampling step that results in reduced image quality.

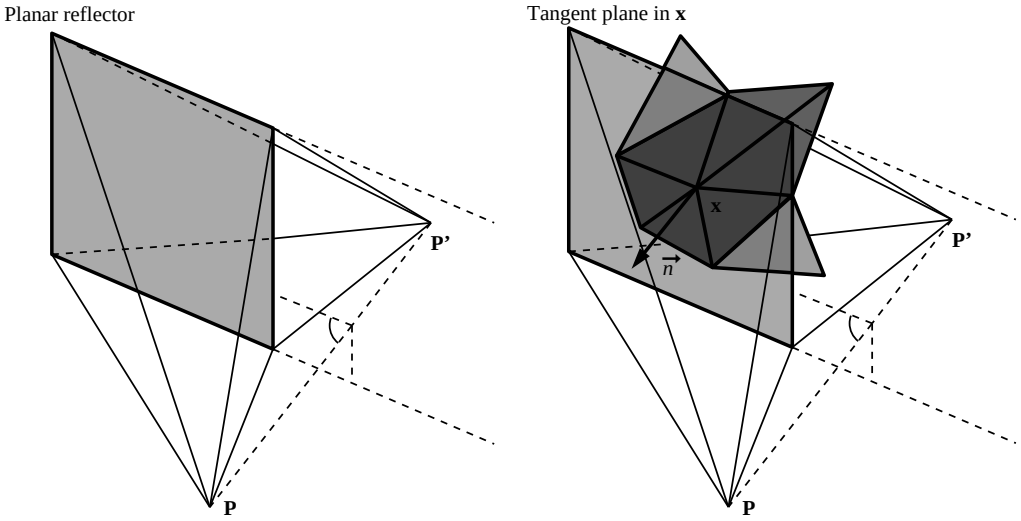

Figure 10.1: Multi-pass mirror reflections in planar and curved objects. While a single reflected eye point $\mathbf{P}'$ exists for planar reflectors, curved reflectors do not have such a uniquely defined point. If a curved object is approximated by a triangle mesh with per-vertex normals, one reflected point can be defined for each vertex in the mesh using the tangent plane in that vertex.

For these reasons, the modified algorithm is an inferior choice for implementing reflections on planar surfaces, but it can be generalized to a naive (and inefficient) method for generating reflections on curved surfaces represented as triangle meshes: for curved surfaces the problem is that the reflected eye point is not constant, but varies across the surface. However, if the surface is reasonably smooth, then it suffices to compute the reflected eye point only at some discrete points on the surface, say the vertices of the triangle mesh, and to interpolate the radiance for each point inside a triangle from the textures obtained for each of the three vertices. Note that each of the textures corresponds to a dynamically generated, 2-dimensional slice of a light field describing the incoming illumination around the reflector, and the interpolation step is nothing but the reconstruction of a novel view from this light field information.

The complete algorithm would then work as follows (see right side of Figure 10.1): For each vertex in the triangle mesh the tangent plane is determined, the eye point is reflected in that plane, and the reflection texture for that tangent plane is rendered. Then, for each vertex, the triangle fan surrounding it is rendered with the reflection applied as a projective texture. During this rendering, the alpha value for the center vertex of the fan is set to 1, the alpha values for all other vertices are set to 0, and the result from texturing is multiplied by the alpha channel. This way, the alpha channel contains the basis functions for the Barycentric coordinates in each pixel. The final image results from adding up all the contributions from the different triangle fans in the frame buffer. This is exactly the interpolation scheme used for the hardware implementation of light fields and Lumigraphs in [Gortler96] and [Sloan97].

Clearly, this approach is not feasible for real time or interactive applications, since the number of vertices (and thus the number of rendering passes) on typical reflectors are often in the order of tens of thousands. On the other hand, for a static scene the incoming light field at the object does not change. Therefore, it is not necessary to rerender the geometry multiple times for each frame in order to generate the 2D slices used as textures. Instead, a practical light field-based method could rely on some amount of precomputation to achieve interactive frame rates.

A geometry-based method for reflections on curved surfaces has recently been introduced in [Ofek98]. For each frame, all vertices of the reflected geometry are individually transformed in software. This approach only works at interactive performance for relatively smooth objects that are either concave or convex. More complicated reflectors with both convex and concave regions need to be subdivided, and a separate copy of the reflected geometry is computed for each of the subdivided parts. Like the texturing method described above, this approach also quickly becomes infeasible.

10.1 Precomputed Light Fields

One approach for implementing reflections in static scenes as precomputed light fields was presented in [Miller98b]. There, the parameterization of the light field is chosen such that u and v are surface parameters of the reflector, and s and t parameterize the hemisphere of directions over the surface point (u, v). The authors call this a *surface light field* because the (u, v) parameters are directly attached to the surface of the reflector.

With this scheme, every sample in the light field is associated with a specific location on the surface, and therefore the local coordinate frame on the surface is implicitly known for each sample. Thus, it is possible to store global illumination solutions for arbitrary reflection models in this light field. For the environment mapping techniques from Chapter 8 this was not possible, because environment maps are decoupled from the surface geometry. Without the local coordinate frame, the glossy prefiltering from Section 8.3.2 was restricted to the Phong model, since this is the only specular reflection model that only depends on the direction of the reflection vector.

In order to render an object with a surface light field as described above, the viewing direction at each vertex has to be transformed into the local coordinate system, and then the s and t parameters have to be computed. With these, a (u, v) slice can be extracted from the light field and used as a texture map. The extraction process involves a bilinear interpolation of the (s, t) parameters, and another bilinear interpolation during the texturing process. Since [Miller98b] uses a block-based compression scheme, it is also necessary to decompress the required parts of the light field during the extraction phase.

The problem with this approach is that the light field is attached to the surface of the re-

flecting object, which makes a costly transformation of the viewing vector into the local coordinate frame at each vertex necessary. If the two-plane parameterization (see Chapter 3 and [Gortler96, Levoy96]) were used for representing the light field, the reconstruction could be implemented with the efficient hardware technique from [Gortler96]. In our own implementation of this method we achieve frame rates of > 20 fps for full screen reconstructions on an SGI O2, independent of the light field resolution. Moreover, on machines with 4D texture mapping, the whole light field can be specified as a single texture from which the images can be extracted in one rendering step, simply by applying the correct s, t, u, and v coordinates as 4D texture coordinates in each vertex. This assumes that the texture memory is large enough to hold the complete light field. This issue can be resolved through the use of vector-quantized light fields, which can be directly rendered in hardware [Heidrich99b, Heidrich99a], using an OpenGL extension called "pixel textures" available from Silicon Graphics.

The major disadvantage of the two-plane parameterization is that the purely image-based reconstruction from [Levoy96] results in a blurring of objects further away from the (u, v) plane. The parameterization by [Miller98b] has the advantage that the reflector itself is sharp and focussed. The same effect, however, can also be achieved with two-plane parameterized light fields using the depth correction proposed in [Gortler96]. This depth corrected version can be implemented with hardware support almost as efficiently as without depth correction.

Both light field representations reach their limits when it comes to mirror reflections and narrow specular highlights from light sources. In both approaches these will still result in some amount of unwanted blurring, due to the limited (s, t) resolution of the light field.

One strategy to overcome this problem, at least as far as specular highlights are concerned, is to separate the direct and the indirect illumination. If only the indirect illumination is stored in the light field, and the direct illumination from light sources is computed on the fly using either standard hardware lighting or the techniques from Chapter 5, crisp highlights can be achieved. Such a separation also has the advantage that some extra blurring for the indirect illumination will in most cases be tolerable since direct illumination is visually much more important than indirect illumination for almost all scenes. This means that one can get away with a lower resolution light field, which saves memory and reduces the acquisition cost for the light field.

To summarize, we argue that it is efficient to represent the precomputed reflections off a non-diffuse, curved object with a two-plane parameterized light field for interactive viewing with graphics hardware. We propose to split the illumination into a direct and an indirect part, and to generate the former on the fly using hardware lighting or one of the techniques from Chapter 5, while the latter is added in a separate rendering pass by exploiting hardware light field rendering as proposed in [Gortler96]. This second pass is extremely efficient and independent of scene complexity.

10.2 Decoupling Illumination from Surface Geometry

However, it is even possible to go one step further. The two-plane parameterization decouples representation of indirect light from the surface complexity. We would like to completely decouple the illumination from the geometry, as in the case of environment maps. The techniques discussed in this section are based on a discussion with Michael Cohen [Cohen97], and were later published in [Heidrich99b]. Decoupling of illumination and geometry is interesting because it simplifies the modeling of a scene. The geometry of the illuminated object, and the environment causing the illumination, can be replaced independently without the need to recompute a complete global illumination solution every time. On the other hand, the disadvantage of this approach is also be clear: as in the case of environment maps, the choice of reflection models is limited, although glossy prefiltering of the illumination with the Phong model is still possible.

The core of the proposed method is the idea to create a "light field" containing geometric information about the object in the form of a color coded direction which can then be used to look up the illumination. The illumination can either be provided in the form of an environment map or in the form of another light field. Thus, to render a complete image, first the geometry light field is rendered, yielding an image of color coded directions. These are then used to look up the illumination from an environment map or a second light field describing the surrounding scene. Both alternatives will be described in the following, although only the variant using environment maps has been implemented so far.

The first case, using environment maps, only makes sense for refractions, since for reflections the result would be the same as if environment mapping were directly applied to the surface geometry.

For this first algorithm using environment maps, the geometry light field should contain the color coded halfway vector between the viewing direction and the refracted viewing ray, since this vector directly represents the texture coordinates for the environment map, as described in Chapter 8. The image resulting from a rendering of this light field contains the indices into an environment map, just as the normal maps in the previous chapter did. Thus, in order to look up the illumination on a per-pixel basis, we can apply pixel textures, just as in Chapter 9.3.

Figure 10.2 shows two images that were generated using this approach. The top left image represents the color coded halfway vectors reconstructed from the light field. The light field itself was generated using the Vision rendering system and a special RenderMan shader. The other images represent the final results after the application of the environment map. This method is very fast, and achieves between 15 and 20 fps on an Octane MXE. The frame rate depends mostly on the resolution of the final image, which determines the amount of work the pixel texture extension has to do.

If the illumination in the scene is also stored in a light field, the geometry light field has to contain the correct u, v, s, and t coordinates to reference it. This means that the relative position

Figure 10.2: Light field rendering with decoupled geometry and illumination, the latter being provided through an environment map. Top left: color coded texture coordinates for the environment map, as extracted from the geometry light field. Top right and bottom: final renderings.

of the two light fields is fixed.[1] To look up the illumination for each pixel, the pixel texture extension is again employed, this time with the second light field as a 4D texture. Combined with the results from [Heidrich99a], this second light field can again be vector-quantized.

10.3 Discussion

In this chapter we have explored two practical techniques for applying light fields to the rendering of non-diffuse reflections in curved objects. Firstly, the use of two-plane parameterized light fields as introduced in [Gortler96] and [Levoy96] for rendering the contribution of the indirect illumination, and secondly the concept of separating the geometry and the illumination into distinct light fields. In both cases, graphics hardware can be efficiently used for the rendering.

The latter approach bears the disadvantage that the number of reflection models is limited. Besides a mirror reflection term and diffuse illumination, only a Phong model can be used through a prefiltering of the incoming illumination information. However, the advantage of this approach is an increased flexibility for modeling. Image-based representations of objects can be positioned in a scene and lit by image-based representations of the illumination in that scene.

[1] An appropriate color matrix can be used for minor adjustments. However, the full range of affine transformations is not available, since the four texture coordinates cannot be interpreted as a homogeneous vector in 3-space.

This idea of storing geometric information instead of simply color values in a light field can be extended even further. For example, if normal vectors are stored in the light field data structure, the local illumination for the object can be computed using the very same techniques applied to normal maps in Chapter 9. Furthermore, using the reflection models from Chapter 5, arbitrarily complex materials can also be simulated. This allows for image-based rendering with changing illumination, an area that has recently been of increased interest in the research community.

Chapter 11

Lens Systems

A very specialized application domain for light field-based refractions is the simulation of realistic lens systems. Since camera lenses are typically relatively smooth, but close to the eye, it is reasonable to treat them differently from other refractive objects, which are typically relatively far from the eye, but might have an arbitrary geometric complexity. In this chapter, we describe a light field-based algorithm that is tailored towards the accurate simulation of realistic lens systems.

The accurate simulation of properties of complex lens systems, including depth of field and geometric aberrations, in particular distortions, are of high importance to many applications of computer graphics. In the past, most approaches for simulating these properties have been based on off-line rendering methods such as distribution ray-tracing [Cook84, Kolb95].

Efforts for improved lens and camera models for interactive computer graphics have mainly been restricted to the simulation of depth of field [Haeberli90, Shinya94]. On the other hand, a model that allows for a more accurate simulation of real lens systems would be particularly useful for interactive graphics, because it could not only be used for photorealistic rendering, but also for combining real and synthetic scenes, for example in augmented reality and virtual studios. For these environments, it is necessary to simulate real lens systems, so that real-world and synthetic objects can be merged into a single image.

In this chapter, we outline an image-based camera model for interactive graphics, which is capable of simulating a variety of properties of real lens systems at high frame rates. The full details of this model have been published in [Heidrich97a]. Similarly to [Kolb95], the model uses the real geometry of lenses and computes an accurate approximation of the exposure on the film plane. However, instead of using ray-tracing, our model approximates the light field [Levoy96, Gortler96] between the lens and the film plane. By making use of coherence in ray-space, common computer graphics hardware can be used for sampling the light field and rendering the final image. Before we introduce our own model, we briefly review other camera models used in the past.

11.1 Camera Models in Computer Graphics

The most commonly used camera models in computer graphics include the pinhole model, thin and thick lens approximations, and finally full geometric descriptions of lens systems. These will be explained in the following.

11.1.1 The Pinhole Model

The pinhole model for cameras is by far the most frequently used camera model in computer graphics. Conceptually, it consists of a box with a small hole of negligible size in one of its sides. Light falls through this hole and projects an upside-down image onto the film on the opposite side of the box. The situation is depicted in Figure 11.1.

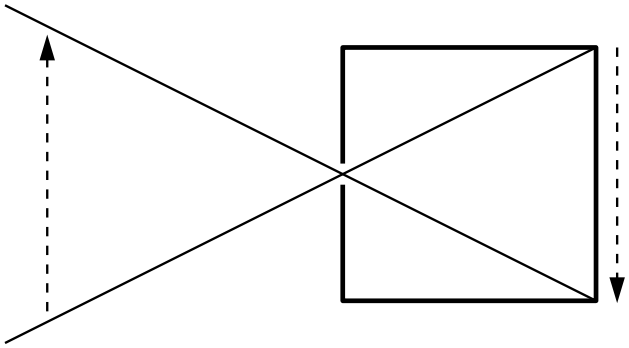

Figure 11.1: A pinhole camera.

The advantage of this model is its simplicity. Because the size of the hole is negligible, the light falling through it can be assumed to be projected through a single point. This projection can be described as a perspective transformation, whose parameters are the dimensions of the film and the distance of the film plane from the hole, along with additional near and far clipping planes (see, for example [Foley90]).

Usually, when dealing with perspective transformations, we do not think of it in terms of a pinhole camera, but as of a perspective projection with a center of projection (COP) and some virtual image plane in front of it. Throughout this chapter, we use the term ”image plane” for a virtual plane in front of the COP of a perspective projection, while the term ”film plane” refers to the plane containing the film in a camera model.

Although it is actually possible to construct a physically working pinhole camera, this is not practical for several reasons. Most importantly, only very little light falls on the film since the hole is so small, and thus the exposure time has to be very long.

11.1.2 The Thin Lens Model

Real lenses have a larger opening, called *aperture*, whose size cannot be neglected. The simplest model of lenses with circular symmetry and finite aperture is the *thin lens approximation*. This model is used in optics and lens design to describe some of the properties of simple lens systems [Born93].

The fundamental assumption of the thin lens model is that the lens is of negligible thickness. As a consequence, light passing through the lens is refracted only in a single plane, the *principal plane*, and moves along a straight line otherwise.

Light coming from a single point **Q** in object space is focused in a single point in image space and vice versa. Incident light from object space parallel to the optical axis is focused in the *focal point* $\mathbf{F}'$ in image space, while parallel light from image space focuses in the focal point **F** in object space. Both **F** and $\mathbf{F}'$ lie on the optical axis of the lens. The situation is depicted in Figure 11.2.

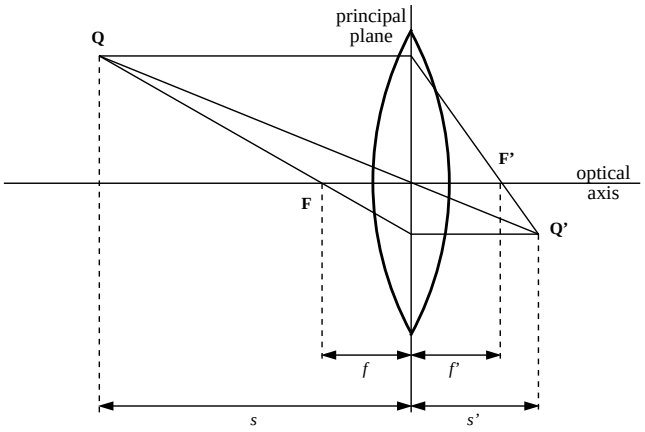

Figure 11.2: The geometry of a thin lens system.

If both object space and image space are in the same medium, the distances f and f' of the focal points from the principal plane are equal, and this distance is called *focal length*. From this property it follows that rays passing through the center of the lens are not bent, but pass straight through the lens system.

With this information it is possible to construct the image of a scene on a film plane at distance s' from the principal plane, given the aperture and the focal length f. For rendering it is often convenient to specify the distance s of the focal plane (the plane that contains all points that are focussed on the film plane) instead of the focal length. This distance can be easily derived from the following well-known relationship

$$\frac{1}{s} + \frac{1}{s'} = \frac{1}{f}.$$

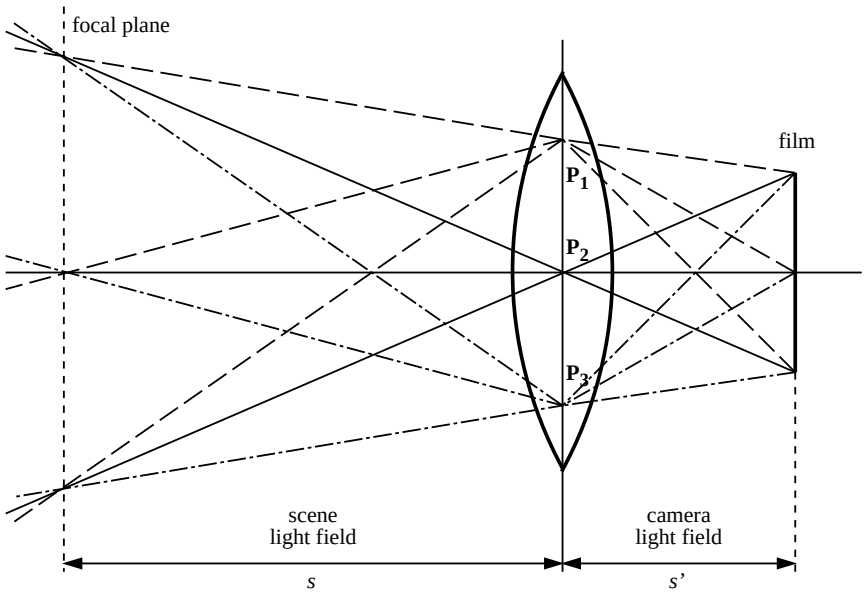

Figure 11.3: Rendering using the thin lens approximation. The rays from all points on the film through a single sample on the aperture form a perspective transformation, which can be used to render a slice of the light field. Rays with the same line style originate from different points on the film, but pass through the same sample point on the lens. All rays with the same line style form a slice of the respective light field.

11.1.3 Rendering Thin Lenses

Typically, rendering of thin lenses is done using distribution ray-tracing [Cook84]. For each sample point on the film, rays are cast through random sample points on the principal plane, and the resulting color values are averaged. Casting of the rays is done by computing the point on the focal plane that corresponds to the point on the film by shooting a ray through the center of the lens. The intersection point is then connected to the chosen sample point on the principal plane.

An alternative approach is to select a fixed set of sample points on the principal plane [Haeberli90]. All rays passing through a single sample point $\mathbf{P}_i$ on the principal plane represent a perspective projection with $\mathbf{P}_i$ as the COP (see Figure 11.3).

Each of the images generated by these perspective projections represent a 2-dimensional slice of the 4-dimensional light field in front of the lens system. In the following, we call this the *scene light field*. Due to the properties of the thin lens model, this slice is identical to a slice of the *camera light field*, defined by the refracted rays between the lens system and the film plane. For both light fields, we can use the two-plane parameterization (see Section 3.3).

Because each slice of the light field can be computed using a standard perspective projection, computer graphics hardware can be used to render the slices. Averaging of slices from different

sample points $\mathbf{P}_i$ to form the final image can be done using an accumulation buffer [Haeberli90].

It should be noted that the simple averaging of the slices of the light field does not yield the correct exposure on the film, as pointed out by [Kolb95]. Our full lens model as described in [Heidrich97a] contains techniques for computing the correct exposure for each point on the film. These techniques can also be applied to thin lenses.

11.1.4 The Thick Lens Model

An improved model for circularly symmetric lenses, where the thickness cannot be neglected, is the *thick lens model*. It is frequently used in optics for complex lens systems composed of several lenses, and allows for a higher accuracy in the approximation.

In contrast to the thin lens, a thick lens has two principal planes. The (signed) distance between the two planes is called the thickness of the lens. Rays from object space hit the object-sided principal plane, then move in parallel to the optical axis, until they hit the image-sided principal plane, where they leave the lens.

From a rendering point of view, a thick lens can be treated very much like a thin lens, except for the shift parallel to the optical axis. Both the thin and the thick lens model yield perfectly undistorted images. Real lenses, however, always show aberrations. Although lens designers usually try to minimize these, they are often not negligible. Visualization and simulation of these effects requires a more sophisticated lens model.

11.1.5 The Geometric Lens Model

The geometric lens model is based on a full geometric description of the lenses together with their index of refraction. The model is evaluated by tracing rays through the true lens geometry, bending it at lens surfaces according to the change in the index of refraction.

The full geometric model correctly simulates all kinds of geometric aberrations, and is the only model that is capable of handling lenses without rotational symmetry, for example bifocal lenses or the progressive addition lenses (PALs) used for eye glasses [Loos98].

This model has been used in [Kolb95] to generate accurate simulations of complex lens systems using distribution ray-tracing. Unfortunately, the model is too expensive in terms of rendering time to be used for interactive graphics.

11.2 An Image-Based Camera Model

In following, we describe an image-based model for camera lens systems, which is capable of simulating aberrations based on the geometric description of the lens. Despite this flexibility, it

is possible to use computer graphics hardware to render images based on this model. It is well suited to interactive applications requiring high frame rates.

Instead of directly using the full geometry of the lens, our model describes a lens as a transformation of the scene light field in front of the lens into the camera light field between the lens and the film. Every property of a lens system is completely defined by such a transformation.

Computer graphics hardware can efficiently generate slices of light fields. Therefore, our model describes a lens system as a mapping from slices of the scene light field into corresponding slices of the camera light field. The mapping consists of two parts: selection of an appropriate slice of the scene light field for a given slice of the camera light field, and a morphing operation correcting for distortions due to the aberrations of the lens system.

We represent a slice of the scene light field as a perspective projection of the scene onto a suitable image plane. Thus, a lens is represented as a set of perspective projections and corresponding image morphing operators.

Similar to rendering with thin lenses, the final image is a composite of a number of slices of the camera light field. These slices are defined by a number of sample points $\mathbf{P}_i$ on the image-sided surface of the lens system (see Figure 11.4).

11.2.1 Approximating Lens Systems

The approximation of real lens systems within the new model consists of two steps. For each slice of the camera light field, a corresponding slice in the scene light field has to be selected. In general, the 2-manifold in the scene light field corresponding to a given slice of the camera light field is unfortunately not planar, and therefore cannot be exactly represented as a perspective projection.

This fact can easily be observed by tracing rays from multiple points on the film through a single point on the lens surface. The rays leaving the system need not intersect in a single common point (which could be used as the COP of a perspective projection, see Figure 11.4). However, in many cases a perspective projection can be found that is a good approximation to the refracted rays.

In order to find the approximating perspective projection, rays are shot from a regular grid on the film through the point $\mathbf{P}_i$ on the image-sided surface of the lens system. Rays are refracted as they pass through the lens, yielding a set of rays leaving the lens system on the object side, as shown in Figure 11.4.

This set of rays is then approximated with a projective transformation, which requires us to select a virtual image plane, and to find an appropriate center of projection. This is described in Section 11.2.3. An alternative way of interpreting this approximation is that the corresponding 2-manifold in ray space is linearly approximated by a plane.

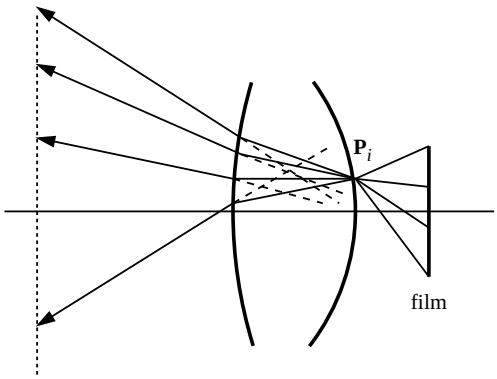

Figure 11.4: Tracing rays from a grid on the film through a single point $\mathbf{P}_i$ on the lens surface yields a set of refracted rays in object space. In general, these do not intersect in a single point, and thus an approximate virtual center of projection has to be found.

The remaining parameters of the perspective transformation, in particular the upper, lower, left and right boundaries on the image plane can then be found by computing the bounding box of ray intersections with the chosen virtual image plane.

Rendering the scene with the computed perspective transformations yields an image containing a slice of the scene light field. At the same time, it approximately corresponds to a distorted slice of the camera light field, and the distortions are directly caused by the aberrations of the lens. We compensate for these distortions using morphing with bilinear interpolation. This is achieved by texture-mapping the slice onto the grid on the film, from which the rays had been shot before. The intersections of the rays with the image plane are used as texture coordinates.

It is important to note that the ray-tracing step, as well as the computation of the COP and the texture coordinates only has to be performed once as long as the geometry of the lens does not change. Only if the lens system is refocused, or the aperture of a lens element changes, for example when adjusting an aperture stop, do these calculations have to be repeated.

This algorithm has been used to render the left image in Figure 11.5. The lens system is an achromatic doublet used in real cameras in the 1920s. Its geometrical description has been taken from [Flügge55]. The right image shows the same scene rendered with distribution ray-tracing. In both images the barrel distortions of the lens are obvious. Moreover, in both images the outer regions of the film are blurred due to lens aberrations.

Our method is the first one to allow for the simulation of these effects at interactive rates: the image-based method achieves a performance of approximately 5 fps. on an SGI O2, and 16 fps. on an Onyx2 BaseReality for 10 sample points on the lens. For the ray-traced image, we used distribution ray-tracing with 10 samples per pixel, which took roughly 2 minutes for a 256×256 resolution on an Onyx2 with a 195 MHz R10k processor (the image was generated using the Vision rendering system [Slusallek95, Slusallek98]). Note that the ray-traced image does not

contain indirect illumination or shadows to allow for performance comparisons between the two algorithms.

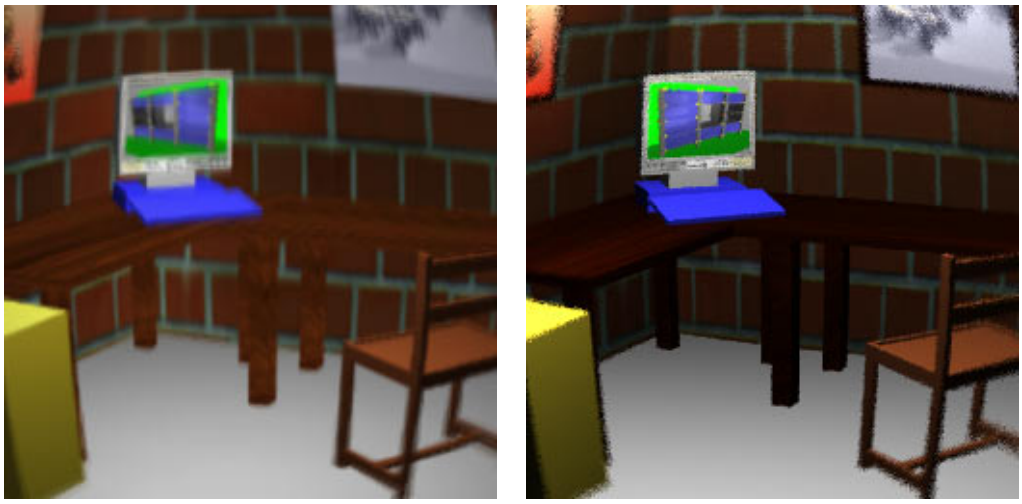

Figure 11.5: A comparison of the image-based lens model (left) with distribution ray-tracing (right) for an achromatic doublet.

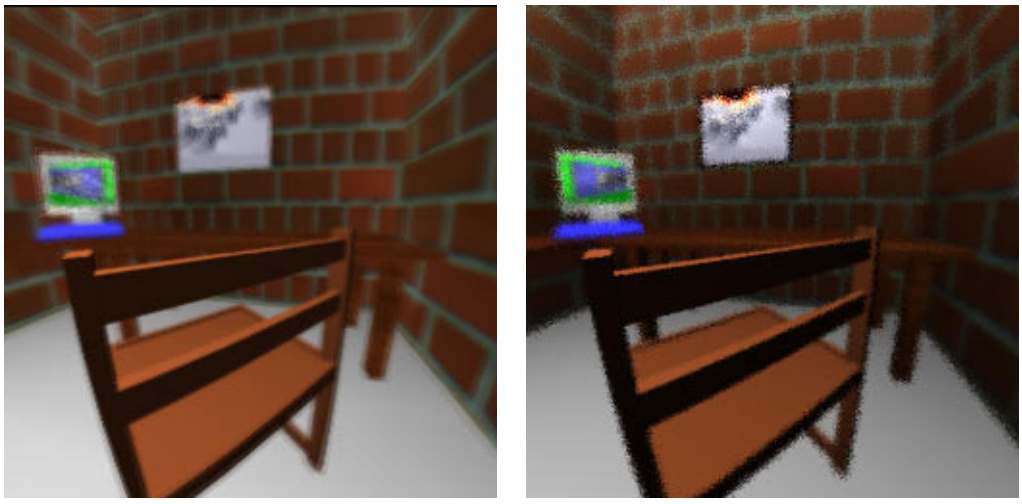

Figure 11.6: A comparison of the image-based lens model (left) with distribution ray-tracing (right) for a biconvex lens with hierarchical subdivision on the image plane.

11.2.2 Hierarchical Subdivision

Lens systems with relatively small aberrations are usually well approximated by a single projective transformation, as described above. For example, this is true for lens systems that have a reasonable thick lens approximation.

However, for lens systems with strong aberrations, an approximation with a single perspective projection introduces a large error. In these cases, the grid on the film can be recursively subdivided and refined using a quad-tree structure. The algorithm is then applied recursively. This corresponds to finding a hierarchical, piecewise linear approximation of the 2-manifold in ray-space.

All quad-tree cells corresponding to a single sample point on the lens form a partition of the film. Combined, they contain the slice of the camera light field required for rendering.

Of course, the hierarchical subdivision introduces an additional rendering cost, since the scene has to be rendered once for every subdivision. Thus, the level of subdivision is a tradeoff between rendering time and image quality. The pseudo-code for determining the perspective transformations is given below.

```
determinePerspectiveTransformation()
{
  /* Perspective Projection */
  generate rays from film plane
  compute virtual COP from the refracted rays
  if COP is good enough
  {
    determine lower left and upper right of the image by
        intersecting the rays with the image plane
  } else {
    subdivide the grid
    for each subgrid
      recurse
  }
}
```

Figure 11.6 shows the scene from Figure 11.5 using a simple, uncorrected biconvex lens. The image demonstrates the barrel distortion of the lens as well as significant depth of field effects. The scene was again rendered with 10 sample points, but this time the film plane was subdivided. As a criterion for the subdivision, we enforced a maximum angle of 0.05 degrees between an original ray and the corresponding approximated ray as described in Section 11.2.3. This resulted in a total of 40 perspective projections to be rendered for one frame. As a result, the frame rate decreased by approximately a factor of 4 to 1 frame per second on the O2 and roughly 4 frames per second on the Onyx2 BaseReality.

11.2.3 Computing the Center of Projection

The sets of rays exiting the lens on the object side represent samples of a possibly complicated transformation that characterizes the lens system. The central task in approximating this transformation with a perspective projection is to find a good virtual COP.

This problem is also known in computer vision, where the aberrations of camera lenses have to be removed based on the measured distortions on a calibration grid [Gremban88]. The approach proposed in [Gremban88] is to choose the COP so that its distance from all rays is minimal in the least-squares sense.

For our purposes, this approach works well as long as there is no hierarchical subdivision. As soon as subdivision is performed, however, intolerable discontinuities occur between adjacent quad-tree cells. An analysis of the problem shows that the angle between the original rays and the corresponding rays in the perspective projection can be relatively large.

As a solution, we found that minimizing the angle between the original rays and those generated by the perspective transformation yields much better results. This way, the maximum angle of the approximation could be reduced by a factor of up to 10. This maximum angle is also a good error criterion for terminating the hierarchical subdivision.

The minimization process, which is described in detail in [Heidrich97a], is implemented by a Newton iteration requiring the solution of a 3×3 linear equation system in each iteration step. We have found that this iteration converges quickly, so that usually two or three iterations are sufficient. It is important to note again that the computation of such an COP only has to be performed once as a preprocessing step, or whenever the lens configuration changes.

11.3 Discussion

In this chapter, we have presented an image-based model for lens systems. The model allows for the use of graphics hardware for rendering, and is therefore well-suited for high-quality, interactive rendering. As mentioned above, the model can be further extended to account for variations of the exposure over the film [Heidrich97a].

The algorithm we use for approximating the mapping between the scene light field and the camera light field relies on the fact that lens systems designed for cameras have very smooth surfaces without any high frequency detail. Thus, the mapping between the two light fields is also a smooth function which can be represented with a small number of subdivisions (Section 11.2.2). While the same hierarchical subdivision scheme could in principle be used for any refractive object, surfaces with a lot of geometric detail would require a large number of subdivisions, and therefore a lot of rendering passes, thereby making the approach infeasible. For these kinds of surfaces, it is better to use a different light field-based approach, such as the one described in

Chapter 10.

The advantage of the approach from this chapter over the one from Chapter 10 is that the latter is restricted to representing the scene as an environment map, which is not the case here. As a consequence, the methods from Chapter 10 are not capable of simulating depth of field effects. Since these effects are negligible if the refractive object is not located directly in front of the eye or camera, this is not a big restriction. In summary, it can be said that the two algorithms, although both present light field techniques for rendering refractive objects, are tailored towards specific application domains. The techniques from Chapter 10 are good for relatively distant objects with a complex surface geometry, while the model in this chapter is best for simulating lens systems of cameras with very little geometric detail.

Although our camera model is capable of simulating a larger variety of lens properties than previous models, there are some aspects of real lenses that cannot be handled. Most importantly, chromatic aberrations and frequency dependent refraction coefficients are not simulated, since contemporary computer graphics hardware only supports RGB rendering. This restriction could be removed with an increased number of rendering passes.

Another limitation of our model is the assumption of instantaneous shutter opening and a constant irradiance during the whole exposure time. More complex shutter functions and motion blur could be implemented by averaging multiple images over time. This, however would be too costly for achieving interactive frame rates on contemporary hardware.

In summary, our model adds a significant amount of realism to the simulation of lens systems in interactive graphics. Although it is not capable of simulating every property of real lenses, it constitutes a good compromise between quality of simulation and rendering performance.

Chapter 12

Conclusions and Future Work

> *In conclusion, there is a need to shift the focus of algorithm development for hardware implementation of 3D graphics. The requirements are changing from "more polygons" and "more pixel fill rate" to more complex dynamic geometry and richer pixels. More complex dynamic geometry does not necessarily mean more triangles or more efficient updates of triangle geometry, but rather better shapes and motion with less data, integrating authoring tools, APIs, and hardware accelerated rendering. Richer pixels does not necessarily mean more pixels rendered but rather that each pixel rendered is the result of far more effort spent lighting, shading, and texturing. The end result will be a higher degree of realism in the interactive experience.*
>
> David B. Kirk, *Unsolved Problems and Opportunities for High-quality, High-performance 3D Graphics on a PC Platform*, Invited Paper, 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware, pp. 11–13, August 1998.

The topic of this dissertation are algorithms for high-quality, realistic shading and lighting with the help of computer graphics hardware. Based on physical models for the interaction of light with various kinds of surfaces, and an abstract model for graphics hardware that is based on existing graphics systems, we have developed a number of algorithms to solve a variety of different problems in image synthesis.

An interesting observation is that all techniques discussed here have one thing in common: they employ a sampling-based approach to achieve certain illumination or shading effects. In Chapter 5, the lighting models are split into several factors which are then stored in 1- or 2-dimensional textures. Shadow maps, as used in Chapter 6, require a discrete sampling of the scene geometry. Bump maps and normal maps (Chapter 9) can be interpreted as discretely

sampled geometric detail, and finally environment maps (Chapter 8) and light fields (Chapters 7, 10, and 11) represent a discrete sampling of the radiance field at a certain point or set of points in space.

In particular the latter two examples, environment maps and light fields, are techniques from image-based rendering, which have here been applied to the shading and lighting of geometric objects. We believe that this sort of algorithm will play an important role in high-quality interactive rendering. At the moment, light field-based approaches push the current graphics hardware to its limits, as they either require a large amount of (texture) memory, as in Chapter 10, or many rendering passes (Chapters 7 and 11). However, the former problem will disappear as the memory prices fall, and as graphics hardware starts to use compressed textures, while the latter problem will be alleviated by increasing performance and direct multi-pass support such as multiple simultaneous textures.

There are several reasons why image-based and, more generally speaking, sample-based approaches are so dominant in this thesis, and are in general becoming popular for interactive applications. Firstly, these representations are the natural result of measurements. All the data required for any of the algorithms used in this thesis can be generated through measurements. This is true for BRDFs (see, for example, [Ward92]), normal maps [Rushmeier97], environment maps [Haeberli93, Debevec98], and light fields [Gortler96, Levoy96].

Secondly, regular samplings are well suited for use with hardware, since they are simply arrays of values. This means that no complicated data structures are involved, which would be inappropriate for hardware implementations. Moreover, often the access patterns to these arrays are also very regular, so that caching schemes can be employed. Finally, sampling based approaches allow for a wide variety of tradeoffs between quality and cost. If future generations of graphics hardware offer larger amounts of dedicated graphics memory, the sampling rate or number of quantization levels can easily be increased, which will immediately results in an improvement in the rendering quality.

An alternative to sampling-based approaches would, for example, be procedural descriptions [Hanrahan90, Pixar89, Olano98]. The two major benefits of this approach are resolution independence and a compact representation. On the other hand, these architectures require very flexible, programmable rasterization hardware, which, at the moment, is still expensive to build. For real time applications, general procedural shaders also pose problems since it is not known in advance how long a procedural shader will take for execution. In particular, the execution time may vary strongly from pixel to pixel and from frame to frame, so that load balancing between parallel rasterization units is hard. Acquisition of the data is also not easily possible, and quality/performance tradeoffs are difficult.

Despite these problems, a trend towards allowing for simple procedural shaders in graphics hardware is clearly noticeable. The problems mentioned above can likely be avoided by developing a new shading language that is specifically designed to be used with graphics hardware. Such

a language could sit on on top of the graphics pipeline, and compile to different hardware platforms, thereby making use of the specific procedural capabilities and other extensions provided by the hardware.

12.1 Suggestions for Future Graphics Hardware

Most of the algorithms proposed in this thesis work efficiently on contemporary hardware, some however, only with certain features that are currently not available on a large number of platforms. These features are

Multiple textures. Some methods, such as the use of alternative lighting models (Chapter 5) or the parabolic parameterization for environment maps (Chapter 8), can use multiple simultaneous textures [SGI97] to reduce the number of rendering passes, and thus improve the performance. For all of the algorithms presented here, multiple textures are not strictly necessary, but for some they are useful.

Imaging operations. Other algorithms, especially the methods for rendering normal maps from Chapter 9, require color matrices and color lookup tables. These are part of the so-called *imaging operations*, which have been formally introduced as a subset in OpenGL version 1.2 [Segal98]. Although support for this subset is not required for OpenGL compliance, it is to be expected that many vendors will supply it in future versions of their hardware. Currently, these operations are available as extensions on some platforms including workstations from SGI on which the algorithms in this thesis have been implemented.

An interesting point is that in this thesis the imaging operations are not used for typical "imaging" algorithms, but for shading and lighting computations. A number of other algorithms also benefit from these features without being imaging applications (see, for example [Westermann98] and [Heidrich99e]). The term "imaging subset" used in [Segal98] is thus somewhat misleading and bears the danger of underestimating the power of these operations.

Pixel textures are used for several algorithms, including normal mapping (Chapter 9) and light field based refractions (Chapter 10). Pixel textures are currently classified as an "experimental extension" from Silicon Graphics, that is only supported on one platform (Octane graphics with texture mapping support). Even this implementation is not fully compliant with the specification [SGI96].

We think that all three of these features are so useful that they should become a standard feature of future graphics platforms. As mentioned above, this is also very likely to happen, at least for the multi texture extension and the imaging subset.

The future of the pixel texture extension is much more unclear. The experimental nature of this extension is a severe limitation. Nonetheless, its usefulness, which has also been demonstrated elsewhere [Heidrich99e], makes it interesting to develop algorithms for it. The extension would be even more useful if projective textures were to be incorporated in the specification, as proposed in Chapters 6 and 9. But even without this change, pixel textures have many applications. The discussed algorithms only show a small part of them, but they demonstrate the potential of the extension for achieving high quality, high performance renderings.

In addition to applying and evaluating features of existing graphics hardware, we have also proposed a number of new extensions throughout this thesis. In short, these are

- A number of new texture generation modes in order to directly support both more advanced lighting models (Chapter 5) and parabolic environment maps (Chapter 8). The same techniques could also be applied to the illumination of bump mapped surfaces (Chapter 9). The proposal for an extension computing the reflection vector has recently been picked up by Kilgard [Kilgard99], who implemented it for the Mesa library and the nVIDIA drivers.

- An additional per-vertex tangent vector to be transformed by the model/view matrix stack. This is necessary in order to implement physically-based anisotropic reflection models.

- In Chapter 5, we have also sketched a completely new sampling-based per-vertex lighting model to replace the traditional Phong lighting. The illumination of a vertex with this approach involves several dot products and one or more table lookups, and is thus not much more expensive than per-vertex Phong lighting.

Besides these concrete proposals, future graphics hardware will have to address a number of other issues. Firstly, methods for reducing the consumption of dedicated texture and framebuffer memory are required. This includes support for compressed textures and textures that are only partially loaded into the dedicated texture memory. The former issue is partially being addressed on some new PC hardware [nVIDIA98], while a solution for the latter issue has been proposed in [Tanner98]. Anisotropic texture filtering is also important for improved rendering quality, and is beginning to appear on newer hardware.

Furthermore, hardware with direct support for bump mapping and per pixel lighting is beginning to appear [nVIDIA98]. These approaches could be combined with our techniques for physically correct lighting models and environment mapping from Chapter 9.

For further improved rendering quality, future graphics hardware should also provide high dynamic range formats for storing textures and intermediate rendering results. Examples for such formats are floating point color channels, or the LogLuv encoding [Larson97]. This is a somewhat problematic issue, since these formats require floating point arithmetic in the rasterization unit, which is not required for implementations of the rendering pipeline as laid out in

Chapter 4. However, hardware that supports bump mapping or floating point depth buffers, such as [nVIDIA98] also requires floating point arithmetic during the rasterization phase, so that the additional cost of implementing high dynamic range color formats should not be too large.

Finally, it is high time for some direct hardware support for shadows. For the reasons discussed in detail in Chapter 6, we favor shadow maps for such an implementation. Support for shadow maps can either come in form of a dedicated extension, as described in [Segal92], or they can be implemented via the alpha test with the algorithms from Chapter 6 and [Heidrich99e]. In the latter case, the hardware should support framebuffer configurations with deep alpha channels of at least 24 bit.

Also, hardware support for new geometric primitives, such as subdivision surfaces, as well a algorithms for reducing the geometric complexity through view frustum- or occlusion culling are important areas of future research. This topic, however, is outside the scope of this thesis.

12.2 Conclusion

This dissertation introduces a set of new algorithms for high quality shading and lighting using computer graphics hardware. In particular, methods for generating various local shading and lighting effects and for visualizing global illumination solutions have been presented. This includes algorithms for shadows, normal mapping, alternative material models, mirror- and glossy reflections off curved surfaces, as well as techniques for realistic lens systems and complex light source models. All these techniques are orthogonal to each other, in the sense that they can be arbitrarily combined, which, however, usually results in an increased number of rendering passes.

In the course of developing these algorithms, we have identified building blocks that are important for future generations of graphics hardware. Some of these are established features of graphics hardware that are used in a new, unexpected way, some are experimental features that are not yet widely used, and some are completely new features that we propose here for the first time.

All presented algorithms employ sampling-based approaches. Future work could extend on these ideas by further studying the use of light fields to describe the illumination on surfaces. We are confident that many more interactive techniques can be developed based on this idea.

Other areas of future research include techniques for volumetric effects and participating media. While it is true that volume rendering based on 3D texture mapping has been explored for visualization applications in medicine and engineering, the requirements for realistic image synthesis are different. Instead of communicating important properties of a data set, this application domain requires realistic effects, such as lighting and shadows cast by volumes. The area of rendering these effects efficiently with graphics hardware has not been covered sufficiently by

researchers so far.

Finally, acquisition of real-world data and models is an important topic. This is true for all areas of computer graphics, but particularly attractive for interactive rendering, since here the measured data can often directly be used with the graphics hardware, for example by applying the techniques described in this dissertation.

Bibliography

[Adelson91] E. H. Adelson and J. R. Bergen. *Computational Models of Visual Processing*, Chapter 1 (The Plenoptic Function and the Elements of Early Vision). MIT Press, Cambridge, MA, 1991.

[Akeley93] Kurt Akeley. RealityEngine graphics. In *Computer Graphics (SIGGRAPH '93 Proceedings)*, pages 109–116, August 1993.

[Ashdown93] Ian Ashdown. Near-Field Photometry: A New Approach. *Journal of the Illuminating Engineering Society*, 22(1):163–180, Winter 1993.

[Ashdown96] Ian Ashdown. Photometry and radiometry – a tour guide for computer graphics enthusiasts. Technical report, Ledalite Architectural Products, Inc., 1996.

[Banks94] David C. Banks. Illumination in diverse codimensions. In *Computer Graphics (Proceedings of SIGGRAPH '94)*, pages 327–334, July 1994.

[Barkans97] Anthony C. Barkans. High-quality rendering using the talisman architecture. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 79–88, August 1997.

[Bastos97] Rui Bastos. Efficient radiosity rendering using textures and bicubic reconstruction. In *Symposium on Interactive 3D Graphics*, 1997.

[Beckmann63] Petr Beckmann and Andre Spizzichino. *The Scattering of Electromagnetic Waves from Rough Surfaces*. McMillan, 1963.

[Bishop94] Gary Bishop, Henry Fuchs, Leonard McMillan, and Ellen J. Scher Zagier. Frameless rendering: Double buffering considered harmful. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 175–176, July 1994.

[Blinn76] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19:542–546, 1976.

[Blinn77] James F. Blinn. Models of light reflection for computer synthesized pictures. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 192–198, July 1977.

[Blinn78] James F. Blinn. Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 286–292, August 1978.

[Blinn88] James F. Blinn. Jim blinn's corner: Me and my (fake) shadow. *IEEE Computer Graphics and Applications*, 8(1):82–86, January 1988.

[Born93] Max Born and Emil Wolf. *Principles of Optics*. Pergamon Press, Oxford, 6 edition, 1993.

[Brockelmann66] R. A. Brockelmann and T. Hagfors. Note on the effect of shadowing on the backscattering of waves from a random rough surface. *IEEE Transactions on Antennas and Propagation*, 14:621–626, September 1966.

[Brotman84] L. S. Brotman and N. I. Badler. Generating soft shadows with a depth buffer algorithm. *IEEE Computer Graphics and Applications*, 4(10):71–81, October 1984.

[Bui-Tuong75] Phong Bui-Tuong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, June 1975.

[Cabral87] Brian Cabral, Nelson Max, and Rebecca Springmeyer. Bidirectional reflection functions from surface bump maps. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, pages 273–281, July 1987.

[Camahort98] Emilio Camahort, Apostolos Lerios, and Donald Fussell. Uniformly sampled light fields. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop)*, pages 117–130, March 1998.

[Cohen93] Michael F. Cohen and John R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press, 1993.

[Cohen97] Michael F. Cohen. Private communication, November 1997.

[Cohen98] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 115–122, July 1998.

[Cook81] Robert L. Cook and Kenneth E. Torrance. A reflectance model for computer graphics. In *Computer Graphics (SIGGRAPH '81 Proceedings)*, pages 307–316, August 1981.

[Cook84] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, pages 134–145, July 1984.

[Crow77] Franklin C. Crow. Shadow algorithms for computer graphics. In *Computer Graphics (SIGGRAPH '77 Proceedings)*, pages 242–248, July 1977.

[Crow84] Franklin C. Crow. Summed-area tables for texture mapping. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, pages 207–212, July 1984.

[Debevec97] Paul E. Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 369–378, August 1997.

[Debevec98] Paul E. Debevec. Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 189–198, July 1998.

[Diefenbach94] Paul J. Diefenbach and Norman Badler. Pipeline Rendering: Interactive refractions, reflections and shadows. *Displays: Special Issue on Interactive Computer Graphics*, 15(3):173–180, 1994.

[Diefenbach96] Paul J. Diefenbach. *Pipeline Rendering: Interaction and Realism Through Hardware-based Multi-Pass Rendering*. PhD thesis, University of Pennsylvania, 3401 Walnut Street, Suite 400A, Philadelphia, PA 19104-6228, June 1996.

[Ebert94] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley. *Texturing and Modeling: A Procedural Approach.* Academic Press, October 1994.

[Encarnação80] J. Encarnação, G. Enderle, K. Kansy, G. Nees, E. G. Schlechtendahl, J. Weiss, and P. Wisskirchen. The workstation concept of GKS and the resulting conceptual differences to the GSPC CORE system. In *Computer Graphics (SIGGRAPH '80 Proceedings)*, pages 226–230, July 1980.

[Ernst98] I. Ernst, H. Rüsseler, H. Schulz, and O. Wittig. Gouraud bump mapping. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 47–54, 1998.

[Eyles97] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. PixelFlow: The realization. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 57–68, August 1997.

[Flügge55] Johannes Flügge. *Das Photographische Objektiv*. Springer Wien, 1955.

[Foley90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990.

[Glassner95] Andrew Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, 1995.

[Gortler96] Steven J. Gortler, Radek Grzeszczuk, Richard Szelinski, and Michael F. Cohen. The Lumigraph. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 43–54, August 1996.

[Greene86] Ned Greene. Applications of world projections. In *Proceedings of Graphics Interface '86*, pages 108–114, May 1986.

[Gremban88] Keith D. Gremban, Charles E. Thorpe, and Takeo Kanade. Geometric camera calibration using systems of linear equations. In *IEEE International Conference on Robotics and Animation*, volume 1, pages 562–567, 1988.

[GSPC79] Graphics Standards Planning Committee. Status report of the graphics standards planning committee. In *Computer Graphics (SIGGRAPH '79 Proceedings)*, page 274, August 1979.

[Gu97] Xianfeng Gu, Steven J. Gortler, and Michael F. Cohen. Polyhedral geometry and the two-plane parameterization. In *Rendering Techniques '97 (Proceedings of Eurographics Rendering Workshop)*, pages 1–12, June 1997.

[Haeberli90] Paul E. Haeberli and Kurt Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, pages 309–318, August 1990.

[Haeberli93] Paul Haeberli and Mark Segal. Texture mapping as a fundamental drawing primitive. In *Fourth Eurographics Workshop on Rendering*, pages 259–266, June 1993.

[Hall89] Roy Hall. *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, New York, 1989.

[Hanrahan90] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, pages 289–298, August 1990.

[Hanrahan93] Pat Hanrahan. *Radiosity and Realistic Image Synthesis*, Chapter Rendering Concepts. Academic Press, 1993.

[Hansen97] Paul Hansen. Introducing pixel texture. In *Developer News*, pages 23–26. Silicon Graphics Inc., May 1997.

[He91] X. D. He, K. E. Torrance, F. X. Sillion, and D. P. Greenberg. A comprehensive physical model for light reflection. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, pages 175–186, July 1991.

[Heidrich97a] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. An image-based model for realistic lens systems in interactive computer graphics. In *Graphics Interface '97*, pages 68–75, 1997.

[Heidrich97b] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic (technical sketch). In *SIGGRAPH '97 Visual Proceedings*, 1997.

[Heidrich98a] Wolfgang Heidrich, Jan Kautz, Philipp Slusallek, and Hans-Peter Seidel. Canned lightsources. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop)*, 1998.

[Heidrich98b] Wolfgang Heidrich and Hans-Peter Seidel. Efficient rendering of anisotropic surfaces using computer graphics hardware. In *Proceedings of the Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP)*, 1998.

[Heidrich98c] Wolfgang Heidrich and Hans-Peter Seidel. A model for anisotropic reflections in OpenGL (technical sketch). In *SIGGRAPH '98 Visual Proceedings*, 1998.

[Heidrich98d] Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 39–45, 1998.

[Heidrich98e] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics*, pages 158–176, 1998.

[Heidrich99a] Wolfgang Heidrich and Hendrik Lensch. Direct rendering of vector-quantized light fields with graphics hardware. Technical report, University of Erlangen-Nürnberg, Computer Graphics Group, 1999. in preparation.

[Heidrich99b] Wolfgang Heidrich, Hendrik Lensch, Michael F. Cohen, and Hans-Peter Seidel. Light field techniques for reflections and refractions. In *Rendering Techniques '99 (Proceedings of Eurographics Rendering Workshop)*, 1999.

[Heidrich99c] Wolfgang Heidrich, Hartmut Schirmacher, and Hans-Peter Seidel. A warping-based refinement of lumigraphs. In *Proceedings of WSCG*, 1999.

[Heidrich99d] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Computer Graphics (SIGGRAPH '99 Proceedings)*, August 1999.

[Heidrich99e] Wolfgang Heidrich, Rüdiger Westermann, Hans-Peter Seidel, and Thomas Ertl. Applications of pixel textures in visualization and realistic image synthesis. In *ACM Symposium on Interactive 3D Graphics*, 1999.

[Kajiya85] James T. Kajiya. Anisotropic reflection models. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, pages 15–21, August 1985.

[Kajiya86] James T. Kajiya. The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, pages 143–150, August 1986.

[Keller97] Alexander Keller. Instant radiosity. *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 49–56, August 1997.

[Kilgard97] Mark J. Kilgard. Realizing OpenGL: Two implementations of one architecture. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, 1997.

[Kilgard99] Mark Kilgard. Personal communication, April 1999.

[Kolb95] Craig Kolb, Don Mitchell, and Pat Hanrahan. A realistic camera model for computer graphics. In *Computer Graphics (SIGGRAPH '95 Proceedings)*, pages 317–324, August 1995.

[Lafortune97] Eric P. F. Lafortune, Sing-Choong Foo, Kenneth E. Torrance, and Donald P. Greenberg. Non-linear approximation of reflectance functions. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 117–126, August 1997.

[Lalonde97a] Paul Lalonde and Alain Fournier. Filtered local shading in the wavelet domain. In *Rendering Techniques '97 (Proceedings of Eurographics Rendering Workshop)*, pages 163–174, June 1997.

[Lalonde97b] Paul Lalonde and Alain Fournier. Generating reflected directions from BRDF data. *Computer Graphics Forum (Proceedings of Eurographics '97)*, 16(3):293–300, August 1997.

[Larson97] Gregory Larson. LogLuv encoding for TIFF images. Technical report, Silicon Graphics, 1997. http://www.sgi.com/Technology/pixformat/tiffluv.html.

[Lengyel97] Jed Lengyel and John Snyder. Rendering with coherent layers. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 233–242, August 1997.

[Levoy96] Marc Levoy and Pat Hanrahan. Light field rendering. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 31–42, August 1996.

[Lewis93] Robert R. Lewis. Making shaders more physically plausible. In *Fourth Eurographics Workshop on Rendering*, pages 47–62, June 1993.

[Loos98] Joachim Loos, Philipp Slusallek, and Hans-Peter Seidel. Using wavefront tracing for the visualization and optimization of progressive lenses. In *Eurographics '99*, pages 255–266, September 1998.

[McReynolds98] Tom McReynolds, David Blythe, Brad Grantham, and Scott Nelson. Advanced graphics programming techniques using OpenGL. In *SIGGRAPH 1998 Course Notes*, July 1998.

[Miller98a] Gavin Miller, Mark Halstead, and Michael Clifton. On-the-fly texture computation for real-time surface shading. *IEEE Computer Graphics & Applications*, 18(2):44–58, March–April 1998.

[Miller98b] Gavin Miller, Steven Rubin, and Dulce Ponceleon. Lazy decompression of surface light fields for precomputed global illumination. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop)*, pages 281–292, March 1998.

[Molnar92] Steven Molnar, John Eyles, and John Poulton. PixelFlow: High-speed rendering using image composition. In *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 231–240, July 1992.

[Montrym97] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal. InfiniteReality: A real-time graphics system. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 293–302, August 1997.

[Myskowski94] Karol Myskowski and Tosiyasu. L. Kunii. Texture mapping as an alternative for meshing during walkthrough animation. In *Photorealistic Rendering Techniques*, pages 389–400. Springer, June 1994.

[Nayar97] Shree K. Nayar. Catadioptric omnidirectional camera. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 482–488, June 1997.

[Nayar98] Shree Nayar. Omnidirectional image sensing. Invited Talk at the 1998 Workshop on Image-Based Modeling and Rendering, March 1998.

[Neider93] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL Programming Guide*. Addison Wesley, 1993.

[nVIDIA98] nVIDIA. *RIVA TNT Product Overview*, 1998. http://www.nvidia.com.

[Ofek98] Eyal Ofek and Ari Rappoport. Interactive reflections on curved objects. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 333–342, July 1998.

[Olano98] Marc Olano and Anselmo Lastra. A shading language on graphics hardware: The PixelFlow shading system. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 159–168, July 1998.

[Paul94] Brian Paul. Mesa web site. http://www.ssec.wisc.edu/ brianp/Mesa.html, 1994.

[Pedrotti93] F. Pedrotti and L. Pedrotti. *Introduction to Optics*. Prentice Hall, second edition, 1993.

[Peercy97] Mark Peercy, John Airey, and Brian Cabral. Efficient bump mapping hardware. In *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 303–306, August 1997.

[Perlin89] Ken Perlin and Eric M. Hoffert. Hypertexture. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, pages 253–262, July 1989.

[Pixar89] Pixar. *The RenderMan Interface*. Pixar, San Rafael, CA, September 1989.

[Poulin90] Pierre Poulin and Alain Fournier. A model for anisotropic reflection. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 273–282, August 1990.

[Rushmeier97] Holly Rushmeier, Gabriel Taubin, and André Guéziec. Applying shape from lighting variation to bump map capture. In *Rendering Techniques '97 (Proceedings of Eurographics Rendering Workshop)*, pages 35–44, June 1997.

[Schilling96] Andreas Schilling, Günter Knittel, and Wolfgang Straßer. Texram: A smart memory for texturing. *IEEE Computer Graphics and Applications*, 16(3):32–41, May 1996.

[Schilling97] Andreas Schilling. Toward real-time photorealistic rendering: Challenges and solutions. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 7–16, August 1997.

[Schlick93] Christophe Schlick. A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering*, pages 73–83, June 1993.

[Segal92] Marc Segal, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. Fast shadow and lighting effects using texture mapping. *Computer Graphics (SIGGRAPH '92 Proceedings)*, 26(2):249–252, July 1992.

[Segal98] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2)*, 1998.

[SGI96] Silicon Graphics Inc. *Pixel Texture Extension*, December 1996. Specification document, available from http://www.opengl.org.

[SGI97] Silicon Graphics Inc. *Multitexture Extension*, September 1997. Specification document, available from http://www.opengl.org.

[Shinya94] Mikio Shinya. Post-filtering for depth of field simulation with ray distribution buffer. In *Proceedings of Graphics Interface '94*, pages 59–66, May 1994.

[Sillion89] Francois X. Sillion and Claude Puech. A general two-pass method integrating specular and diffuse reflection. In *Computer Graphics (SIGGRAPH '89 Proceedings)*, pages 335–344, July 1989.

[Sillion91] Francois X. Sillion, James R. Arvo, Stephen H. Westin, and Donald P. Greenberg. A global illumination solution for general reflectance distributions. In *Computer Graphics (SIGGRAPH '91 Proceedings)*, pages 187–196, July 1991.

[Sillion94] Francois X. Sillion and Claude Puech. *Radiosity & Global Illumination*. Morgan Kaufmann, 1994.

[Sloan97] Peter-Pike Sloan, Michael F. Cohen, and Steven J. Gortler. Time critical Lumigraph rendering. In *Symposium on Interactive 3D Graphics*, 1997.

[Sloan99] Peter-Pike Sloan. Private communication, January 1999.

[Slusallek95] Philipp Slusallek and Hans-Peter Seidel. Vision: An architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics*, 1(1):77–96, March 1995.

[Slusallek98] Philipp Slusallek, Marc Stamminger, Wolfgang Heidrich, Jan-Christian Popp, and Hans-Peter Seidel. Composite lighting simulations with lighting networks. *IEEE Computer Graphics and Applications*, 18(2):22–31, March 1998.

[Smith67] Bruce G. Smith. Geometrical shadowing of a random rough surface. *IEEE Transactions on Antennas and Propagation*, 15(5):668–671, September 1967.

[Snyder98] John Snyder and Jed Lengyel. Visibility sorting and compositing without splitting for image layer decomposition. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 219–230, July 1998.

[Stalling97] Detlev Stalling, Malte Zöckler, and Hans-Christian Hege. Fast display of illuminated field lines. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):118–128, 1997.

[Stamminger95] Marc Stamminger, Philipp Slusallek, and Hans-Peter Seidel. Interactive walkthroughs and higher order global illumination. In *Modeling, Virtual Worlds, Distributed Graphics*, pages 121–128, November 1995.

[Stürzlinger97] Wolfgang Stürzlinger and Rui Bastos. Interactive rendering of globally illuminated glossy scenes. In *Rendering Techniques '97*, pages 93–102, 1997.

[Tanner98] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: A virtual mipmap. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 151–158, July 1998.

[Torborg96] Jay Torborg and Jim Kajiya. Talisman: Commodity Real-time 3D graphics for the PC. In *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 353–364, August 1996.

[Torrance66] Kenneth E. Torrance, E. M. Sparrow, and R. C. Birkebak. Polarization, directional distribution, and off-specular peak phenomena in light reflected from roughened surfaces. *Journal of the Optical Society of America*, 56(7):916–925, July 1966.

[Torrance67] Kenneth E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces. *Journal of the Optical Society of America*, 57(9):1105–1114, September 1967.

[Tsang98] Glenn Tsang, Sherif Ghali, Eugene L. Fiume, and Anastasios N. Venetsanopoulos. A novel parameterization of the light field. In *Proceedings of the Image and Multi-dimensional Digital Signal Processing Workshop (IMDSP)*, 1998.

[Voorhies94] D. Voorhies and J. Foran. Reflection vector shading hardware. In *Computer Graphics (SIGGRAPH '94 Proceedings)*, pages 163–166, July 1994.

[Walter97] Bruce Walter, Gün Alppay, Eric Lafortune, Sebastian Fernandez, and Donald P. Greenberg. Fitting virtual lights for non-diffuse walkthroughs. *Computer Graphics (SIGGRAPH '97 Proceedings)*, pages 45–48, August 1997.

[Ward92] Gregory J. Ward. Measuring and modeling anisotropic reflection. *Computer Graphics (SIGGRAPH '92 Proceedings)*, pages 265–273, July 1992.

[Westermann98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH '98 Proceedings)*, pages 169–178, July 1998.

[Williams78] Lance Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, August 1978.

[Williams83] Lance Williams. Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, pages 1–11, July 1983.

[Zöckler96] Malte Zöckler, Detlev Stalling, and Hans-Christian Hege. Interactive visualization of 3D-vector fields using illuminated stream lines. In *IEEE Visualization '96*, pages 107–113, 1996.

Deutschsprachige Teile

Inhaltsverzeichnis

Einleitung

> *Interaktive Graphik ist ein Gebiet dessen Zeit gekommen ist. Bis vor kurzem war sie eine esoterische Spezialität, die teuere Hardware, beachtliche Computerresourcen und eigentümliche Software erforderte. In den letzten Jahren hat sie jedoch von der kontinuierlichen und manchmal spektakulären Reduktion des Preis/Leistungsverhältnisses für Hardware (z.B. für PCs mit ihren standardmäßig verfügbaren graphischen Displays), sowie von der Entwicklung von mächtigen, Hardware-unabhängigen Graphikpaketen profitiert, die eine rationale und einfache Graphikprogrammierung ermöglichen.*
>
> James Foley und Andries van Dam, Vorwort zu *Fundamentals of Interactive Computer Graphics*, 1982.

In den frühen 1980er Jahren, als dieses Zitat zu Papier gebracht wurde, charakterisierte es es eine Situation, in der die zuvor dominierende Vektorgraphik zügig durch Rastergraphik ersetzt wurde, und sich die ersten Graphikstandards wie Core [GSPC79] und GKS [Encarnação80] entwickelten. Obwohl weder der IBM PC, noch der Apple Macintosh zu dieser Zeit bereits eingeführt waren, wurde die Framebufferhardware immer preiswerter, zweidimensionale Lininengraphik wurde auf vergleichsweise billigen Systemen verfügbar, und graphische Benutzerschnittstellen erschienen. Die Zeit für interaktive Graphik war in der Tat gekommen.

Etwa 15 Jahre später, in der Mitte der 1990er, ergab sich eine ähnliche Situation für dreidimensionale Graphik. Allm”ahlich konnten Personalcomputer schattierte, beleuchtete und texturierte Dreiecke hinreichend schnell darstellen, um für praxisrelevante Aufgaben eingesetzt zu werden. Dreidimensionale Graphik wurde zunehmend im Massenmarkt eingesetzt, zunächst nur im Spielebereich, bald aber auch im Bereich betriebswirtschaftlicher und ingenieurwissenschaftlicher Anwendungen. Heute, im Jahre 1999, wird kaum ein Computer mehr ohne substantielle Spezialhardware für dreidimensionale Graphik verkauft.

Die Reduktion des Preis/Leistungsverhältnisses läßt sich vor allem auf zwei Tatsachen zurückführen. Erstens sind alle modernen Prozessoren schnell genug, um Geometrieverarbeitung

in Software durchzuführen. Das hat zur Folge, daß spezielle Geometrieeinheiten für kleine und mittlere Systeme nicht mehr erforderlich sind. Zweitens gingen die Preise für Speicherchips in den letzten Jahren drastisch zurück. Dies hat es ermöglicht, signifikante Mengen von dezidiertem Graphik- und Texturspeicher eng an die Rasterisierungshardware anzukoppeln.

Auf der Softwareseite ersetzte der de-facto-Standard OpenGL zu großen Teilen proprietäre Bibliotheken wie Starbase (Hewlett Packard), Iris GL (Silicon Graphics) und XGL (Sun Microsystems). Dadurch isr es Programmierern inzwischen möglich, graphische Anwendungen für eine Vielzahl von Plattformen zu entwickeln, wodurch die Entwicklungskosten für solche Anwendungen dramatisch reduziert werden konnten. Das Thema der Programmierschnittstellen wird weiter unten genauer betrachtet.

Das Hauptaugenmerk bei der Entwicklung neuer Hardware galt bis vor kurzem der schnelleren Abarbeitung der traditionellen Graphikpipeline. Heute sind selbst auf Geräten der unteren Preisklasse mehrere Millionen texturierte, beleuchtete Dreiecke pro Sekunde erreichbar. Das hat zur Auswirkung, daß sich der Schwerpunkt von höherer Geschwindigkeit hin zu höherer Qualität und erweiterter Funktionalität verlagert, wodurch die Hardware für eine vollkommen neue Klasse von Anwendungen eingesetzt werden kann.

Diese Dissertation stellt eine Reihe neuer Algorithmen für die qualitativ hochwertige Beleuchtungsberechnung und Schattierung unter Zuhilfenahme von Graphikhardware vor. Im Verlauf der Entwicklung dieser Algorithmen identifizieren wir Bausteine, die wir als wichtig für zukünftige Generationen von Graphikhardware ansehen. Einige davon sind etablierte Funktionen der Graphikhardware, die in einer innovativen, unerwarteten Weise eingesetzt werden. Andere sind experimentelle Funktionen, die noch nicht weit verbreitet sind, und wieder andere werden von uns neu eingeführt.

Die Arbeit trägt daher auf zwei Arten zum Gebiet der Computergraphik bei: einerseits wird eine Reihe neuer Algorithmen für die realistische Bildsynthese unter Zuhilfenahme von Graphikhardware vorgestellt. Diese sind in der Lage, Bilder von einer Qualität wie sie einfachen Raytracern entspricht, in interaktiver Geschwindigkeit auf heutigen Graphiksystemen zu erzeugen. Andererseits definiert und identifiziert sie Funktionen und Bausteine, welche für die realistische Beleuchtungsberechnung und Schattierung wichtig sind, und trägt damit dazu bei, daß in der Zukunft bessere Hardware gebaut werden kann.

Architekturen für Graphikhardware

Als Grundlage für die Diskussion von Algorithmen, welche Graphikhardware ausnutzen, und zur Einführung zukünftiger Erweiterungen, ist es hilfreich durch ein Modell von einer spezifischen Implementierung zu abstrahieren. Der oben erwähnte Standardisierungsprozeß hat ein solches abstraktes Modell, die *Renderingpipeline*, hervorgebracht. Der überwiegende Teil der im Augen-

blick verfügbaren Graphikhardware basiert auf diesem Modell, welches wir in Kapitel 4 genauer beschreiben. Es hat sich auch gezeigt, daß die Rendering Pipeline flexibel genug ist, um neue Funktionalität zu integrieren, ohne bestehende Anwendungen unbrauchbar zu machen.

In den vergangenen Jahren gab es auch eine Reihe von Forschungsprojekten zu alternativen Graphikarchitekturen. Die wichtigsten von ihnen sind:

Frameless Rendering: Eines der Probleme herkömmlicher interaktiver Bildsynthesesysteme ist, daß Veränderungen erst dann sichtbar werden, wenn bereits die ganze Szene gezeichnet wurde. Dies fällt insbesondere dann negativ auf, wenn die Szene zu groß ist, um mit interaktiven Bildwiederholraten dargestellt werden zu können. Die dadurch entstehende Verzögerung zwischen Interaktion und visueller Antwort verursacht beim Betrachter ein Schwindelgefühl welches als *Motion Sickness* bekannt geworden ist. Dies gilt insbesondere, wenn immersive Ausgabegeräte wie etwa Head-Mounted Displays verwendet werden.

Frameless rendering [Bishop94] hat zum Ziel, dieses Problem durch die Darstellung von Detailergebnissen der Benutzerinteraktion in den Griff zu bekommen. Zufällig ausgewählte Pixel auf dem Bildschirm werden auf den neuesten Stand gebracht, wobei die jeweils neueste Objekt- und Augposition Verwendung findet. Dies bewirkt, daß das Bild nach einer starken Bewegung verrauscht oder unscharf wirkt, aber schnell gegen das eigentliche Bild konvergiert, sobald die Bewegung aufhört. Da jede Interaktion durch die Änderung einzelner Pixel unmittelbar sichtbar wird, wird die Gefahr der Motion Sickness dramatisch reduziert. Gleichzeitig scheinen erste Studien zu belegen, daß das menschliche visuelle System nicht übermäßig durch das Rauschen und die Unschärfe irritiert wird.

Der große Nachteil von Frameless Rendering ist der Verlust an Kohärenz. Wo herkömmliche Graphiksysteme effiziente Scanlinetechniken verwenden können, um die Scankonvertierung von Dreiecken durchzuführen, ist Raycasting praktisch die einzige Möglichkeit, um zufällig ausgewählte Pixel auf den neuesten Stand zu bringen. Daher steht zu erwarten, daß leistungsfähigere Hardware benötigt wird, um Pixelfüllraten zu erreichen, wie sie mit herkömmlicher Hardware erreichbar sind. Zudem scheint Frameless Rendering ungeeignet für alle Anwendungen, bei denen der Anwender feine Details identifizieren oder verfolgen muß. Solche Details sind nur dann erkennbar, wenn für eine gewisse Zeit keine Bewegung stattfindet.

Im Augenblick ist keine Hardwareimplementierung von Frameless Rendering bekannt. In [Bishop94] wurde eine Softwaresimulation verwendet, um das Konzept zu bewerten.

Talisman: Das Talisman-Projekt ist eine Initiative von Microsoft, welche auf das untere Preissegment abzielt. Es unterscheidet sich von der herkömmlichen Renderingpipeline durch zwei wesentliche Punkte. Erstens wird, anstatt die Szenendatenbank nur einmal pro Bild

zu durchlaufen und dabei alle Polygone unabhängig von ihrer Bildschirmposition zu zeichnen, der Framebuffer in Rechtecke von 32×32 Pixeln unterteilt. Jedes Rechteck besitzt eine Liste der geometrischen Primitive, die in ihm sichtbar sind. Diese Liste wird in einem Vorverarbeitungsschritt für jedes Bild neu erzeugt. Durch diesen Ansatz wird dezidierter Graphikspeicher eingespart, da der Tiefenpuffer und andere, nicht direkt sichtbare Speicherbereiche wie etwa der Alphakanal von allen Rechtecken gemeinsam genutzt werden können. Aufgrund der fallenden Speicherpreise ist allerdings fraglich, wie signifikant dieser Vorteil tatsächlich ist.

Der andere große Unterschied zu herkömmlichen Systemen ist das Konzept der Wiederverwendung zuvor gezeichneter Bildteile für neue Bilder. Zu diesem Zweck wird die Szene in eine Reihe unabhängiger Schichten zerlegt, welche von hinten nach vorne sortiert werden. Diese Schichten werden separat voneinander gezeichnet und nachträglich zu einem Gesamtbild zusammengefügt. Wenn sich der Augpunkt ändert und sich Objekte in nachfolgenden Bildern bewegen, kann ein gewisser Teil der Ebenen wiederverwendet werden, indem zweidimensionale affine Transformationen auf sie angewendet werden. Erst wenn der Fehler, der durch dieses Verfahren verursacht wird, eine gewisse Schranke überschreitet, muß die Geometrie neu gezeichnet werden.

Die Schwierigkeit im Zusammenhang mit der Talisman-Architektur ist es, eine gute Unterteilung in Schichten zu finden, und den Fehler zuverlässig abzuschätzen, der dadurch entsteht, daß die bestehenden Schichten wiederverwendet werden, anstatt die Geometrie neu zu zeichnen. Es ist auch wichtig, plötzliche Änderungen zwischen wiederverwendeten und neugezeichneten Schichten zu vermeiden. In all diesen Bereichen hat es seit der ersten Vorstellung des Talisman Projektes einige Fortschritte gegeben [Lengyel97, Snyder98], aber einige Fragen sind noch immer offen. Eine davon ist, wie eine Programmierschnittstelle für diese Architektur aussehen sollte. Keine der bekannten Schnittstellen scheint ohne deutliche Veränderungen für die Unterteilung in Rechtecke und Schichten geeignet. Diese Veränderungen würden wiederum zu Inkompatibilitäten mit bestehenden Anwendungen führen. Im Moment existiert keine kommerziell verfügbare Implementierung der Talisman-Architektur, allerdings scheint Microsoft zusammen mit anderen Firmen an einer solchen zu arbeiten.

PixelFlow: Das auf Bildkomposition basierende PixelFlow-Projekt [Molnar92] ist wohl eine der vielversprechendsten Alternativen zur Renderingpipeline. Die Szenendatenbank wird gleichmäßig auf eine Reihe von Graphikkarten aufgeteilt, welche jeweils sowohl eine Geometrieeinheit als auch einen Scankonverter enthalten. Im Gegensatz zum Talisman-Projekt ist diese Unterteilung beliebig und nicht an die Tiefensortierung in Schichten gebunden. Jede der Graphikkarten zeichnet ein komplettes Bild ihrer Geometrie in voller Bildschirmauflösung. Wie im Talisman-Projekt wird auch hier der Framebuffer in Rechtecke unterteilt, diesmal in einer Größe von 128×128 Pixeln. Für jedes Pixel in einem

Rechteck existiert ein separater Pixelprozessor. In SIMD-Art wird so jedes Dreieck parallel scankonvertiert.

Jedes fertige Rechteck wird dann mit dem entsprechen Rechteck der vorhergehenden Graphikkarte verknüpft, und an die nächste Karte weitergereicht. Der Vorteil dieses Ansatzes liegt darin begründet, daß die benötigte Bandbreite unabhängig von der Komplexität der Szene ist, und daß das System durch das Hinzufügen neuer Karten gut skaliert.

Die Punkte, in denen sich PixelFlow am deutlichsten von anderen Ansätzen unterscheidet, sind die verzögerte und prozedurale Schattierung. Während in der normalen Renderingpipeline die Beleuchtung von Polygonen nur an den Knoten berechnet wird, und die resultierenden Farbwerte dann interpoliert werden, interpoliert die verzögerte Beleuchtungsberechnung die Normalen sowie andere Informationen, und berechnet die Beleuchtung dann separat für jedes Pixel. Wenn dies mit prozeduralen Oberflächenbeschreibungen kombiniert wird, ergibt sich die Möglichkeit zur Generierung äußerst komplexer und realistischer Bilder. Einige der Effekte, die in dieser Arbeit vorgestellt werden, sind, zumindest prinzipiell, einfach auf PixelFlow zu implementieren [Olano98].

Allerdings hat auch die PixelFlow-Architektur Nachteile. Die verzögerte Beleuchtungsberechnung erhöht die benötigte Bandbreite. Zudem erhöht sich durch das Konzept der Bildkomposition und die Unterteilung in Rechtecke die Latenz, so daß Motion Sickness zum Problem werden kann. Antialiasing und Transparenz sind mit dieser Architektur schwierig zu erreichen, da die Schattierung erst nach der Verknüpfung der Rechtecke aller Graphikkarten erfolgt. Dies hat zur Folge, daß Information über Pixel hinter teilweise transparenten Objekten zum Zeitpunkt der Schattierung nicht mehr zur Verfügung steht. Zwar gibt es Verfahren, um transparente Objekte sowie Primitive mit Antialiasing zu zeichnen, jedoch erhöhen diese weiter die Latenz, und beeinträchtigen auch die Performanz.

Ein letzter Nachteil, der sowohl PixelFlow als auch die anderen beiden Architekturen betrifft, ist die Notwendigkeit, die Szenendatenbank explizit auf der Graphikkarte vorzuhalten. Nicht nur, daß dies die Speicheranforderungen auf der Graphikkarte deutlich erhöht (Szenen, welche nicht in diesen Speicher passen, können nicht verarbeitet werden), es ergibt sich auch eine niedrigere Performanz und eine höhere Latenz für hochgradig dynamische Szenen (Immediate Mode Rendering). Natürlich treffen aufgrund der Unterteilung in Rechtecke auch die Hinweise zu Programmierschnittstellen zu, die im Hinblick auf das Talisman Projekt gegeben wurden.

Dennoch ist das PixelFlow-Projekt eine vielversprechende neue Architektur, welche mit hoher Wahrscheinlichkeit andere Neuentwicklungen bei der Graphikhardware beinflußt. Bei Fertigstellung dieser Arbeit existieren einige Prototypen von PixelFlow, welche von der Universität von Nord Carolina in Chapel Hill und von Hewlett Packard realisiert wur-

den. Eine Kommerzialisierung durch Hewlett Packard wurde jedoch abgebrochen.

Die vorangehende Diskussion zeigt, daß, obwohl Forschungsarbeiten im Bereich alternativer Graphikhardware durchgeführt werden, Systeme, die auf der herkömmlichen Renderingpipeline beruhen, noch für eine Weile die Mehrzahl der kommerziellen Systeme stellen werden. Daher ist es sinnvoll, sich mit der Entwicklung von Erweiterungen dieser Pipeline auseinanderzusetzten, mit denen sowohl eine Leistungssteigerung als auch eine verbesserte Realitätsnähe erreicht werden kann. Die Vergangenheit hat gezeigt, daß dies oft ohne inkompatible Änderungen durch Hinzufügen neuer Funktionalität in der entsprechenden Stufe der Pipeline möglich ist. Diese Flexibilität der Renderingpipeline zusammen mit ihrer großen Verfügbarkeit ist auch der Grund, warum wir sie als Basis für diese Arbeit gewählt haben.

Programmierschnittstellen

Programmierschnittstellen haben einen wichtigen Beitrag zu der Verbreitung interaktiver 3-dimensionaler Computergraphik geleistet. Obwohl Abhängigkeiten von einer konkreten Schnittstelle in dem Rest dieser Arbeit weitestgehend vermieden werden, scheint es dennoch angebracht, an dieser Stelle einige der damit verbundenen Problemstellungen zu diskutieren.

Die Programmierschnittstellen für Graphikhardware können in drei Kategorien unterteilt werden. Auf der untersten Ebene gibt es die sogenannten *Immediate-Mode*-Schnittstellen, welche als Hardwareabstraktionsschicht dienen und eine oft nur sehr dünne Schicht über der Graphikhardware bilden. Die nächsthöhere Schicht ist die der *Retained-Mode*-, oder Szenengraphschnittstelle, welche die Szene in der Form eines ungerichteten, azyklischen Graphen (DAG) speichert. Auf der höchsten Abstraktionsebene finden sich schließlich Schnittstellen für die Handhabung großer Modelle, welche Freiformflächen wie etwa NURBS und Subdivision-Flächen verarbeiten. Diese führen auch eine Polygonreduktion und andere Optimierungen der Geometrie, wie etwa Viewfrustum- oder Occlusionculling, durch.

Für die Zwecke dieser Dissertation sind Szenengraphschnittstellen und Schnittstellen für große Modelle von untergeordneter Bedeutung, da die hier betrachteten Verfahren direkt auf der zugrundeliegenden Hardware aufbauen. Im Bereich der Immediate-Mode-Schnittstellen fand in den letzten Jahren ein beträchtlicher Standardisierungsprozeß statt. Auf Workstations sowie im Bereich professioneller PC Anwendungen wurden die vorhergehenden proprietären Schnittstellen wie Starbase (Hewlett Packard), Iris GL (Silicon Graphics) und XGL (Sun Microsystems) schrittweise durch den Industriestandard OpenGL ersetzt. Zur gleichen Zeit hat sich die Direct 3D Immediate-Mode-Schnittstelle vor allem für die Spieleentwicklung auf PCs etabliert.

Da beide Schnittstellen auf der Annahme basieren, daß die zugrundeliegende Hardware auf dem Konzept der Renderingpipeline (siehe auch Kapitel 4) aufbaut, ist es nicht weiter verwun-

derlich, daß sich beider Funktionalität mit den letzten Versionen immer weiter aneinander angenähert hat. Noch existierende Unterschiede im Funktionsumfang beruhen weniger auf prinzipiellen Differenzen, sondern auf unterschiedlichen Anforderungen der jeweiligen Marktsegmente, auf die die Schnittstellen abzielen.

Diese Tatsache erlaubt es uns, die Diskussion in den nachfolgenden Kapitel weitgehend unabhängig von einer konkreten Schnittstelle zu gestalten. Obwohl das abstrakte System, welches als Grundlage für unsere Betrachtungen dient (siehe Kapitel 4), im wesentlichen auf der Definition von OpenGL [Segal98] beruht, sind praktisch alle Ergebnisse auch auf andere Immediate-Mode-Schnittstellen übertragbar. OpenGL wurde ausgewählt, weil es eine offene Struktur besitzt, und zudem gut spezifiziert und dokumentiert ist.

Kapitelüberblick

Der Rest dieser Arbeit ist wie folgt aufgebaut. In Kapitel 2 werden kurz die physikalischen Grundlagen der Bildsynthese zusammengefaßt. Kapitel 3 diskutiert dann andere relevante Arbeiten, auf denen diese Dissertation aufbaut, und Kapitel 4 definiert den Funktionsumfang der Graphikhardware, den wir im Verlauf der Arbeit voraussetzen.

Anschließend besprechen wir eine Reihe von Verfahren, die auf ihre Art zu erhöhter Realitätstreue synthetischer Bilder beitragen. Den Anfang macht in Kapitel 5 eine Beschreibung von Techniken zur Verwendung hochwertiger, physikalisch basierter Beleuchtungsmodelle zur lokalen Beleuchtungsberechnung mit Graphikhardware. Wir diskutieren insbesondere anisotrope Reflexionen und die Verwendung des Torrance-Sparrow Modells.

Diesem Kapitel folgt eine Beschreibung von Schattenalgorithmen in Kapitel 6. Dazu gehören insbesondere projizierte Geometrie, Schattenvolumina und Schattenkarten. Wir führen einen neuen Algorithmus zur Behandlung von Schattenkarten ein, basierend auf dem, in Kapitel 4 vorgestellten Funktionsumfang von Graphikhardware. Darauf folgend präsentieren wir ein auf Lichtfeldern basierendes Modell zur Repräsentation komplexer Lichtquellen in Kapitel 7.

Reflexionen und Refraktionen basierend auf Umgebungskarten sind dann das Thema in Kapitel 8. Dies beinhaltet die Entwicklung einer neuen Parametrisierung für solche Karten, sowie Techniken zur Vorfilterung, um matte Reflexionen zu erzielen. Die vorgestellten Algorithmen können dazu verwendet werden, vorberechnete Lösungen des globalen Beleuchtungsproblems auf nicht-diffusen, gekrümmten Flächen interaktiv zu visualisieren.

In Kapitel 9 diskutieren wir dann Bumpmaps und Normalenkarten. Letztere können mit den Techniken der Kapitel 5 und 8 verbunden werden.

Darauf folgen in Kapitel 10 Lichtfeld-basierte Ansätze für Reflexions- und Refraktionseffekte, welche einen noch höheren Grad an Realismus erzielen, als die in Kapitel 8 vorgestellten Verfahren. Bildbasierte Verfahren werden dann auch in Kapitel 11 zur Simulation realistischer

Linsensysteme eingesetzt.

In Kapitel 12, schließen wir die Arbeit mit einer Zusammenfassung der vorgeschlagenen Erweiterungen der Renderingpipeline und einer Diskussion ihrer Auswirkungen auf zukünftige Graphikhardware ab.

Zusammenfassung und Ausblick

> *Zusammenfassend ist festzustellen, daß eine Notwendigkeit besteht, den Schwerpunkt bei der Entwicklung neuer Algorithmen für die Implementierung von 3D Graphikhardware zu verschieben. Die Anforderungen entwickeln sich weg von "mehr Polygone" und "höhere Pixelfüllrate", hin zu komplexerer, dynamischer Geometrie, und gehaltvolleren Pixeln. Komplexere, dynamische Geometrie heißt nicht notwendigerweise mehr Dreiecke, oder schnellere Aktualisierung von Dreiecken, sondern bessere geometrische Primitive und Bewegung mit geringerem Datenaufkommen, integrierten Werkzeugen zur Inhaltserzeugung, Programmierschnittstellen, und hardwarebeschleunigtem Rendering. Gehaltvollere Pixel heißt nicht notwendigerweise mehr dargestellte Pixel, sondern daß jedes dargestellte Pixel das Ergebnis von aufwendigeren Verfahren für Beleuchtung, Schattierung und Texturierung ist. Das Endergebnis wird ein höherer Grad an Realismus bei der interaktiven Erfahrung sein.*
>
> David B. Kirk, *Unsolved Problems and Opportunities for High-quality, High-performance 3D Graphics on a PC Platform*, Eingeladener Vortrag, 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware ACM Computer Graphics, pp. 11–13, August 1998.

Gegenstand dieser Dissertation sind Algorithmen zur qualitativ hochwertigen Schattierung und Beleuchtungsberechnung unter Zuhilfenahme von Graphikhardware. Basierend auf physikalischen Modellen für die Interaktion von Licht mit allen möglichen Arten von Oberflächen, sowie auf einem abstrakten Modell für Graphikhardware, welches weitgehend auf bestehenden Graphiksystemen beruht, wurde eine Reihe von Algorithmen entwickelt, um unterschiedliche Probleme der Bildsynthese zu lösen.

Eine interessante Beobachtung ergibt sich aus der Tatsache, daß alle hier diskutierten Techniken eine Gemeinsamkeit aufweisen: in allen werden diskret abgetastete Werte verwendet,

um bestimmte Beleuchtungs- oder Schattierungseffekte zu erzielen. In Kapitel 5 werden die Beleuchtungsmodelle in mehrere Faktoren unterteilt, welche dann als 1- oder 2-dimensionale Texturen gespeichert werden können. Schattenkarten, wie sie in Kapitel 6 zum Einsatz kommen, verwenden eine diskrete Abtastung der Geometrie einer Szene. Bumpmaps und Normalenkarten (Kapitel 9) können als diskret abgetastetes geometrisches Detail angesehen werden, und schließlich repräsentieren Umgebungskarten (Kapitel 8) und Lichtfelder (Kapitel 7, 10 und 11) eine diskrete Abtastung der Strahlungsdichte für einen konkreten Punkt oder eine Reihe von Punkten im Raum.

Insbesondere die letzten beiden Beispiele, Umgebungskarten und Lichtfelder, sind vom bildbasierten Rendering kommende Techniken, welche hier auf die Beleuchtung und Schattierung geometrischer Objekte angewandt wurden. Wir glauben, daß dieser Art von Algorithmen eine wichtige Rolle im qualitativ hochwertigen, interaktiven Rendering zukommen wird. Im Moment stoßen die Lichtfeld-basierten Ansätze zwar an die Grenzen der verfügbaren Graphikhardware, da sie entweder, wie in Kapitel 10, große Mengen von Texturspeicher, oder aber eine große Anzahl von Renderingschritten (Kapitel 7 und 11) benötigen. Jedoch wird ersteres Problem durch weitere Preisreduktionen bei Speicherbausteinen sowie durch Graphikhardware, welche komprimierte Texturen verwendet verschwinden. Das zweite Problem wird abgemildert durch steigende Performanz sowie direkte Unterstützung von Mehrschrittverfahren, wie etwa die Möglichkeit, mehrere Texturen zur gleichen Zeit zu verwenden.

Es gibt mehrere Gründe, warum bildbasierte Verfahren, oder allgemeiner gesagt, Verfahren, welche auf Abtastwerten basieren, in dieser Arbeit dominieren, und auch ganz allgemein immer mehr Anwendung finden. Zunächst sind diese Daten das natürliche Ergebnis aller Arten von Messungen. Alle Werte, welche für Algorithmen in dieser Arbeit benötigt werden, lassen sich direkt durch Messungen generieren. Dies gilt sowohl für BRDFs, wie zum Beispiel in [Ward92] nachzulesen ist, als auch für Normalenkarten [Rushmeier97], Umgebungskarten [Haeberli93, Debevec98] und Lichtfelder [Gortler96, Levoy96].

Weiterhin sind regelmäßig angeordnete Abtastwerte in höchstem Maße geeignet für hardwarebasierte Anwendungen, da sie einfach Felder von Werten darstellen. Komplexe, für Hardwareimplementierungen ungeeignete Datenstrukturen entfallen auf diese Art. Zudem sind die Zugriffsmuster auf solche Felder oft ebenfalls sehr regelmäßig, so daß Techniken zur Pufferung eingesetzt werden können. Ein letzter Punkt ist, daß Verfahren, welche auf Abtastwerten basieren, eine große Anzahl von Stufen mit unterschiedlichem Kosten/Nutzen Verhältnis ermöglichen. Wenn in zukünftigen Generationen von Hardware mehr dezidierter Graphikspeicher verfügbar ist, kann ohne großen Aufwand die Abtastrate oder Anzahl der Quantisierungsstufen erhöht werden, was eine unmittelbare Verbesserung der Bildqualität zur Folge hat.

Eine Alternative zu abtastbasierten Ansätzen wäre beispielsweise die Verwendung prozeduraler Beschreibungen [Hanrahan90, Pixar89, Olano98]. Die beiden großen Vorteile dieses Ansatzes sind sowohl die Auflösungsunabhängigkeit, als auch eine kompakte Repräsentation.

Auf der anderen Seite benötigen solche Architekturen äußerst flexible, programmierbare Rasterisierungshardware, welche in der Herstellung teuer ist. Auch für Echtzeitanwendungen stellt der prozedurale Ansatz ein Problem dar, da im voraus nicht bekannt ist, wie lange die Ausführung einer prozeduralen Oberflächenbeschreibung dauern wird. Insbesondere kann diese Zeit von Pixel zu Pixel, aber auch von Bild zu Bild stark variieren, so daß eine gleichmäßige Lastverteilung zwischen parallelen Rasterisierungseinheiten schwierig ist. Die Aquisition von Ausgangsdaten und Kosten/Nutzen-Abwägungen sind ebenfalls kompliziert.

Vorschläge für zukünftige Generationen von Graphikhardware

Die meisten der hier vorgeschlagenen Algorithmen arbeiten effizient auf zeitgemäßer Graphikhardware, einige verwenden dabei jedoch bestimmte Funktionalitäten, welche noch nicht auf einer großen Anzahl von Plattformen zur Verfügung stehen. Diese Funktionsgruppen sind im Einzelnen:

Mehrere Texturen zur gleichen Zeit. Einige der Verfahren, wie etwa die Verwendung alternativer Beleuchtungsmodelle (Kapitel 5), oder die parabolische Parametrisierung für Umgebungskarten (Kapitel 8), profitieren von der Möglichkeit, gleichzeitig mehrere Texturen auf ein Objekt anzuwenden [SGI97], indem dadurch die Anzahl der Renderingschritte reduziert wird. Für keinen der hier präsentierten Algorithmen sind mehrere Texturen absolut notwendig, aber für einige sind sie hilfreich.

Bildverarbeitungsoperationen Einige Algorithmen, insbesondere die Verfahren zur Darstellung von Normalenkarten von Kapitel 9, benötigen Farbmatrizen und Farbtabellen. Diese sind ein Teil der sogenannten *Bildverarbeitungsoperationen*, welche in OpenGL Version 1.2 formal als eine separate Untermenge spezifiziert wurden [Segal98]. Obwohl die Unterstützung dieser Untermenge keine Voraussetzung für OpenGL-Kompatibilität ist, steht zu erwarten, daß viele Hersteller sie in zukünftigen Versionen ihrer Hardware unterstützen werden. Im Moment sind sie als Erweiterungen auf einer Reihe von Workstations verfügbar, inklusive derer von Silicon Graphics, für welche die Algorithmen dieser Arbeit implementiert wurden.

Interessanterweise werden diese Operationen in der vorliegenden Arbeit nicht etwa für typische Bildverarbeitungsalgorithmen eingesetzt, sondern für die Beleuchtungs- und Schattierungsverfahren. Eine Reihe anderer Algorithmen profitiert ebenfalls von dieser Funktionalität, ohne daß es Bildverarbeitungsanwendungen währen (siehe [Westermann98] und [Heidrich99e]). Die Bezeichnung “Imaging Subset”, welche in [Segal98] eingeführt wird,

ist daher etwas irreführend, und birgt die Gefahr, daß die Mächtigkeit dieser Operationen unterschätzt wird.

Pixeltexturen werden in der vorliegenden Arbeit von mehreren Algorithmen, einschließlich den Normalenkarten (Kapitel 9) und lichtfeldbasierten Brechungen (Kapitel 10) verwendet. Pixeltexturen werden augenblicklich von Silicon Graphics als "experimentelle Erweiterung" klassifiziert, die nur auf einer Plattform (Octane-Graphik mit Unterstützung für Texturemapping) verfügbar ist. Selbst diese Implementierung folgt nicht in allen Details der Spezifikation [SGI96].

Wir sind der Meinung, daß alle drei Funktionsgruppen so hilfreich sind, daß sie zum Funktionsumfang zukünftiger Graphikhardware gehören sollten. Wie schon oben erwähnt, kann dies für Mehrfachtexturen und die Bildverarbeitungsoperationen auch als gegeben angenommen werden.

Die Zukunft der Pixeltexturen ist weitaus weniger klar. Der experimentelle Charakter der Erweiterung ist eine ernstzunehmende Einschränkung. Ihre Nützlichkeit, welche auch an anderer Stelle belegt ist [Heidrich99e], macht die Entwicklung von Algorithmen für sie dennoch interessant. Dies wäre umsomehr der Fall, wenn projektive Texturen in die Spezifikation mit aufgenommen würden, wie in den Kapiteln 6 und 9 vorgeschlagen. Doch selbst ohne diese Änderung haben Pixeltexturen eine ganze Reihe von Anwendungen. Die hier beschriebenen Algorithmen zeigen nur einen kleinen Teil davon, demonstrieren aber das Potential dieser Erweiterung für die effiziente, qualitativ hochwertige Bildsynthese.

Zusätzlich zur Verwendung und Bewertung des Funktionsumfangs bestehender Graphikhardware werden in der vorliegenden Arbeit auch einige neue Erweiterungen vorgeschlagen. Zusammenfassend sind dies

- eine Reihe neuer Modi zur automatischen Generierung von Texturkoordinaten. Diese dienen der direkten Unterstützung der alternativen Beleuchtungsmodelle von Kapitel 5 und der parabolischen Umgebungskarten von Kapitel 8. Dieselben Verfahren könnten auch zur Beleuchtung von Bumpmaps herangezogen werden (Kapitel 9).

- ein zusätzlicher Tangentenvektor, welcher für jeden Knoten angegeben werden kann und entsprechend der Modellierungsmatrix transformiert wird. Dieser wird benötigt, um physikalisch basierte anisotrope Reflexionsmodelle zu implementieren.

- In Kapitel 5 wurde auch ein komplett neues abtastbasiertes Beleuchtungsmodell vorgestellt, welches geeignet ist, das herkömmliche Phong-Modell zu ersetzen. Damit besteht die Beleuchtungsberechnung für einen Knoten aus mehreren Skalarprodukten und dem Nachschlagen in einer Tabelle, und ist somit nicht wesentlich teurer als die knotenbasierte Phong-Beleuchtung.

Zusätzlich zu diesen konkreten Vorschlägen wird sich zukünftige Graphikhardware mit einer Reihe anderer Themen auseinandersetzen müssen. Zunächst werden Methoden für die Reduktion des Bedarfs an dezidiertem Textur- Bildschirmspeicher benötigt. Dies beinhaltet die Unterstützung von komprimierten Texturen, sowie die Möglichkeit, nur Ausschnitte einer großen Textur im Texturspeicher vorzuhalten. Das erstere Themengebiet wird bereits von einiger neuer PC Hardware in Angriff genommen [nVIDIA98], während in [Tanner98] ein Ansatz für das zweite vorgeschlagen wurde.

Weiterhin sind bereits erste Implementierungen mit direkter Unterstützung für Bumpmaps und Beleuchtungsberechnung pro Pixel verfügbar [nVIDIA98]. Diese Ansätze könnten mit unseren Techniken für physikalisch basierte Beleuchtungsmodelle von Kapitel 9 kombiniert werden.

Für eine weitere Verbesserung der Renderingqualität sollte zukünftige Graphikhardware auch Formate mit hohem Dynamikbereich für die Speicherung von Texturen und berechneten Zwischenbildern vorsehen. Zwei Beispiele für solche Formate sind Fließkommarepräsentationen für Farbkanäle, und das LogLuv Format [Larson97]. Dieser Themenkomplex bringt einige Probleme mit sich, da solche Formate Fließkommaarithmetik in der Rasterisierungseinheit voraussetzen, was für eine Implementierung der Rendering Pipeline nach Kapitel 4 nicht notwendig ist. Jedoch benötigt Hardware für Bumpmapping oder Fließkomma-Tiefenpuffer, wie etwa [nVIDIA98], während der Rasterisierungsphase ebenfalls Fließkommaarithmetik, so daß die zusätzlichen Kosten für Formate mit hohem Dynamikbereich nicht zu groß ausfallen dürften.

Schließlich wird es endlich Zeit für einer direkte Hardwareunterstützung von Schatten. Aus Gründen, die in Kapitel 6 genauer erläutert werden, halten wir hierfür Schattenkarten für am besten geeignet. Dies kann entweder in Form spezieller Erweiterungen geschehen, wie etwa in [Segal92] vorgeschlagen, oder durch den Alphatest unter Verwendung der Algorithmen von Kapitel 6 und [Heidrich99e]. Im letzteren Fall sollte die Hardware Framebufferkonfigurationen mit besonders tiefen Alphakanälen (mindestens 24 bit) unterstützen.

Auch Hardwareunterstützung für neue geometrische Primitive, wie etwa Subdivision-Flächen, oder auch Algorithmen zur Reduktion der geometrischen Komplexität durch Viewfrustum- oder Occlusionculling sind wichtige Schwerpunkte zukünftiger Forschungsarbeiten. Diese liegen jedoch außerhalb des Rahmens der vorliegenden Arbeit.

Zusammenfassung

Die vorliegende Dissertation führt eine Reihe neuer Algorithmen für qualitativ hochwertige Schattierung und Beleuchtungsberechnung unter Zuhilfenahme von Graphikhardware ein. Insbesondere wurden Verfahren zur Erzeugung verschiedener lokaler Schattierungs- und Beleuchtungseffekte, sowie zur Visualisierung globaler Beleuchtungseffekte vorgestellt. Dies schließt

Algorithmen für Schatten, Normalenkarten, alternative Beleuchtungsmodelle, spiegelnde und matt spiegelnde Reflexionen auf gekrümmten Flächen, sowie realistische Linsenmodelle und Lichtquellen ein. Alle diese Verfahren sind zueinander orthogonal in dem Sinne, daß sie beliebig miteinander kombiniert werden können, wobei sich jedoch die Anzahl der Renderingschritte erhöht.

Im Verlauf der Entwicklung dieser Verfahren haben wir wichtige Funktionsblöcke für zukünftige Generationen von Graphikhardware identifiziert. Einige von diesen sind etablierte Operationen, welche auf eine neue, innovative Art verwendet werden, andere sind experimentelle Operationen, welche noch keine weite Verbreitung gefunden haben, und wieder andere sind vollkommen neue Erweiterungen, die von uns eingeführt wurden.

Alle vorgestellten Algorithmen basieren auf der Verwendung diskreter Abtastwerte. Zukünftige Arbeiten können, auf diesen Ideen aufbauend, weitere Methoden der Anwendung von Lichtfeldern für Beleuchtungseffekte auf Oberflächen untersuchen. Wir sind zuversichtlich, daß noch viele andere interaktive Techniken aufbauend auf dieser Grundidee entwickelt werden können.

Andere Gebiete für zukünftige Arbeiten sind beispielsweise Verfahren für Volumeneffekte und partizipierende Medien. Zwar trifft es zu, daß die Darstellung von Volumina basierend auf einer 3D-Texturierung für Visualisierungsanwendungen im medizinischen oder ingenieurwissenschafftlichen Bereich untersucht wurden, allerdings sind die Anforderungen für die realistische Bildsynthese anderer Art. Anstatt wichtige Eigenschaften eines Volumendatensatzes vermitteln zu wollen, verlangt dieser Anwendungsbereich nach realistischen Effekten wie etwa Beleuchtung oder Schatten, welche von einem Volumen geworfen werden. Die effiziente Darstellung solcher Effekte mit Hilfe von Graphikhardware wird zur Zeit nicht hinreichend von wissenschaftlichen Arbeiten abgedeckt.

Schließlich ist auch die Aquisition von realen Daten und Modellen ein wichtiges Anliegen. Dies trifft auf alle Bereiche der Computergraphik zu, ist für interaktive Anwendungen aber besonders attraktiv, da hier die gemessenen Daten oft direkt mit der Graphikhardware weiterverarbeitet werden können, zum Beispiel unter Zuhilfenahme der Techniken die in der vorliegenden Dissertation entwickelt wurden.