Advanced

Lighting and Materials with Shaders

Kelly Dempski
Emmanuel Viale

WORDWARE

Advanced Lighting and Materials with Shaders

Kelly Dempski and Emmanuel Viale

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Dempski, Kelly.
Advanced lighting and materials with shaders / by Kelly Dempski and Emmanuel Viale.
p. cm.
Includes bibliographical references and index.
ISBN 1-55622-292-0 (pbk., companion cd-rom)
1. Computer graphics. 2. Light—Computer simulation. I. Viale, Emmanuel. II. Title.
T385.D467 2004
006.6'9—dc22 2004018415

2320 Los Rios Boulevard
Plano, Texas 75074

Printed in the United States of America

ISBN 1-55622-292-0

10 9 8 7 6 5 4 3 2 1
0409

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

In memory of Helen Gaede
KD

To Alexandre, Nicolas, and Valérie
EV

This page intentionally left blank.

Contents

This page intentionally left blank.

Foreword

The description of surface appearance is at the core of image synthesis, and creating convincing images with the computer remains to date a complex process, requiring advanced specialized skills. In reality, appearance is controlled by the complex interplay of light (electromagnetic radiation coming from various sources) and the materials composing the objects in the scene, with complex scattering effects taking place all along the path of light.

The physical laws governing this process have long been known at the scale of the light wavelength; yet in order to create a computational model that can be used by a computer, a discrete model is required, and the vast body of research conducted in computer graphics in the last three decades has aimed at providing simplified models for the calculation of lighting effects.

Nowadays computers are equipped with powerful graphics units specialized in massively parallel computations at the vertex or pixel level. The complex calculations needed for proper appearance rendering therefore become affordable in real-time applications, provided they can be incorporated in small programs called shaders. This book therefore provides a very timely contribution by demonstrating in detail how to create these shaders for advanced lighting.

The authors have chosen an ambitious and rigorous path by first presenting the physics and detailed lighting equations. Their discussion of lighting effects, advanced rendering algorithms such as raytracing and radiosity, and material descriptions will be useful to anyone first approaching computer graphics applications.

The remainder of the book provides ample detail on how to incorporate the most advanced lighting effects in shaders. The authors went as far as covering recent advances such as "precomputed radiance transfer" techniques, in which the objects can be illuminated in real time from all directions.

The combination of the theoretical presentation of the underlying concepts and the practical implementation methods for direct usage make this book a complete and self-contained manual. I very much enjoyed reading it and I am certain that it will prove a valuable resource for a great many professionals in the gaming or special effects industry.

Francois Sillion
Grenoble, France

Acknowledgments

Every book of this kind includes the knowledge and expertise of many people, both directly and indirectly involved with the book itself. First, we are indebted to the people who developed the theory and techniques that are described here. Their work is referenced at the end of each chapter and we encourage readers to use those sources to explore further. Of the people who worked directly with the book, we would like to thank Jason Mitchell, Scott Thompson, Wes Beckwith, and Jim Hill. Their help and suggestions were excellent.

We would also like to thank our families and colleagues. Their support is invaluable when working on a book like this.

Introduction

The Importance of Light

When you take a careful look at the real world around you, you begin to realize that most of the detail you see is the result of subtle lighting effects. Areas of light and shadow define the furrowed bark of trees. An elderly face can appear wiser simply because of the depth of its creases. Before a storm, ordinary sunlight takes on an ominous tone, and objects appear less defined. The same room can look very different when it is lit by a television or by candlelight. Each of these examples is an instance where your perception of reality is affected by very subtle changes in the way light interacts with the objects around you. Your eyes and brain are tuned to look for those subtle cues. Therefore, if your goal is to recreate reality (or to create a new reality), you must include as many of those effects as possible.

However, most game development books dramatically understate the importance of lighting and the interactions between light and physical materials. This is largely the result of the fact that, until very recently, consumer graphics hardware was not capable of rendering complex lighting effects in real time. This is changing, and game programmers are finding that they have the power to render complex effects that more closely approximate reality. In the past, the ability to render complex scenes was limited to the raw performance of the hardware. Performance will always be a concern, but complexity is now also dependent on the ability to use the capabilities of the hardware effectively.

With the correct algorithms and approaches, graphics hardware is capable of reproducing the subtle differences between elements of the virtual environment. These subtleties are the key to realism. Now characters need not be limited to simple textured models. They can be comprised of complex materials such as matte cloth, gleaming shields, and smooth skin. This level of realism is now possible, but it requires a deeper understanding of the underlying theory of how light interacts with matter.

The Structure of the Book

This book explains the underlying principles of lighting more deeply than most game development books. However, unlike other theoretical books, this book takes a holistic approach that couples theory with practical implementations that address the features and limitations of modern hardware.

The first chapters of the book deal with theory. They lay the conceptual foundations for the remaining chapters. The bulk of the book is comprised of chapters that each focus on a specific technique. Each chapter will explain the theory as well as give a sample implementation. Some readers look for books that give them a fully built out gaming engine. That is not our intention here. In fact, we have tried to strip each sample down to its bare essentials in order to highlight a given technique. This should make it easier to put the technique into any engine you need.

This book can be read nonlinearly. Advanced readers might want to jump to specific chapters. Beginning readers might want to start with the theory before they jump through sets of theory/approach/implementation to see how everything fits together. Regardless of how they are read, the chapters provide a solid theoretical and practical foundation on which additional techniques and further optimizations can be based.

Included with the book is a CD-ROM that contains all of the source code used in the sample applications. We have written the applications with both OpenGL and DirectX. In some cases, we have used both for a given technique. In other cases, we've used either DirectX or OpenGL. When we use only one, it is either because the other implementation is unnecessary (it is supported in a helper library like D3DX, for example) or because porting one implementation to another is trivial (like using HLSL shaders in an OpenGL/Cg environment).

Contact Information

The more you learn about a given topic, the more questions you are bound to ask. As you read the material, please feel free to contact us with any questions or comments. We have created a web site for this book at: http://www.advancedrenderingtechniques.com, where you will find different sections that complement the content of this book and its companion CD-ROM (FAQ, errata, downloads, etc.). Alternatively, we can be reached at Graphics_Book@hotmail.com (Kelly) and lightingbook@advancedrenderingtechniques.com (Emmanuel).

Chapter 1

The Physics of Light

Introduction

When game developers talk about physics, they are usually talking about motion, forces, and collisions. They almost never talk about the physics of the light that illuminates their scenes. Graphics programmers are more concerned with efficient implementations of higher-level lighting equations. However, in order to truly understand those equations, one must understand the deeper theoretical nature of light itself. This chapter is a quick primer on the physics of light. It provides you with a basic understanding of light and introduces you to key concepts that permeate the rest of the chapters. The problem is, an explanation of light requires two explanations of light.

1.1 The Duality of Light

The physicists of the 1920s were faced with an interesting problem. Newton had originally believed that light consisted of small particles. Much later, experiments by Thomas Young and others showed that light behaves like an electromagnetic wave [1]. Simple, repeatable experiments yielded results that were exactly consistent with wave behavior. A couple of years later, Einstein and others showed that, in fact, light does behave like particles in different experiments. Einstein's experiments demonstrated the existence of photons and eventually won him a Nobel Prize [2]. Two credible

experiments yielded two different results, and physicists found themselves struggling with the wave-particle duality of light.

This all might be very interesting to physicists, but what does it mean for graphics programmers? It's quite important actually, because graphics programmers use both forms nearly every day. Reflection and refraction are most easily described by thinking of light as particles. On the other hand, color is fundamentally described in terms of waves. A more complete understanding of illumination requires both. Physicists now have ways of dealing with the duality of light, but the equations they use are not directly relevant to computer graphics. Instead, the remainder of this chapter will address each form of light separately. Just remember that the two explanations describe the same light; they just describe it in two different ways.

1.2 Light as Particles

I will begin with an explanation of light as particles simply because it is the easiest to visualize. Light (in the particle model) is comprised of photons that are not physical particles in the same sense as dust particles, atoms, or electrons. Instead, you can think of them as very small packets of energy that move at a constant velocity and have no actual mass. With those constraints in mind, you can safely think of photons in the same way you think about billiard balls. Your intuition about how billiard balls behave on a pool table will hold approximately true for photons. You can begin to think about light reflecting off surfaces in roughly the same way you think about balls ricocheting off the sides of a pool table.

Anyone who has played pool knows that a billiard ball will bounce off a surface at an angle equal and opposite to its incoming angle (θ) as shown in Figure 1.1. The same is true for light reflecting off a surface.

Each time the ball hits the side of the table, a small amount of energy is lost in the collision. A small amount of energy is also lost to the table itself as the friction between the ball and the table generates a small amount of heat and sound. The ball continues to move across the table, striking surfaces, until gradually all of its kinetic energy is lost and the ball comes to rest.

Reflection of light is similar, although the exact mechanisms of energy transfer are different. Each ray of light can be thought of as a stream of many photons. Each time the stream strikes an object, some percentage of photons are absorbed and transformed to heat. The amount of absorption is dependent on the material properties of the object. The remaining photons are reflected and don't slow down, but there are fewer of them, and thus there is less total energy in the stream. The stream of remaining photons will continue to move through space until all of the energy has been absorbed by the objects in the environment. Figure 1.2 shows that,

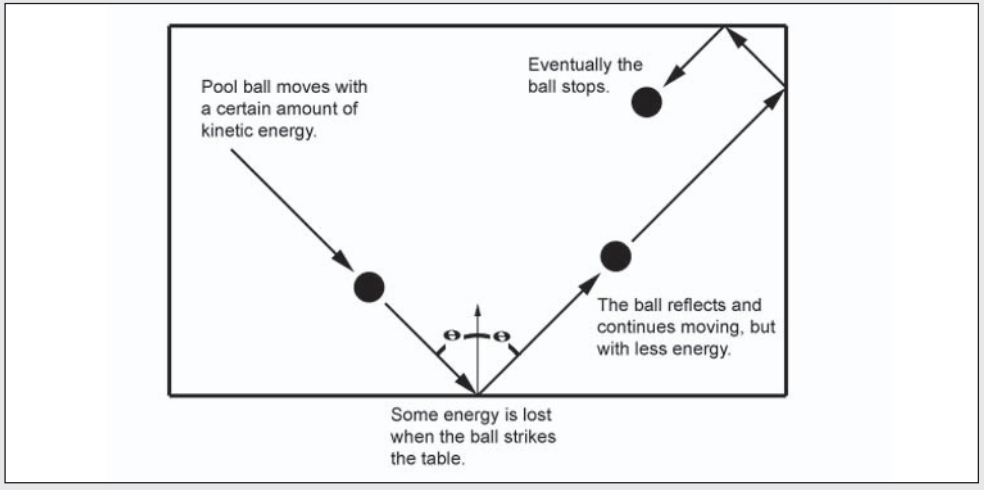


Figure 1.1. Simple ricochet of a ball on a pool table

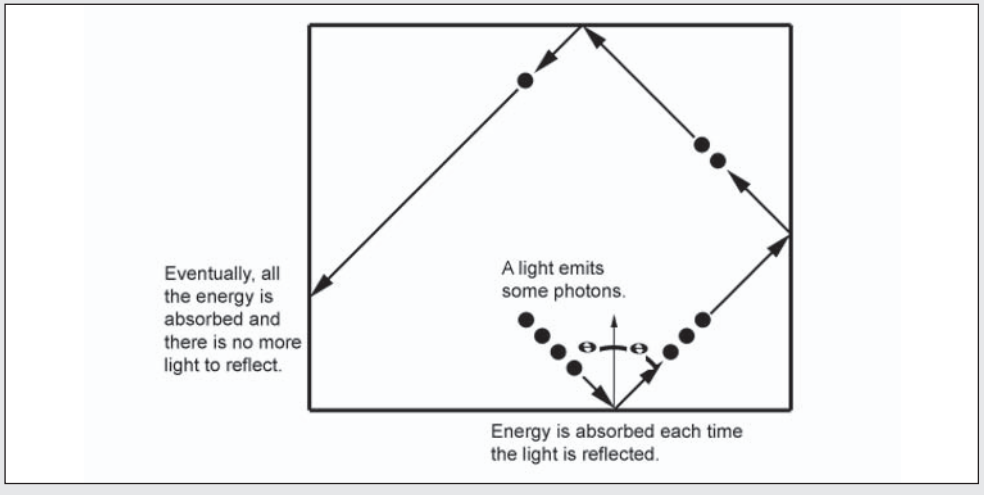


Figure 1.2. Simple reflection of light on a set of surfaces

unlike the billiard ball, the stream of photons never comes to rest. Instead, there comes a point where all of the photons have simply been absorbed.

At first glance, the two situations might seem very different, but at a high level they are the same. You can think of both examples as systems in which a certain amount of energy continues to travel through an environment until all of that energy is transferred to the environment. In the case of a billiard ball, less energy is lost on a smoother table, and the ball rolls farther. In the case of light, less energy is lost if the surfaces are more reflective, and more light travels farther.

NOTE:

Conservation of energy dictates that energy does not simply disappear, but instead changes form. In both examples, energy is transformed to heat and transferred to the environment.

One key point to take away from this section is that your intuitions about physical objects reflecting off of surfaces will mostly hold true for light

reflecting off of surfaces. Another point is that the overall impact of light in a given scene is highly dependent on its interactions with the materials in the scene. A white room lit by a dim light might appear brighter than a black room lit by a very bright light in much the same way that a slow billiard ball on a smooth surface might travel farther than a fast ball on a rough surface. The purpose of this book is to delve more deeply into those types of interactions.

1.3 Light as Waves

Despite all of the particle-like properties of light, it also has all the properties of an electromagnetic wave. In this sense, it behaves just like radio waves, x-rays, and other forms of electromagnetic waves. The basic properties of a wave are shown in Figure 1.3.

The amplitude of a wave corresponds to the intensity of the light. The *wavelength* is the distance covered by a single wave, which is usually very short for waves of light (measured in nanometers). The *frequency* is the number of waves that pass a given point in space in a single second (measured in hertz, or "cycles per second"). The relationship between wavelength and frequency is given by:

$$f = \frac{c}{\lambda}$$

where f is the frequency, c is the speed of light, and lambda (λ) represents the wavelength. Because the speed of light is constant, there is a one-to-one correspondence between a given frequency and the matching wavelength.

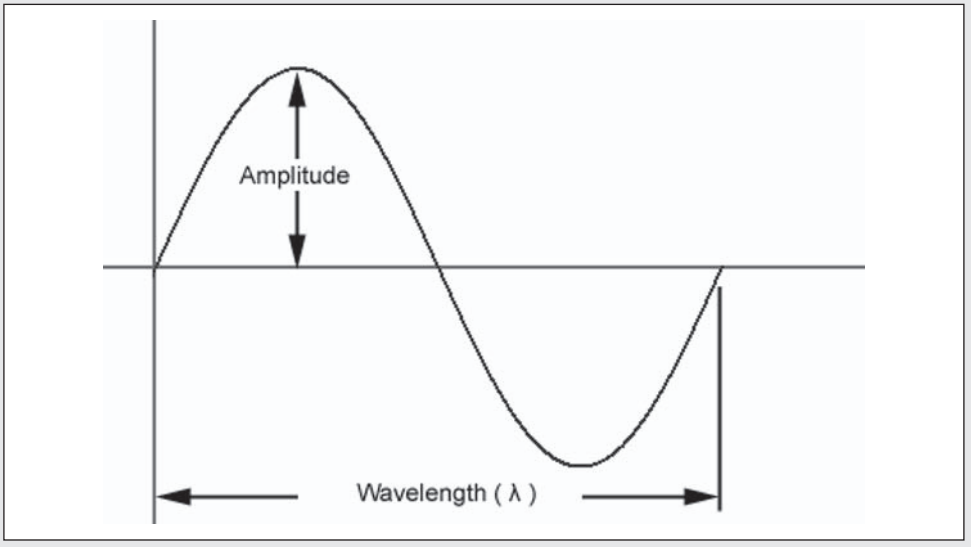


Figure 1.3. Properties of electromagnetic waves

NOTE:

Light also has another property called *polarity*, which is the orientation of the wave in space. You can think of polarity as the angle of revolution about the x-axis in Figure 1.3. The polarity of a light source is largely ignored in most graphics applications because most light sources emit light that is randomly and uniformly polarized.

1.3.1 Wavelength and Color

The color of visible light is dependent on the wavelength, and a given source of light might emit waves of several different wavelengths. Electromagnetic waves are classified by ranges of wavelengths. There are many classes of electromagnetic waves that have wavelengths above and below the visible range. Visible light falls into the range shown in Figure 1.4 below.

Therefore, an emitter of perfect red light only emits waves with a "red wavelength." Color is also a function of human perception. An orange light might be either light of a given wavelength or a set of different wavelengths that when combined we perceive as being orange [3]. White light is considered to be a mixture of waves covering the entire visible spectrum.

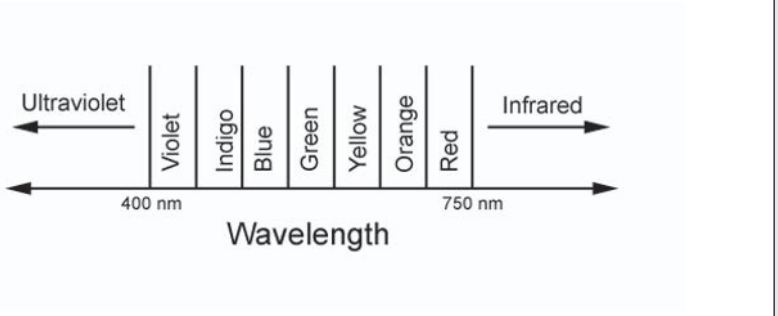


Figure 1.4. The visible spectrum

Some materials absorb light of certain wavelengths and not others. A red object illuminated by a white light reflects only the red portion of the light, which is why we see it as red (the other wavelengths are absorbed). Similarly, most conventional colored lights are white lights with colored filters. On the other hand, high precision lasers actually emit light in a very narrow range of colors.

Most graphics equations treat color as a simple property and safely ignore the fact that it is fundamentally tied to wavelength. However, you will see some lighting models that account for the fact that the features of some surfaces are roughly the same size as a given wavelength of light. For those surfaces, wavelength plays a key role because the topology of the surface is just the right size to create interference between waves of a given wavelength.

1.3.2 Phase and Interference

Another property of all waves is *phase*, which is an offset distance between two waves, as shown in Figure 1.5.

The light emitted by most real-world lights is not uniform enough to consider the phase differences between two waves. However, phase is important as it relates to interference.

When two or more waves overlap in space, they interfere with each other either constructively or destructively. As Figure 1.6 shows, interference is additive. The waveforms of two waves can be added to produce a third.

This is true for all classes of waves. Two ocean waves might come together to produce one larger wave. Noise-canceling headphones generate sound waves to destructively interfere with and cancel out environmental sounds. The same effects hold true for light. Interference between two waves of light can cause the final light to appear lighter or darker.

You can see from these figures that phase plays an important role in how two waves interfere and whether or not the interference is constructive or

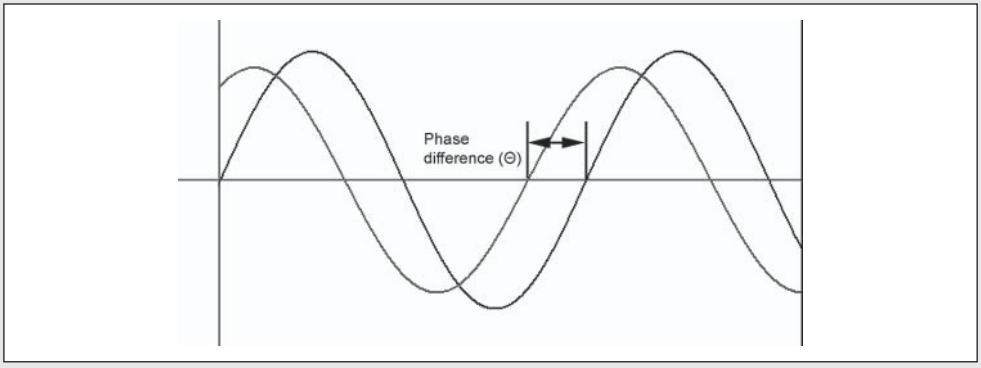


Figure 1.5. Two waves with equivalent wavelength and frequency that are out of phase

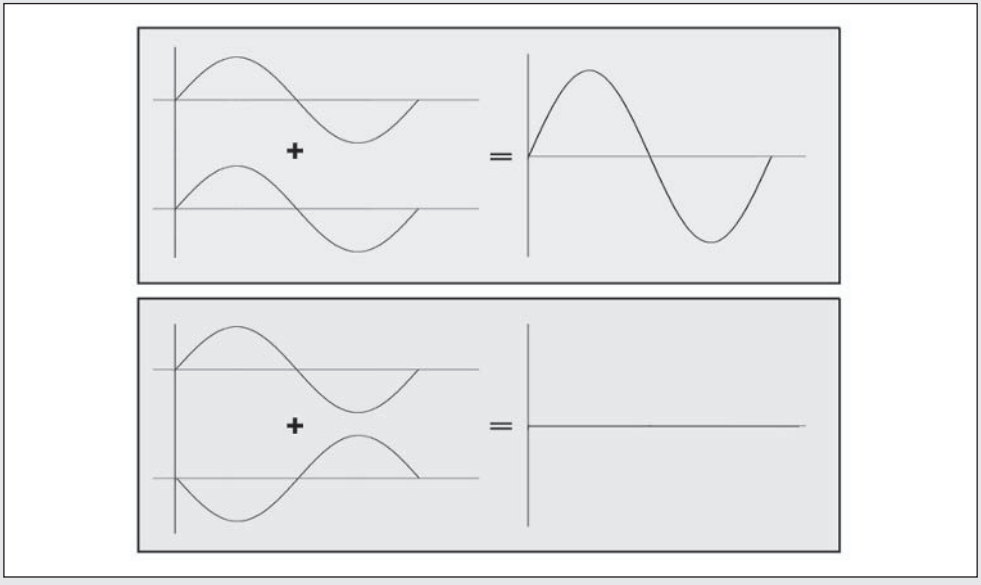


Figure 1.6. Constructive and destructive interference between waves

destructive. Figure 1.6 shows waves of the same wavelength and amplitude. If the phase difference between the two is exactly one half of their wavelength, they will cancel each other out. If they are exactly in phase, the final wave will be double the amplitude.

These interactions are not modeled in most lighting equations, but they do appear in some. Chapter 5 presents a lighting model that accounts for the effects of very small microfacets on the surface of certain materials. The phase difference between two rays of light reflecting off of an irregular surface can sometimes be enough to amplify some colors and mute others [3]. Also, very thin materials (where the thickness of the object is close to the wavelengths in the visible spectrum) can cause interference, resulting in the characteristic bands of color such as is seen on soap bubbles.

1.4 Light as Energy

Although many game development books ignore this fact, light is another form of energy. Therefore, it must follow the physical laws that govern the transfer of energy. The most important of these laws is conservation of energy. Simply put, this law says that energy is never created or destroyed, but it can be transformed [4]. For instance, the potential energy found in coal is used to create kinetic energy in steam, which is transferred to kinetic energy in a turbine, which creates electricity. The electricity travels to your house, your lightbulb transforms that electricity to light energy, the light energy hits the page, it reflects from the page, is absorbed by your retina, and here you are reading. Other rays of light bounce off of your forehead, end up in some distant corner of the room, and are eventually absorbed elsewhere.

This might seem completely esoteric from a game programmer's point of view, and you certainly don't need to worry about this with simple diffuse lighting. However, in the following chapters, you will see how conservation of energy explains basic properties such as attenuation. It is also a component of global illumination models which, in fact, have their roots in thermodynamic research and were only later applied to illumination for computer graphics.

Conclusion and Further Reading

Admittedly, you could probably render amazing scenes without ever learning about the physics of light. However, many books do an excellent job of explaining the "how" without ever explaining the "why." The purpose of this chapter was to give you a brief glimpse into the "why." As you read the subsequent chapters, you will find that this background information makes it much easier to understand some of the higher-level concepts.

I have only given you a cursory look into the physics of light and the history surrounding it. A complete discussion could fill many, many books. The references below are just a few of the books you can read to get a deeper understanding. Personally, I highly recommend books by Richard Feynman, both for his accessible explanations and his incredible wit.

References

[1] Einstein, A. and L. Infeld, *The Evolution of Physics*, Touchstone, 1967, 112.

[2] Halliday, D. and R. Resnick, *Physics*, John Wiley & Sons, 1978.

[3] Watt, A. and M. Watt, *Advanced Animation and Rendering Techniques*, Addison-Wesley, 1992, 51-52.

[4] Feynman, R. P., *Six Easy Pieces*, Addison-Wesley, 1995, 69-86.

Chapter 2

Modeling Real-World Lights

Introduction

In the real world, lights come in a variety of shapes, colors, and sizes. Outside, you might have sunlight as well as man-made lights such as streetlights and neon lights. Inside, you might have lamps, candles, and television sets. Even small items such as lit cigarettes and LED displays contribute light to a scene. Adding light to the real world is usually as easy as flipping a switch or lighting a match.

Adding light to a virtual scene is much more difficult. Offline rendering applications offer a variety of different lights that a 3D artist can use to approximate the physical world. An artist might use area lights to reproduce the effect of overhead lighting and a low-intensity point light to add the glow of a cigarette. In an outdoor scene, the effects of sunlight can be approximated with directional lighting and a complex global illumination solution. The results are very good, but some scenes require several minutes to render a single frame and every new light adds to the computational overhead.

Real-time applications such as games must render frames in fractions of a second, and therefore use a much smaller subset of lighting features. The computational cost of each light in the scene must be kept to a minimum. Most gaming engines are limited to combinations of directional lights, point lights, and spotlights. Many books explain these three lights without putting them in the context of their real-world and offline rendering equivalents. The reader might know how to set attenuation values with no clear idea of what they really mean. In this chapter, I explain how the features of

real-time lights map to real-world lighting effects with the following topics:

- Ideal point lights
- Geometric attenuation
- Attenuation through matter
- Directional lights
- Area lights
- Spotlights
- Local and global illumination models
- Ambient lights

2.1 Ideal Point Lights

The most idealized representation of a light is a point light. Point lights are also prototypical in that they serve as a basis for the other light types. The fundamental features of point lights will hold true for other light types, and they will help define special cases such as directional lights. At the most basic level, all lights are characterized by the fact that they add energy to a scene in the form of visible light.

2.1.1 Point Lights as Energy Sources

The simplest explanation of a *point light* is that it is a single point in space that emits energy in a uniform spherical field, as shown in Figure 2.1.

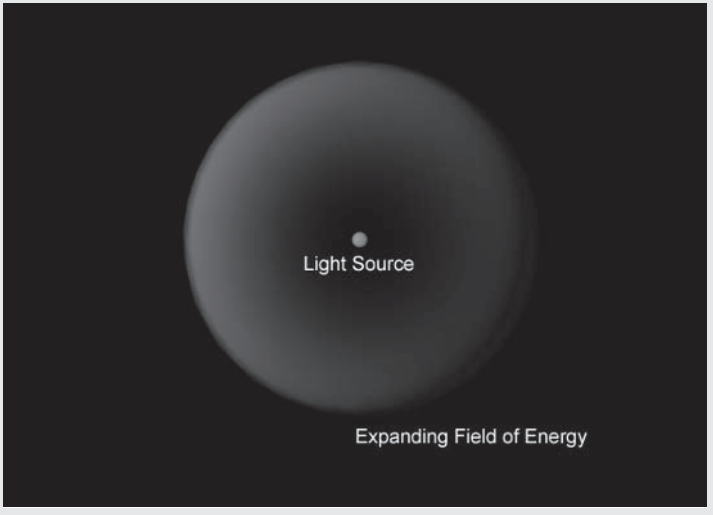


Figure 2.1. A simple point light

NOTE:

A sphere can be divided into solid angles (just as circles are divided into angles). These solid angles are measured in units called steradians and there are 4π (pi) steradians in a complete sphere.

An ideal point light is a point in the mathematical sense. It has no size or shape of its own. Instead, it is merely a

location in space that serves as the center of the spherical field of energy radiating from the light. In more explicit terms, a point light is a source of *luminous flux* (power) measured in lumens. Basically, this means that, in a given period of time, a certain amount of energy is added to the environment. In the context of lighting, you are most interested in light energy. The change in energy over time is the "flux," and the fact that it is light energy is what makes it "luminous." In the idealized case, this energy is emitted in a perfect sphere, so it is convenient to talk about luminous intensity as the energy emitted per unit solid angle of that sphere. For visible light, luminous intensity is measured in a unit called a *candela* (lumens per unit solid angle of a sphere). Figure 2.2 expands on Figure 2.1 to include these points.

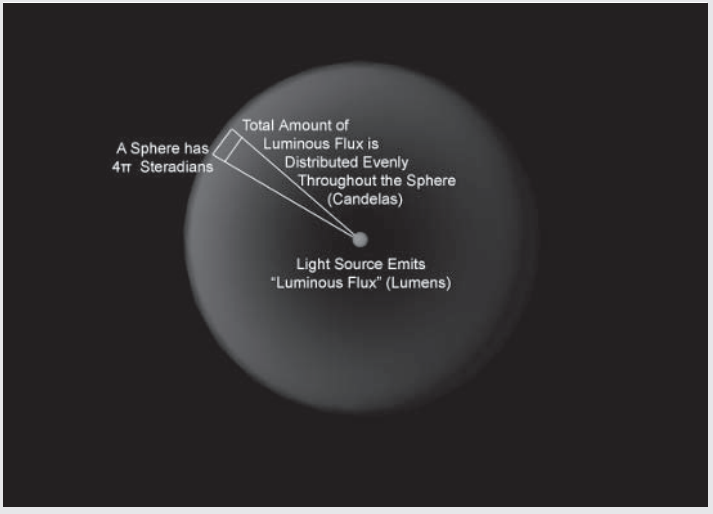


Figure 2.2. A point light described in terms of power output

NOTE:

Power is a measurement of energy output per unit of time. Light intensity is measured in power, but I will often talk about energy to make points about the conservation of energy. Units of power and energy are *not* interchangeable, so I run the risk of introducing bad habits, but it is convenient to ignore time and equate the two for the sake of discussing the higher-level concepts. The total amount of energy is power * time, and I'm essentially multiplying the power by one unit of time to get energy.

It is no coincidence that a typical candle emits approximately one candela of luminous intensity. This can serve as a basis for understanding how the intensities of different lights in a scene relate to each other. You will probably never write a game engine that expresses light values in terms of candelas, but it might be worthwhile to have some rough ideas of the relative output of different light types. For instance, a lightbulb emits roughly 100 candelas, which means that it is roughly 100 times brighter than a candle. This provides a good rule of thumb if you ever have an environment that features both a candle and a lightbulb. Chances are that you'd want to tweak the relative values until you found something that looked good, but the real-world values would provide a good starting point. There are many online resources for intensity values, and some products list them as part of their specifications.

NOTE:

Some product specifications will express brightness in terms of candelas, lumens, candlepower, nits, and other units. These units are not necessarily interchangeable (numerically or semantically), and you might need to do some unit conversion if you really want to use the published values.

Regardless of the value of the luminous intensity, the amount of energy that actually reaches a given object depends on the distance between the light and the object. Referring back to Figures 2.1 and 2.2, a light is a source of a certain amount of energy (lumens). That energy is (in the ideal case) evenly distributed across solid angles of a sphere (candelas). If you were to enclose the light source in a sphere of a given radius, a certain amount of light would strike each unit of area of that sphere. This measurement is the luminance at that point, measured in candelas per square meter. This is the value that is most important when illuminating surfaces and, as you will see, the amount of luminance at any given point is a function of the radius of the sphere of light at that point. This is the basis for geometric attenuation.

2.1.2 Geometric Attenuation

Attenuation is a process whereby something is weakened in intensity. For electromagnetic fields, attenuation is a function both of distance from the source and the properties of the medium through which the energy is traveling. The first part of the attenuation function is called geometric attenuation because it is a byproduct of the geometric relationship between the energy source and the receiver.

In section 2.1.1, a point light was described as a source of power (energy per unit of time) with its power radiating in an expanding sphere. Now imagine that you encase the light inside a sphere of some radius. For a given amount of power, a smaller sphere receives more energy per unit of surface area than a larger sphere does. Conservation of energy dictates that a sphere with a larger amount of surface area must receive less energy per unit of surface area. If it was somehow the same (or greater) amount per unit of surface area, that would mean that more energy was somehow created as the sphere became larger, which does not conform to the law

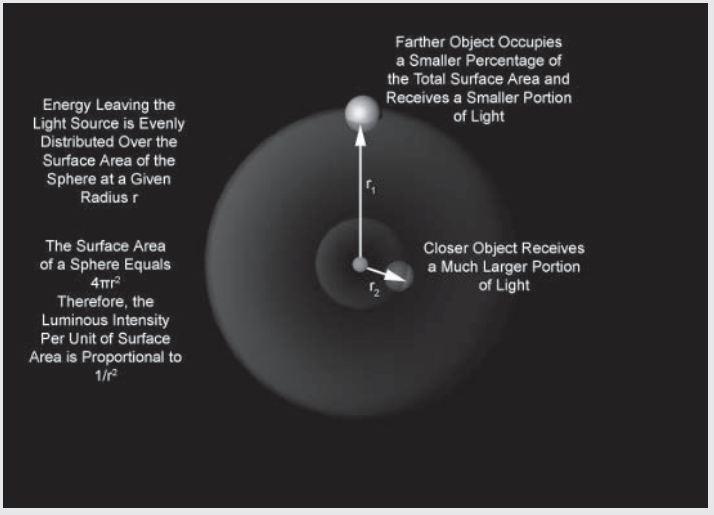


Figure 2.3. The inverse square rule of geometric attenuation

of conservation of energy.

The relationship between the source intensity and the destination intensity can be derived from Figure 2.2 and a little bit of geometry. Figure 2.3 shows that the intensity weakens with the inverse square of the distance.

This means that every time an object doubles its distance it quarters the amount of light energy it receives. Figure 2.4 shows this falloff curve.

This is mathematically correct, but often results in unnatural lighting in 3D scenes. The reason for this is simple. Natural lights are not ideal point lights, so natural lighting cannot be accurately modeled with idealized equations. Your eye doesn't care about the raw mathematics. It looks for cues that are familiar from the real world. I will return to this topic when I discuss area lights in section 2.3.

2.1.3 Attenuation through Matter

Though it's not typically modeled in basic lighting equations, light intensity does attenuate as it moves through matter. This can easily be seen when shining a light through fog or murky water. When passing through fog, the rules of geometric attenuation do not change, but light is also absorbed or scattered by the particles of water in the fog.

Keep in mind that energy is not lost. Rather, it is absorbed or scattered. Figure 2.5 shows this effect; the illustration on the left does not include smoke, while the illustration on the right shows the light passing through a column of smoke before it hits the sphere. Some of the light is scattered by the smoke, which means there's less light traveling to the sphere. Some of the light is scattered toward the camera (making the fog visible).

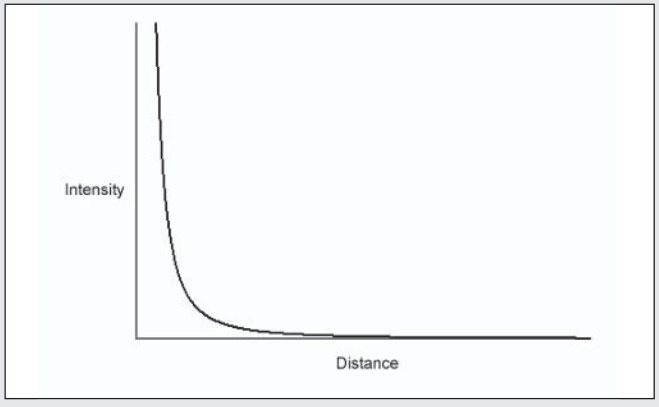


Figure 2.4. A plot of the inverse square rule

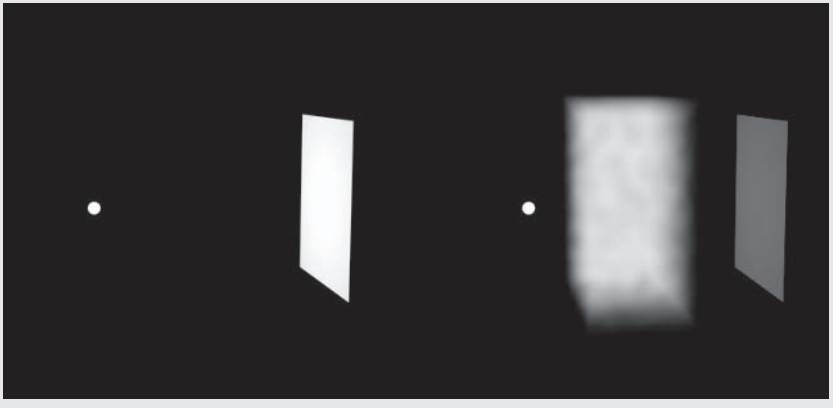


Figure 2.5. The effects of passing light through smoke

Realistic scattering and absorption is not typically used in game engines because of the relatively high computational cost. Also, the overall effect is minimal unless the scene contains heavy fog or smoke. I will revisit this topic in more depth in Chapter 5, but the important point to remember is that fog and smoke will affect attenuation. In some cases you might want to implement a physically based effect, and in other cases you might want to simply tweak attenuation values to approximate its effects.

As a final point on attenuation, notice that the attenuation equations for geometric attenuation and most forms of scattering will never result in a value of zero. In the real world, a light source at an extremely large distance will still illuminate an object with some extremely small amount of light. However, in practical terms it's convenient to set a maximum distance. While not exactly correct, it will save you calculations that would yield miniscule values that provide no actual effect on the scene due to representation and precision of the color values. In practical terms, there is some distance at which the total illumination cannot be represented meaningfully by an 8- or 16-bit color value. If you know that distance, there is no need to do the full calculation. However, be sure that your maximum distance is set far enough that it does not cause a harsh change in lighting conditions by prematurely clamping the value to zero.

2.1.4 Point Lights and Lighting Equations

In Chapter 5, I will begin to introduce lighting equations that are functions of a light's intensity, its angle relative to the receiver, and other factors. Without talking about specific lighting equations, Figure 2.6 illustrates how the topics from this section serve as inputs to a lighting equation.

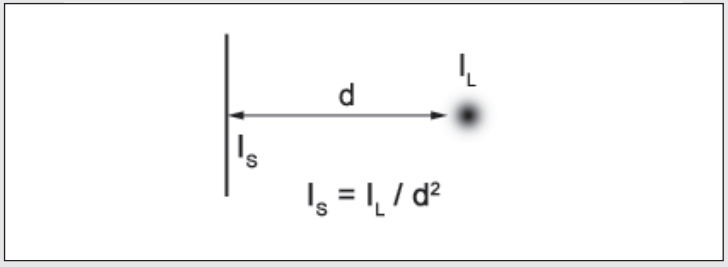


Figure 2.6. The properties of a point light as they relate to lighting equations. Where I_L represents the illumination at the light source, I_s represents the illumination at the surface, and d represents the distance between the light and the surface.

2.2 Directional Lights

Directional lights are a special case of point lights. As an object moves farther away from a light source, it occupies a smaller solid angle within the spherical field. This creates the opportunity to simplify the lighting calculations.

2.2.1 The Relationship between Point and Directional Lights

Geometry and trigonometry will tell you that independent radial rays emitted from the light source to some receiving object become closer to parallel as the object gets farther away. This is shown in Figure 2.7. This is also true if the object stays in the same position but becomes smaller.

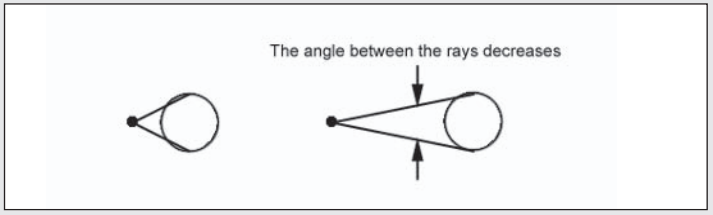


Figure 2.7. Rays becoming closer to parallel with increasing distance

When the distance is infinite or the object is infinitely small, the rays are parallel. This means that you can treat very distant lights as simple directional lights, which are computationally less expensive than point lights. Of course, this is only an approximation, but Figure 2.8 shows that this approximation is more than reasonable for sunlight since the maximum angle of light from the sun is less than five thousandths of a degree!

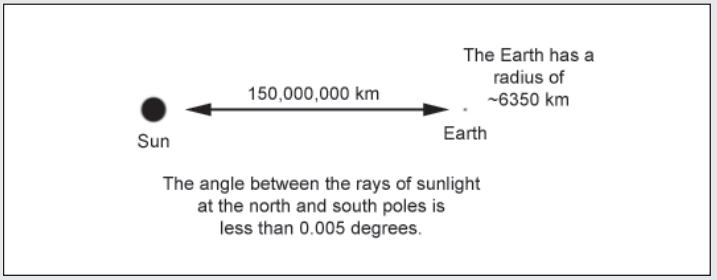


Figure 2.8. Sunlight as a directional light

The same factors that allow you to approximate distant point lights with directional lights also allow you to disregard the effects of attenuation. Based on the distances in Figure 2.8, you can compute the intensity difference from one side of the Earth to the other. The difference is extremely small (ignoring the fact that the Earth itself is blocking the light). Any attenuation of sunlight within a given scene would be effectively zero, especially considering the resolution of the color values. This allows you to ignore attenuation completely when dealing with sunlight hitting a single surface. There might be attenuation due to atmospheric effects, but that would be modeled a little differently, as you will see in Chapter 5.

Figure 2.8 illustrated the effectiveness of the directional lighting approximation for sunlight, but it might also be appropriate in many other instances. For instance, you could probably model a streetlamp as a vertical directional light if the lit object below was very small and stationary. However, the same light should probably be modeled as a point light if a car passes underneath because the larger object and changing angle (relative to the light direction) will have a greater visual impact.

Now that you understand the relationship between point lights and directional lights, you can optimize accordingly. There is always a trade-off. Directional lights are less computationally expensive but may not look as good (except in cases like sunlight). Point lights can give a more realistic appearance, but require more calculations.

2.2.2 Directional Lights and Lighting Equations

In Figure 2.6, you saw that the inputs to the lighting equation required some calculations based on the geometric relationship between the object and the light. Directional lights require no such calculation because the angle of the light doesn't change and there is no attenuation. Figure 2.9 illustrates this.

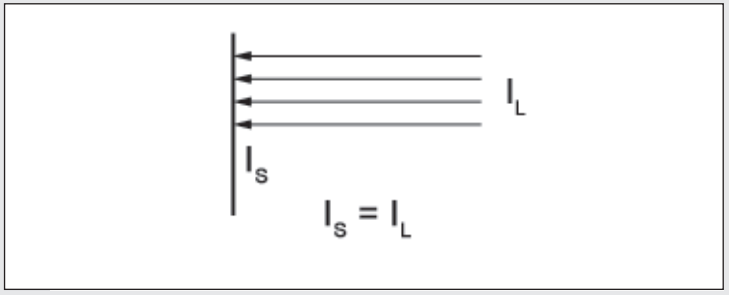


Figure 2.9. The properties of a directional light as they relate to lighting equations

2.3 Area Lights

Point lights and directional lights are frequently used in real-time graphics, but area lights are usually reserved for offline rendering applications because they are more computationally expensive. As such, this discussion of area lights might seem like an abrupt departure from the previous material. The reason for this departure is that area lights serve as a foundation for discussions about spotlights and other forms of illumination.

2.3.1 The Relationship between Point and Area Lights

Ideal point lights do not exist in the real world. Every real light source has some amount of surface area and takes up some finite amount of space. Consider a typical frosted lightbulb. Inside, the glowing filament might be relatively small (but not infinitely so), and the frosted exterior glass diffuses that light from all points on the much larger surface of the bulb. For all practical purposes, light is uniformly emitted from the entire surface area of the bulb. In a sense, every location on the frosted surface becomes a point light. The result is that a physical lightbulb creates a much softer lighting effect than a single ideal point light because of this larger light-emitting surface area. Figure 2.10 shows the difference between a scene lit by a single point light and the same scene lit by an area light. The degree of the effect is a function of the size of the light.

3D rendering applications usually recreate this effect by

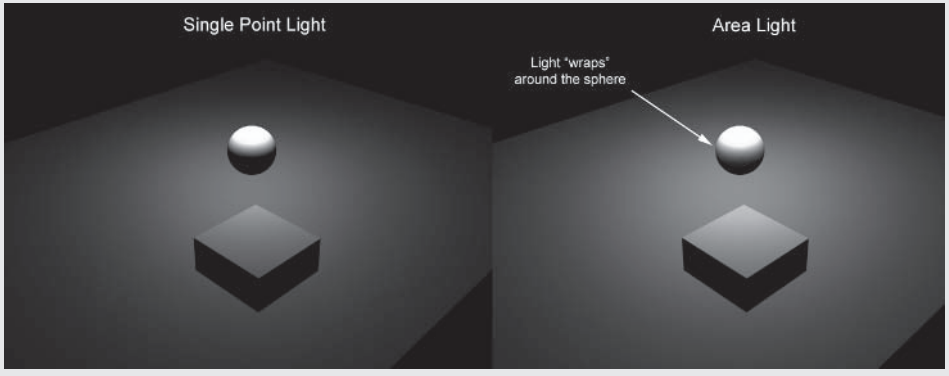


Figure 2.10. Lighting with a point light and an area light

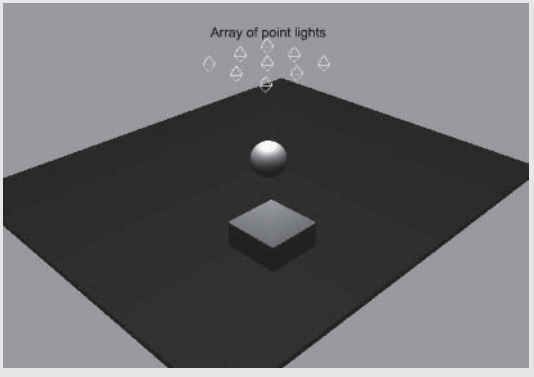


Figure 2.11. An array of lights can simulate a diffuse ceiling light.

using a collection of point lights. Figure 2.11 shows a conceptual view of this. The more point lights that are used, the more accurate the effect and the higher the computational cost.

If you want to create an area light effect, you can create a set of point lights in any configuration you need. For instance, a spherical cluster of points could be used to recreate the effect of a lightbulb. A flat, rectangular array of lights could be used to model an overhead fluorescent light. A set of lights in a curve could be used to recreate the shape and effect of a neon sign. These basic primitives are available in some modeling packages or can be easily created in others. The only limiting factor is computational cost. For instance, the right side of Figure 2.10 contains nine point lights and it took approximately nine times longer to render. In the past, this computational overhead was unacceptable for games, but better hardware is making it possible to create these effects in real time.

This section is not included as an attempt to convince you to use many true area lights in your games. However, there are some effects that people do want to see in games that are, in the real world, the result of area lights. Understanding area lights is the key to understanding the effect. One such effect is soft shadowing. Another effect is softer attenuation.

2.3.2 Attenuation and Area Lights

Attenuation is a function of distance, which is easy to compute for an ideal point light. For an area light, things change because there is no one discrete point and therefore no one discrete distance from the light to some point in space. If you continue to model an area light as a collection of point lights, you can compute the attenuation of each light and add the attenuated values to find the total illumination for a given point in space. Figure 2.12 shows a very simple diagram in which the area light is linear and oriented perpendicular to a surface. You can imagine the light as a neon tube.

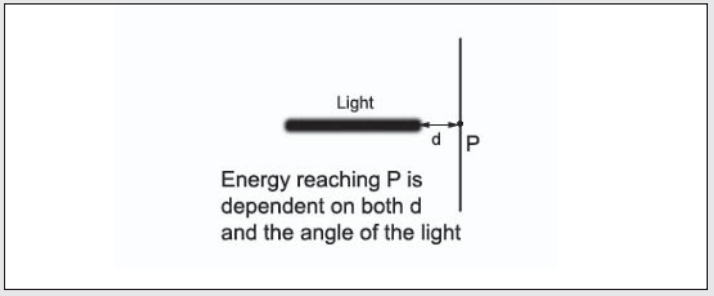


Figure 2.12. Attenuated light from a linear area light

Based on Figure 2.12, you can see that the linear light is very different from a single point in space. The point P will be lit from the entire length of the light, but the light will be attenuated differently for different points along the line. If you wanted to treat the light as a single entity with a single attenuation equation you could, but the attenuation would be

different from the inverse square you saw previously. Other shapes would yield more complex attenuation profiles. Ideally, you would like to have a flexible way to at least approximate the attenuation of light for lights of different shapes and sizes. The equation below is a typical instantiation of such an equation, and it is the generalized attenuation equation used in DirectX and other tools.

$$Attenuation = \frac{1}{Ad^2 + Bd + C}$$

The value of A determines how quickly the light attenuates as a function of the square of the distance. For the inverse square rule, the value of A would be 1.0. The value of B determines the linear behavior of attenuation. In some cases, you might want a smaller value of A but a larger value of B, thereby creating a more gentle attenuation effect than what you get with the inverse square. Finally, the value of C can be used to bias the effect. The values of these coefficients determine the overall shape of the attenuation curve as shown in Figure 2.13.

For the most part, it is reasonable to choose values based on what looks good. In most games, you might want to create a more gradual falloff effect than the inverse square, but you are probably not concerned with accurately modeling the shape characteristics of the light. Figure 2.14 shows the difference between the ideal inverse square case and an inverse attenuation function. Notice the effect of softer lighting.

While area lights are not a defined light type in most 3D APIs, their properties are built into one that is defined. The spotlight inherits its behavior from the fact that, at some level, it is an area light.

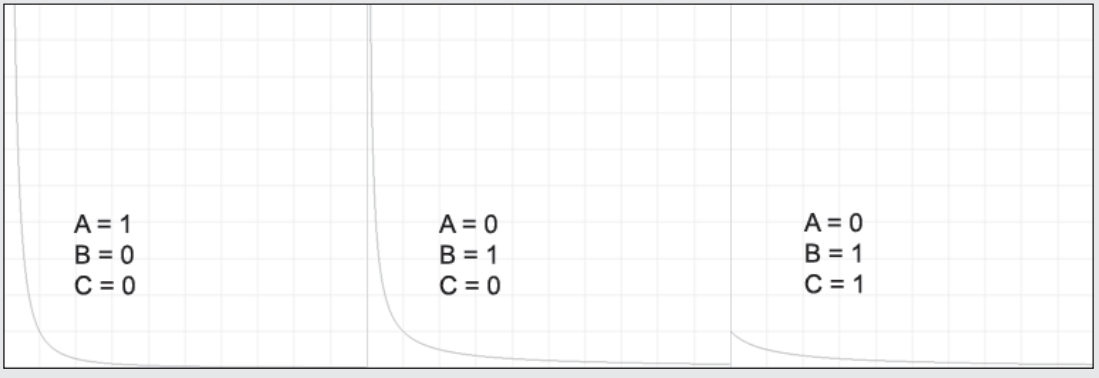


Figure 2.13. Attenuation coefficients and their effects on falloff curves

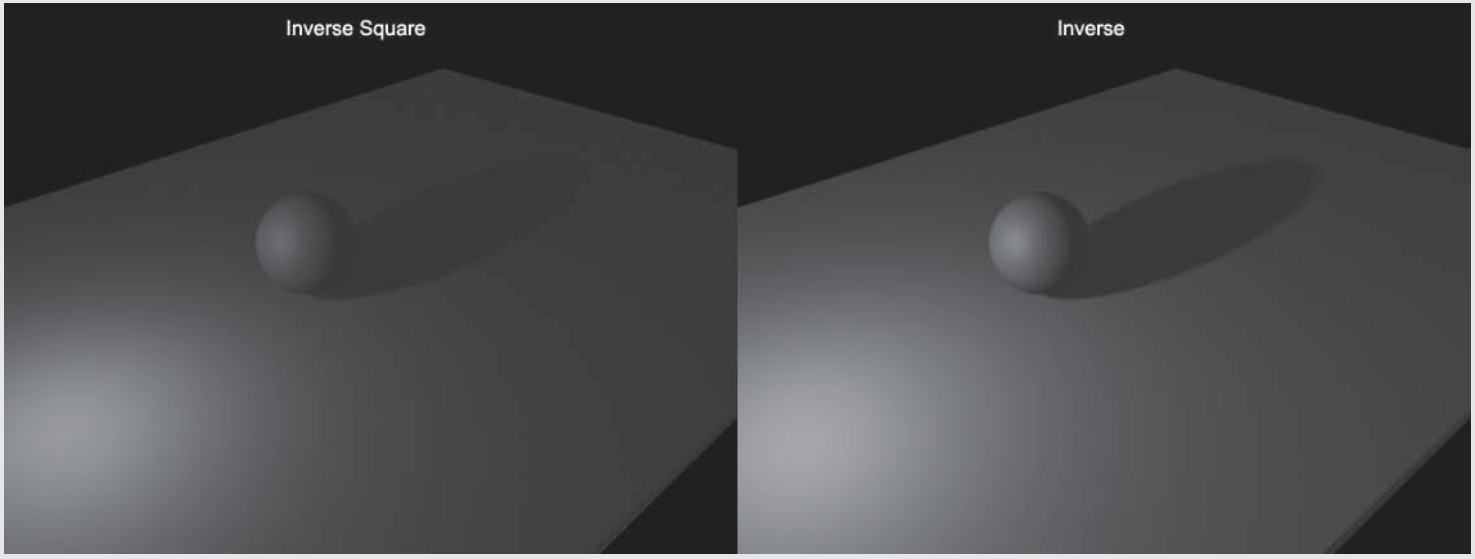


Figure 2.14. Comparing attenuation functions

2.4 Spotlights

Spotlights are characterized by the fact that they are more directional than point lights and that they have some of the "soft" qualities of area lights in regions called the umbra and penumbra. I will explain what these words mean, but first I would like to talk about why they happen.

2.4.1 Spotlights as Physical Lights

You can make a spotlight pretty easily. Get a lightbulb and place a can around it. The can will block most of the light except for that which escapes from the open end. Reshape the can into more of a parabolic shape, add a lens, and you have yourself a pretty good spotlight. Now, take another look at your light. In the abstract, you have an area light (the lightbulb) and a shell that obstructs some of the light that is leaving the bulb. This is shown in Figure 2.15.

Light is leaving the lightbulb in all directions, but the can blocks a large portion of the light. The rest leaves the spotlight in a cone shape. Ignoring the effects of the shape of the can or a lens, the size of the cone is dependent on the position of the light inside the can. Adjusting the position of the light changes the range of angles through which the light is allowed to leave and therefore resizes the cone of light. This is shown in Figure 2.16.

For the sake of simplicity, Figure 2.16 shows the cones with the assumption that the light inside has no physical size. In the case of an ideal point light, the can would either block or not block the light at a given angle. In the case of an area light, the situation is less clear-cut. At some angles (measured relative to the axis of the light), the can will not block out the light at all. This portion of the spotlight cone is

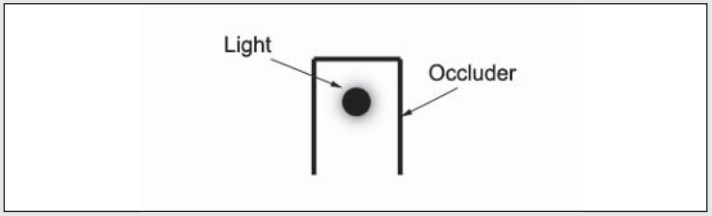


Figure 2.15. A basic spotlight

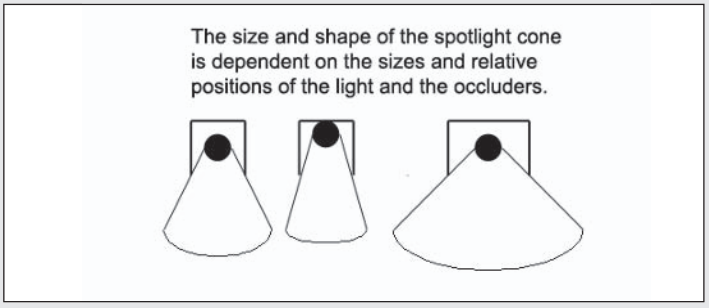


Figure 2.16. Resizing the cone of light

called the *umbra*. Points within the umbra will be lit as if there were no can at all. At larger angles, the can will begin to block an increasingly large portion of the light. This area is called the *penumbra*. Points within the penumbra will be lit less as they approach the edge of the penumbra. Finally, there is an angle at which the can blocks the entire light. The sizes of the umbra and penumbra are affected by the position of the light within the can and the size of the light relative to the size of the can. Figure 2.17 shows different spotlight variations and how they affect the umbra and penumbra.

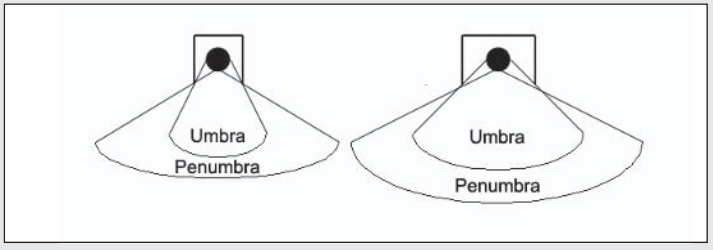


Figure 2.17. Changing the umbra and penumbra

Notice how the sizes of the umbrae nearly double, but the penumbrae are roughly the same size. This is because of the positions and the sizes of the lights. In this example, it just happened to turn out that way. Other configurations could yield different results.

You can verify these effects with a physical light, or you can build your own spotlight in a 3D rendering program using an area light and an occluding shape. In real-time graphics, there are simpler alternatives that boil the concepts above down to a few equations.

NOTE:

Be careful if you make your own physical spotlight as they can get quite hot. All the light that doesn't escape is changed into heat. Conservation of energy strikes again.

2.4.2 Spotlights and Lighting Equations

A spotlight is really just an extension of an area light when the lighting effect is constrained by the angles of the umbra and penumbra. Attenuation over distance can be computed using the equation shown in section 2.3.2, but you need extra terms for the effects of the occluding shell and the falloff within the penumbra. Therefore, you need terms that describe attenuation in terms of an angle relative to the axis of the light.

First, the spotlight itself needs three new parameters for light direction and the angles of the umbra and penumbra. This is shown in Figure 2.18.

When you compute the intensity of a point light on a surface, you are only interested in the distance between the light and a point on that surface. Now you need to know the angle between the light direction and the light-to-point direction. This is shown in Figure 2.19.

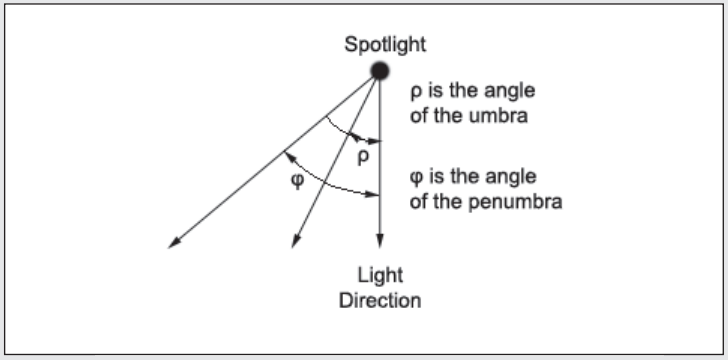


Figure 2.18. Extending an area light to a spotlight

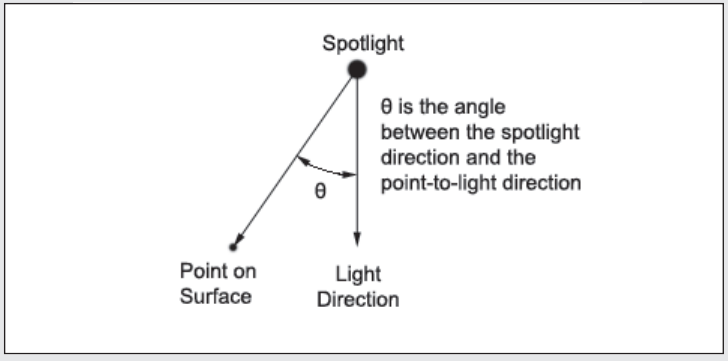


Figure 2.19. Finding the angle between the light direction and the light-to-point direction

Based on this, the equations for the intensity of light inside the umbra and outside the penumbra become trivial.

$$I = 1.0 \text{ if } \theta < \rho$$
$$0.0 \text{ if } \theta > \varphi$$

Within the penumbra, the attenuation caused by the occluding shell increasingly blocking the light is called *falloff*. In the real world, the rate of falloff is dependent on the light shape and other factors, but it can be approximated by a simple flexible equation (much like the extended attenuation equation). This equation is a function of the spotlight angles, the angle between the axis and light-to-point vector, and some falloff factor *F*. Therefore, the following becomes the equation for the intensity of light within the penumbra.

$$I = 1.0 \text{ if } \theta < \rho$$
$$0.0 \text{ if } \theta > \varphi$$
$$\text{else}$$
$$= \left(\frac{\cos\theta - \cos\rho}{\cos\varphi - \cos\rho} \right)^{F}$$

As you can see, computing the intensity of light within the penumbra can be quite computationally expensive. In many cases, using inverse attenuation and a falloff of 1.0 can yield good results with considerably less overhead.

2.4.3 Other Spotlight Models

The explanation above was designed to give a physical rationale for the most commonly used attributes of hardware spotlights, but there are other ways to model the light itself. At the most basic level, spotlights are light sources that are partially blocked by some form of occluder or reflector. Different shapes of occluders will shape the outgoing light field. Also, different lenses might shape the light field, although I won't get into that explicitly.

To simplify things, consider only a point light. An unoccluded point light will emit light in a uniform spherical field as seen in the early part of this chapter. If you take the same light source and place a planar reflector next to it, the shape of the emitted field will look similar to Figure 2.20.

In the direction of the plane's normal vector, the intensity of the light doubles. This is because light that strikes the plane parallel to the normal is reflected back in the direction of the normal. For essentially the same reason, very little light is emitted perpendicular to the normal because light tends to be reflected away from the plane rather than along it. The exact shape and orientation of the light field will depend on the distance from the light to the plane.

Likewise, more complex occluders will shape the light field in different ways. In general, the shape of the emitted light could take the shape of something much more ellipsoid than spherical. Figure 2.21 shows another equation based on Warn's spotlight model.

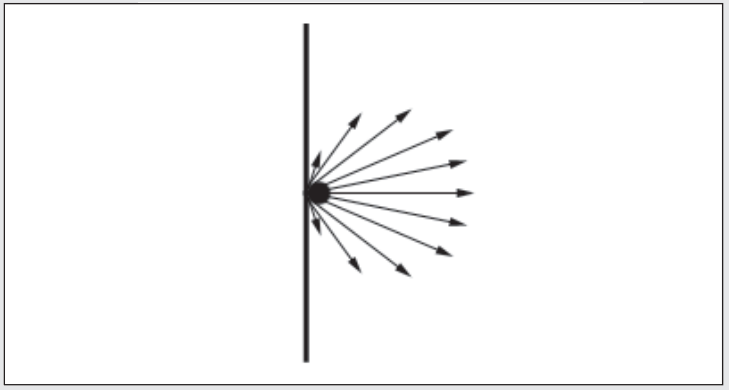


Figure 2.20. A point light with planar reflector

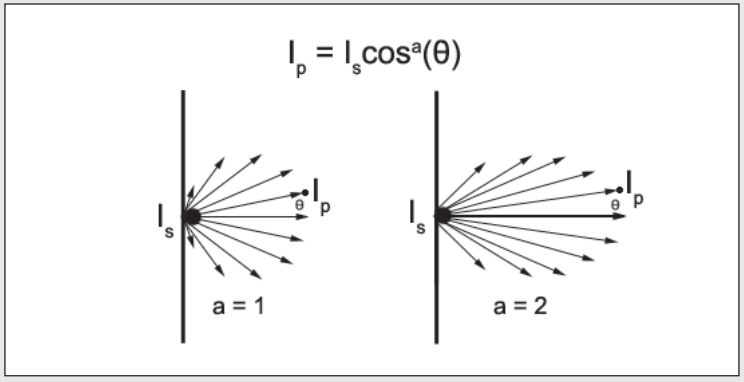


Figure 2.21. Warn's spotlight model gives a different equation for spotlights.

For some applications, it might make sense to model the spotlight to correctly match some real-world light source. For others, a general shape with a set of desirable characteristics might suffice. For instance, Figure 2.21 shows a spotlight model that is probably appropriate for a handheld flashlight, although the values might not exactly match any specific real-world flashlight.

2.5 Global Illumination

This chapter has explained how to model real-world lights for the purpose of reproducing them in a virtual scene. The equations are based on the relationship between a light and a point on a surface. In the real world, surfaces receive light from other surfaces even though they are not directly affected by a given light.

If you imagine sunlight (a directional light) streaming through a window, the sunlight will never directly hit the ceiling of the room, yet the ceiling is lit. This is because sunlight enters the room, reflects off the floor and the walls, and eventually hits the ceiling. This effect is ubiquitous in the real world, but is very difficult to reproduce in a virtual scene.

2.5.1 Global Illumination vs. Local Illumination

Most real-time 3D graphics are rendered using a local illumination model. This means that each element of the scene is lit as if there were no other elements in the scene. Figure 2.22 shows how a simple scene is lit with local illumination.

This type of rendering is not very realistic, and your eyes tell you that the lighting isn't correct. Most obviously, there is no shadow where the ball would block the light from hitting the floor. There should also be some amount of light reflecting off of the floor and hitting the ball. These second-order reflections would light the underside of the ball and would be affected or attenuated in some way by the floor. For instance, light reflected from a red floor would light the underside of the ball with a red light. Likewise, light that reflects from the floor to the wall to the ball would be a third-order reflection and have some of the properties of both the wall and the floor. Figure 2.23 is the same scene lit with global illumination. The effect is much more realistic.

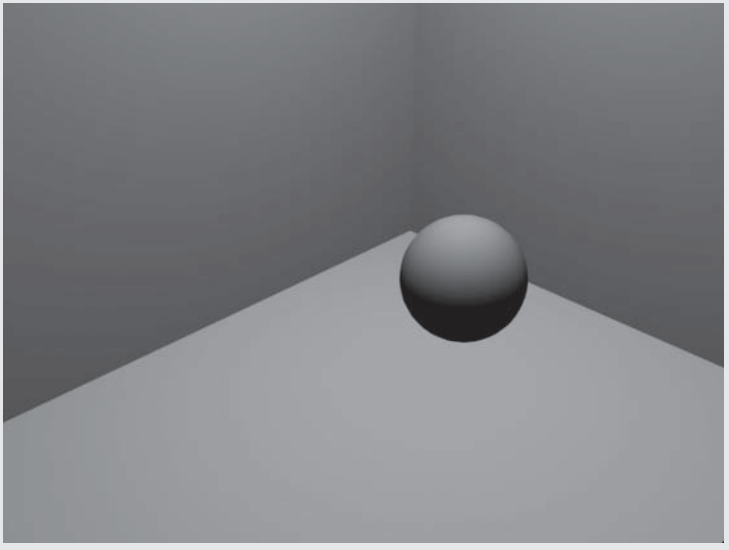


Figure 2.22. Each object is rendered as if there were no other objects.

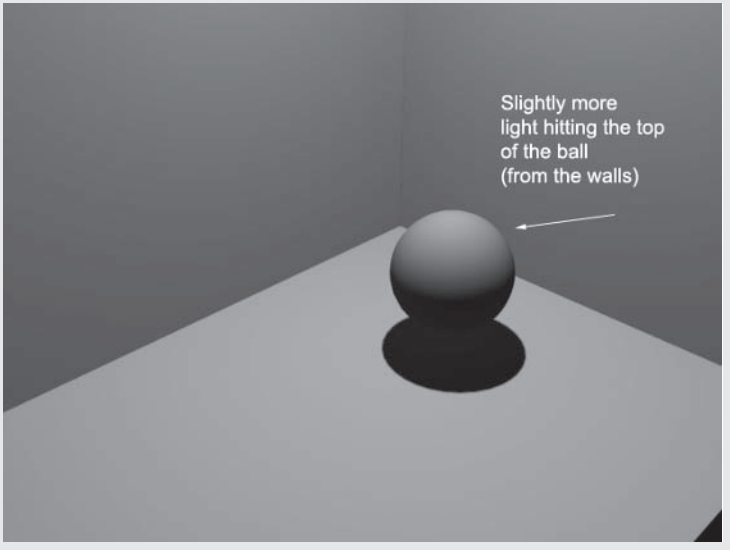


Figure 2.23. Global illumination accounts for reflections between objects.

This effect is very difficult to reproduce in real time because it is very difficult to account for all of the possible interactions between objects. The next chapter briefly describes some offline global illumination techniques, but the most common approximation of the effect is to use ambient light.

2.5.2 Ambient Light

Ambient light is used in most scenes to minimize unlit areas and to approximate global lighting. Although it can be valuable, it should be used sparingly. The problem is that it increases the brightness of everything in the scene, which can wash out detail. For instance, the scene in Figures 2.22 and 2.23 includes a single directional light. In Figure 2.22, the light illuminates the top of the object but not the bottom,

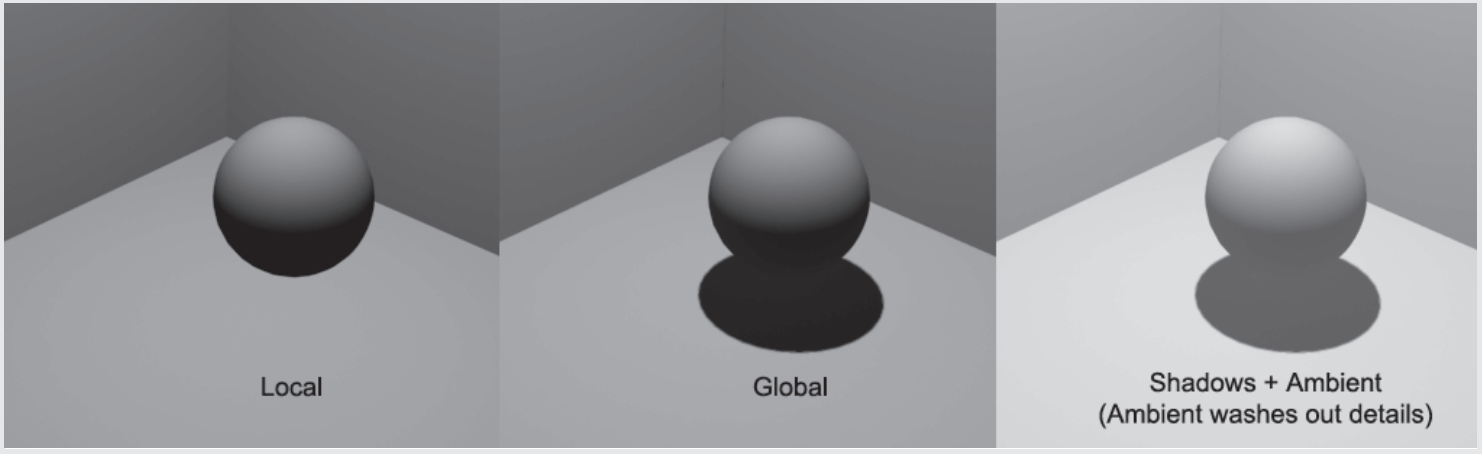


Figure 2.24. Ambient lighting is not a good global lighting technique.

producing a harsh, unrealistic shadow. You can light the underside if you increase the ambient light, but in doing so, you will increase the brightness on the top of the object. This is not really correct either. Figure 2.24 shows the differences between local illumination, global illumination, and local illumination with ambient lighting.

Ambient light should be used sparingly. As hardware capabilities increase, it becomes more reasonable to approximate global effects with a wider array of lights or other techniques. The chapters on different materials will concentrate on local illumination, but the properties of the materials will remain constant across most lighting models. As you experiment with different lights and scenes, keep the effects of global illumination in the back of your mind.

Conclusion

Many readers will have used lights without really understanding why light is attenuated or how directional lights relate to point lights. For some, a deeper understanding isn't really a necessity. However, a deeper understanding of the underlying principles can be very useful when optimizing a shader, choosing attenuation coefficients, or choosing which types of lights to use in your scene. Very rarely will you need to worry about how many candelas a given light emits, but an understanding of all of the underlying concepts might help you when you are tweaking light values. As you get into more complex techniques, you will find that greater depth of knowledge leads to better intuition.

I have not included a sample application for this chapter simply because lighting is also dependent upon material properties, which I have not yet discussed. Also, many books and SDK examples feature different light types. My goal in this chapter was to discuss those common types with more depth than you commonly see.

Most of the remaining chapters will use simple directional lighting to demonstrate material properties. Directional lighting keeps the equations relatively simple, and the material properties are the same for other light types. Chapter 6 will discuss how to implement the different light types in vertex and pixel shaders and these shaders will be carried forward to the subsequent chapters. If you like, you can skip to Chapter 6 now and see how the lights are implemented, or you can continue on to Chapter 3, which discusses raytracing and other global illumination techniques.

Chapter 3

Raytracing and Related Techniques

Introduction

The first two chapters presented the physics behind lights and the different types of lights that can be found in the world that surrounds us. In this chapter, I present an algorithm that uses the physical laws of light rays to produce photorealistic images. As you can see by this simple definition, the raytracing algorithm, as well as its extensions and related techniques, is primarily a rendering algorithm, not a shading algorithm per se. However, it is simple and elegant enough to be understood without advanced skills or knowledge in mathematics and physics. And, as with any rendering algorithm, it involves the use of various shading techniques, such as those presented in this book.

In this chapter, I talk about what raytracing is and what it covers, both in terms of lighting and shading, and give you enough knowledge and pointers to start implementing a simple raytracer. I also explain why, presently, it cannot be used in game engine programming, even though technology is getting closer to achieving this!

The last sections of this chapter present the different techniques and algorithms related to Monte Carlo raytracing that provide even more photorealistic renderings.

3.1 The Raytracing Algorithm

The basic idea behind the raytracing algorithm is to simulate the propagation of light in the virtual world using the physical laws of optics. In this section, I explain the different aspects of the algorithm.

3.1.1 Backward Raytracing

In the real world, light rays emanate from light sources. The rays travel through different types of mediums (air, water, etc.), bounce off reflective surfaces (mirrors, plastic surfaces, etc.), go through translucent materials, etc., many times before reaching the eyes of an observer. Raytracing simply tries to reproduce this complex process in a simple way by modeling how light physically interacts with the objects in a virtual scene.

In raytracing, the observer is like a camera. Just like a camera receives a portion of the light that has traveled in the scene and made it to the lens, the observer also receives a portion of that light. However, not all rays emanating from the various lights of the scene will reach the observer. In fact, only a very small portion of those rays will reach the observer. Therefore, it is much more efficient to trace rays from the observer out into the world than trying to trace all of the rays coming from all of the lights that, for the most part, will never reach the observer. Raytracing does exactly this by sending out rays in the surrounding world, looking for possible intersections with objects and finding the color of the surface at the point of intersection depending on the surface properties (reflection, transparency, etc.) and the position of the object relative to the different light sources. Since this process is achieved in reverse order from what happens in the real world, raytracing is in fact backward raytracing.

To illustrate this, let's take the example of a scene composed of several lights and objects. The observer is a camera: It has a position in space, it is looking in a given direction, and it captures the color information it receives from the surrounding world on film. With backward raytracing, you simply do the following:

1. The film of the camera is divided into a grid of discrete elements (picture elements — also known as *pixels*). See section 3.1.2, "Camera Models," for more explanation on how the grid is created. The goal is to determine the color of the pixels forming the image (*image plane*) as shown in Figure 3.1.
2. Through each of these pixels, trace a ray from the position of the camera in the direction that the camera is pointing, as shown in Figure 3.2.
3. For this ray, check if it intercepts an object in the scene. If the ray intercepts an object, proceed to step 4. If it does not intercept an object in the scene, assign the background color of the scene to the current pixel. Additionally, you can add effects (such as fog, for instance) or assign a color to the current pixel based on a background image. Continue to the next pixel and return to step 2.

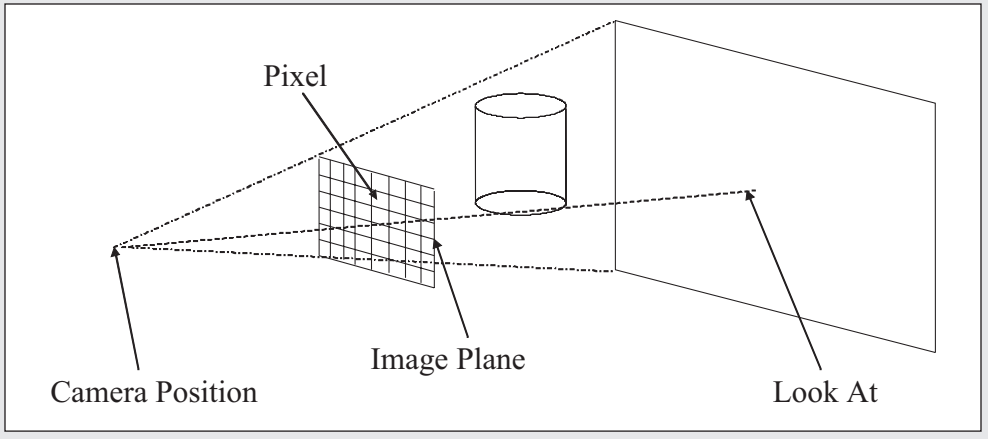


Figure 3.1. Camera model and projection

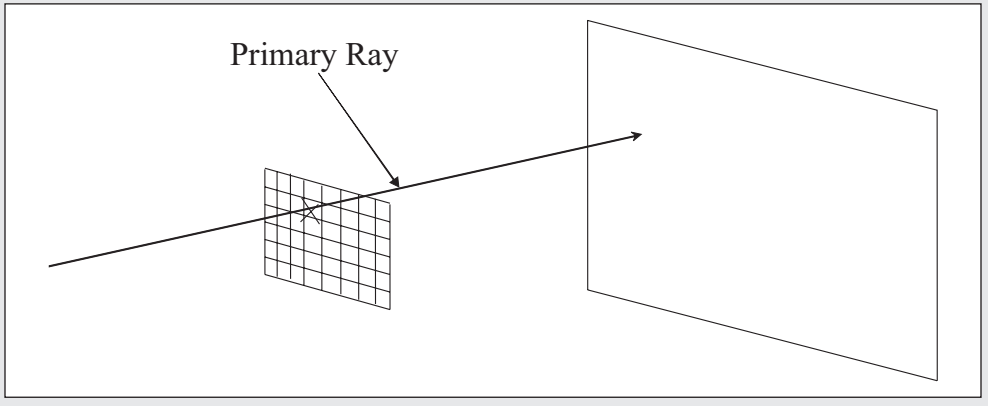


Figure 3.2. Primary ray

4. If the ray does intercept an object in the scene (see Figure 3.3), compute the color at the point of the object intersection. Since the pixel on the image plane is the projection of the point of intersection, determining the color at this point is the same as determining the color of the pixel. This is performed in several steps:
   a. First check for the contribution of light energy by each light. To do this, trace new rays toward each light (*light ray* or *shadow ray*) as shown in Figure 3.4. It is therefore possible to determine if the object, at the point of intersection, is completely lit, partially lit, or not lit at all. During this step, you simply determine shadows.
   b. If the surface is reflective (for example, a mirror), calculate the direction of reflection of the initial ray (see Figure 3.5). Then trace another ray (*reflection ray*) and repeat the process outlined in step 3.

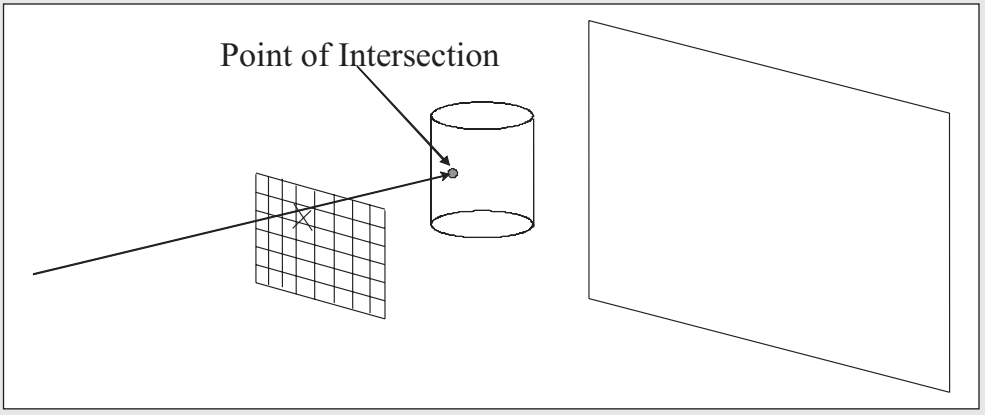


Figure 3.3. Point of intersection and color

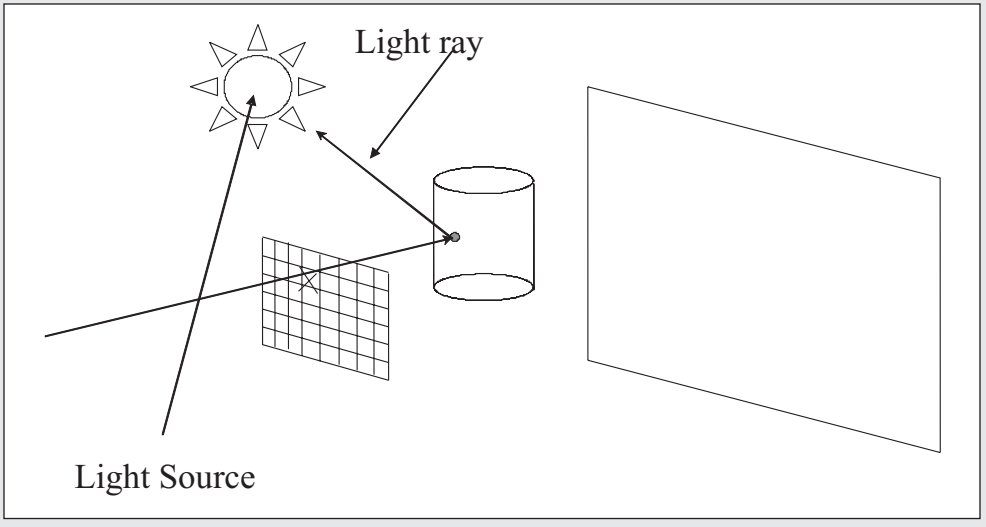


Figure 3.4. Shadow and light rays

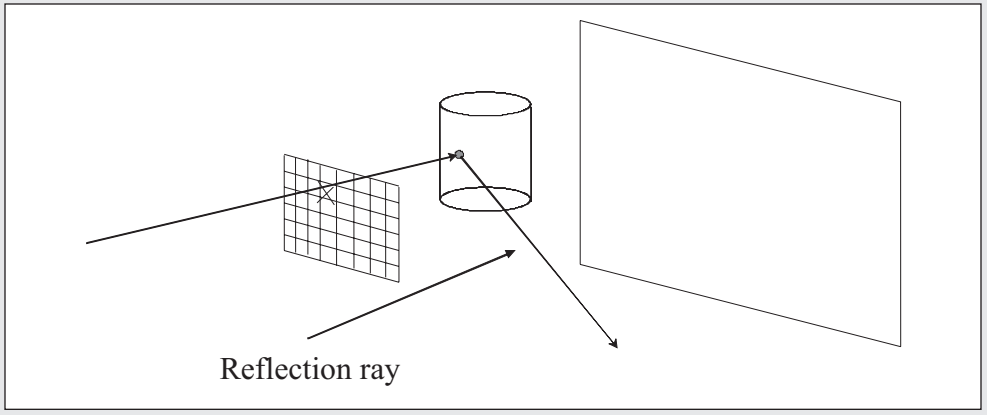


Figure 3.5. Reflection ray

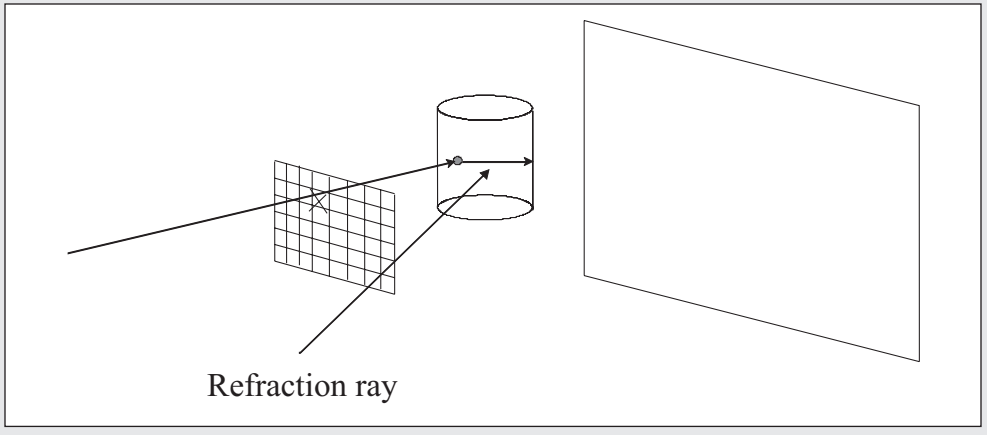


Figure 3.6. Refraction ray

c. If the surface is refractive (for example, a glass of water), calculate the direction of refraction of the initial ray (see Figure 3.6). Then trace another ray (*refraction ray*) and repeat the process outlined in step 3.

d. Finally, based on the surface properties (reflection index, refraction index, etc.) and the colors calculated with the different types of rays sent out into the virtual world, it is possible to determine the final color of the point of intersection, thus determining the color of the image pixel.

5. Continue to the next pixel and return to step 2. Continue this process until there are no more pixels at either step 2 or step 5.

3.1.2 Camera Models

In the previous section, I referred to the different positions from which the world is seen as the different positions of the observer. The observer is represented by a camera in computer graphics; its role is to give a 2D representation of the 3D world. To go from the 3D coordinates of the world to the 2D representation of an image, you use projections. There are several types of projections, each corresponding to a different camera model: perspective projection, spherical projection (fish-eye), orthographic projection, cylindrical projection, ultrawide-angle projection, panoramic projection, etc.

The most commonly used camera model is that of the pinhole camera. It corresponds to the perspective projection. In this model (perspective projection), there is a center of projection, which corresponds to the position of the camera (observer). The camera is pointing in a direction and has a field of view. The model is represented as a pyramid (called the viewing pyramid or viewing *frustum*). The position of the observer is at the point of the pyramid. The field of view (FOV) is defined as an angle (the viewing angle). With small angular values, the region of the world seen from the position of the observer will be smaller, and faraway objects will appear bigger. With large values, the opposite effect is produced.

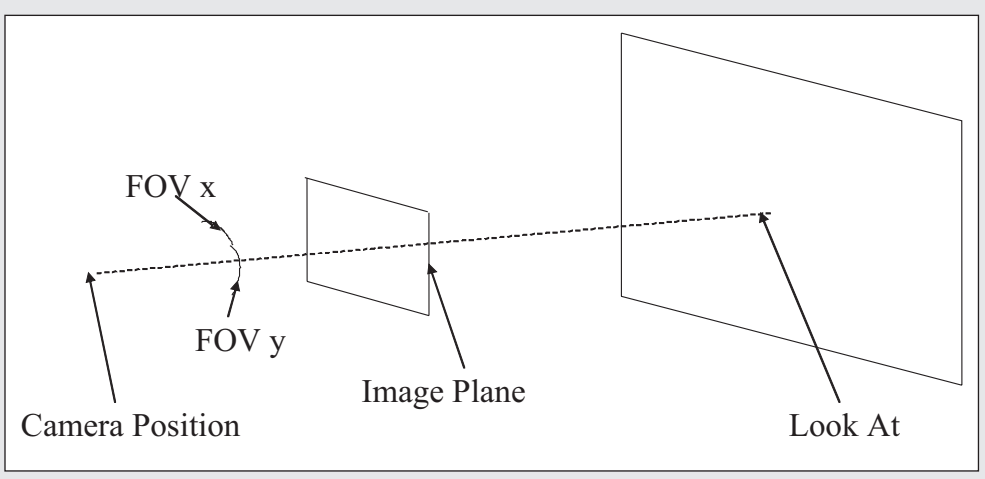


Figure 3.7. The pinhole camera model

Figure 3.7 summarizes the pinhole camera model (perspective projection).

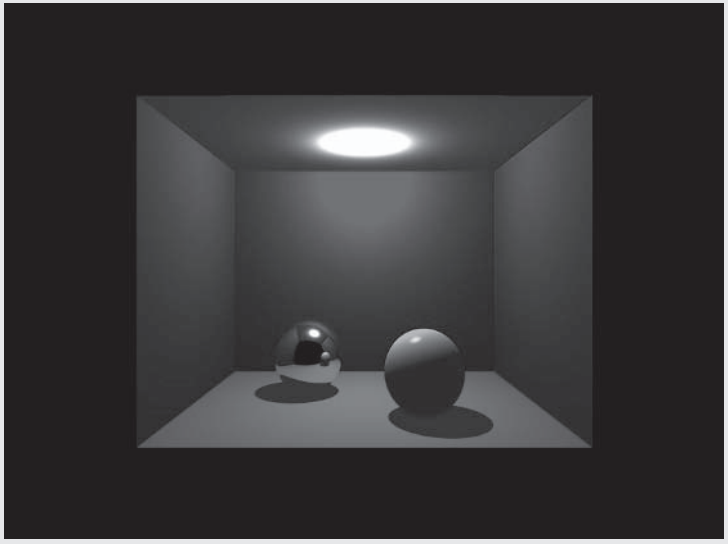


Figure 3.8. Simple scene with a field of view of 80 degrees

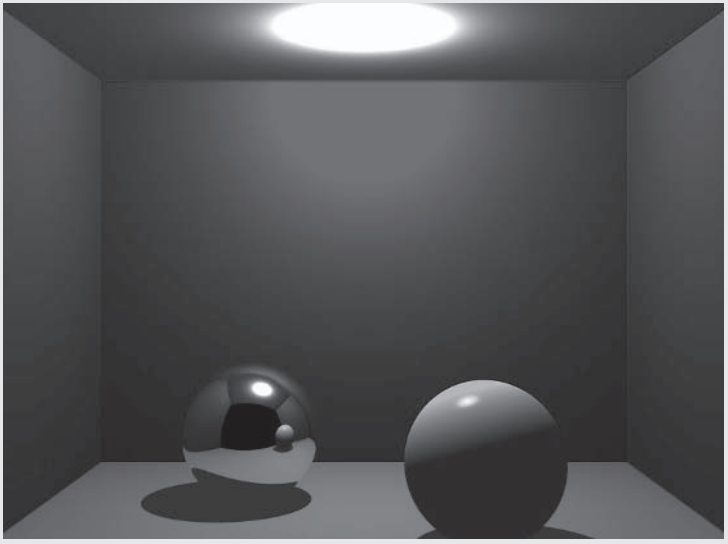


Figure 3.9. Simple scene with a field of view of 45 degrees

The image in Figure 3.8 is a simple scene rendered with a large field of view (80 degrees). Changing the field of view angle is called zooming. Making the angle smaller is called zooming in, and making the angle larger is called zooming out.

The image in Figure 3.9 is the same scene rendered with a smaller field of view (45 degrees).

These two images clearly show the zoom effect in the pinhole camera model.

3.1.3 The Different Types of Rays

From the brief explanation of the algorithm that I gave in the previous section, you can see that there are four main types of rays involved in the calculation of the color of a pixel: primary, shadow/light, reflection, and refraction.

- Primary rays: These rays are traced from the observer out into the world. The pinhole camera model (any other camera model can be used, only the projection formulae will be different) is used to trace these rays as shown in Figure 3.10.
- Shadow (or light) rays: These rays are traced from the point of intersection to the different lights of the scene. For each of the lights in the scene, you trace a ray in the direction of that light. If the ray doesn't intersect an object of the scene before it reaches the light, then that light source contributes to the lighting at the point of intersection. If, on the other hand, the ray does intersect an object in the scene, the original point of intersection will be

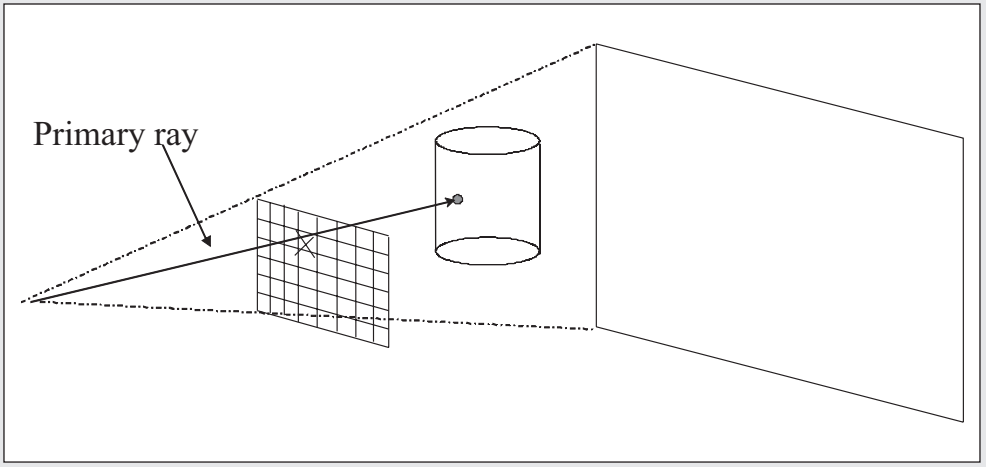


Figure 3.10. Primary ray and camera model

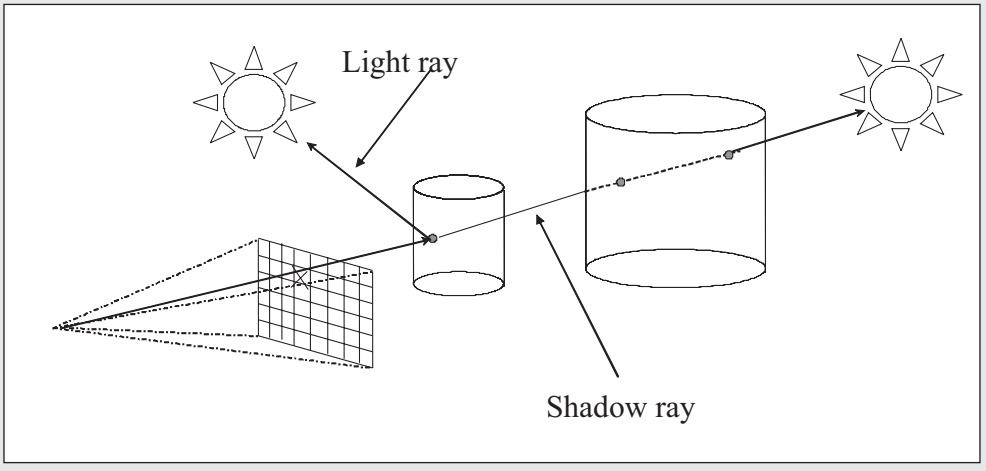


Figure 3.11. Shadow rays and light rays

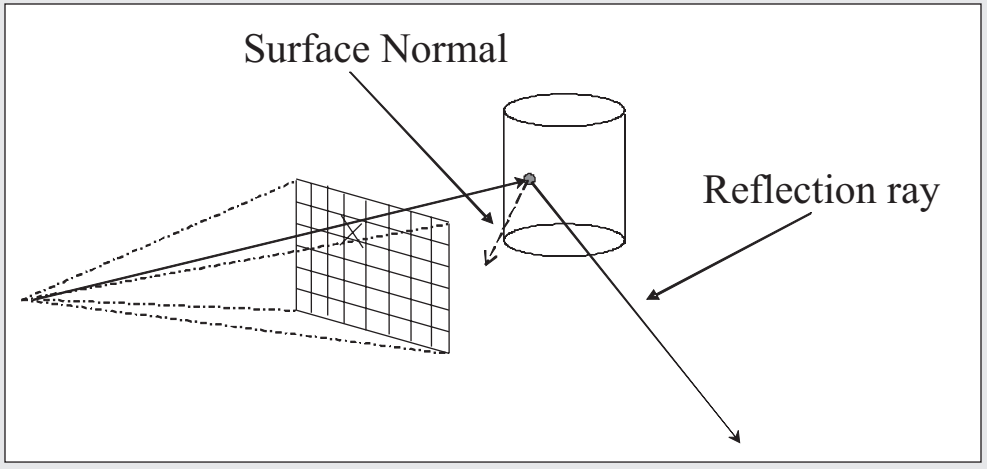


Figure 3.12. Reflection rays

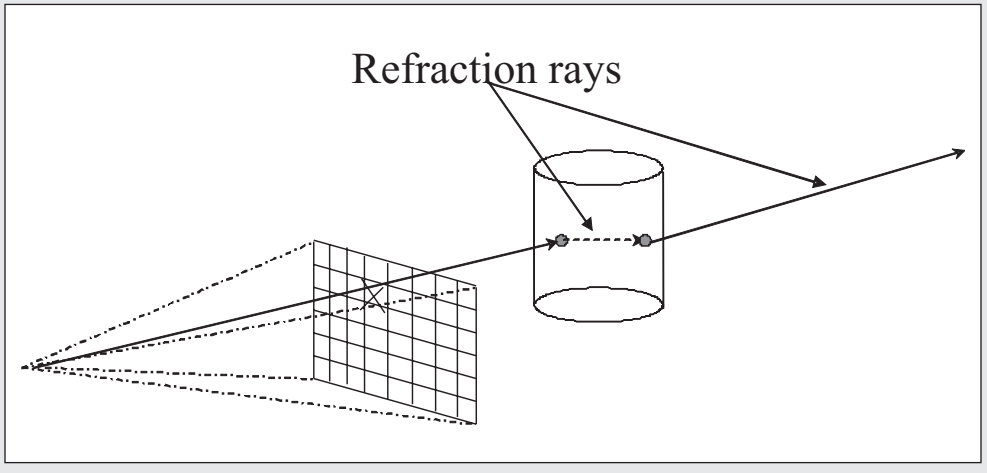


Figure 3.13. Refraction rays

in the shadow of that light source, and that light source will not directly contribute to the lighting at the original point of intersection. This is illustrated in Figure 3.11.

- Reflection rays: When an object's surface is partially or totally reflective (for instance, a mirror), part of the light energy bounces off the surface and continues its path in the scene. Reflection rays are used to simulate this effect. A new ray is traced from the point of intersection out into the world following Descartes-Snell laws, as shown in Figure 3.12.
- Refracted rays: When an object's surface is refractive and partially transparent (for example, at the surface of a glass of water), part of the light energy continues its path inside the object. Refraction rays are used to simulate this effect. A new ray is traced from the point of intersection and into the interior of the object following Descartes-Snell laws, as illustrated in Figure 3.13.

3.1.4 Recursion

As you saw in the previous sections, the raytracing algorithm is, in essence, a recursive algorithm. You simply trace rays in different directions, depending on the surface properties and the different light sources present in the scene, and create new rays as you intersect the various items.

As with any recursive algorithm, you must be careful to stop the process at some stage to avoid infinite recursion. Indeed, let's take the example of two parallel mirrors facing each other. When a reflective ray is traced from one of the mirrors, there is a possibility that this ray might be normal to the surface of the mirror. When hitting the opposite mirror, the algorithm will create a new reflective ray, normal to the surface of that mirror and pointing in the direction of the opposite mirror (the first mirror). At this stage, the ray will be trapped, and the recursion process will never end. Every single ray traced from one mirror to the other will lead to the creation of a new ray on the opposite mirror, and so on, and so on. If you don't set a maximum level of recursion, your raytracing program will never end!

The other reason to set a maximum level of recursion is subsequent rays' contribution: Generally, surfaces are partially reflective and/or refractive, and after a few iterations, the contribution of the rays traced won't be significant to the final color of the pixel. However, the process will have been executed several times for a result that is nearly imperceptible, thus spending precious CPU cycles for nothing useful.

3.1.5 Ray and Object Intersections

At the core of a raytracing algorithm you need to calculate the intersections between a given ray and the objects of the scene. Every time you trace a ray, you need to check for intersections with the objects in the scene. This is a simple mathematical problem of a line intersecting a geometric object.

Even though you could solve the mathematical equations behind every single intersection test, because of the huge number (in millions) of rays traced in a typical scene, this represents an even larger number of calculations than is necessary if you implement some optimizations. These extra calculations simply translate into long rendering times. As you will see later, it is paramount to not only tune the intersection functions, but also, and above all, to try to minimize the number of calls to these functions. Some of the different methods to get around this problem are listed in section 3.1.8.

3.1.6 Texturing and Shading

When a ray hits a surface, you need to determine the normal vector as well as the color at the given point of intersection. This consists of two main steps: texturing and shading.

The texturing phase basically consists of finding the intrinsic color of the surface hit by the ray and determining the surface normal. They can both originate from data (color and normal mapping) or be determined mathematically (procedural texturing). For instance, it is possible to use a 2D picture as an image map to determine the color on a surface. A scanned picture of grass can be used to give a more realistic appearance to the ground in the scene. This is called image mapping (or texture mapping), and will be described in more detail in later chapters in the context of real-time rendering. It is also possible to use a 2D picture to modify the normal of the surface; this is called bump mapping. Usually, grayscale images are used in this case to perturb the normal based on the color of the bump map at that point. The surface of the object is no longer flat, but more or less bumpy (a black pixel in the bump map corresponds to low altitudes on the surface, white pixels correspond to high altitudes, and gray pixels are in between). In later chapters you will see bump mapping techniques applied to real-time rendering.

Procedural texturing consists of mathematically describing the variation of color and normal vector on the surface. It is possible, for instance, to mathematically make the surface look like wood or marble. It is also possible to make it look like the surface of a pool with waves of different heights without having to model that surface using numerous polygons. Texturing is a vast subject that could be covered in several books. See [1] for a very good reference on the subject.

The shading phase is what this book is all about! Most (if not all) techniques presented here can be used to shade the objects of a scene rendered using the raytracing algorithm. Most raytracing programs use the following simple shading models:

- Perfect diffuse reflection: For a non-directional point light source, perfect diffuse reflection takes into account the contribution of a light source based on the angle between the surface normal and the direction of the light source as seen from the point of intersection (angle α). This is summarized in Figure 3.14.

- Perfect diffuse transmission: In this case, the incident ray passes through the surface of the object and is partially attenuated. This is illustrated in Figure 3.15.
- Perfect specular reflection: This corresponds to the contribution of the reflected ray. This doesn't take into account absorption or reradiation of the light from the object. This is described in Figure 3.16.

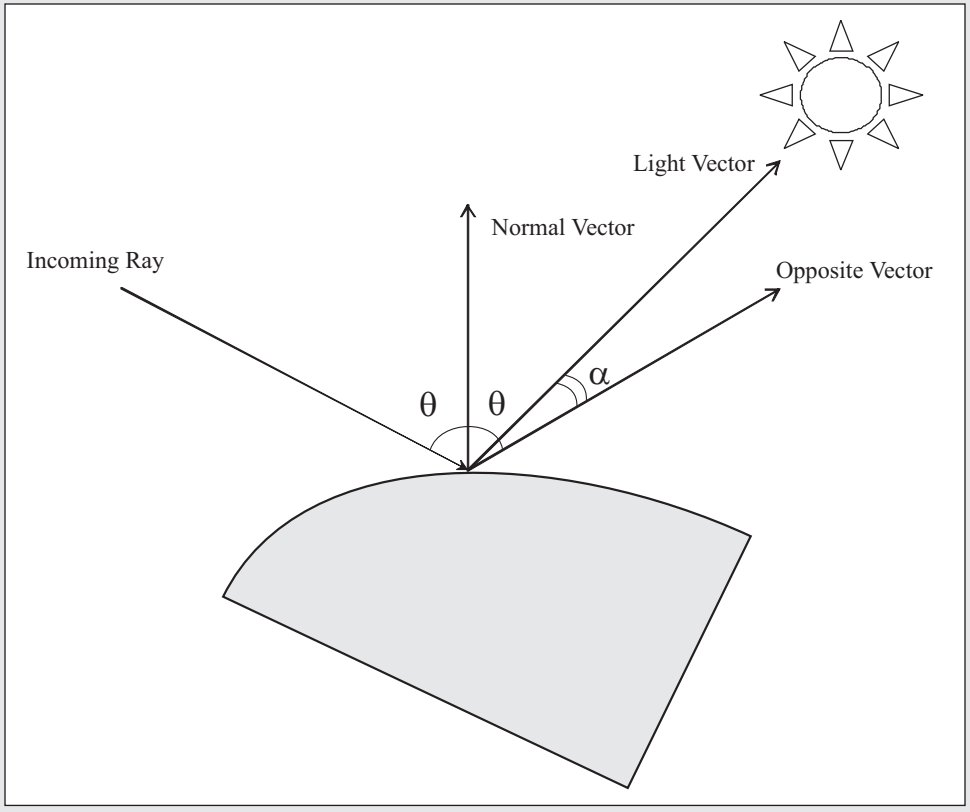


Figure 3.14. Perfect diffuse reflection

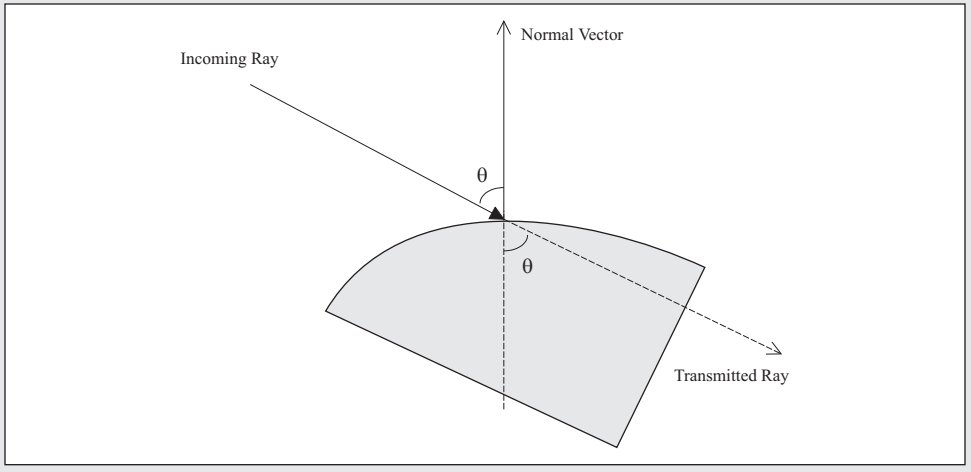


Figure 3.15. Perfect diffuse transmission

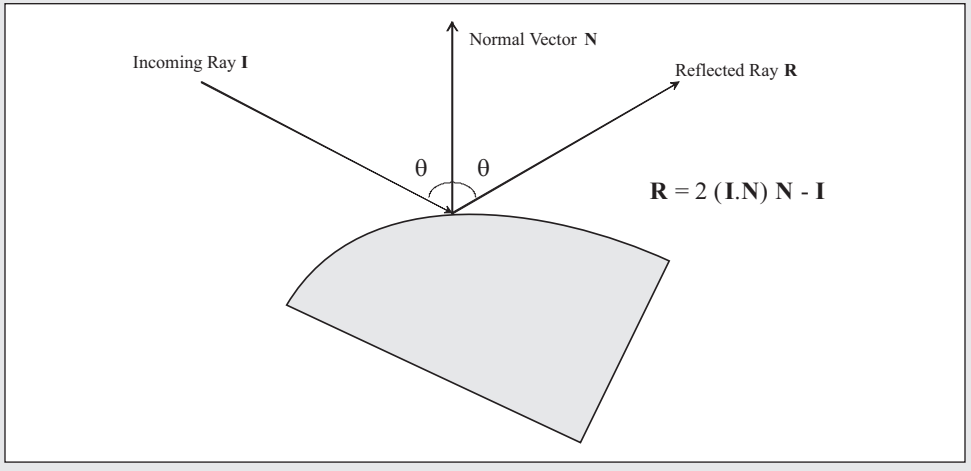


Figure 3.16. Perfect specular reflection

- Perfect specular transmission (refraction): In this case the ray is refracted according to Snell-Descartes laws, as explained in Figure 3.17.

You will see later how the different elements of this simple shading model can be applied and implemented in the context of real-time rendering (e.g., in a gaming engine).

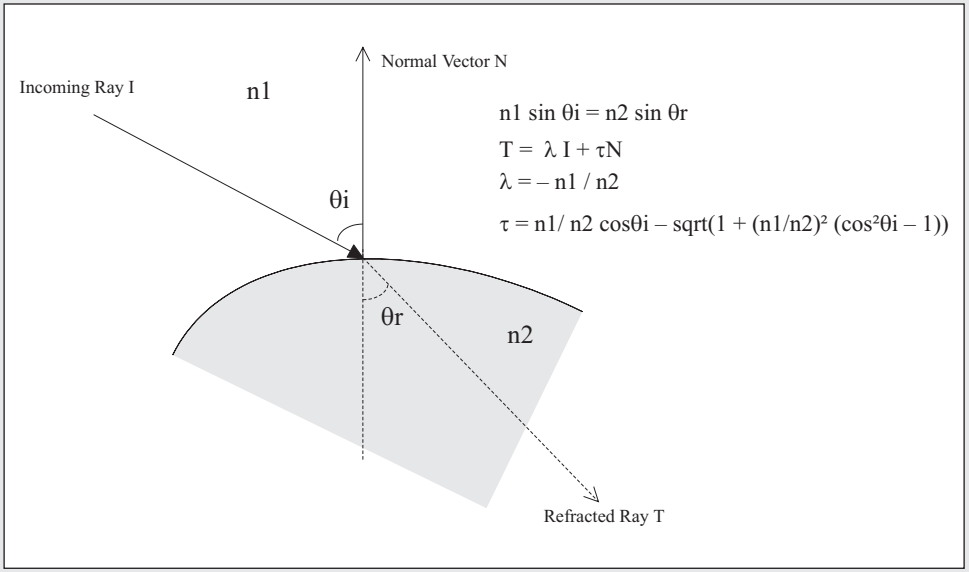


Figure 3.17. Perfect specular transmission (refraction)

3.1.7 Problems and Limitations

As with any other algorithm, raytracing comes with its inherent problems. The following list describes the most common problems and limitations encountered with raytracing.

- Performance: This is the main limitation of the algorithm. Because of its recursive nature and the number of rays traced, images rendered can take from a second (or even less) to several hours. On average, 80 to 90 percent of the time spent to render an image is dedicated to calculating the intersections between the rays traced and the objects of the scene. It is therefore paramount to tune the intersection routines, as well as to try to limit the number of calls to these routines as much as possible. Several techniques exist to limit this number of calls to intersection routines; I list some of them in the next section.
- Aliasing: Aliasing artifacts, in computer graphics, most commonly take the form of stairsteps or jaggy edges on the screen. This is due to the fact that you use a screen with a finite number of pixels and that you trace a finite number of rays. The image in Figure 3.18 shows a typical aliasing artifact in a raytraced scene (*spatial aliasing*). Aliasing effects also appear when rendering subsequent frames of an animation (*temporal aliasing*). You will see in the next section that antialiasing algorithms can come to the rescue to eliminate these artifacts.
- Sharp shadows: The basic raytracing algorithm can only produce sharp shadows (because it simulates point lights). Although this can be the case in the real world, most shadows you see around you are not sharp. Additional techniques can be used to produce soft shadows (also known as smooth shadows or penumbra). Figure 3.19 shows an example of a raytraced scene with soft shadows.
- Local lighting and shading: By its nature, the raytracing algorithm traces only a very small number of rays in the scene. As you saw previously, only four types of rays are traced recursively: primary rays, shadow/light rays, reflected rays, and refracted rays. As a result, interdiffuse reflections of light between objects are not taken into account. Simply put: The raytracing algorithm doesn't include global illumination.

Fortunately, for all problems and limitations, solutions and workarounds exist! This is the topic of the next section.

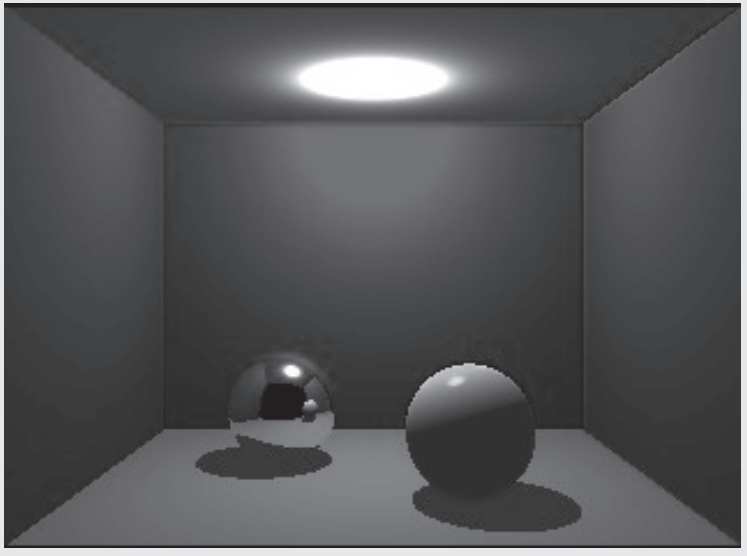


Figure 3.18. Aliasing artifacts in raytracing

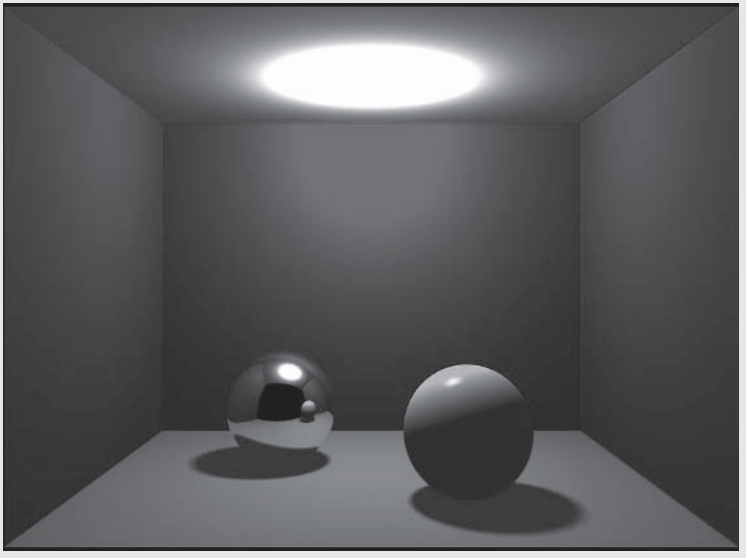


Figure 3.19. Soft shadows

3.1.8 Solutions and Workarounds

For the problems and limitations I enumerated in the previous section, I now give some solutions and workarounds:

- Performance: As you saw in the previous section, the main problem with raytracing is performance. A huge number of rays are traced in the virtual world, and each ray can potentially intersect an object of the scene. On average, 80 to 90 percent of the rendering time is spent on trying to figure out if rays intersect objects. Several tricks and techniques can come to the rescue to help reduce rendering times in raytracing:
  - Use more or better hardware: The first obvious solution is to throw more hardware at it! Moore's law tells you that computers double in speed (hence in performance) every eighteen months. Although easy to put into practice, this solution isn't the most effective one, or applicable in all cases. Even today, the fastest computer will come to a crawl when running a poorly written raytracer. Better solutions exist; most of them can be combined to achieve even better performance.
  - Massively parallelize computations: The raytracing algorithm is massively parallel in essence. As a matter of fact, all rays traced are independent from each other. This means that the raytracing algorithm can take advantage of parallel computing. This approach consists of splitting the image to render into zones that are then assigned to different instances of the same raytracing algorithm running on a multiprocessor machine or running on different networked machines (grid-computing, clusters, etc.). This also means splitting the image to render into zones that are assigned to different threads of the same raytracing program. Parallelizing a raytracer (or any other piece of software as a matter of fact) is an art in itself — it is tricky but gives very good results, and most of all is a rewarding activity!
- Limit the number of intersection tests: Many techniques exist to try to limit the number of intersection tests. The most common techniques use hierarchies of bounding volumes — the idea is to quickly determine if a ray will or will not intersect a group of objects. The objects are grouped and enclosed in bounding volumes for which the intersection tests are quicker and easier to calculate. For instance, one can use bounding boxes or bounding spheres (the sphere is the easiest and quickest shape to test against a ray intersection). Other techniques use spatial subdivision to limit the number of intersection tests such as octrees, BSPs (binary space partitioning), and grids. More information can be found in [2].
- Optimize the intersection tests: Many algorithms have been created to optimize the intersection tests between a ray and all sorts of objects. Some of these algorithms are presented in [3].
- Limit the number of rays traced: Examples of such techniques include, for example, using an adaptive recursive algorithm. Instead of setting a maximum level of recursion levels, the idea here is to limit the recursion based on the contribution of subsequent rays

to the current pixel's color. A threshold is set over which subsequent rays won't be traced because their contribution to the final pixel color will be almost insignificant (i.e., their contribution will be imperceptible in the final rendered image), but the time to trace them will increase the overall rendering time. This technique can be used for all types of rays (primary, light, reflected, and refracted).

- Aliasing: The aliasing artifacts inherent to raytracing can be reduced greatly with the help of techniques such as super-sampling, adaptive antialiasing, and jittering.
  - Trace additional primary rays and average the results — also known as *super-sampling*: Instead of tracing one ray per pixel, you trace a given number of rays through each pixel. Each pixel is thus divided into sub-pixels, through which you send out primary rays. When all sub-pixels have been processed, you simply average the different colors and

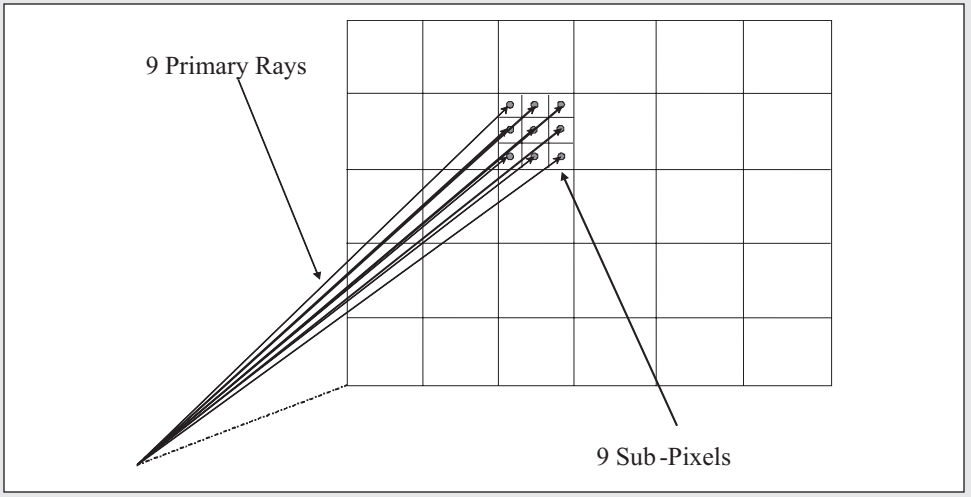


Figure 3.20. Antialiasing with super-sampling

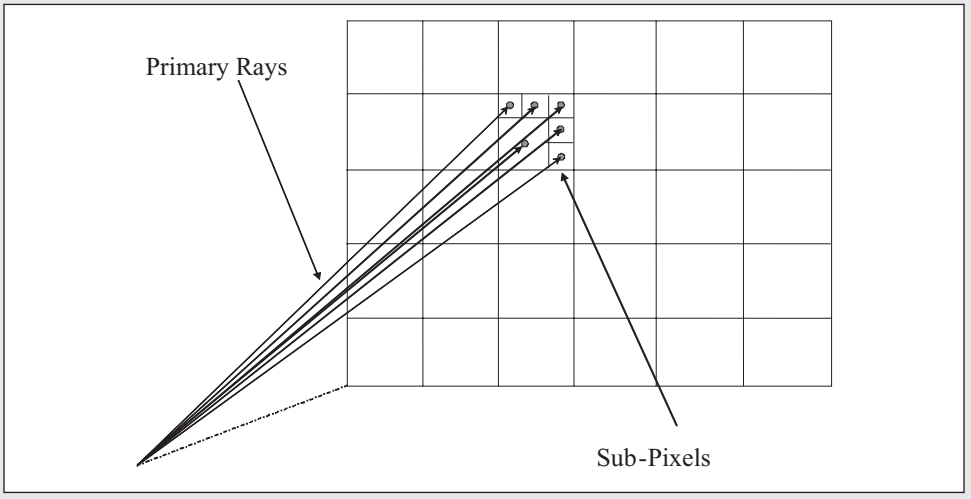


Figure 3.21. Antialiasing and adaptive sampling

assign this value to the image pixel. Though very simple and easy to implement, this technique increases rendering times dramatically as more primary rays are traced. Even worse, in some cases, more primary rays are traced for nothing, which can be the case in instances where regions of the image vary in color only slightly across pixels. Images in which the color varies greatly from one pixel to the next will still present aliasing artifacts. In this case, more primary rays would have been needed. This super-sampling technique is illustrated in Figure 3.20, where each pixel is divided into a regular grid of nine sub-pixels.

- Use adaptive antialiasing: An even better approach consists of concentrating the efforts where needed in the image. In this case, you trace more primary rays in regions of the image where there are abrupt changes of colors, and you trace a minimum amount of primary rays in regions where the color doesn't change much. This technique is called adaptive antialiasing or adaptive super-sampling. You use a threshold to determine whether or not each pixel needs to be split into sub-pixels. This is shown in Figure 3.21, where a pixel is divided into only six sub-pixels instead of nine.
- Use stochastic antialiasing: The idea here, as you will see in section 3.2, which is dedicated to Monte Carlo techniques, is to use stochastic sampling in place of regular sampling. In the case of antialiasing techniques, it simply means tracing primary rays in a pseudo-random distribution in place of the regular distributions I described earlier. This technique (also referred to as *jittering*) can be combined with the two techniques described above. It reduces the stairstep artifact by introducing noise. Figure 3.22 shows an example of jittering primary rays coupled with an adaptive super-sampling scheme.

- Sharp shadows: The basic raytracing algorithm produces sharp shadows. This is due to the fact that you use point light sources and that only one shadow ray (or light ray) is traced per intersection. One can use different techniques to produce smooth, and thus more realistic, shadows.
  - Area lights: In this case, a grid of point lights is used to simulate an area light source. At each point of intersection between a ray and an object, you simply trace as many light rays as there are point lights in the grid of a given area light in the scene. Although simple, this technique presents two major drawbacks: Rendering times increase because more rays are traced, and aliasing artifacts appear on the shadows generated. To get around these aliasing artifacts, you can simply increase the size of the grid used to simulate an area light; this reduces the artifacts but greatly increases rendering times. You can also use a stochastic distribution of point lights in the grid instead of a regular distribution. This reduces the aliasing artifacts but introduces noise, as with any stochastic distribution. Figure 3.23 shows how light (or shadow!) rays are traced in the case of a regular grid area light.
  - Monte Carlo raytracing for extended light source: The idea here is to use statistical supersampling to produce more realistic shadows without introducing too many aliasing artifacts. The light source is modeled as a sphere, not a point. Shadow rays are traced stochastically not in the direction of a point, but in the direction of a point on the sphere representing the light source. The colors of these shadow rays are then averaged to give the final shadow color at the given point of intersection. I present Monte Carlo raytracing (or

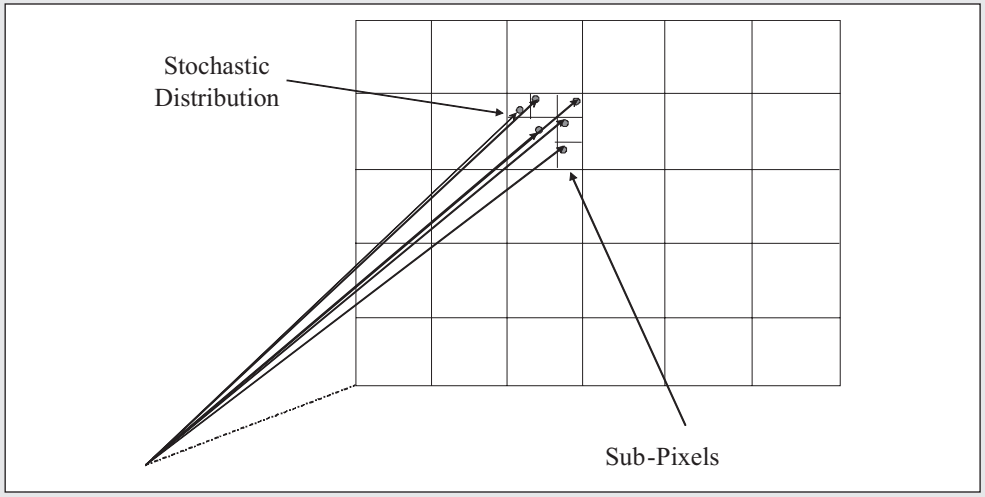


Figure 3.22. Antialiasing and jittering

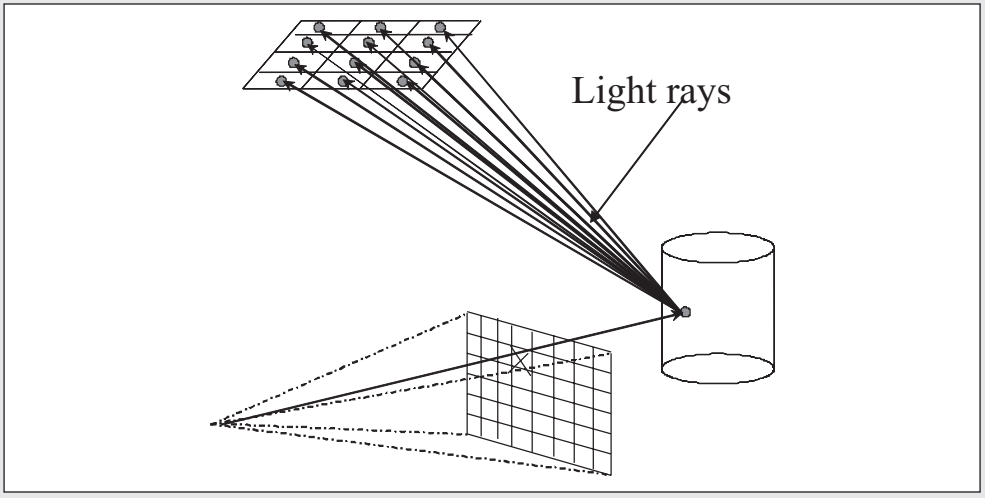


Figure 3.23. Area light and smooth shadows

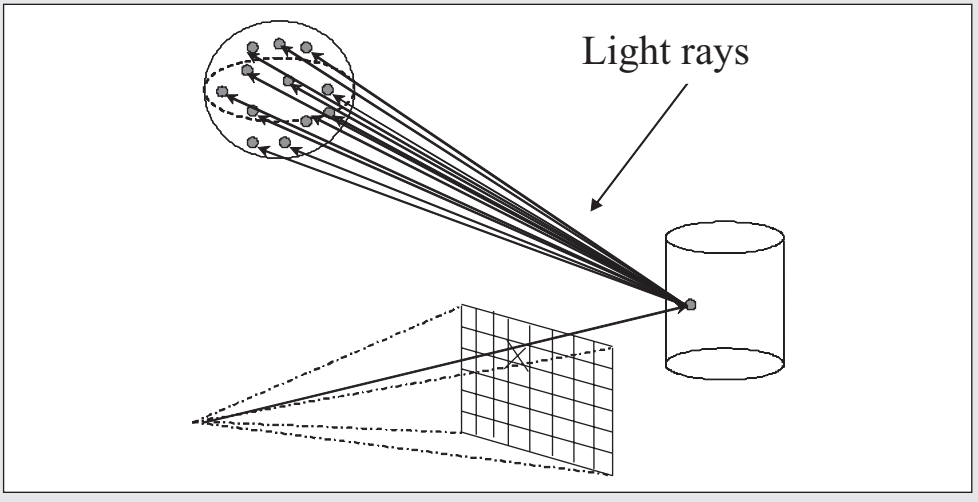


Figure 3.24. Spherical light and smooth shadows

stochastic raytracing) in more detail in section 3.2. Figure 3.24 gives an example of a spherical light.

- Global lighting and shading: You can also use Monte Carlo raytracing to determine in a very effective way the global lighting contribution in the scene at a given point of intersection between a ray and an object. The different techniques that have been developed to achieve this are presented in more detail in section 3.2. Another technique that can be combined with raytracing to take into account global illumination is radiosity. Radiosity is an algorithm that uses meshes to calculate the interdiffuse reflection of light in a scene. The algorithm is iterative and based on thermodynamics theory (where energy transfers are studied). Hybrid algorithms combining radiosity and raytracing have been developed to do this. More information can be found in [4], [5], [6], and [7].

3.1.9 The Algorithm

The following pseudocode summarizes the raytracing algorithm. It includes the main characteristics of a raytracer and can serve as the basis for an implementation.

```
Function Raytrace(Scene World)
{
   for (each pixel of the image)
   {
      Calculate the ray corresponding to the pixel
           (projection);
      Pixel color = trace(ray,0);
   }
}

color trace(Ray myRay, integer recurs_level)
{
   if (myRay intersects an object Obj)
   {
      Calculate normal N at point of intersection Pi;
      Calculate surface color SC at point of intersection
           Pi;
      Final_Color = Shade(Obj, myRay, N, SC,Pi,
           recurs_level);
   }
   else
   {
      Calculate background color BkgC;
      Final_Color = BkgC;
   }

   return Final_Color;
}
```

```
color Shade(Object obj, Ray myRay, Normal N, Color SC,
     Point Pi, integer recurslev)
{
   increment recurslev;
   if (recurslev > MAX_RECURSION_LEVEL) return 0;

   for (each light source)
   {
      calculate light_ray (Pi towards light source);
      if (light_ray doesn't intersect an object)
      {
         add light contribution to color based on
         angle between light_ray and myRay;
      }
   }

   calculate reflec_ray;
   reflect_color = trace(reflec_ray, recurslev++);
   calculate refrac_ray;
   refract_color = trace(refrac_ray, recurslev++);

   return average(color, reflect_color, refract_color);
}
```

3.2 Extending the Raytracing Algorithm

In this section I present a set of algorithms based on stochastic sampling of rays, which extend the basic raytracing algorithm you saw earlier. All these algorithms are commonly referred to as stochastic raytracing or Monte Carlo raytracing. The end goal of Monte Carlo raytracing is to achieve even greater realism in the rendering. At the end of this section, I also briefly present the photon mapping technique that extends raytracing but doesn't solely use Monte Carlo raytracing.

3.2.1 Stochastic Sampling

As you saw earlier, the raytracing algorithm traces a limited number of rays in the virtual world. Although the total number of rays traced can be huge even for a simple image, this is insignificant compared to what happens in the real world. In scientific terms, what you have is a sampling process.

Sampling can be achieved in many different ways. One common method is uniform sampling. Take a simple example: a mathematical function that you would like to plot over a given interval. With uniform sampling, you simply divide the interval into small regular intervals (i.e., intervals that have equal width), calculate the value of the function in the middle of the interval, and finally plot the points calculated. Figure 3.25 illustrates regular sampling of a mathematical function.

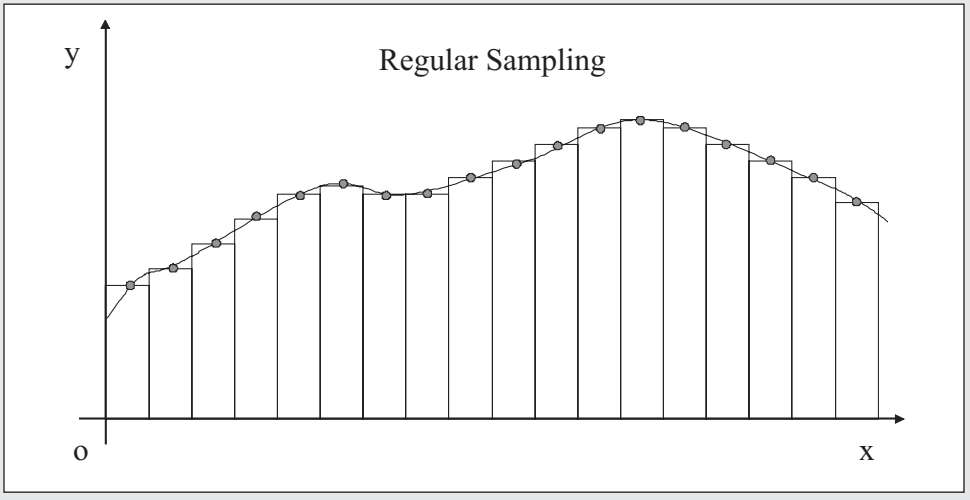


Figure 3.25. Regular sampling

The problem with uniform sampling is that it can lead to wrong results. If you don't have enough intervals, you might get an incorrect representation of the function. For periodic functions (e.g., sine waves), this leads to a very incorrect representation of the function sampled, as shown in Figure 3.26.

If we go back to the sampling processes you find in computer graphics in general, and in raytracing in particular, this translates into the aliasing artifacts I presented earlier: jaggy edges in the image rendered, stairstep effects, jaggy highlights, jaggy shadows (e.g., with uniform area lights), etc. Stochastic sampling cannot remove all of these artifacts but does attempt to reduce them in an effective way.

If you take the example of the mathematical function that you would like to sample over a given interval, with stochastic sampling the idea is to take intervals of pseudorandom width (i.e., intervals having varying widths) instead of regular intervals. The process to plot the function or to calculate its integral will remain the same; however, the aliasing artifacts will be reduced. Figure 3.27 shows the same function as before,

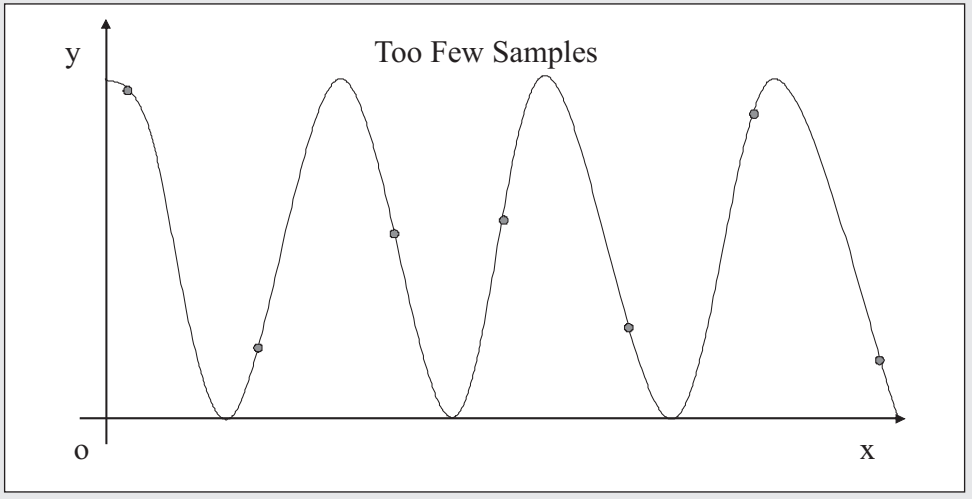


Figure 3.26. Sampling problems with too few samples

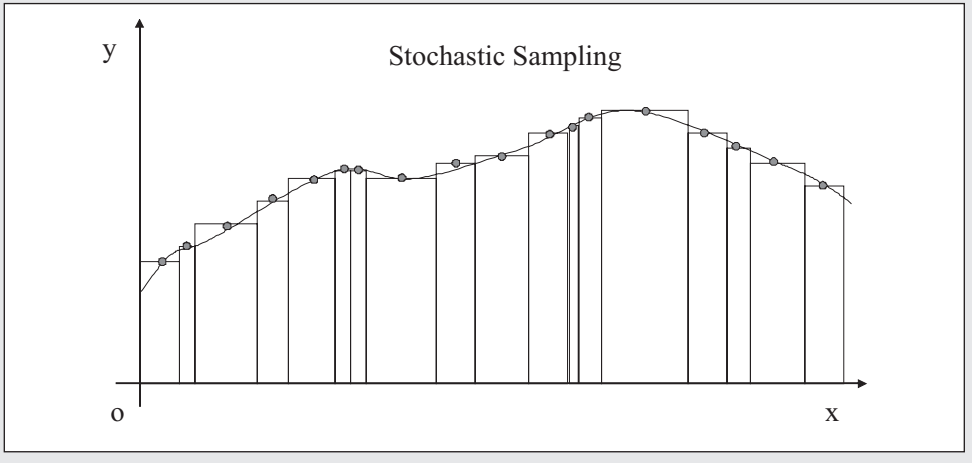


Figure 3.27. Stochastic sampling

but with stochastic sampling.

Now, when you apply stochastic sampling to raytracing, the result is that you reduce aliasing by introducing noise. Your eyes are less prone to notice noise than aliasing artifacts. Stochastic sampling also allows extensions to the simple raytracing algorithm you saw earlier.

- You can now render smooth shadows that are more realistic — *stochastic sampling of shadow rays*.
- You can render reflections that are no longer sharp but fuzzy, and thus more realistic — *stochastic sampling of reflected rays*.
- You can render refractions that are no longer sharp but fuzzy, and thus more realistic — *stochastic sampling of refracted rays*.
- You can take into account the depth of field effect that you see with real photographs: objects that are not in focus appear blurry — *stochastic sampling of primary rays*.
- You can take into account the motion blur effect that you see with real photographs and in movies: temporal aliasing makes objects in movement appear blurry — *stochastic sampling in time, not in space, of primary rays*.

Figure 3.28 shows a scene using stochastic raytracing to render fuzzy reflections of a sphere.

Figure 3.29 shows the same scene using stochastic raytracing to render smooth shadows.

Figure 3.30 shows a scene using stochastic raytracing to produce a depth of field effect.

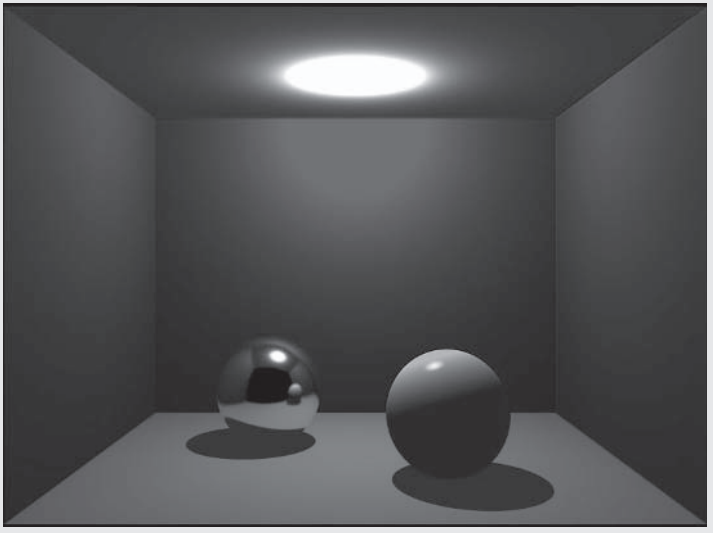


Figure 3.28. Stochastic raytracing and fuzzy reflections

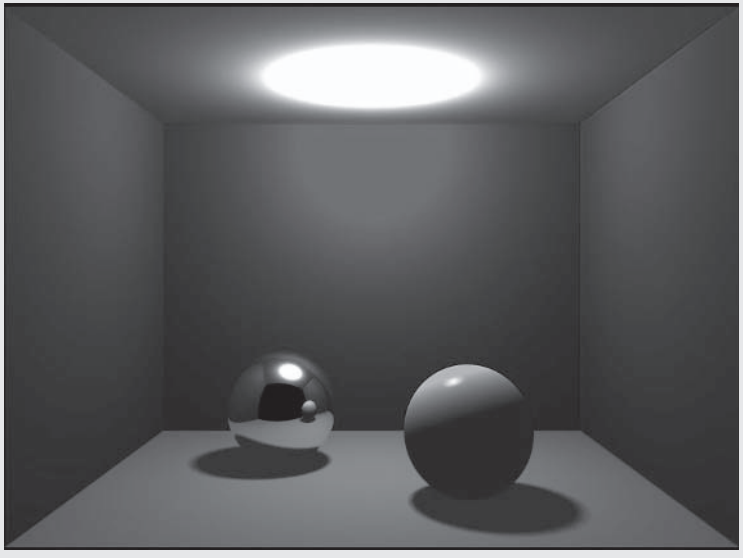


Figure 3.29. Stochastic raytracing and smooth shadows

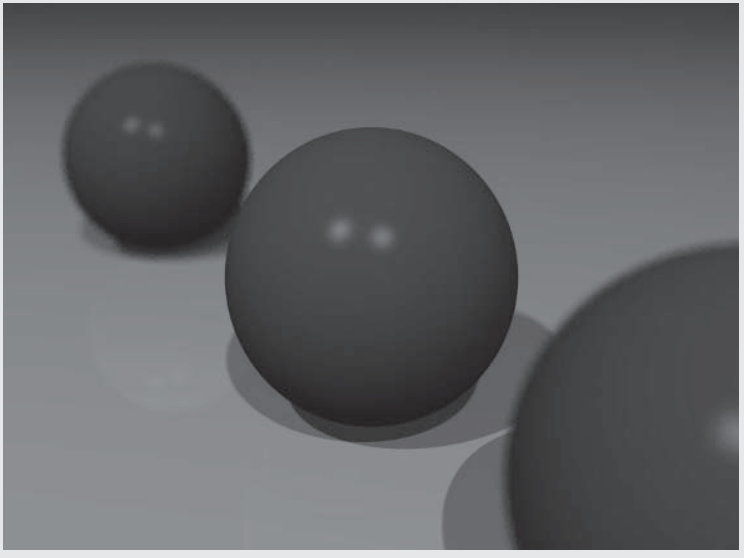


Figure 3.30. Stochastic raytracing and depth of field

3.2.2 Path Tracing and Related Techniques

In this section, I present a few techniques using Monte Carlo sampling to add a major feature to raytracing: global illumination. This brief introduction gives you pointers to further extend the raytracing algorithm you saw earlier.

Path tracing, and its related algorithms, extends the raytracing algorithm to take into account global lighting in a virtual scene. As you saw earlier, raytracing only traces four types of rays (primary, shadow, reflected, and refracted), none of which are used to calculate the light received from the rest of the scene at the point of intersection on the surface. Path tracing and the different algorithms that extend it do exactly this — tracing rays out into the world using stochastic sampling to compute the global illumination of the scene.

In the path tracing algorithm, indirect lighting is calculated by tracing rays in all possible paths surrounding the point of intersection on the surface. To avoid infinite rendering times, all of these possible light paths are sampled using stochastic sampling. The distribution of the rays traced in this way is usually a hemisphere (half a sphere) whose center is the point of intersection, and aligned with the surface normal at that point. As you already saw, if you don't trace enough rays and you use stochastic sampling, you won't get aliasing artifacts but rather noise. The major drawbacks of path tracing are:

- Long(er) rendering times: Since you want to stochastically sample all possible paths for each intersection, this leads to tracing many more rays, testing even more for ray/object intersections, etc. This leads to rendering times that are much longer than with raytracing.
- Noise: When not enough rays are traced to compute global illumination, the final image appears noisy. The only solution in this case is to trace more rays.

On the other hand, path tracing presents several advantages, the main ones being simplicity and the level of realism achieved in the rendering. Path tracing is also very efficient in the case of outdoor scenes where skylight is used. In this particular case, the final image presents very little noise because the light slowly varies in all directions. In other words, since the function you are trying to sample varies very slowly, taking a small number of samples still gives a very good approximation of that function. Path tracing is discussed in more detail in [8].

Figure 3.31. Raytraced Cornell room

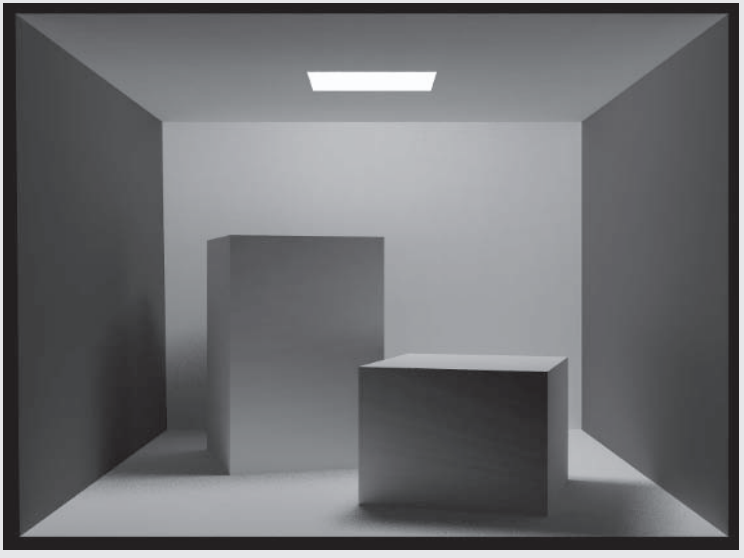


Figure 3.32. Path-traced Cornell room

Figures 3.31 and 3.32 present the same scene (the famous Cornell room) rendered with raytracing (Figure 3.31) and with path tracing (Figure 3.32). There is a clear difference in terms of realism. However, as stated above, this comes at a cost: The raytraced image took a few seconds to render, whereas the path-traced image took three hours to render, using exactly the same settings, i.e., same hardware, same program (with path tracing turned on and off), same geometry, same antialiasing settings. In the path-traced image, you can also notice the noise I just talked about (especially on the ceiling that appears "patchy" in places, with color bleeding), although 500 samples were used to calculate the global illumination contribution at each point. These images were rendered using a modified version of the Povray raytracer (see [9] for more details).

Figure 3.33 shows the basic concepts of path tracing.

Bidirectional path tracing extends path tracing, thus further extending raytracing, by tracing light paths as is done with path tracing, but also tracing light paths from the light sources of the scene. The end goal is simply to try to reduce the number of samples compared to path tracing. Bidirectional path tracing still presents the same problems as

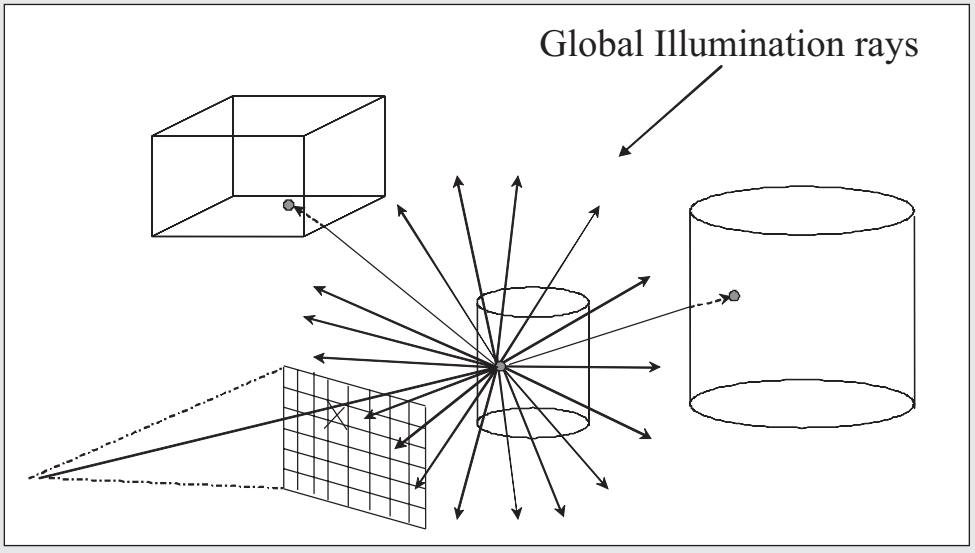


Figure 3.33. Path tracing

with path tracing — long rendering times and noise in the final image rendered. However, fewer samples are used to compute global lighting. Bidirectional path tracing is discussed in more detail in [8].

Finally, Metropolis Light Transport is another technique that can be used to compute the global illumination solution of a scene. It is different in the way it samples the light paths I talked about earlier, as the idea is to adapt the number of samples where they are needed in the scene. More details about Metropolis Light Transport can be found in [8]. Beware though; it is not for the faint-of-heart when it comes to mathematics!

3.2.3 Photon Mapping

To conclude this section, I present a technique that is not purely based on stochastic sampling: photon mapping, a two-pass rendering algorithm that extends the raytracing algorithm. Its main characteristics are to take into account global lighting and interdiffuse reflections and produce caustics effects, among other things. Figure 3.34 shows the use of photon mapping to render caustics (image rendered with the Virtualight renderer; see [10] for more details).

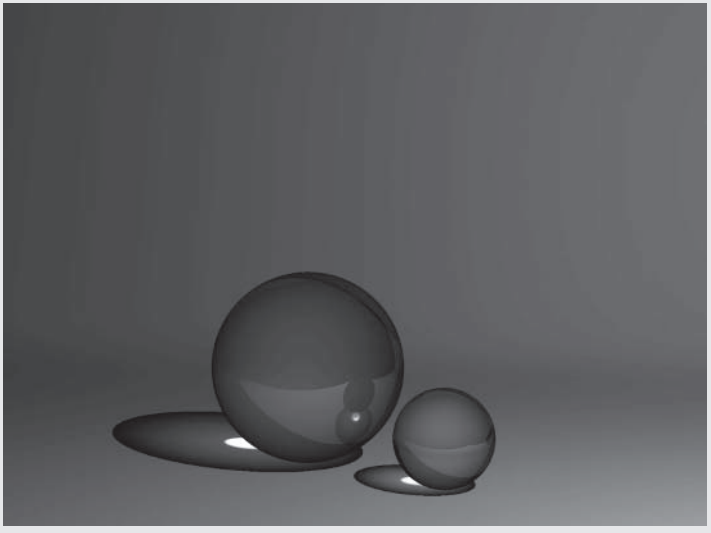


Figure 3.34. Caustics rendered with photon mapping

Photon mapping works in two passes: photon tracing, or building the photon maps, and illumination gathering, or rendering using the photon maps.

During the first pass, rays of light, or photons, are traced from the different light sources of the scene out into the virtual world. The difference from raytracing is that the photons carry a fraction of the energy emitted by the light source from which they originate, hence the use of the term "photon" in place of the term "ray." The photons are traced in the outer world using the light source geometry and properties. Any type of light can be used: point lights, area lights, spotlights, etc. Even complex lights can be used, as long as their properties are known (direction, energy radiated, etc.).

When a photon is traced in the virtual world, it can be reflected, transmitted, or absorbed, depending on the surface properties it hits. At a point of intersection on a surface, you need to decide what to do with a given photon. In order to efficiently trace photons, you can use a new probabilistic sampling method called Russian roulette. With this Monte Carlo sampling technique, the goal is to keep the useful photons and get rid of the useless ones. The Russian roulette technique is described in more detail in [8]. Just keep in mind that Russian roulette allows you to keep the important photons and get rid of those for which tracing would have been a waste of time!

When a photon hits a diffuse surface, you store the incoming energy (the incoming *flux*) on the surface at the point of intersection. For all diffuse surfaces hit by the photon, you store that information until the photon is absorbed. To efficiently store this incoming illumination, you use a map. This map simply gives information on the incoming energy at a

given point in space. There is no need to store the geometry of the scene. A photon map is usually implemented as a k-d tree data structure.

Once a given number of photons have been sent and the information has been gathered in the photon map, the rendering phase can start. The raytracing algorithm can be used to render the scene. At a point of intersection between a ray and an object, the information stored in the photon map is used to estimate the illumination (direct and indirect). The estimated illumination is based on the nearest photon found in the map.

As you can see, photon mapping is an efficient way of computing global illumination of a scene. It can be easily coupled with a raytracing algorithm to achieve a high level of realism. Compared to radiosity, where the virtual scene needs to be meshed, photon mapping can handle a scene with complex geometry as easily as as simple scenes.

3.3 Real-time Raytracing

This book is about advanced lighting and shading techniques for real-time rendering, yet you have seen that raytracing is very computer intensive and doesn't fill the requirements of a real-time rendering algorithm. Nonetheless, the term "real-time raytracing" has been around for some years now. Real-time raytracing's primary focus is not to render high-quality pictures, but to render pictures as fast as possible. Many factors can help achieve this, including:

- Moore's law: As you saw earlier, advances in hardware makes it possible to render complex scenes in a few seconds. This depends of course on the resolution and the complexity of the scene. Images that used to take hours to render now take only a few minutes, if not seconds.
- Massive parallelism: As stated before, a raytracer is massively parallel in essence. Clusters of machines can render several frames per second. Coupled with advances in hardware, this tends to reduce rendering times even more.
- Trade-offs: The idea is to limit some features of the raytracer so as to speed up rendering times: level of recursion, number of primary rays, etc.

Even when considering these factors, you can use additional techniques to reduce real-time raytracing computation times even further. Simple techniques include: simplified geometry, no antialiasing techniques, simple light sources, simple shadows, limited reflection and refraction, etc.

One common technique implemented in real-time raytracing programs consists of optimizing how primary rays are traced. For instance, visibility in the virtual scene can be determined by using a Z-buffer algorithm; only secondary rays have to be traced. This technique considerably reduces the number of calculations needed to render a frame. Modern 3D APIs implementing the Z-buffer algorithm in hardware, such as Direct3D and OpenGL, can be used to determine the

visibility in the scene.

A similar technique can be used for shadow rays. During a preprocessing step, the Z-buffer algorithm is used from all light sources in the scene to construct shadow buffers, thus reducing the number of shadow rays to trace during the rendering phase. This technique is applicable in scenes with static geometry (otherwise, the shadow buffers change when the geometry or lights move). Here again, one can use 3D APIs and 3D graphics cards to construct shadow buffers.

Another optimization technique, called *undersampling* or *subsampling*, can be used to reduce the number of rays traced. In this case, only a limited number of primary rays are traced through the pixels of the image plane. When a ray intersects an object, the primary rays sent through the neighboring pixels are traced to refine the visibility of the scene and check if other objects are intersected. This technique reduces rendering times but has a flaw: Since not all primary rays are traced, objects in the scene can be missed (if the width of a given object as seen from the observer is smaller than the distance between two adjacent rays).

Other real-time raytracing optimization techniques focus on efficient algorithm and data structures to avoid ray/object intersections as much as possible: voxels (volume elements; the equivalent of pixels, but in 3D), k-d trees, nested grids, uniform grids, 3DDDA, etc.

As you can see, real-time raytracing is a vast subject. It implies many optimizations and trade-offs to enable scenes to be rendered within milliseconds. I only presented a few examples of the most common techniques. Interested readers will find pointers to advanced techniques and related papers in the References and Other Resources sections of this chapter. Whatever the recent advances in this domain, real-time raytracing hasn't yet been implemented in games as the primary rendering algorithm. Nonetheless, this could happen sooner than you may think!

3.4 Raytracing Concepts for Other Techniques

Raytracing as described in this chapter is a method for producing a final rendered image of a virtual scene, but raytracing concepts can be used to support other techniques. For instance, later chapters cover techniques such as spherical harmonics, which use raytracing concepts when computing data that will be used in real-time rendering. Also, raytracing can be used to produce detailed cube maps used for environmental mapping. Remember that the concepts described in this chapter can be used for a wide range of techniques.

Conclusion

This chapter was only an introduction to this fascinating and complex rendering algorithm. You saw that, even though simple and elegant, raytracing in its primary form is slow and limited — it takes a very long time to produce jaggy images that don't even include important subtle effects such as interdiffuse reflections of light. Hopefully, researchers and scientists around the world are extending the idea and finding solutions for all its problems and limitations. All of the presented techniques are usually referred to as Monte Carlo raytracing or stochastic raytracing. Using these techniques it is possible to produce images that are indiscernible from real photographs, at the cost of long rendering times. Finally, with the recent advances in hardware, it is now possible to imagine using raytracing in the context of real-time rendering, at the cost of some trade-offs in rendering quality.

In the following chapters of this book, I discuss shading techniques that are applicable to raytracing. However, all the examples and implementation I present focus on real-time rendering and gaming.

References

[1] Ebert, D., F. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing and Modeling: A Procedural Approach*, AP Professional, 2000.

[2] Glassner, Andrew S., *An Introduction to Raytracing*, A.K. Peters, 1989.

[3] Many contributors, *Graphics Gems*, Vol. I to V, Academic Press, Harcourt Brace Jovanovich, and AP Professional, 1990-1995.

[4] Cohen, M. and J. Wallace, *Radiosity and Realistic Image Synthesis*, Academic Press Professional, 1993.

[5] Sillion, F. and C. Puech, *Radiosity and Global Illumination*, Morgan Kaufmann Publishers, Inc., 1994.

[6] Larson, G. W. and R. Shakespeare, *Rendering with Radiance*, Morgan Kaufmann Publishers, Inc., 1998.

[7] Ashdown, I., *Radiosity: A Programmer's Perspective*, Wiley, 1994.

[8] Jensen, H. W., *Realistic Image Synthesis Using Photon Mapping*, A.K. Peters, 2001.

[9] Marty, S., "Stochastic Global Illumination Povray," http://rendering.e-viale.net/SMSGIPov/

[10] Marty S., "Virtualight Renderer," http://www.3dvirtualight.com

Other Resources

Watkins, D., S. Coy, and M. Finlay, *Photorealism and Raytracing in C*, M&T Books, 1992.

Site dedicated to real-time rendering in general: http://www.realtimerendering.com

OpenRT — an open library for (real-time) raytracing: http://www.openrt.de

Real-time raytracing-related publications: http://graphics.cs.uni-sb.de/Publications/index.html

A commercial real-time raytracing hardware solution: http://www.realtime-raytracing.com

OpenRT-related publications: http://www.openrt.de/Publications/index.html

Chapter 4

Objects and Materials

Introduction

Beyond this chapter, most of this book concentrates on complex mathematical equations and computer graphics techniques that are useful when trying to reconstruct real-world phenomena in a virtual scene. Readers and writers can both be guilty of becoming immersed in the math without taking the time to point out the fact that we are literally surrounded by perfectly realistic implementations of these effects. Before diving into the math, I want to spend some time in the real world.

In this chapter, I explain lighting phenomena purely in terms of the way you see them, without much in the way of geometric or mathematic explanation. This is just to increase your awareness. In the next chapter, I'll offer deeper explanations with plenty of equations.

As a game developer, you might be interested in reproducing environments that most people have never seen. If you're lucky, you don't live in an enemy camp, a mysterious forest, or an alien cocoon, but chances are you might have access to a tent, a tree, or slime of one sort or another. You can begin to develop an intuition for how different materials should look by observing those materials in your own environment. As you develop complex materials and scenes, it will help to have a mental or physical model to use as a starting point. Character animators frequently have mirrors next to their workstations so they can see how their own mouths or limbs move. Interior designers develop drawings and a mental model of what they think a space should contain and then use that mental model to decide whether or not a new

item fits into the space. Likewise, you can improve the look and feel of your own scenes by making them match objects that are familiar to you and the people that play your games. The following is just a short list of some things you might see:

- Plastics
- Wood
- Leaves and vegetation
- Metals
- Concrete and stone
- Skin
- Hair and fur
- Air and the atmosphere
- Transparent materials
- Paint
- New and worn materials

4.1 Plastics

Many of the objects seen in 3D scenes have an appearance that looks very much like plastic because some of the most widely used illumination models produce that effect. Granted, there is a lot of plastic in the world, but there is also a lot of wood, metal, and organic material.

Plastic objects come in many colors, shapes, and finishes, but the material itself is fairly consistent in terms of composition. Most plastics are made up of a white or transparent substrate that has been infused with dye particles, as shown in Figure 4.1. If a product is available in more than one color of plastic, chances are very high that the basic plastic material is the same color throughout and it's the difference in the dye particles that creates the difference in colors.

When light strikes a plastic object, some of the light reflects off of the outermost surface. At a particular angle, the viewer will see this light, and that part of the object will appear white (the color of the plastic substrate). Some of the light will penetrate the object and reflect off of the dye particles. These particles will absorb some of the wavelengths of

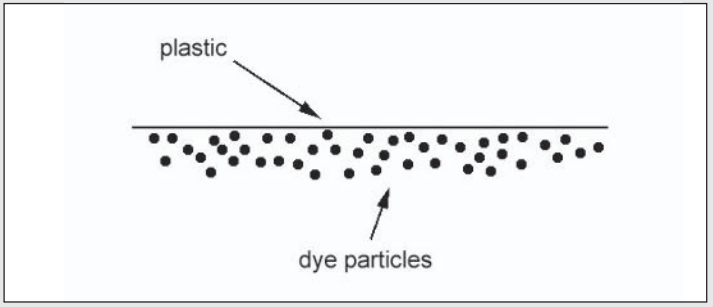


Figure 4.1. Plastic with dye particles

the light and not others. As shown in Figure 4.2, at some angles, you're seeing the plastic substrate, and at other angles, you're seeing the dye particles.

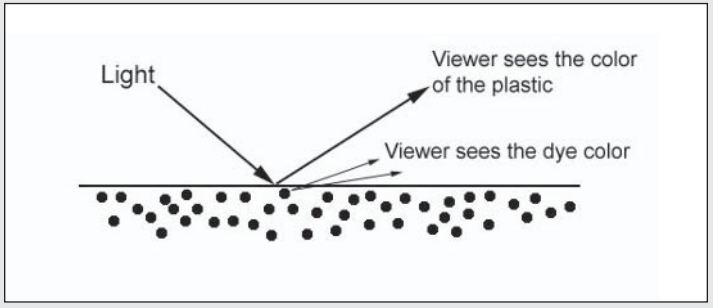


Figure 4.2. Smooth plastic seen from different angles

Figure 4.3. Matte plastic with subtle highlights

This effect is especially pronounced in very smooth objects because the angle of the light relative to the surface is fairly consistent over large portions of the surface.

Plastic can also have a matte finish. In this case, the surface is rougher and light penetrates and reflects much less evenly over the surface. In this case, you still get lighter highlights, but they are less even and less pronounced. There are two different reasons for this. First, the rough surface causes uneven reflection/penetration patterns. Secondly, the roughness of the surface causes some of the bumps to cast very small shadows on other parts of the surface. In the end, you get the subtle highlights shown in Figure 4.3. Notice that part of the spacebar has been worn smooth over time.

Many basic lighting tutorials tell the reader to set the specular color of a material to white. This is a decent rule of thumb for many plastics, but it doesn't necessarily apply to other materials. In plastic, the white color comes from the substrate, but what about materials such as wood that don't have a substrate?

4.2 Wood

Wood is a very common material in both real and virtual environments. Unfortunately, it's also very difficult to render in a completely realistic way because the patterns and surface properties can be very complex. To compound the problem, wood appears in many different forms. You can talk about trees in the forest, varnished furniture in a hotel lobby, or the raw lumber used to build the enemy camp. Each form has a different look defined by its physical attributes.

4.2.1 Trees

One of the strongest visual cues that separates trees from metal poles is the existence of bark. In nature, the rough pattern of the bark is caused by the growth of the tree. Most of the visual detail is made up of the areas of light and dark in the peaks and troughs of the bark as shown in Figure 4.4.

A very simple view of Figure 4.4 could be that there are light and dark areas of the bark. This is not the case. The color of the bark doesn't really change all that much over the surface of the tree. Instead, nearly all of that detail comes from the geometry of the bark. The areas of light and dark are caused by differences in illumination, not by changes in the color of the bark itself. If you take a photograph of bark and then try to use it as a texture, you will encode the lighting conditions in the texture, which will decrease realism in a scene with different lighting.

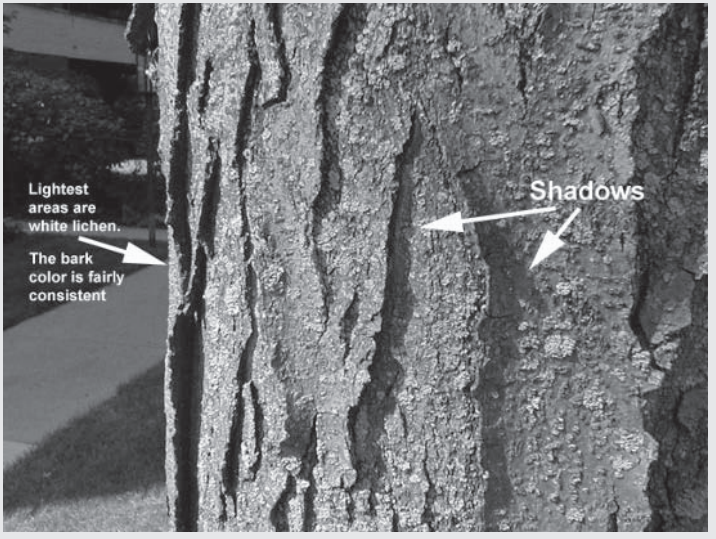


Figure 4.4. Tree bark

This is not to say that all bark is uniform in color. The paper-like bark of a birch tree has areas of different color and roughness. In addition to color changes and surface roughness, the papery areas tend to be a bit shinier than the other areas. In a realistic scene, this type of bark would be fairly difficult to reproduce accurately. In general, you want to include all of these factors, but you don't want your trees to look like shiny plastic props.

4.2.2 Lumber

I'm defining lumber as any sort of bare wood, from arrow shafts to construction lumber. The most notable feature of lumber is that it is cut from the tree and shows the grain of the wood. Figure 4.5 shows several examples.

Figure 4.5. Several examples of lumber

Grain patterns can be viewed as changes in color, but they often contain other features. Parts of the grain might have more bumps and ridges. Other parts might be shinier or duller than the rest of the wood (perhaps because of changes in concentrations of sap or oils), although most raw wood has a matte finish. Furthermore, pieces of lumber exhibit some amount of *anisotropy*, meaning that the direction of the grain affects how light reflects from different angles.

4.2.3 Finished Wood

Items such as wood furniture usually have some sort of finish like a stain or a varnish. In this case, the wood might have all of the color features of lumber, but also have a higher amount of specular reflectivity. From some viewing angles, more light is seen reflected from the varnish than from the underlying wood, creating more of a shiny highlight as shown in Figure 4.6. Furthermore, different shades of varnish might contribute to different shades of specular color.

The amount of shininess helps to communicate age and quality. With varied levels of shininess, the same grain texture could be used to communicate either highly polished new furniture or rustic antiques. However, you should make sure that the overall texture matches the amount of shininess. Some scenes have wood elements where the texture is of battered wood, but the lighting equations make it shiny. This creates two conflicting impressions: In terms of color and texture, the wood looks very old, but in terms of shininess it looks like freshly lacquered furniture. Usually, this doesn't look realistic.

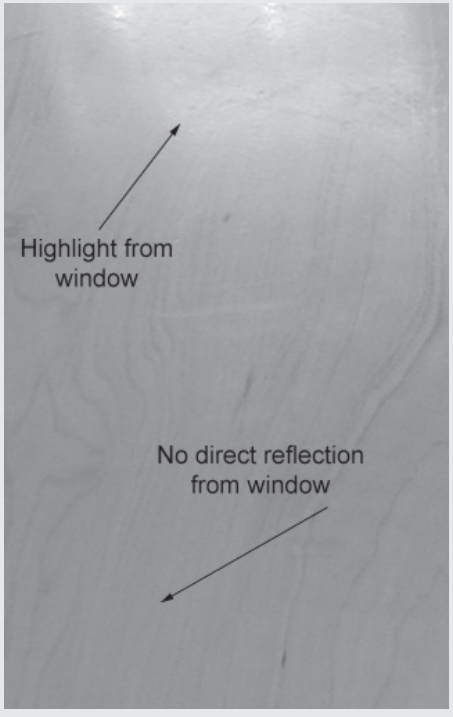


Figure 4.6. Varnished wood with highlights

4.3 Leaves and Vegetation

In addition to trees and wood, many indoor and outdoor scenes include leaves, grass, and other forms of vegetation. Right now, most plants in real-time scenes are rendered with green leaf textures, but things aren't that simple. Like tree bark, much of the detail on leaves comes from the way that light interacts with features of the surfaces. There are raised veins and ridges. Many leaves have a shiny side and a more matte side. Other leaves have small hairs or spines that interact with light in interesting ways. As with bark, this detail can't be accurately captured in the colors of a single texture. The leaves in Figure 4.7 are mostly homogeneous in color. The detail that you're seeing is caused by illumination of the surface details.

Furthermore, most leaves are translucent, meaning that they allow a small amount of light to pass through them. This causes a couple of effects. First, a leaf can become brighter when a light appears behind it. For example, a dark green leaf can appear more emerald green when backlit. Also, the light can reveal some of the inner structure of the leaf. In this case, you are seeing aspects of the leaf that have nothing to do with the outer surfaces. This effect is not easy to capture in a grayscale photograph, but it's easy to see in most houseplants. Shine a flashlight behind various leaves and you will see different details depending on the structure and thickness of the leaf.

Finally, many leafy environments are dense and block much of the light traveling through them. One common example is a grassy lawn. The color of the blades of grass is fairly consistent over the length of the blade, but it appears as if the grass gets darker closer to the ground. This is because the blades collectively block the light and less gets to the lower extremities of the blades. The same is true for light beneath a dense jungle canopy. In addition to blocking light, translucent leaves will also give the light more of a

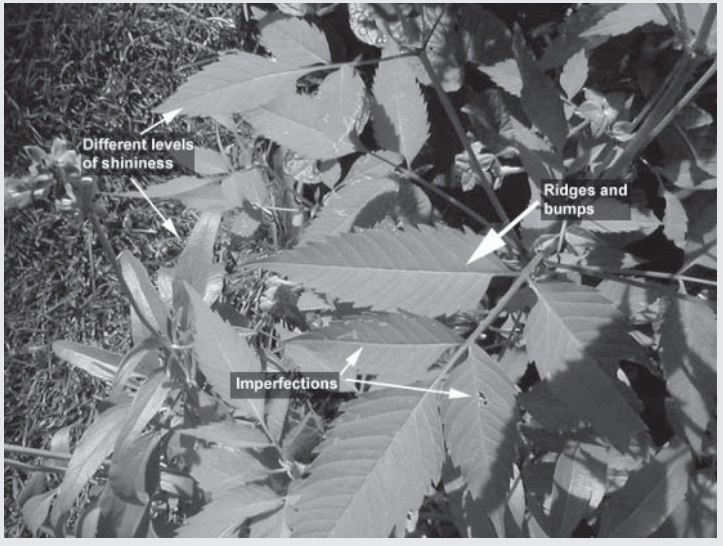


Figure 4.7 Surface details on leaves

greenish color. This effect is important to remember when your commando team is moving through a dense rainforest. Generally, the leaves above might be very green, but things are darker at the ground level. If you look up, overhead leaves might appear much brighter even though you are looking at the undersides.

There are actually many more features of plants, many of which are very expensive to render in real time. This is just a taste of what you should be looking for. If your rescue team is going to be spending a lot of time sniping from tall grass, spend a little bit of time in some tall grass and see what things look like.

4.4 Metals

Bare metal might be one of those materials that appear more often in games than they do in real life. Very rarely will you find yourself running down some titanium tube trying to get out of an alien base, yet this seems to happen all the time in games.

Metal is similar to plastic in that it can have strong specular highlights. Unlike plastic, these highlights tend to take on the color of the metal rather than the color of the light. Yellow plastic tends to have white highlights. Yellow gold has yellow highlights. Part of the reason for this is that metals react to electromagnetic waves differently from nonconductive materials. As a result, they tend to behave differently when illuminated. Also, they obviously have no plastic substrate, unless they have been painted or coated with some clear material.

Also, different metals have subtle differences in the way that they reflect light. These differences are caused by the fact that different metals have different molecular structures and therefore reflect light differently on a microscopic level. These structures can cause the metal surface to reflect different colors differently. On top of that, different objects have different finishes, which also reflect differently. Therefore, copper might look different from bronze, but burnished bronze will be different from shiny bronze. Figure 4.8 shows two examples of steel with different finishes. One is very smooth, while the other has a coarse, brushed look. Both are essentially the same material with different surface properties.

Figure 4.8. Two samples of steel

4.5 Concrete and Stone

We probably see more mineral-based materials in our day-to-day lives than we see bare metal. On a short walk down any city street, you might see hundreds of different types of concrete, brick, and stone. Each of these materials is very different, but there are some higher-level features that are consistent across all of them.

4.5.1 Concrete

If you take a close look at a sidewalk, you can see that, like bark, much of the visual detail comes from the roughness and the mix of materials that make up the surface. Many of the light and dark areas of the sidewalk are caused by cracks or small shadows that one part of the surface casts on another. As the position of the sun changes, the small shadows will also change in very subtle ways. In short, the color of some forms of concrete is much more uniform than it might appear, but shading introduces more visual detail than you might expect.

On the other hand, concrete is a composite material made up of a binding material and many little stones. Therefore, you can have flecks of color, the density and mixture of which is dependent on the mix of the concrete. This is shown in a close-up in Figure 4.9.

Similarly, you might have small areas where the color is more or less the same, but the shininess changes. In some cities, you will see sidewalks that appear to sparkle in the right light. This is caused by a high concentration of reflective material in the mix.

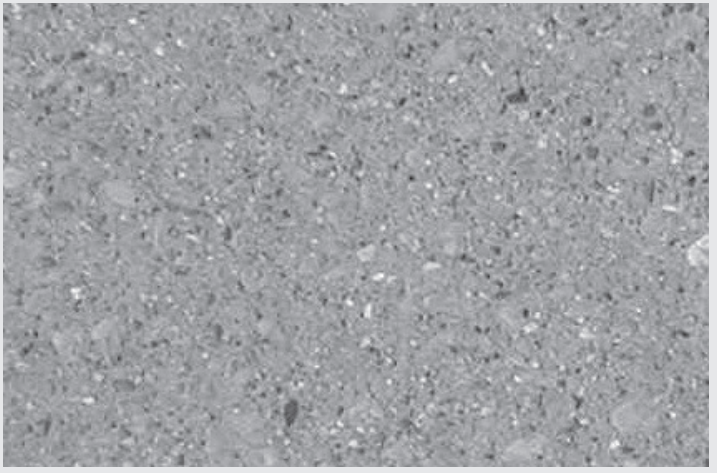


Figure 4.9. Close-up of a sidewalk

Although it is not quite the same as concrete, asphalt has similar properties. A brand new racetrack will appear mostly black because the viewer sees the layer of tar on the surface. As that layer is worn away, the track appears gray because the viewer is seeing the stones that comprise the asphalt. Tire tracks will not only darken the track, but the rubber residue will also reflect light differently.

4.5.2 Brick

Brick is another surface in which most of its visual detail comes from pits and bumps on its surface. On a typical red brick, you might see patches of different shades of red, but the smaller texture and color features are due to the surface detail. Also, the brick and the mortar might have very different properties above and beyond color. Once you take a careful look at a brick wall, you'll see that the subtle effects would be difficult to reproduce with a simple texture. Even a bump map does not completely reproduce the smaller details.

Brick walls are very often used to demonstrate the power of bump mapping and other shader effects. In addition to demonstrating the basics of bump mapping, these applications demonstrate the fact that it is very difficult to use a photograph of a texture like brick while avoiding the lighting conditions in the photograph. For instance, a photograph taken at noon will be lit from above, meaning any undercut parts of the brick wall will be in shadow. If that photograph is used as a texture and the textured object is lit from the front, the photographed shadows will contribute to an incorrect final rendering.

4.5.3 Natural Stone

Natural stone has all of the aforementioned features and more. An accurate depiction of different kinds of stone can help to give game players a sense of where they really are. For instance, the loose sandstone found in the American West tends to be of a fairly uniform color with a very matte finish. The granite of the Alps can have a very sharp appearance. A volcanic region might have a mix of several types of rock.

Also, many natural rocks are heterogeneous. They contain veins of other materials with different properties. Most rocks do not have uniform shininess or texture. Like most natural objects, many stones and rocks have many variations in color and surface detail. Many games feature cliff faces with low geometric detail. This is understandable given polygon budgets, but real rocks have a huge amount of detail in terms of both color and geometry.

4.6 Skin

If there is one thing that humans are good at scrutinizing, it is other humans. This is why virtual actors are so difficult to make completely lifelike. It's very easy to spot a fake human even if we can't articulate exactly why we think it looks unnatural. This is an impediment in games because you really want players to feel as if they are interacting with other people. This is especially true for online games where they actually *are* interacting with other humans. The more natural you can make the character look, the better. A completely natural appearance requires more than lighting and materials. Proper animation is a large part of it, but that is beyond the scope of this book. Instead, I will concentrate on the appearance of skin.

Skin is mostly uniform in color, but many of the other surface details change. Some areas might be rougher than others. Some areas might be shinier. In addition to being visibly different, those features communicate something about the character to us. The face of a soldier might be pitted and scarred, while the face of the elf queen might be very smooth. A high degree of shininess on the top of a head might be used to accentuate baldness. A low degree of shininess might be used to portray grime.

Smaller details like pores might contribute to larger effects. For instance, as you make different facial expressions, the wrinkles and pores on your face stretch in different ways. These changes have subtle effects on the way your face reflects light from different directions.

Finally, since skin is translucent, the interactions between different translucent layers of skin produce subsurface light scattering effects. Light penetrates the surface of the skin, scatters, and then reflects back out. This contributes to a sense of the skin having depth. If you stand next to a mannequin with a similar skin tone, you'll find that your skin has a softer, deeper look. This is due to the fact that your skin allows light to penetrate while the plastic skin of the mannequin does not. If you do the same comparison at a wax museum, you will see less of a difference because the wax is also translucent (hence the choice of wax in the first place).

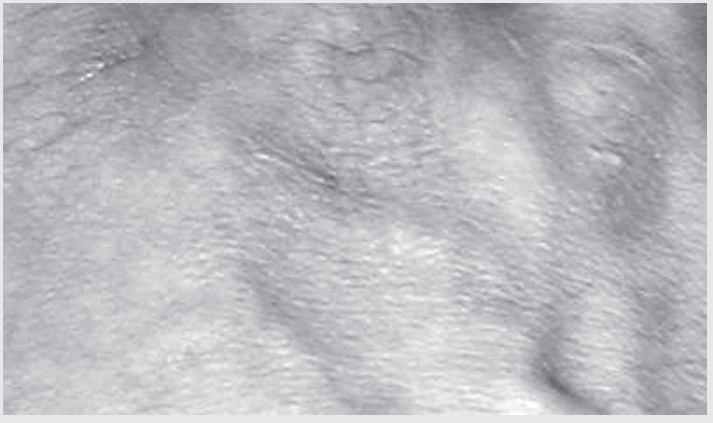


Figure 4.10. Close-up of a hand

The effects of translucency and scattering are visible from any angle, but they are most apparent when the light is behind the skin. This is especially true if the object is thin or the light is very bright.

Figure 4.10 shows a contrast-enhanced close-up of skin on the back of a hand. Notice how the knuckles at the top of the photo have more enhanced details. Also, the larger bumps at the bottom of the photo are veins that move as the hand flexes.

4.7 Hair and Fur

If you look at hairs individually, it is very difficult to come up with a set of properties that tells you anything about how to render them efficiently. When viewed as a mass of hairs, some patterns become clear. When many hairs are lying in the same direction, as shown in Figure 4.11, it can be viewed as essentially one surface with many small grooves. This causes a very characteristic sheen that is similar to a record or brushed metal.

At the other end of the spectrum is short fur that stands more or less straight up. This type of fur tends to be fairly uniform and thus easier to render. As hair or fur becomes less uniform, it becomes harder to generalize. If you look at your hair when you get up in the morning, you will see that there is not much order. At that point, the look of your hair is dependent on the spatial properties of your hair, the uniformity of the color, the amount of shadowing and reflection between hairs, and more.

Like skin, hair can be used to communicate something about a character. A person or a dog with clean, shiny hair is more appealing than one with dull, matted hair. If a bear attacks a character, you want to make sure the bear does not have the smooth, regular hair of a stuffed animal. Likewise, a rabid hellhound typically does not have a healthy, shiny coat.

Figure 4.11. Shiny hair

4.8 Air and the Atmosphere

Air quality and atmospheric effects tell you a lot about an environment and its scale. A clear day with high visibility feels very clean and natural. A hazy, smoggy day feels more urban and claustrophobic. Dense fog is disorienting, while mist on the ground seems mysterious and sinister. A yellow haze gives the impression of being distinctly unhealthy.

These effects might seem obvious, but it might be worthwhile to spend some time identifying effects that convey different feelings. In graphics applications, fog is frequently used to convey distance, but it is also capable of setting a general mood or tone for an environment. Often, environments can change dramatically with changes to the color or quality of the sky. In the movie *The Matrix*, the sky was tinted green to be more consistent with the color scheme of the movie. In *Seven*, filming on rainy, overcast days created a very dreary view of Los Angeles. Figure 4.12 shows one example of this. On the left, the hazy sky hangs over a city, creating a washed-out and slightly ominous look. On the right, the sky is clear, the scene is brighter, and the objects have more clearly defined areas of light and shadow. Each of these factors contributes to a much more inviting feeling.

Figure 4.12. Scenes from Beijing and Shanghai

4.9 Transparent Materials

Transparent materials are characterized by the fact that you can see through them to some degree. The ability to create semitransparent objects in 3D scenes has been available for a long time, but it has only recently become possible to create all of the subtle effects that go along with transparency such as reflection, refraction, and chromatic aberration.

4.9.1 Glass

Glass has been a part of virtual scenes for a while, but only in the sense that it is rendered as a transparent material. Real glass exhibits a much wider range of effects. First, like all transparent materials, glass refracts light. Refraction becomes much more apparent when you look at curved surfaces and objects with variable thickness. If you look at a wineglass, you will see that refraction gives you visual cues about its thickness.

Smooth, transparent glass also reflects the environment around it. However, the amount of light that reflects from the glass (vs. the amount that is allowed to transmit through) is dependent on the angle of the light. When you look at a curved glass, you will see that the glass is more reflective at the edges than it is on the side closest to you, as shown in Figure 4.13. This is due to the fact that you are only seeing reflections of light that is striking the glass at shallow angles.

There are many forms of glass and they all have different effects. Frosted glass isn't very transparent at all. Mirrors are made of glass with a highly reflective coating on the reverse side. Spend a little bit of time looking at glass. You might find that it is more reflective and refractive than you thought.

Figure 4.13. Reflections on a wineglass

4.9.2 Water

Water can also have many subtle qualities, many of which are caused by refraction. For instance, refraction causes underwater objects to appear displaced when viewed from above the water. Shooting fish in a barrel isn't quite as easy as it might sound. Refraction also causes the caustic patterns you see on the bottom of a pool. Waves on the surface of the water redirect light into different patterns based on the depth of the water at different points. These same surface patterns also redirect light that is reflected off of the surface of the water. On a sunny day, waves will reflect caustic patterns onto the sides of ships. Light reflecting off of waves gives you visual cues about the water itself. Small ripples tell you it's a pleasant day. Large wavy patches tell you the wind is stronger. Regular patterns and wakes tell you something about the movement of objects above or below the surface.

Finally, water can be filled with other, less pleasant materials. It can have an oily sheen on its surface or a thin layer of floating grime that dulls reflections. It can also be filled with silt and other matter that makes it difficult to see into the water. Take a close look at dirty water. From a distance, it appears to be a uniform color. Close up, you can see that light penetrates the water before being reflected by the silt. Reproducing this effect can help you prevent a muddy river from looking like chocolate syrup.

4.10 Paint

Many objects are painted, but there are many different kinds of paint and finishes. In homes, paints are made with a variety of materials and are available in a number of different finishes. Textured paints give rooms a sense of warmth and richness. Kitchens might have more of a gloss finish to aid with cleaning. Ask anyone who has recently painted their home and they'll tell you there's more to worry about than color.

Paint applied with a brush looks different from spray paint. Brushed paint will tend to be a bit thicker. In some cases, you might be able to see the brush strokes. Spray paint is usually thinner and more uniform. It doesn't have a large effect on the texture of the surface.

In the context of games, paint jobs are perhaps most important for cars. Auto paint can include many of the features seen in other materials. For instance, many cars have a clear coat that provides a greater sense of depth and will contribute to strong specular highlights. Some paints have small flecks of metal embedded in them that give the paint a sparkling appearance.

Exotic auto paints give the car much more than color. Multiple coats of paint give more expensive cars a very thick, luxurious look. Iridescent paint can change color with the viewing angle. These properties can lend a lot of realism to a sports car racing game.

However, cars shouldn't always be shiny and new looking. They get rusty and dusty. Paint chips and flakes over time. As a car gets older, the clear coat will deteriorate. Even if the car isn't rusty, it will take on a matte appearance. It will have less of a shine and generally look less interesting than a car with a newer coat of paint. This is just one example of the difference between new and worn materials.

4.11 Worn Materials

Regardless of the material, objects wear with age. Finishes get duller, colors fade, and surfaces get pitted and scratched. Accurately simulating these effects is important if you want to make sure that the player understands the environment. The ancient castle should never contain anything that looks like shiny plastic. Likewise, futuristic usually means shiny and new looking. These might sound like subtleties, but consider the *Star Wars* movies. In the first movie, high-tech spaceships look worn and old, a visual cue that tells you that it takes place after a long galactic war. The prequels introduce the queen's perfect chrome ship early on, telling you that this part of the story takes place during better times. The wear and tear on the ships becomes part of the story. Figure 4.14 shows everyday objects that are far from pristine. Would these objects "tell the same story" if you rendered them as shiny and new?

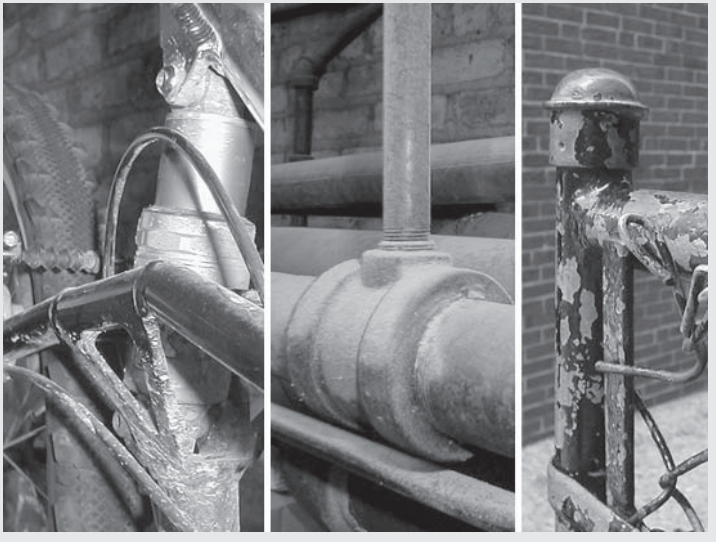


Figure 4.14. Dirty and worn objects

Wear and tear does not necessarily take place over the span of years. A rallysport car shouldn't be as shiny as a new sportscar. Warriors should look dirty and worn after a long battle. Changes in the way that characters look will help to convey a sense of time and experience. These cues might seem subtle, but they help to tell a story.

Conclusion

I have only described a handful of the many phenomena that you probably see every minute of every day. One could compile an entire book of examples, but I have chosen to highlight a few features that relate to the techniques discussed in later chapters. I encourage you to take a close look at real materials. As you look around, you will find many things I haven't mentioned. You will also find exceptions to the points I made. You will begin to acquire a mental model of what things really look like.

Until recently, many of the more subtle effects were not possible or not worthwhile to recreate in real time. As the hardware advances, so will people's expectations. Also, the gaming audience is expanding. Casual gamers are drawn to pleasing graphics and rich environments. People who would not think of playing Space Invaders suddenly find themselves spending forty hours a week in the rich environments of role-playing games. In the next several chapters, I show you techniques that can help you recreate real effects, but everything comes with a cost. Having a good sense of what things really look like will help you decide what is really important and prioritize effectively.

This page intentionally left blank.

Chapter 5

Lighting and Reflectance Models

Introduction

Chapters 1 through 4 primarily focused on describing the ways in which light travels through an environment before reaching a surface. Chapter 4 illustrated how different real-world objects appear when illuminated. This chapter begins to bring all the pieces together by giving you the theoretical and mathematical tools needed to reproduce those effects in your own virtual scenes.

I begin with an explanation of what happens when light reaches a surface and then provide several models that explain how that light is reflected from the surface. In doing so, I cover the following concepts:

- The rendering equation
- Basic illumination definitions
- Illumination and Lambert's law
- Bidirectional reflectance distribution functions (BRDFs)
- Diffuse materials
- Specular materials
- Diffuse reflection models
- Specular and metallic reflection models

5.1 The Rendering Equation

The contents of the previous chapters have been leading up to one key point: For a surface to be visible to the eye, there must be light that is reflected from, emitted from, or transmitted through that surface. In other words, the visible qualities of a surface can be defined solely by the way that that surface interacts with or emits light. For the sake of simplicity, I will assume that surfaces of interest do not emit light. I will also assume that the surfaces are fully opaque. This means that you can describe the visual appearance of any surface with a greatly simplified *rendering equation* as shown below.

Outgoing light = EmittedLight + ReflectanceFunction * IncomingLight
(assume no emitted light)
Outgoing Light = ReflectanceFunction * IncomingLight

Figure 5.1. A simplified rendering equation

NOTE:

This equation omits many intricate mathematical details that will be filled in as the chapter progresses. If you are also interested in a much deeper mathematical view, sources such as [1] are very highly recommended.

The equation in Figure 5.1 shows that the amount of light reflected from a surface is a function of the amount of incoming light energy incident upon the surface. Computing the amount of energy is slightly more involved than the previously shown attenuation functions (Chapter 2), and is the first topic covered in this chapter.

The second topic is the reflectance function. This can take many different forms. Some forms are based heavily on realistic physics. Others are more ad hoc and simply produce good visual results. In either case, the purpose of the reflectance function is to modulate the incoming light in a manner that reproduces the visual qualities of the surface. This is the most interesting and difficult part of rendering a surface realistically. However, before you can talk about reflection, you must know how much energy to reflect.

5.2 Basic Illumination Definitions

As you read through the literature, you will see such terms as radiance, irradiance, luminance, illumination, illuminance, and much more. In some contexts, it might seem as if these words all mean basically the same thing, but each is a term for a different phenomenon. The definitions below will be used throughout this book, but an easy-to-read glossary of older and deprecated terms can be found online in the "Lighting Design Glossary" [2].

5.2.1 Irradiance and Illuminance

In some texts, especially those not primarily concerned with graphics, you might find the term "irradiance." *Irradiance* is the total amount of energy received per unit area of a surface. For a given energy source and a given surface, irradiance is the measure of energy that arrives at the surface after accounting for the effects of attenuation and the geometric relationship between the energy source and the surface.

At the highest level, illuminance is essentially the same thing. The difference is that *illuminance* measures the amount of visible light energy in photometric terms. As Figure 5.2 shows, a lamp might emit both visible light and other forms of radiation, such as heat. Irradiance would include all forms of radiation, while illuminance includes only weighted values of the visible light energy.

Illumination is synonymous with illuminance. Therefore, "illumination models" are used to describe how visible light arrives at a given surface. They don't say anything about what happens once it strikes the surface. That is the role of radiance and luminance.

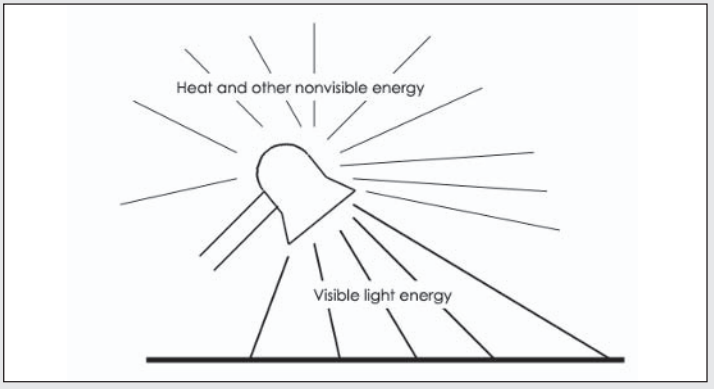


Figure 5.2. Irradiance and illuminance

5.2.2 Radiance and Luminance

Once energy arrives at a surface, the surface will reflect, transmit, or absorb different amounts of that energy. Similar to irradiance, *radiance* is the measure of energy that is reflected by the surface. *Luminance* is the measure of photometrically weighted light energy that leaves the surface, and is therefore the more interesting term in a graphics context. *Luminous intensity* is the amount of light energy that is emitted by the surface in a given direction. Both are functions of the material properties of the surface. However, just as

illumination is a function of the geometric relationship between the surface and the light, the luminous intensity as seen by a particular viewer is a function of the relationships between the light and the surface, the viewer and the surface, and the viewer and the light. For instance, different materials might reflect different amounts of light, even when illuminated by the same amount of light. Likewise, the same material might reflect different amounts of light in different directions, as shown in Figure 5.3.

Also, both are the sums of light energy that is reflected, emitted, or transmitted from the surface. Light that reaches a viewer from a glass surface might be the sum of light that is reflected from its surface and light that transmits through the glass object. Both contribute to the amount of light the user sees, although the mathematical formulae used for each contribution might be very different.

This chapter focuses on the mechanics of light reflecting from a surface. Transmitted light and other effects are explained in later chapters. The first four chapters explained the mechanisms by which light arrives to a given point. This chapter explains what happens once it gets there.

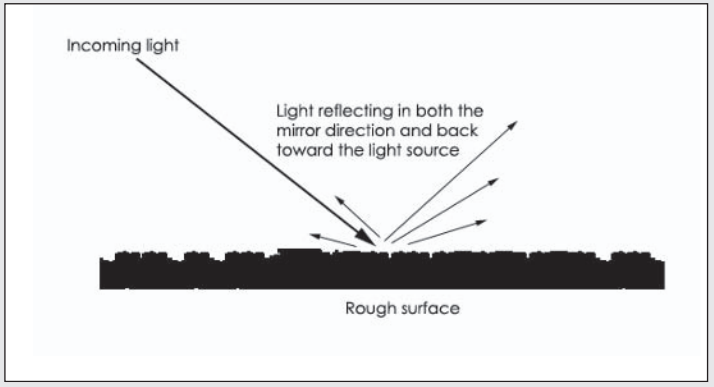


Figure 5.3. Outgoing light in different angles

5.3 Lambert's Law for Illumination

The attenuation equation in Chapter 2 is based on the assumption that light is striking an ideal point when in reality light strikes a very small but finite area. The ideal point approximation is acceptable for attenuation, but to fully understand illumination you need to remember that light is measured in terms of energy per unit of area. Figure 5.4 shows the idealized case of light striking a point and the more realistic case of light striking a very small but finite area on a surface.

Remember, the output of a light source is measured in terms of energy per unit of area. This means that if you minimize the amount of area receiving the light, you will maximize the amount of energy per unit of area. Geometry and trigonometry can be used to prove that the amount of affected area increases as you increase the angle between the

surface normal vector and the vector between the light and the point on the surface. This is shown graphically in Figure 5.5.

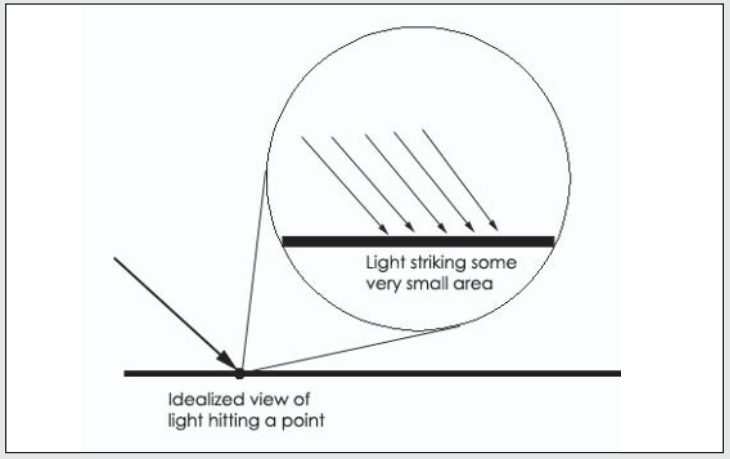


Figure 5.4. Light striking a point on a surface

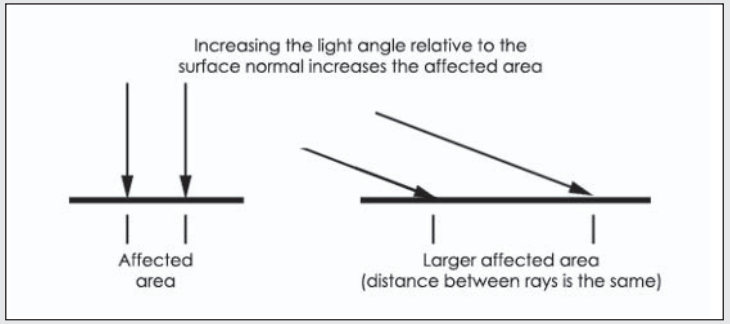


Figure 5.5. Increasing the angle of the surface relative to the light vector increases the affected area.

NOTE:

"Rays" of light as depicted in Figure 5.5 will be emitted in different directions along radii of a sphere, yet Figure 5.5 shows parallel rays. As you saw in Chapter 2, you can approximate rays as parallel if the angle between them is very small. This is a safe approximation for very small areas and simplifies Figure 5.5.

By increasing the affected area, larger angles decrease the amount of energy per unit of area assuming the amount of energy from the light remains constant. The law of conservation of energy says this must be the case. The rightmost diagram in Figure 5.5 demonstrates what happens when the surface normal is perpendicular to the light direction. In this case, all light energy would travel past the surface, never hitting it. Of course, this is only the case for an ideal surface. A non-ideal surface has a lot of variation, so small details will inevitably receive and reflect a portion of the energy.

Glassner [1] describes the same phenomenon in a slightly different way. If you think in terms of an element of a surface that is smaller than a given volume of light, the projected area of that element will get smaller as the angle increases, as shown in Figure 5.6. This means that the surface element will receive less of the energy present in the fixed volume of light. Both explanations are equivalent, although it might not seem like it.

More succinctly, the amount of energy that each unit of area receives is a function of the incoming energy and the cosine of the angle between the surface normal and the light

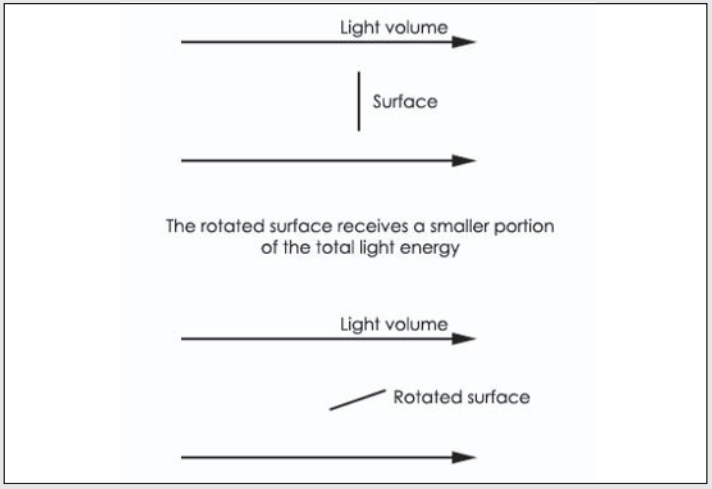


Figure 5.6. Increasing the angle of the surface relative to the light vector decreases the projected area.

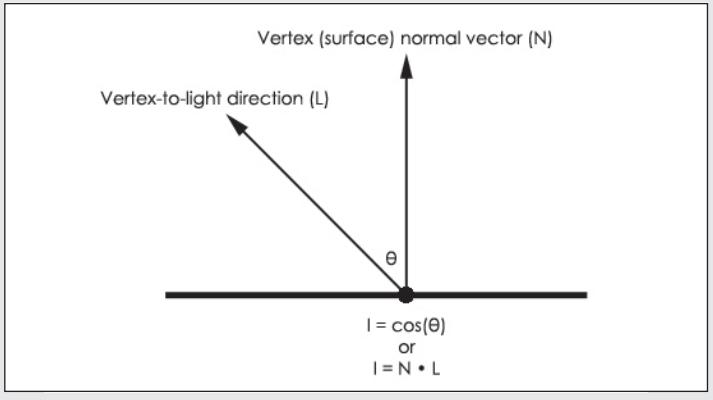


Figure 5.7. The relationship between the angle and the amount of illumination

direction vector. This is referred to as the cosine law, as first described by Lambert. Figure 5.7 shows this graphically.

Figure 5.7 implies that the amount of illumination is clamped to positive values between 0 and 1. When the cosine of the angle between the two vectors is less than 0, the light is behind the point on the surface, so the amount of light must be 0. If you forget to clamp the value, lights that are positioned behind the point will effectively remove energy from the surface. This is obviously incorrect.

You now have all the pieces to describe how much light energy reaches a point on a surface. In order for the surface to be visible, some of that energy needs to be reflected toward the viewer. The equation in Figure 5.8 revisits the equation seen in Figure 5.1 at the beginning of the chapter. Notice that the equation does not explicitly include attenuation effects. This equation is meant to describe how a material on a surface reflects light. How the light was attenuated before it got there is immaterial.

$$I_0 = \text{ReflectanceFunction} \times I_L \cos(\theta)$$

Figure 5.8. A rudimentary illumination model

In Figure 5.8, and for the rest of this chapter, it is important to note that the intensity, *I*, is a color value that is usually encoded as three RGB values. Therefore, the equation in Figure 5.8 really represents three separate equations. In some cases, a material might have different reflecting behavior for the different color channels. If not, you usually

don't need to solve separate equations explicitly because shaders and other tools are good at computing vector values. The remainder of this chapter will be focused on defining a set of functions that describe how the light is reflected.

Every material reflects light in subtly different ways, but a common way of describing reflectance is with a bidirectional reflectance distribution function, or BRDF.

5.4 Bidirectional Reflectance Distribution Functions (BRDFs)

A BRDF is a function that describes how light from a given incoming direction is reflected in an outgoing direction. In most cases, the outgoing direction of interest is toward the viewer. Consider glare on your monitor. If you have glare, you can usually minimize it by either moving your head, moving the light source, or lowering the intensity of the light source. This would imply that the amount of reflected light you are seeing is some function of your viewing angle relative to the surface of the screen, the light angle relative to the surface of the screen, and the intensity of the light itself. You might call this function a "reflectance distribution function" because it describes how reflected light is distributed over a range of viewing angles.

If you experiment further, you might find that, for many materials, the amount of glare you see is the same if you and the light source switch positions. This would imply that the function was bidirectional, so you might find yourself talking about "bidirectional reflectance distribution functions." While not applicable to every material, the general idea of such a function provides a useful framework for much of what you see in the real world.

In short, a BRDF is a mathematical function designed to yield some of the same kinds of results that you might have observed when reading Chapter 4. Figure 5.9 extends Figure 5.8. Here, a BRDF is used as the reflectance function.

$$I_0 = \mathrm{BDRF}(....) \times I_L \cos(\theta)$$

Figure 5.9. The role of a BRDF in the lighting equation

When light strikes a surface of a given material, it is reflected, transmitted, or absorbed. Reflected light leaves the surface at one or more angles. For some materials, a BRDF can be used to solve the outgoing intensity of light at any of the possible exiting angles. Note that a BRDF is not always adequate for some materials. You will see some other representations in later chapters.

BRDFs can take many forms, but as a rule they have two fundamental properties [1]. The first is that they are bidirectional. This means that you can swap the incoming light direction and outgoing direction and the resulting values would be the same. The second property is that a true BRDF

will obey the law of conservation of energy. You can, of course, break this rule by picking bad values or formulae for your BRDFs, but this is a good rule to keep in mind because it will help to make sure that your code doesn't yield something that looks unrealistic. Having said that, many reflection models are *phenomenological* — they produce desirable effects even though they don't adhere to these two rules. I'll return to this point later in this chapter, once I define all of the pieces of BRDFs themselves.

5.4.1 Parameters to a BRDF

Figures 5.8 and 5.9 describe the role of a BRDF in the general illumination equation, but you still need to know the parameters for the function itself. Imagine that a hemisphere surrounds every point on a surface. This hemisphere defines all of the possible directions from which light can either enter or exit the surface. Thinking in terms of a hemisphere allows you to think of the incoming and outgoing directions in terms of spherical coordinates, wherein you have two angles for both the incoming direction and the outgoing direction. These four angles become the four major parameters of the BRDF. Figure 5.10 shows this graphically.

BRDFs are not limited to these parameters, but they are the most basic set, since you are interested in how light coming from one direction is reflected to another. However, it's important to note that some BRDFs might not use all four parameters. Others might represent the directions as vectors. Conceptually, the two representations are equivalent.

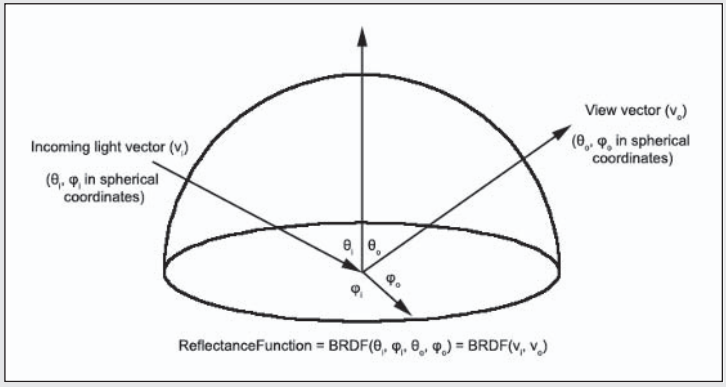


Figure 5.10. Incoming and outgoing directions as parameters to the BRDF

Also, some BRDFs can be dependent on wavelength, adding a fifth parameter. Other BRDFs might have tens of parameters to model many subtle effects. Some can be dependent on the actual location of a given point on the surface. Wood and rock are good examples of this. Different areas of the wood grain or rock surface might reflect light differently. For the most part, the BRDFs in this chapter assume a consistent material across the surface. Also, the BRDFs will be assumed to be wavelength independent unless otherwise noted. For the moment, I will concentrate on simpler BRDFs where the incoming and outgoing angles are the most important factors.

5.4.2 Isotropic vs. Anisotropic Materials

Almost all materials exhibit some amount of *anisotropy*, meaning that their reflective properties are dependent on the viewer's angle around the surface normal. An extreme example of an anisotropic material is brushed metal. As you turn a piece of brushed metal, it will exhibit different amounts of reflection at different angles. However, there are many materials for which the anisotropic effects can be easily ignored. A smooth piece of plastic doesn't exhibit much anisotropy as you spin it.

Although anisotropy might be caused by many different surface and material attributes, the effect is most easily demonstrated by looking at a grooved surface such as a record. Figure 5.11 shows how such a surface reflects light. On the left, the light direction is parallel to the grooves and it reflects just as it would from a smooth surface. On the right, the light is perpendicular to the grooves, causing some of the light to be reflected differently in the troughs.

Sometimes, anisotropic effects can be dynamic and change as the surface changes. One example of this is skin. Figure 5.12 shows a very simple representation of pores in the surface of human skin. On the left, the pore is circular, which would yield isotropic reflection effects. On the right, the skin is stretched and the elongated pore would cause subtle anisotropic effects.

There are a couple of things to remember when looking at Figure 5.12. First, the direction of anisotropy in something like skin is highly dependent on the type of stretching. A frown would probably yield a slightly different effect than a smile. Secondly, these effects can be extremely subtle, but arguably important. We are very good at recognizing human skin, so extremely subtle effects are often the difference between something that looks lifelike and something that looks like a mannequin. On the other hand, simulating the dynamic anisotropic effects on an inflating balloon is arguably a waste of resources.

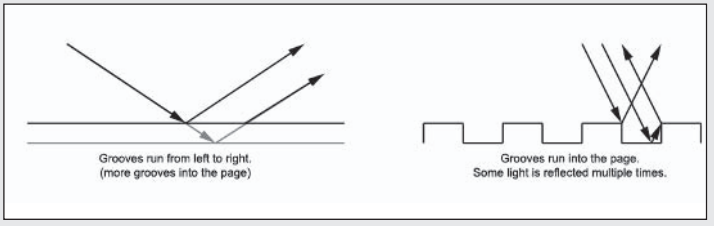


Figure 5.11. Reflections from an anisotropic surface

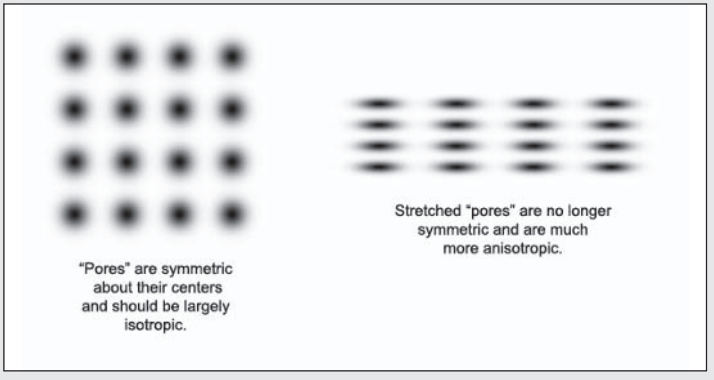


Figure 5.12. Stretched pores affect anistropy.

This chapter covers examples of both isotropic and anisotropic materials. In the case of isotropic materials, such as smooth plastic, the models ignore any anisotropic contribution because it is so small relative to the other surface features. For other materials, such as brushed metal, the anisotropic effects contribute significantly to the overall appearance of the object, requiring an anisotropic shading model.

5.4.3 BRDFs vs. Shading Models

The last several pages outlined the concepts of illumination and the reflectance of that illumination as dictated by the BRDF of a material. A BRDF is an excellent way to describe the properties of a surface in real physical terms. It also provides a nice framework in which to think about the angles and factors that affect reflection. Having said that, real-time shading models used to actually render a surface are usually more ad hoc. Frequently used models such as the Phong model comply with the spirit of BRDFs without being completely physically correct. This is because, in the case of real-time graphics, good-looking, computationally inexpensive operations frequently win over theoretically rigorous calculations.

Therefore, the remainder of this chapter focuses on describing shading models, which might or might not be true BRDFs in that they might not comply with all of the rules of BRDFs. Still, it is important to remember that the overall concepts remain the same even when some models are not, strictly speaking, BRDFs.

5.5 Diffuse Materials

Some of the examples in Chapter 4 were of materials that scatter light evenly in every direction. For example, a material such as matte paint will appear equally bright no matter what angle you view it from. The *cosine law* says that the total amount of illumination is dependent on the angle between the light direction and the surface normal, but for matte paint, the amount of reflected light the viewer sees is nearly the same from any angle. This is because the paint has a very rough surface. When light hits the paint, it is scattered in many different directions. If you change your viewpoint, you won't see much variation in brightness. On average, it appears as though the same amount of light is scattered in every direction.

5.5.1 A Simple Diffuse Shading Model

Matte paint is one of many materials that can be classified as a diffuse material, or one that reflects light evenly in all directions. Figure 5.13 shows the idealized case. Every point on the hemisphere represents the amount of reflected light in that particular direction. For a purely diffuse material, all of the intensities are equal.

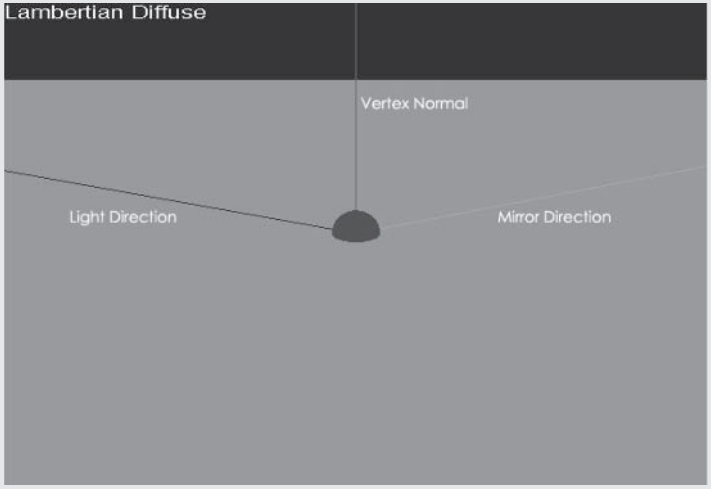


Figure 5.13. Visualizing the outgoing intensity of a diffuse material

This is an idealized model because most real materials do not scatter light in a completely uniform pattern. If you were to revise Figure 5.13 to show the real reflection distribution for matte paint, the surface of the hemisphere would be much less uniform. It would probably also tend to lean in one direction or another. Finally, a real material isn't completely consistent. You would probably get different results for different points on the surface. However, the differences are small enough that you can approximate many materials with the idealized case. Remember, the fidelity of the color on the display isn't good enough to render an entirely accurate reproduction, even if you added the math to do so.

All things considered, the shading model for a diffuse material is simply the illumination value given by the cosine law multiplied by some scaling value. This is usually referred to as a Lambertian diffuse model because of its dependence on Lambert's law.

$$I_0 = D \times I_L \cos(\theta)$$

This scaling value is the diffuse color. If you set the diffuse color to red, only the red component of the light will be reflected. Using black as the color of a perfectly diffuse material would mean that the material absorbed all light energy. In other materials that are not perfectly diffuse, a color of black would simply mean that the material did not diffusely reflect light, although it does not necessarily mean that all the light energy was absorbed. It might mean that it was reflected in some other way.

5.5.2 Diffuse Materials and Conservation of Energy

Although it might not affect your simpler diffuse calculations, it might be a good idea to be aware of the relationship between BRDFs and the law of conservation of energy. Although the law applies to all materials, the effect is easiest to demonstrate with diffuse materials.

Imagine you have a material in your gaming world that is pure white. Being pure white, you give it a diffuse color of 1.0 for all color channels. This yields pleasant results, but consider the following. Imagine shining a red laser directly

onto the surface as shown in Figure 5.14. Because the light is very bright pure red, you would give it a value of 1.0 for the red channel and 0.0 for the other channels.

Because the laser is shining directly onto the surface, it receives the full power of the laser (the cosine law yields a value of 1.0). Because it is a perfectly white diffuse material (1.0 for all color channels), the surface reflects the light at full intensity in all directions. In the real world, this would mean that two viewers looking at the surface from different directions would both see as much energy as they would if they were looking directly into the laser. Ocular health issues aside, this essentially means that the surface has doubled the overall intensity of the laser. This is clearly not possible. Stated more formally, the total amount of outgoing energy must be less than or equal to the amount of incoming energy. For a real material, the total amount of outgoing energy will always be less because there is always some amount lost to absorption.

In reality, a diffuse material would redistribute the energy from the laser and reflect some fraction of the energy in different directions, meaning that the "diffuse color" cannot be perfectly white (in the sense that it reflects 100 percent of the energy). Instead, a real material would have a diffuse color value that is significantly less than 1.

This raises some questions. Most importantly, if the above is true, how can one see a white object in the real world? The answer is that our eyes are very different from monitor screens. In the real world, a white bedroom wall is significantly less bright than a white-hot filament. What we see in the real world is usually different shades of gray. Both the

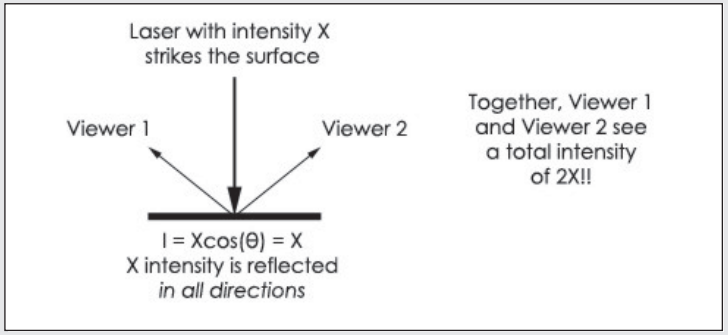


Figure 5.14. Shining a laser directly onto a surface

monitor and the renderer (in most cases) have less range and less color resolution than the real world, meaning that a white wall in a brightly lit room and a white light appear as the same intensity when shown in most rendered scenes. Reconsider the laser example. A monitor is incapable of reproducing the brightness of the laser. Therefore, a laser shone on a white surface in a virtual scene does reflect back to the real viewer with some intensity that is less than that of a real laser. Because of the limitations of the monitor, everything works out in the end in a way that mostly satisfies the conservation of energy and our eyes even if it's not exactly visually correct. In reality, a full intensity laser light would be much brighter than a red pixel on a monitor.

So, why am I making such a big deal of this? There are two main reasons. The first is that a deeper understanding leads to better intuition about how to implement different effects. The second reason is that these finer points will become

more important as you begin to see cards and techniques with higher dynamic range images and full floating-point operations. Once you see how to handle images with a higher dynamic range, you should keep the conservation of energy in mind because it will be possible to represent the laser light as a very bright value and adjust other values accordingly. The exact mechanics of this will be discussed in a later chapter.

5.5.3 Purely Diffuse Materials

In the previous laser example, I touched on the fact that a diffuse color of pure white can lead to unrealistic results because too much light is being reflected in too many different directions. This begs the question: What is the maximum value for a purely diffuse material? It is relatively easy to solve for this with a little bit of calculus. I have omitted the mathematics, but Glassner [1] shows that you can find the value by obeying the law of conservation of energy and integrating the reflectance function over a hemisphere. If you do this, you will find that the maximum value is $1/\pi$.

As you read through the literature, you will find that many sources include this value as a normalization factor that helps reflectance functions stay within a realistic range. Most game programming books omit this term, probably because diffuse color values are more subject to artistic and aesthetic constraints than physical ones. With that in mind, I have omitted the term here as well.

5.6 Specular Materials

Pure specular materials could be thought of as the opposite of purely diffuse materials. Where a pure diffuse material will reflect light evenly in all directions, a pure specular material will only reflect light in the mirror direction, as shown in Figure 5.15.

5.6.1 Purely Specular Materials

A mirror is a good example of a material that is very specular (although real mirrors are not quite purely specular). If you shine a focused light on a mirror and view the mirror from any angle other than the mirror angle, you will probably not

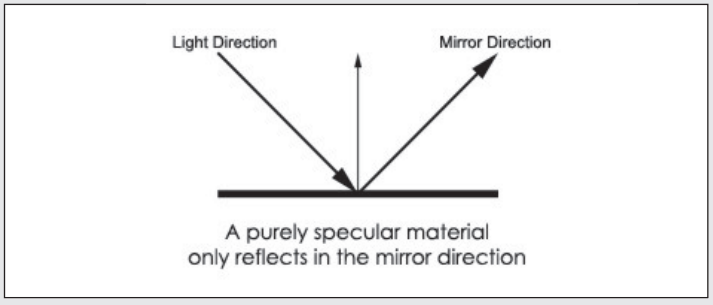


Figure 5.15. A purely specular material

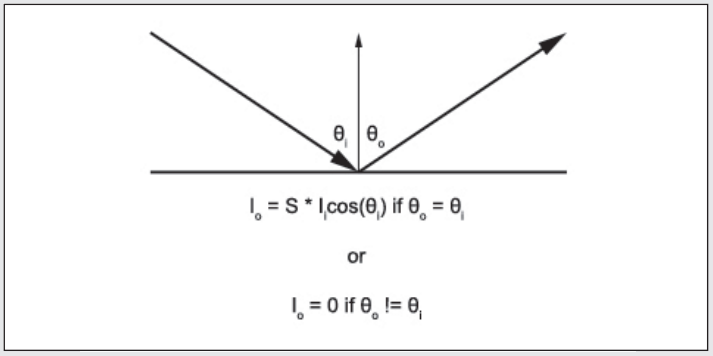


Figure 5.16. The components of a purely specular shading model

$$I_o = S * (R \cdot v_o)^n * I_i$$
or
$$I_o = S * (R \cdot E)^n * I_i$$

$$R = 2N(v_i \cdot N) - v_i = 2N(L \cdot N) - L$$

R is the mirror reflection vector
N is the surface normal vector
E is the vector to the eye
L is the vector to the light

Figure 5.17. The Phong specular model

see the light. Tilt the mirror so that you are viewing it along the mirror angle and you will see the light. Very high-quality mirrors are very close to being purely specular, which is why they can be used to redirect laser beams with minimal loss of the total energy content of the laser.

Specular materials also have the first instance of an interesting shading model. The model for a diffuse material was simply a scaling factor. For a specular material, the outgoing energy as seen by a viewer is a function of the viewing angle, the incoming angle, and the surface normal at a given point on the surface. Figure 5.16 expresses this graphically.

In this case, the model is comprised of a specular color factor multiplied by a factor of either 0.0 or 1.0, depending on whether or not the viewing angle matches the mirror reflection angle. However, purely specular materials are not terribly interesting because they are relatively rare in nature. Instead, you need a model that relates to a wider range of real materials.

5.6.2 Specular and the Phong Model

Real materials will exhibit some amount of specular reflection in a range of angles around the mirror reflection angle. Very shiny surfaces specularly reflect over a very small range of angles. Less shiny materials reflect specular highlights over a wider range.

Mathematically, this phenomenon can be expressed as an equation that finds the similarity between the mirror reflection angle and the view angle and raises that value to the power of some exponent, as shown in Figure 5.17. The equation, shown in Figure 5.17, was developed by Phong Bui-Tuong in 1975 and is nearly ubiquitous for real-time specular reflection.

Figure 5.17 shows the equation in two equivalent forms. The first form refers to specular reflection for general input and output vectors. The second form is a more typical form that explicitly refers to the light and eye vectors. The equation above is usually accompanied by the diffuse term shown in section 5.5.1. This is not necessarily a physically correct equation for many materials, but it does reproduce results that appear realistic. The dot product is used to find how close the viewing angle is to the mirror reflection angle. The exponent, or *specular power*, is used to constrain the area of the specular highlight. Higher specular power values accentuate the differences between the view vector and the mirror reflection vector. The result is a BRDF that drops sharply in intensity as the view vector moves away from the reflection vector, as shown on the left-hand side of Figure 5.18. A lower specular power produces a gentler falloff value, yielding something similar to the right-hand side of Figure 5.18.

5.6.3 The Blinn-Phong Model

Jim Blinn adapted the Phong model two years after it was first developed in a way that simplifies the calculations. Instead of calculating the true reflection vector, you can calculate a *half vector*, which is the vector halfway between the light vector and the eye vector. You can then find the dot product between the half vector and the surface normal as a measure of how close you are to the mirror direction. When the viewer is looking along the mirror direction, the half vector and the surface normal will be equal. As the viewer moves away from the mirror direction, so will the half angle vector deviate from the surface normal vector. The difference between the half vector and the surface normal provides an approximation of how close the view direction is to the mirror direction. Figure 5.19 shows a graphical representation of this and the corresponding equations.

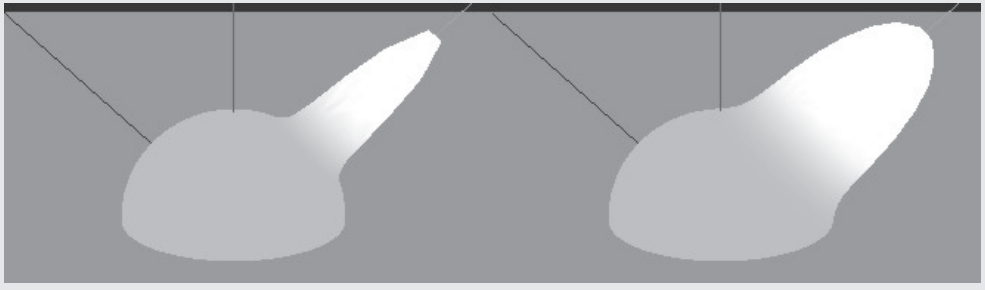


Figure 5.18. Different levels of specular power

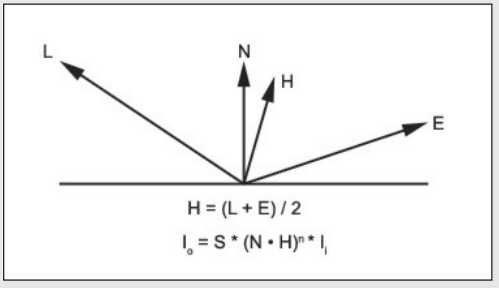


Figure 5.19. Lighting with the Blinn-Phong equation

This approximation with the half vector does yield slightly different values than those seen in Figure 5.18. Figure 5.20 shows the two methods side by side. The methods are different, but all material values are the same. Chapter 7 contains the sample code that will let you view the different distributions in real time to get a better feel for the difference.

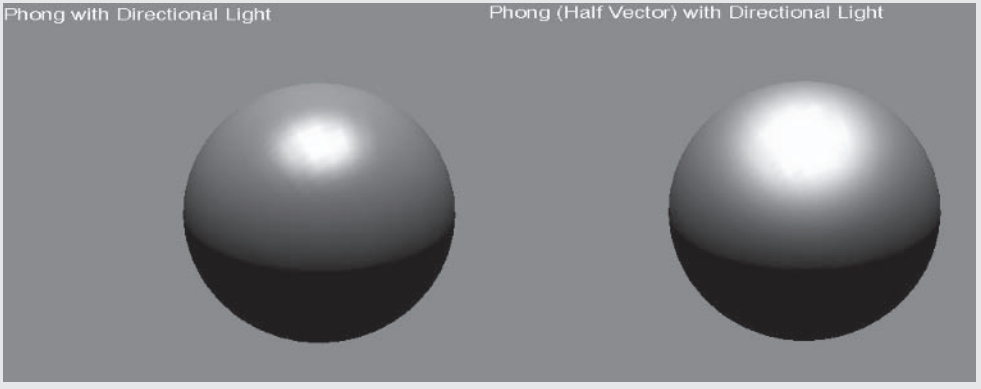


Figure 5.20. Phong with true reflection vector (left) and half vector approximation (right)

In addition to being a simpler calculation, the half vector creates some opportunities to optimize. For example, directional lighting allows you to calculate the half vector once for all vertices because the light and view directions are always the same. Once the half vector is calculated, it can be sent to a vertex shader to be reused for every vertex calculation. However, this optimization is yet another approximation that assumes that the eye-to-vertex angle does not change dramatically over the surface of the object. Therefore, it can improve performance, but should be used with care.

Specular materials rendered with the Blinn-Phong model are perhaps the most ubiquitous materials in real-time graphics because they are both computationally inexpensive and they yield acceptable visual results. Also, materials usually exhibit some combination of diffuse and specular reflection. To recreate this effect, simple use the two techniques together.

5.6.4 Combining Diffuse and Specular Reflection

Most texts present the Phong model with both diffuse and specular components. I have presented the two components separately to make a subtle point. Occasionally, you might want to deal with the two components in very different ways. For instance, you might calculate the diffuse component mathematically in a vertex shader, but use a pixel shader to render per-pixel specularity with a gloss map. Similarly, you might compute a diffuse lighting solution with spherical harmonics and mathematically add a specular component in a vertex shader. The main point is that the diffuse and specular components do not necessarily need to be viewed as a single entity.

The combination of the Blinn-Phong specular model and the Lambertian diffuse model is what is used in most real-time 3D graphics you see today. It is the built-in lighting model for both DirectX and OpenGL, but it isn't perfect. Nor is it capable of recreating all material effects. The remainder of this chapter focuses on other lighting models that are made possible by programmable hardware.

5.7 Diffuse Reflection Models

The Lambertian diffuse reflection model is simple and convenient, but it cannot accurately depict all diffuse materials. In section 5.5, I described a diffuse material as one for which small, rough surface features scatter light evenly in all directions. However, there are some materials that scatter the outgoing light unevenly, leading to subtle but important visual differences.

5.7.1 Oren-Nayar Diffuse Reflection

The Oren-Nayar diffuse shader model [3] models the fact that some diffuse materials exhibit different behavior than that described by the Lambertian model. Instead of diffusing light uniformly in all directions, some rough surfaces such as dirt, clay, and some types of cloth exhibit a high degree of retroreflection (reflection back in the direction of the light source). This creates a flatter impression than the Lambertian model, especially when the viewer is looking along the light vector. Both [3] and [4] demonstrate good examples of real-world objects that exhibit this behavior. Figure 5.21 shows screen shots from sample applications found later in this book. In these two samples, the light and view directions are the same, but the object on the right is more uniformly lit than the Lambertian object on the left. This corresponds well to natural materials such as the clay used in pottery, as well as objects such as the moon.

Whether you look at the moon or a clay pot, the rationale for this flattening effect is the same. Oren and Nayar recognize that small, rough facets on an object are significantly smaller than the amount of area in a single pixel of the

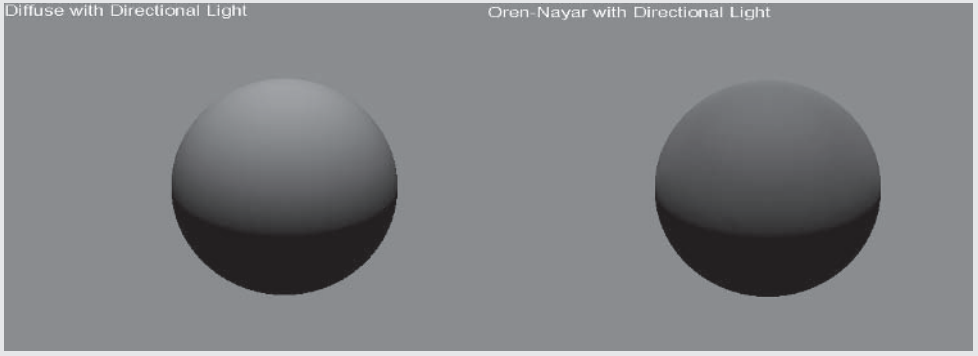


Figure 5.21. Lambertian and Oren-Nayar diffuse objects

rendered scene. In other words, each pixel contains a very large number of facets facing in different directions. They also show that, while a single facet might be Lambertian, the behavior of the larger collection is not. For the type of materials they are interested in, the facets tend to reflect a greater distribution of light back toward the light source and this behavior is determined by the roughness of the material.

To model roughness, they chose to describe the directions of the facets as a random Gaussian distribution with a mean value of 0. From there, the standard distribution of the facet directions can be treated as a measure of roughness. A higher standard deviation means that the facet directions are more different, and the surface is rougher. Therefore, roughness is represented by the Greek symbol for standard deviation, sigma (σ).

The original paper [3] by Oren and Nayar goes into the rationale and mathematics needed to implement the model. They also present the complete model and a simplified version. However, Ron Fosner does a very good job of making the equation more programmer friendly in his *Gamasutra* article on HLSL [4]. The final form incorporates a surface roughness parameter, as well as the view and light angles relative to the vertex normal.

The equation in Figure 5.22 is basically the same as the one shown in [4] with some changes to the notation.

$$I_0 = D * (N \bullet L) * (A + B * \sin(\alpha) * \tan(\beta) * MAX(0, \cos(C))) * I_1$$

$$A = 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.09}$$

$$\alpha = MAX(\text{acos}(N \bullet L), \text{acos}(N \bullet V))$$

$$\beta = MIN(\text{acos}(N \bullet L), \text{acos}(N \bullet V))$$

C = the azimuth angle between the light vector and the view vector

Figure 5.22. The Oren-Nayar equations

Each of the terms is relatively easy and straightforward to compute, with the possible exception of *C*. Remember, the normal vector defines a plane. If you project the view and light vectors onto that plane, *C* is the angle between those two projected vectors. Computing this angle is not quite as trivial as finding the dot product. I will revisit this point when I show the actual shader implementation.

This equation can be implemented in a shader and produces the effects seen in Figure 5.21. The effect can be quite nice in certain situations, especially where the view and light directions are similar. However, the BRDF has a significantly higher computational cost than simple diffuse lighting, so you might want to be careful about not using it when the effect will not be very noticeable.

5.7.2 Minnaert Reflection

Although it was originally developed for other purposes, a shading model developed by Marcel Minnaert [5] has proven to be very good for modeling the appearance of velvet [6]. Minnaert's model is characterized by creating "darkening limbs" along edges, the result of which is an effect that looks very much like velvet. Figure 5.23 compares a diffuse object with a Minnaert object. The object on the right has a softer appearance, as if it is coated with velvet or a similar material.

The reflection equation is shown in Figure 5.24 below.

$$I_0 = D * ((N \bullet L)(N \bullet V))^{m-1} * I_1$$

Figure 5.24. The equation for Minnaert reflection

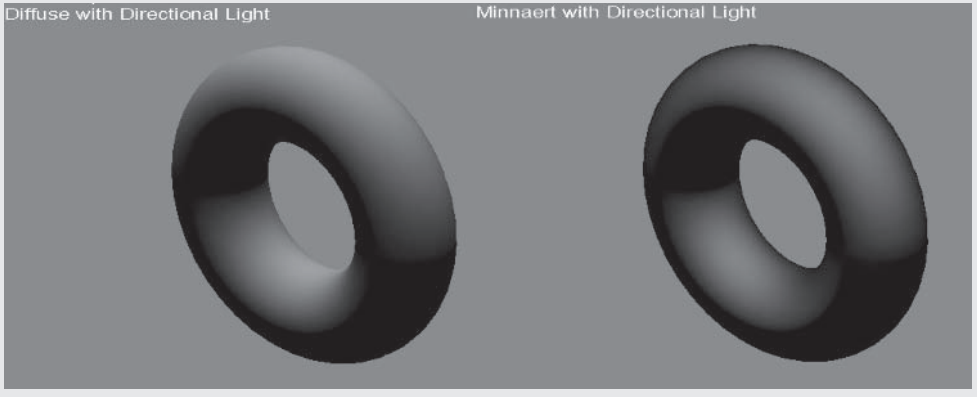


Figure 5.23. Lambertian and Minnaert objects

Different values of *m* will produce subtle changes to the overall look of the material. Chapter 7 includes code that will allow us to visualize the effect and experiment with different values.

5.8 Specular and Metallic Reflection Models

The Phong model discussed earlier in this chapter is frequently "credited" with making most of the shiny objects in computer-generated scenes appear plastic. You need other models if you want to achieve different effects that more accurately depict metals and other shiny materials. There are many models available, but the following sub-sections present a couple of useful models that provide good results and also provide a good conceptual starting point.

One thing to keep in mind as you read this section is that specular models reproduce the shape and intensity of the highlights on shiny materials, but the color of the highlight is dependent on the material. For instance, white might be a good specular color for plastic and other insulators, but conductors like metals will typically have highlights that are the same color as the metal. For instance, gold will have a brighter gold highlight and bronze will have a more reddish color. This might not come across in the grayscale screen shots, but it's important to keep in mind.

5.8.1 Ward Reflection Model

The reflection model developed by Ward [1] is interesting for a couple of reasons. First, it is more physically accurate than the Phong model. As such, it can be used as a true BRDF (where both the diffuse and specular components scale the irradiance term). Secondly, it can be expressed in both isotropic and anisotropic forms.

In both forms, the diffuse component is simply a constant value as described in section 5.5. The specular component is more involved than that of the Phong model. Figure 5.25 shows the isotropic form. The specular highlight is controlled by a roughness factor that is constant across the surface. This roughness factor is denoted by the Greek symbol sigma (σ).

$$I_o = \left(D + S * \frac{e^{-\tan^2\gamma/\sigma^2}}{2\pi\sigma^2\sqrt{(N \bullet L)(N \bullet V)}} \right) I_i (N \bullet L)$$

$$\gamma = \operatorname{acos}(N \bullet H)$$

Figure 5.25. The equation for the isotropic Ward model

In the anisotropic case, the roughness factor is defined in two orthogonal directions on the surface. This creates an anisotropic effect in the specular highlight as the view direction changes. Figure 5.26 shows the anisotropic form.

$$I_o = \left(D + S * \frac{e^{-\tan^2\gamma\left(\frac{\cos^2\phi}{\sigma_x^2} + \frac{\sin^2\phi}{\sigma_y^2}\right)}}{2\pi\sigma_x\sigma_y\sqrt{(N \bullet L)(N \bullet V)}} \right) I_i (N \bullet L)$$

$$\gamma = \operatorname{acos}(N \bullet H)$$

ϕ = the azimuth angle of the light vector on the tangent plane

Figure 5.26. The equation for the anisotropic Ward model

The result of the isotropic model is shown in Figure 5.27. As you can see, the isotropic version is very similar to the Phong model, although you might notice subtle differences when experimenting with the two interactively.

Figure 5.28 shows two variations of the anisotropic model. On the left, the direction of anisotropy is largely horizontal. On the right, the direction is more vertical.

The equation in Figure 5.26 includes ϕ, the azimuth angle of the light vector on the tangent plane. In plain English, this accounts for the fact that anisotropic surfaces have some direction of anisotropy. When the light vector is projected on the tangent plane, the angle between the resulting projected vector and the direction of anisotropy strongly influence the amount of specular reflection.

Finally, Figure 5.29 shows visualizations of the isotropic and anisotropic BRDFs. On the left, the isotropic version looks very similar to the Phong visualization shown in Figure 5.18. On the right, the anisotropic version has more of a fin-like appearance that demonstrates the tendency to reflect more light along a given direction.

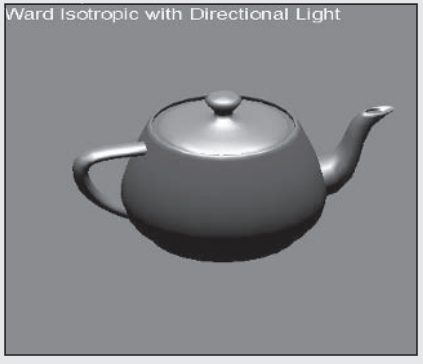


Figure 5.27. Shading with the isotropic Ward model

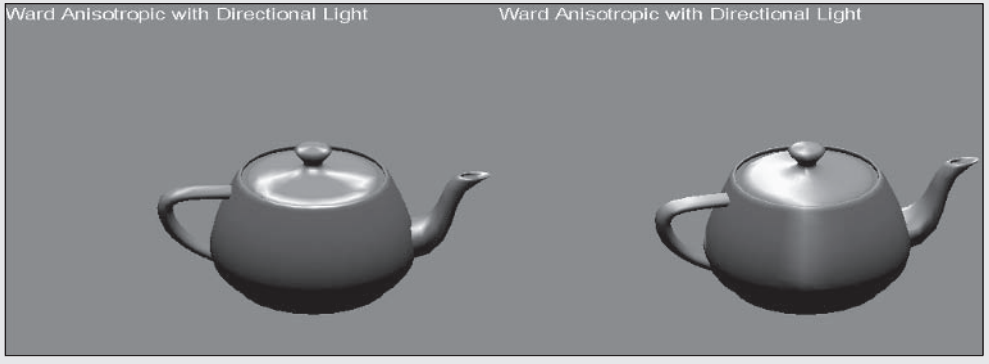


Figure 5.28. Shading with the anisotropic Ward model. Notice the anisotropy is largely horizontal on the left and vertical on the right.

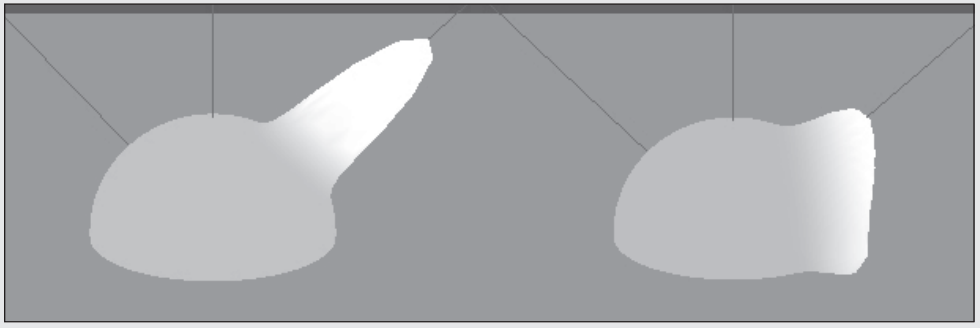


Figure 5.29. Visualizing the Ward model

5.8.2 Schlick Reflection Model

If the Ward model improves on Phong by making it more physically plausible, the Schlick model [7] improves upon it by making it faster. This is probably a more interesting improvement in the context of games. Schlick's model produces results that are similar to those achieved with the Phong model, but it eliminates the exponent, thereby reducing computational overhead. The equation for this model is shown in Figure 5.30 and a sample rendering is shown in Figure 5.31.

As you can see, the results are again similar to Phong, although the computational overhead could be much less. It's certainly much less than the Ward model. Having said that, there are subtle differences between the three models, and one might be better for some applications than others.

$$I_o = D * I_i * (N \bullet L) + S * I_i * \left(\frac{(R \bullet V)}{n - ((n-1) * (R \bullet V))} \right)$$

n is the shininess factor

Figure 5.30. The Schlick model

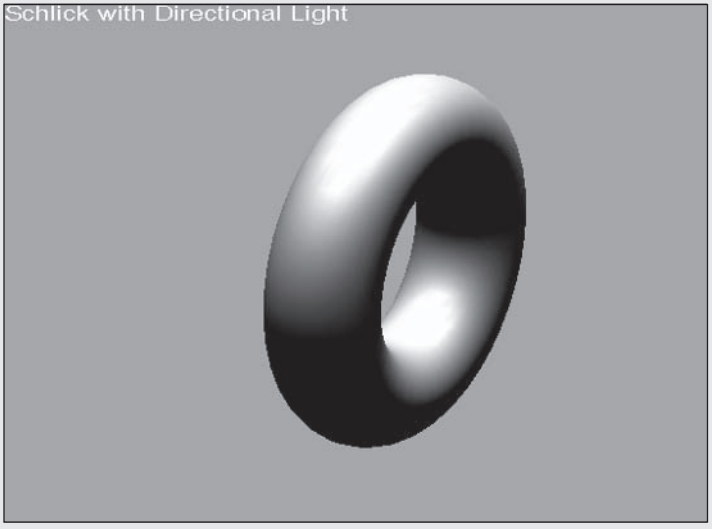


Figure 5.31. Rendering with the Schlick model

5.8.3 Cook-Torrance Model

The Cook-Torrance model [8] is the last model described in this chapter. It is based on the work of Torrance and Sparrow, which focused on developing a BRDF for metallic surfaces. Eventually, the work was reformulated for computer graphics with the help of Cook, and (in some forms) Blinn, Beckmann, and others.

The model describes a surface as a set of many tiny microfacets that reflect light in many different directions. It also accounts for the fact that some microfacets block incoming light from striking nearby microfacets (referred to as *shadowing*) and others block outgoing light leaving nearby microfacets (referred to as *masking*). These phenomena affect the amount of diffuse and specular reflection back to the viewer. The model also incorporates a Fresnel term to control the amount of reflected light at different angles of incidence. The complete equation is shown in Figure 5.32.

$$I_o = I_i * \left(\frac{FDG}{(N \bullet L)(N \bullet V)} \right)$$

Figure 5.32. The Cook-Torrance model

The three main terms (D, G, and F) each control different effects as described below. In previous sections, D has been used to denote the diffuse color of the material. In this section, D is used to denote a roughness term. Although diffuse reflection and roughness are related, it is important to note that D is not simply the diffuse color in the context of this model.

5.8.3.1 The Geometric Term

In Figure 5.32, the G term accounts for the effects of shadowing and masking between microfacets, which help determine the amount of specular reflection in a given direction. Figure 5.33 expands G, which includes factors you have seen in other specular terms.

$$G = \min\left\{1, \frac{2(N \bullet H)(N \bullet V)}{(V \bullet H)}, \frac{2(N \bullet H)(N \bullet L)}{(V \bullet H)}\right\}$$

Figure 5.33. The geometric term

The G term doesn't have any factors that are dependent on the physical properties of the surface. The remaining two terms do.

5.8.3.2 The Fresnel Term

The Fresnel term determines the amount of reflection from each microfacet and how metallic the object looks. The Fresnel term is discussed in greater depth in [1] and [9], but is shown in Figure 5.34 for completeness.

$$F = \frac{(g-c)^2}{2(g+c)^2}\left(1 + \frac{(c(g+c)-1)^2}{(c(g+c)+1)^2}\right)$$

$$c = (V \bullet H)$$

$$g = \sqrt{\eta^2 + c^2 - 1}$$

Figure 5.34. The Fresnel term

The Fresnel term can be wavelength (or at least color channel) dependent, yielding richer effects. In some cases, it's easier to use acquired data placed in a texture than it is to formulate an equation to reproduce the correct effect.

5.8.3.3 The Roughness Term

The final D term describes the roughness of the surface as the distribution of the slopes of the microfacets. There are a couple of different formulations for this. One formulation is the Beckmann distribution function shown in Figure 5.35.

$$D = \frac{e^{-\left(\frac{\tan\alpha}{m}\right)^2}}{m^2 \cos^4\alpha}$$

m is the average slope of the microfacets

$\alpha = \text{acos}(N \bullet H)$

Figure 5.35. The Beckmann distribution function

A simpler alternate distribution function based on a Gaussian distribution was suggested by Blinn [1]. This form is faster to compute and is shown in Figure 5.36.

$$D = ce^{-\left(\frac{\alpha}{m}\right)^2}$$

c is a user-selected constant

Figure 5.36. The alternate Blinn distribution function

5.8.3.4 The Complete Cook-Torrance Model

All three terms are combined to produce the final reflection model for metals. Note that the Fresnel term takes the place of the "specular color" seen in many of the other models. This can produce richer effects that more closely approximate real metals.

Figure 5.37 demonstrates the difference between an object rendered with the Phong model (on the left) and the same object rendered with the Cook-Torrance model (on the right).

Between the three terms, there are many parameters that influence the final look and feel of the surface. Rather than discuss specific values, I encourage you to experiment with different combinations in the interactive demos in later chapters. This type of experimentation can be more illustrative than a lot of textual discussions.

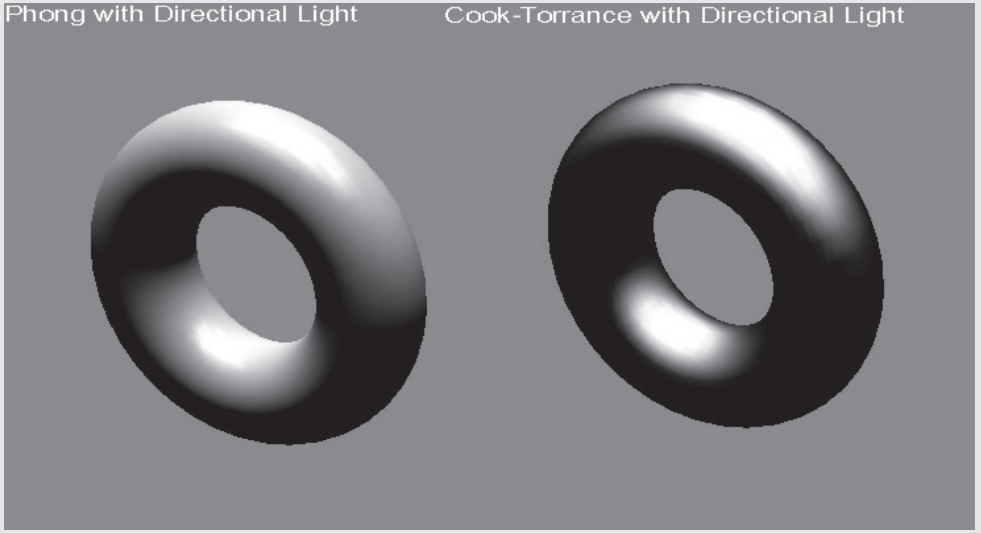


Figure 5.37. Phong and Cook-Torrance objects

Conclusion

The models presented here are just samples of a much larger body of models and BRDFs that can be used to create realistic renderings. These models were chosen to give you a feel for some of the math and methodology behind some of the equations you might find when searching the web or reading other texts.

You will also find that many models are variants of the ones presented here. Some are simplified versions of the Cook-Torrance model. Others optimize or augment these models to increase accuracy or decrease computational cost. In others, you might see the Fresnel term or distribution functions used in combination with other terms. In any case, the concepts presented in this chapter will help you understand the other models that are available or those yet to be invented.

There are some who might argue that many of these models are overly complicated and that simple Phong shading and texturing can handle the bulk of your rendering tasks without the computational overhead incurred with some of these models. That might be true for many applications, but I believe that will change as the bar is raised on realism in real-time applications. Just as movies must use more intricate shading models, so too will games and other forms of interactive media.

In presenting these models, I have tried to stay agnostic about which model is best and which model is fastest. Best depends on the task, fastest depends on the implementation. Implementation can take the form of a vertex shader, a pixel shader, or some combination of the two. The remaining chapters of this book will focus on implementation details using a variety of different strategies.

References

There is a wealth of resources on these subjects available both on the web and in print. The following references are just a small selection, but the information you find here should help you craft more web searches for your specific needs.

[1] Glassner, Andrew, *Principles of Digital Image Synthesis*, Morgan Kaufmann Publishers, Inc., 1995.

[2] "Lighting Design Glossary," http://www.schorsch.com/kbase/glossary/

[3] Nayar, S. K. and M. Oren, "Generalization of the Lambertian Model and Implications for Machine Vision," *International Journal of Computer Vision*, Vol. 14, 1995.

[4] Fosner, Ron, "Implementing Modular HLSL with RenderMonkey," http://www.gamasutra.com/features/20030514/fosner_01.shtml

[5] Minnaert, Marcel, "The Reciprocity Principle in Lunar Photometry," *Astrophysical Journal*, Vol. 93, 1941.

[6] NVIDIA Developer web site, "Minnaert Lighting," http://developer.nvidia.com/object/Minneart_Lighting.html

[7] Schlick, Christophe, "An Inexpensive BRDF Model for Physically-based Rendering," *Computer Graphics Forum*, 13(3):233-246, 1994.

[8] Cook, Robert and Kenneth Torrance, "A Reflectance Model for Computer Graphics," *ACM Transactions on Graphics*, January 1982.

[9] "Fresnel Reflection" (NVIDIA) http://developer.nvidia.com/object/fresnel_wp.html

This page intentionally left blank.

Chapter 6

Implementing Lights in Shaders

Introduction

This chapter is the first chapter that focuses on implementation and the first of two chapters that focus on implementations of lighting and lighting models. Since this is an advanced book, I'm not going to spend much time going over the implementations of basic light types (directional, point, and spot). You can find those implementations in many books and freely available sources listed in the reference section at the end of the chapter.

Instead, I will review some of the math concepts that are at the heart of many, if not all, lighting shaders. For many, this will be review, but I wanted to revisit material that is sometimes glossed over in samples and online articles. The basics will also provide a way to make sure that all readers are "on the same page" going forward.

Once I explain the basics, I will show per-vertex and per-pixel implementations of Warn spotlights. Warn lights are not necessarily the most ubiquitous lights, but they have properties that make for interesting vertex and pixel shaders. The concepts I will show for Warn lights will apply to other light types.

All of the shaders in this chapter are used within simple application frameworks meant to highlight the shaders while simplifying the actual application code. These frameworks can be easily understood by experienced users of DirectX

and/or OpenGL, but you might want to at least skim through Appendix C before moving forward. This chapter does not explicitly discuss the framework, but it does cover the following topics:

- Color modulation
- Object space light vectors
- Per-vertex Warn lights in DirectX
- Per-pixel Warn lights in DirectX

6.1 Basic Lighting Math

Before delving into individual implementations, it's important to remember that there are a couple of techniques that are common to all of the light types. They are addressed conceptually here, and they will be built into all of the shader implementations in this chapter.

6.1.1 Color Modulation

This topic has been addressed implicitly in many chapters, but it is addressed here explicitly because it sometimes causes confusion. The equations in Chapter 5 dealt primarily with intensity. Color values were three component vectors, but the three components were collapsed to one for the sake of simplicity. Remember, however, that the relationship between the incoming light and the resulting reflective color is the product of the light color and the material color (with some scaling factor provided by the reflectance function).

In other words, a purely green object lit by a purely red light will appear black because the red portion of the incoming light (1.0) will be multiplied by the red portion of the object's material (0.0). This recreates the physical reality of all of the red light being absorbed and only the green light (of which there is none) being reflected.

Occasionally, people will be surprised when an object lit by colored light appears black. Although it's mathematically correct, our minds immediately think something is wrong. The reason that it appears to be wrong is that pure colors are very rare in real scenes. It would be very difficult, if not impossible, to construct a scene where a purely green object is lit by a purely red light. Instead, materials and lights typically reflect a wide range of colors in varying degrees. In the real world, the green object might reflect at least a small amount of the red light and not appear completely black.

There are two important points to consider here. The first is that the relationship between input and output colors is multiplicative, as will be seen in all of the shaders. The other is that it might be advantageous to avoid using pure colors except in very specific instances. The world is not a perfect place, and your renderings will appear more realistic when you account for that.

6.1.2 Object Space Light Vectors

Many shaders transform light directions and positions into object space vectors and occasionally the reasons are not fully explained. The rationale is very simple and is most easily explained by example.

Imagine that you have defined a plane with four vertices. The plane is oriented so that its normal vector is pointing straight up. As such, the normal vector of each of the four vertices is directed straight up. Now imagine you have a simple directional light. With the light direction, the vertex normals, and simple diffuse lighting, you can easily and correctly light the plane. If only all of life were as simple as this scenario shown in Figure 6.1.

Now imagine you rotate your plane about one of its edges. If you light this plane, the result should be different from the first example. However, the normal vectors are still defined as pointing straight up as shown in Figure 6.2.

In order to correctly light the plane, you would need to also transform the four normal vectors. Once transformed, they can be used with the light vector to yield the correct solution. However, this requires several extra operations for every vertex.

Alternately, you could perform fewer calculations if you transform the light into object space. Continuing with the scenario of a simple plane, you would rotate the vertex positions in order to get the correct screen positions, but you leave the vertices unchanged. Instead, you rotate the light vector in an equal and opposite direction and use the transformed light vector with the untransformed normal vectors.

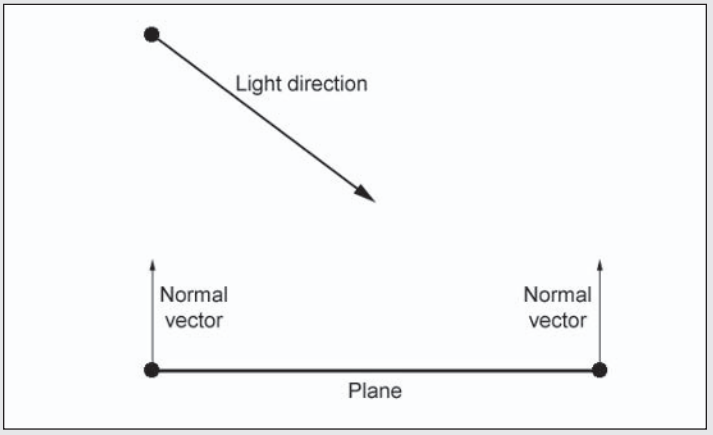


Figure 6.1. A simple lit plane

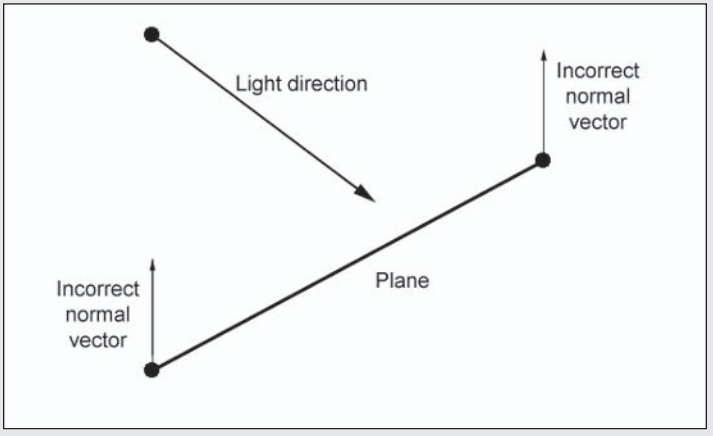


Figure 6.2. A rotated plane without rotated normals

This yields a correct and equivalent solution, as shown in Figure 6.3, but you transformed one vector instead of four. For more complex models, the difference in computation time could be dramatic.

This transformation is straightforward if you are using matrices and you assume that the lights and the objects are defined relative to some common reference frame. When you want to move or rotate an object, you multiply its position by some transformation matrix. This matrix transforms its position to some new position in the world. The inverse of this matrix is used to perform equal and opposite transformations on other entities, such as lights. Transforming the light position with the inverse matrix has the effect of placing the light in object space, accounting for the new position and orientation of the object. For the purposes of the lighting calculations, you are moving the light to the correct position relative to the light instead of the reverse. This is shown graphically in Figure 6.3.

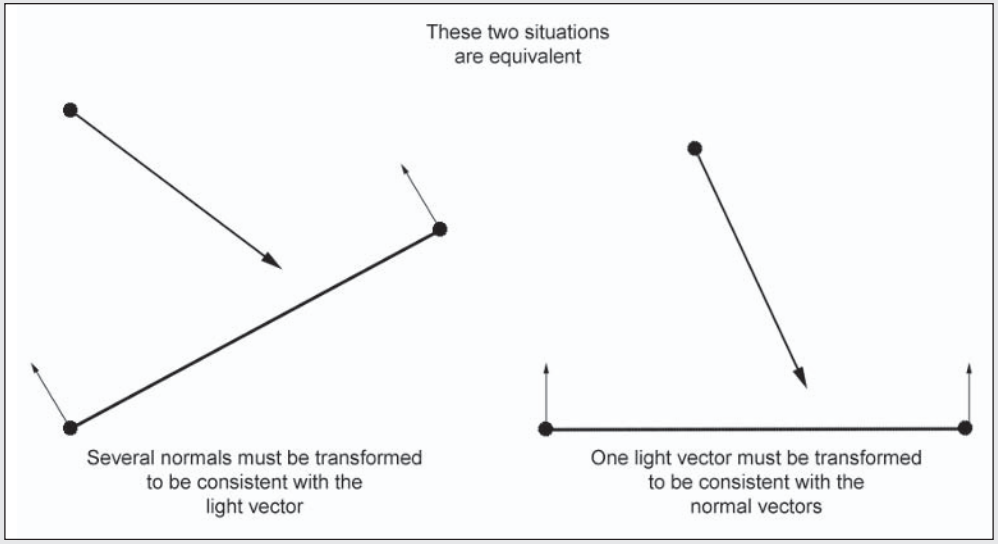


Figure 6.3. Transforming the light vector to object space

This allows you to simplify the shaders because you can pass the object space light vectors to a shader before rendering many vertices in a single object. However, you must remember to transform the light for each individual object. I will show the actual implementation of this when I show the vertex shader implementation for directional lights in section 6.2.1.

In some instances, it's not feasible to do this transformation because the shader itself transforms the vertex in some interesting way. If this is the case, you might not be able to benefit from this optimization, but it is often an easy way to save a few shader instructions.

6.1.3 Putting the Basics Together

Before moving to the actual shaders, I will briefly recap the core mathematics that will be used in every shader. In most lighting equations, you will need to find the light intensity based on some function of the light position, the surface normal, and the properties of the light and the surface itself. These functions were outlined in the previous chapter. In many cases, you can simplify your calculations if you leave the surface normal unchanged, but you place the light in object space. Once you have calculated the intensity, the final step is to multiply the light intensity by the color of the material. Because this chapter is concerned more with lights than materials, all the objects shown in the screen shots and in the code will be made of a perfectly diffuse white material.

6.2 Per-Vertex Warn Lights

Figure 6.4 revisits the Warn light equations shown in Chapter 2. In this chapter, I provide shader based implementations of this equation using a couple of different approaches.

This equation really only describes the way that light propagates from the source as a function of the angle between the light direction and the light-to-point direction. The complete solution needs to account for attenuation; points at different distances will be lit differently even if they are at the same angle. Finally, the shader has to account for the material of the object. This chapter assumes a perfectly diffuse white object, so I am using basic Lambertian reflection. Accounting for all of these factors, the final lighting equation implemented in the shaders is shown in the following equation:

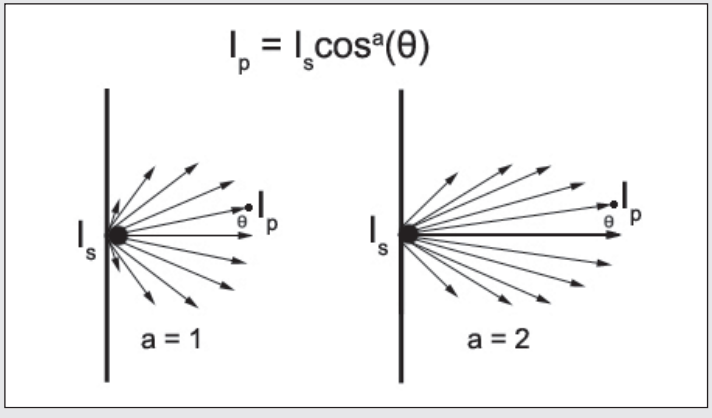


Figure 6.4. Warn lights

$$I_v = \frac{I_L}{r^2} D \cos^{\alpha}(\theta_{VL}) \cos(\theta_{NL})$$

$\frac{I_L}{r^2}$ = attenuated light intensity

D = object color = 1.0

α = Warn exponent

θ_{VL} = angle between light direction and light to vertex direction

θ_{NL} = angle between light direction and normal vector

In a vertex shader, this equation can be implemented fairly simply. I begin by showing the shader, followed by the DirectX code that feeds data and constants to the shader. Keep in mind that these shaders are meant to be more illustrative than optimal. It's probable that one could develop a faster shader, especially with lower level shader code. In this case, the shader is written in a higher level language and the following code is usable in both DirectX and OpenGL applications. This shader can be found on the companion CD as \Code\Shaders\Diffuse_Warn.hsl.

6.2.1 The Warn Shader

The shader code begins with declarations of the input and output structures. This code might not be used in a cg application, but it doesn't adversely affect it either. These examples are very simple, so the vertex only has a position and a normal. The output of the shader is limited to the final transformed position and the color.

```
struct VS_INPUT
{
   float4 Position    : POSITION;
   float3 Normal      : NORMAL;
};

struct VS_OUTPUT
{
   float4 Position    : POSITION;
   float3 Diffuse     : COLOR0;
};
```

The next five parameters are inputs that are constant across all vertices in a given object. They include the transformation matrix for the vertex, the position, direction, and color of the light, and the exponent that controls the falloff of the light. In support of the simplest DirectX application, these parameters are tied to specific input constant registers. In more sophisticated systems such as cg or DirectX's Effect system, these constants might be accessed more elegantly.

```
matrix Transform         : register(c0);
float3 LightVector       : register(c4);
float3 LightPosition     : register(c5);
float3 LightColor        : register(c6);
float  WarnExponent      : register(c8);
```

The actual shader code begins with two basic functions that determine the direction of the light as well as the attenuated intensity of the light. These functions are simple enough that they could have just as easily been put in the main shader code, but I broke them out into separate functions to better highlight the specific Warn computations in the main function.

```
float3 GetLightDirection(float4 VertexPosition)
{
   return normalize(LightPosition.xyz - VertexPosition);
}
```

The attenuation function is the "correct" function of the inverse square of the distance. You can change the attenuation behavior here without adversely affecting the Warn-specific behavior of the light.

```
float3 GetLightIntensity(float4 VertexPosition)
{
   float DistanceToVertex = distance
      (VertexPosition.xyz, LightPosition);
   return LightColor / (DistanceToVertex *
      DistanceToVertex);
}
```

This is the main function of the shader. It takes a vertex as input and outputs the transformed and lit vertex to the pipeline. The first thing the shader will do is create an output vertex that will be filled with the resulting data.

```
VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;
```

First, get the vertex to light direction and the attenuated intensity. These two lines could also be the start of many other lighting shaders.

```
   float3 VertexToLight = GetLightDirection
      (InVertex.Position);
   float3 Intensity     = GetLightIntensity
      (InVertex.Position);
```

This next line does all the Warn-specific processing. Here, the shader finds the cosine of the angle between the light direction and the vertex to light direction using the dot product. Then, that result is raised to the power of the Warn exponent parameter. Finally, the result is multiplied by the attenuated intensity. At this point, the value of the intensity variable is the amount of light energy that is incident upon the vertex, accounting for attenuation and the behavior of the Warn light. Nothing has been determined about the amount of light that is actually reflected by the surface material.

```
   Intensity = Intensity * pow(dot(VertexToLight,
      LightVector), WarnExponent);
```

This next line is where you would add the reflective behavior. Here, the shader accounts for Lambertian diffuse reflection with the dot product and multiplies that with the incoming intensity. If you were worried about a vertex color, you would multiply that here. If your material were more complex than a simple diffuse material, you could account for that here as well. Because I am more concerned with talking about lights than about surfaces, I am using a simple white diffuse material, meaning that I implicitly multiply the result by a color value of 1.0.

```
OutVertex.Diffuse  = Intensity * max(0.0f, dot
   (VertexToLight, InVertex.Normal));
```

Finally, the shader transforms the point using the transformation matrix. If you wanted to do something more complex with the position, such as shader based animation, you could do that here as well. If you do that, just remember that

animation would probably change the normal vector as well, and that new vector should be included in the calculations above.

```
   OutVertex.Position = mul(Transform, InVertex.Position);

      return OutVertex;
   }
```

This shader works well with both DirectX and cg, assuming the application instantiates and "feeds" it properly.

6.2.2 The Warn Application

I chose DirectX to demonstrate the implementation of the application that feeds the shader. The D3DX library provides a set of functions to instantiate the shader as well as a set of functions to do basic vector transformations for the object space light vectors. The same functionality is available for OpenGL users, but different users use different libraries. My aim here is to present the basic functionality with DirectX and D3DX. If you like, this functionality can be easily implemented with OpenGL and any number of libraries.

The following code snippet assumes that the shader and the models have been loaded properly. For simplicity, I am only showing the Render function. The entire application can be found on the companion CD in the \Code\Chapter 06 - Warn With VS directory.

```
   void CLightingApplication::Render()
   {
```

First, I set the light direction and transform it to an object space vector. The inverse world matrix is computed every time the world matrix is changed, such as when the object is rotated (see Appendix C). The code sends this transformed vector to the shader with SetVertexShaderConstant and it assumes there is only one object. If there were more, the light vector would need to be transformed to each object space.

```
      D3DXVECTOR4 LightVector(0.0f, 1.0f, 0.0f, 0.0f);
      D3DXVec4Transform(&LightVector, &LightVector,
         &m_InverseWorld);
      D3DXVec4Normalize(&LightVector, &LightVector);
      m_pD3DDevice->SetVertexShaderConstantF(4,
         (float *)&LightVector, 1);
```

The same goes for the light position. For a simple directional light, you don't need to set the light position. A directional light assumes the light is infinitely far from the object. You might notice that some samples and online examples pass the world space position and vector to the shader and rely on the shader to compute either the object space light vector or the world space normal. In these cases, the same computations are repeated for each and every vertex. This is much more efficient for rigid objects.

```
      D3DXVECTOR4 LightPosition(0.0f, 50.0f, 0.0f, 0.0f);
      D3DXVec4Transform(&LightPosition, &LightPosition,
         &m_InverseWorld);
      m_pD3DDevice->SetVertexShaderConstantF(5,
         (float *)&LightPosition, 1);
```

I also set the light color/brightness.

```
D3DXVECTOR4 LightColor(m_LightBrightness,
   m_LightBrightness, m_LightBrightness, 1.0f);
   m_pD3DDevice->SetVertexShaderConstantF(6,
      (float *)&LightColor, 1);
```

Next, I set the eye position, which is primarily used to set up the view matrix. If you wanted to create a specular lighting model, you would use the eye position as part of the lighting equation, but here it is simply passed to the D3DXMatrixLookAtLH function.

```
    D3DXVECTOR4 EyePosition(-30.0f, 15.0f, 0.0f, 0.0f);
    D3DXMatrixLookAtLH(&m_View, (D3DXVECTOR3 *)
       &EyePosition, &D3DXVECTOR3(0.0f, 0.0f, 0.0f),
       &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
```

The three matrices are concatenated and sent to the vertex shader. This final transformation matrix is used by the shader to transform the vertex position.

```
    m_ShaderMatrix = m_World * m_View * m_Projection;
    m_pD3DDevice->SetVertexShaderConstantF(0,
       (float *)&m_ShaderMatrix, 4);
```

This last constant is the most important in the context of creating a Warn spotlight. Here, I send the Warn exponent to the shader. If you wanted to use the shader constants more efficiently, you could probably pack the light brightness or other information into the three unused float values.

```
    D3DXVECTOR4 MiscLightParams(m_WarnExponent, 0.0f,
       0.0f, 0.0f);
    m_pD3DDevice->SetVertexShaderConstantF(8,
       (float *)&MiscLightParams, 1);
```

At this point, all of the relevant shader constants that you saw in the previous section have been sent to the shader. The remaining lines of this function set the shader, set the stream source for the vertices, and render the selected model.

```
    m_pD3DDevice->SetVertexDeclaration
       (m_pVertexDeclaration);
    m_pD3DDevice->SetVertexShader
       (m_VertexShaders[m_CurrentVertexShader]
       .pShader);
    m_pD3DDevice->SetStreamSource(0, m_Models
       [m_CurrentModel].pModelVertices, 0, m_Stride);
    m_pD3DDevice->SetIndices(m_Models[m_CurrentModel]
       .pModelIndices);
    m_pD3DDevice->DrawIndexedPrimitive(D3DPT_
       TRIANGLELIST, 0, 0, m_Models[m_CurrentModel]
       .NumVertices, 0, m_Models[m_CurrentModel]
       .NumFaces);
}
```

The same basic procedure applies to other APIs and shader paradigms. Compute the values for the constants, send the constants to the shader through the API-specific

mechanism, and render the final model in the API-specific way. In most cases, the shader itself will not change between different rendering engines.

6.2.3 The Results

Figure 6.5 shows the final rendered images of objects lit by several Warn lights with different exponents. As you can see, the spotlight falloff looks good on well-tessellated models.

When rendering with a vertex shader, the overall quality of the final render will be a function of the number of vertices that make up the mesh. The vertex shader will compute a lighting solution for each of those vertices and the results will be interpolated over the surface of the mesh. If the number of vertices is high enough, the results can be quite satisfactory. If, however, the number of vertices is too low, the results can be entirely incorrect.

Figure 6.6 shows a cube lit by the same light as was used for Figure 6.5. Here, the top of the cube should have a circular highlight that falls off closer to the edges. Instead, the entire top of the cube is evenly lit.

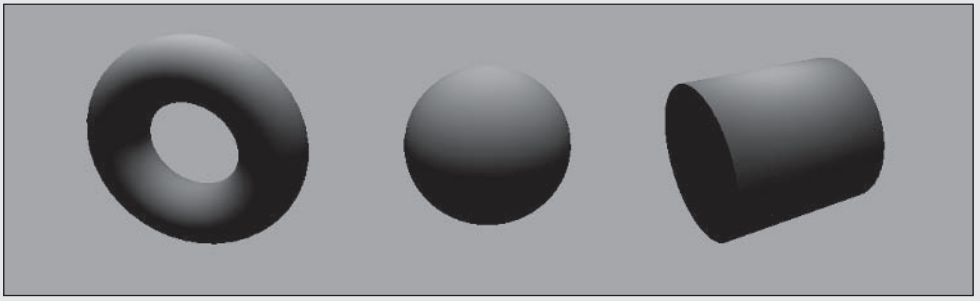


Figure 6.5. Objects illuminated by Warn spotlights

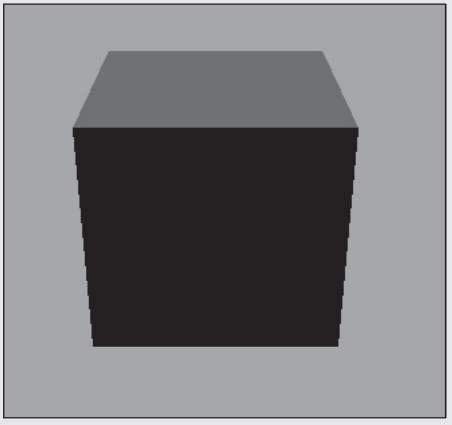


Figure 6.6. Incorrectly lit cube

The reason for this incorrect result lies in the fact that this cube is only defined by its corner vertices. In this case, not much light is incident upon the corners, making them quite dark. This dark result is interpolated across the top of the cube, creating an overall dark face. In a way, this is the exact opposite of what should be happening. Figure 6.7 shows the "real" situation graphically and compares it to the result of the vertex shader.

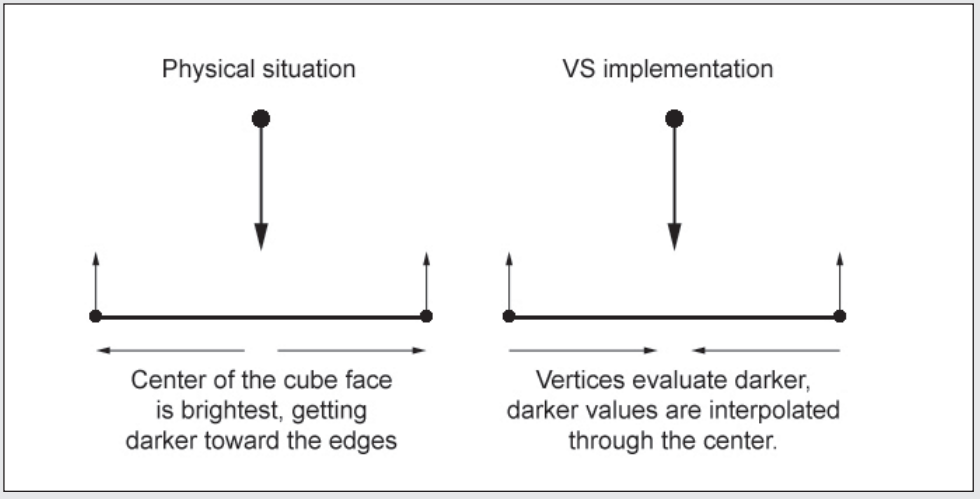


Figure 6.7. Lighting the cube top

There are a couple of different ways to fix this. The first approach might be to increase the number of vertices that make up each cube face. Geometrically, these vertices would be unnecessary overhead, but they would allow you to achieve a good result with a vertex shader. There are plenty of situations where this would be the right approach.

On the other hand, graphics cards with good pixel shader support allow you to compute the lighting solution on a per-pixel basis. This can be especially useful if the lighting solution will be used with other per-pixel effects such as bump mapping.

6.3 Per-Pixel Warn Lights

Pixel shaders allow you to compute a lighting solution for each pixel, but these final solutions are still dependent on vertex properties such as the positions and normal vectors. There are a couple of steps needed to create the final per-pixel solution. The first step is to compute a few values in a vertex shader that will be interpolated and sent to the pixel shader as input values. The next step is to actually compute the per-pixel solution using these values and more operations. In some cases, these steps will require additional input in the form of lookup values stored in textures. This is especially true with pixel shader versions earlier than PS2.0. In this section, I discuss the steps needed to compute a per-pixel solution with PS2.0 (or OpenGL equivalent). After that, I show how to approximate the same operations in lower end shaders with the help of a few lookup textures.

6.3.1 PS2.0 Lighting

Throughout this section, I will refer to a class of shaders defined by the DirectX 9 Pixel Shader 2.0 specification. However, the same class of hardware is capable of supporting some form of OpenGL equivalent. I am using the PS2.0 moniker as an indication of a level of capability. As you saw in the previous chapter, some shaders are usable across multiple APIs and shader paradigms.

I am starting with PS2.0 because these shaders are capable of computing the complete lighting solution (at least for our Warn light) with no additional lookup textures or other helpers. The instruction set and available instruction count is more than adequate for the fairly complex Warn light. Still, a few values need to come from the vertex shader so that the pixel shader knows where the light is and the relative orientation of the vertex normals. To this end, the following shader is very simple and computes only enough data to feed the shader with geometric information needed for the per-pixel lighting solution. This shader can be found on the companion CD as \Code\Shaders\Basic_PS20_Setup.hsl.

The input vertex structure remains the same, containing only enough information to demonstrate lighting.

```
struct VS_INPUT
{
   float4 Position    : POSITION;
   float3 Normal      : NORMAL;
};
```

The output structure has changed. Now, the output structure contains the raw position data and a normal vector. I named the untransformed position data “InterPos” because the pixel shader will act on the interpolated position data between the vertices. These vectors will be sent to the pixel shader as texture coordinates, but the shader will not use them for texture lookups. Instead, they will be used in the lighting calculations.

```
struct VS_OUTPUT
{
   float4 Position    : POSITION;
   float3 InterPos    : TEXCOORD0;
   float3 Normal      : TEXCOORD1;
};
```

The only calculations performed by the vertex shader are those needed for geometric processing, so only the transformation matrix is needed as a parameter.

```
matrix Transform      : register(c0);
```

The shader functionality is very simple. It transforms the vertex position and then passes the vertex position and normal data to the pixel shader for further processing. The pixel shader does the real work.

```
VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;

   OutVertex.Position = mul(Transform, InVertex.Position);
   OutVertex.InterPos = InVertex.Position.xyz;
   OutVertex.Normal   = InVertex.Normal;

   return OutVertex;
}
```

One could argue that perhaps the vertex shader should be doing more precalculations of factors that are more tolerant of the linear interpolation between vertices. That might be true in some cases. In this section, however, I want to show the capabilities of more advanced pixel shaders. The optimal distribution of work through the pipeline will ultimately be dependent on many factors that aren't present in these simple samples.

In this example, the output vertex contains the same data as the input vertex. This allows you to write the pixel shader using essentially the same variables as the vertex shader seen earlier. There is, however, one crucial difference. Any per-vertex values coming out of the vertex shader are interpolated across the related pixels. This creates some side effects when dealing with vector values.

For example, the vertex shader above passes the normal vector to the pixel shader as a set of texture coordinates. These coordinates are interpolated over the related pixels, but each component is interpolated separately. This means that the direction of the interpolated normal vectors is correct, but they are unlikely to be normalized. Figure 6.8 shows this graphically. The middle vector has the correct direction, but its length is much less than 1.0.

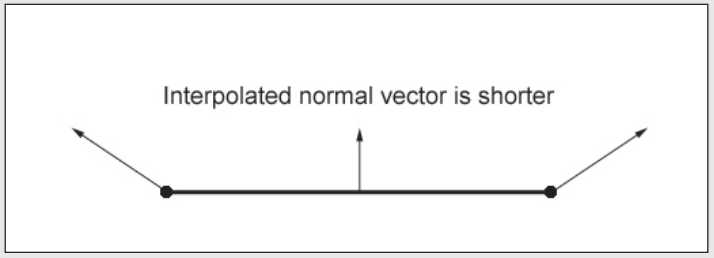


Figure 6.8. Interpolated normal vector

Because it's not normalized, the per-pixel normal vector would not yield the correct results if used with an operation such as the dot product. Therefore, in most cases you must normalize the vector before you can do anything useful with it.

With the first generation of pixel shaders, normalization with pixel shader instructions was often too heavy; it would be far too costly in terms of instruction count. An alternative solution was to use a normalization cube map. This would yield a normalized vector, but it could be costly in terms of texture count or memory.

With more advanced pixel shaders, you can comfortably normalize the vector with arithmetic instructions. This can be more costly in terms of instruction count, but it also frees a texture for more interesting uses. It might also be faster than the texture lookup, and will yield better precision. For a more rigorous discussion of the pros and cons of each approach, see NVIDIA's whitepaper on the subject [1].

The shader below uses the same variables you saw earlier, along with some normalization, to reproduce the same Warn effect seen in the earlier vertex shader. This pixel shader can be found on the CD as \Code\Shaders\Diffuse_Warn_PS20.hsl.

First, I define an output structure. This pixel shader is only concerned with the final color of the output pixel. Also, I'm not interested in alpha blending, so I am somewhat reckless with the final value of the alpha channel.

```
struct PS_OUTPUT
{
   float4 Color : COLOR0;
};
```

The vertex structure is the same as you saw in the setup vertex shader.

```
struct VS_OUTPUT
{
   float4 Position    : POSITION;
   float3 InterPos    : TEXCOORD0;
   float3 Normal      : TEXCOORD1;
};
```

Now, the same parameters that were sent to the vertex shader must be sent to the pixel shader. I am not explicitly showing the application code in this instance because it is very similar to the application code shown earlier. The only real difference is that these constants must be set with SetPixelShaderConstantF instead of SetVertexShaderConstantF. Otherwise, all values and semantics for these parameters remain the same.

```
float3  LightVector   : register(c0);
float   WarnExponent  : register(c1);
float   LightColor    : register(c3);
float3  LightPosition : register(c5);
```

Here is the main function that is executed for each pixel:

```
PS_OUTPUT main(const VS_OUTPUT In)
{
   PS_OUTPUT OutPixel;
```

PixelToLight is the vector from the pixel to the light. Both the direction and the magnitude of this vector will be used to compute the irradiance at that pixel.

```
    float3 PixelToLight = LightPosition - In.InterPos;
```

Next, you need to normalize the input vectors for the reasons discussed above. These lines could be replaced by a texture lookup into a normalization cube map. Depending on the target hardware, the complexity of the overall shader, and other factors, either method could be more optimal, but this approach isn't limited by factors such as the resolution of the texture or the precision of the pixel values.

```
    float3 NewNormal = normalize(In.Normal);
    float3 NewDir    = normalize(PixelToLight);
```

This next line computes the per-pixel incoming intensity factor based on the shape of the spotlight. This value doesn't account for attenuation or diffuse reflection.

```
    float WarnValue = pow(dot(NewDir, LightVector),
       WarnExponent);
```

Finally, all of the pieces are brought together. The following line accounts for the shape of the spotlight, the light brightness and attenuation, and diffuse reflection. Again, the color of the object is assumed to be 1.0 (full white).

```
    OutPixel.Color = WarnValue * dot(NewNormal, NewDir)
       * LightColor / pow(length(PixelToLight), 2.0);

    return OutPixel;
}
```

6.3.2 The Results

The end result can be seen in Figure 6.9. As you can see, the per-pixel solution is not dramatically different from the per-vertex solution when applied to highly tessellated models such as the sphere. However, the difference is striking when the two are applied to the cube.

6.3.3 Lookup Textures

Earlier in this chapter, I alluded to previous versions of pixel shaders and the need to use lookup textures such as a normalization cube map. Readers might also be wary of instructions such as "pow," which is used twice in the PS2.0 pixel shader. In earlier pixel shader versions, such operations often required the use of a lookup texture. For instance, you might have a term like $\sin(x)$. Instead of solving that arithmetically (with a Taylor series, for instance), you could have a 1D texture that stored the value of $\sin(x)$ for every value of x. Once you have that texture, solving for that term would involve one texture lookup instead of several instructions.

Such an approach is easily extensible to a 2D function. The Warn term is one example, where the solution is dependent on a dot product value with a range from 0 to 1 and an exponent. Figure 6.10 shows a lookup texture that stores the Warn results for different values of the dot product and different exponent values. Each horizontal row stores the results for increasing values of the dot product. The Warn exponent increases as you move down each column.

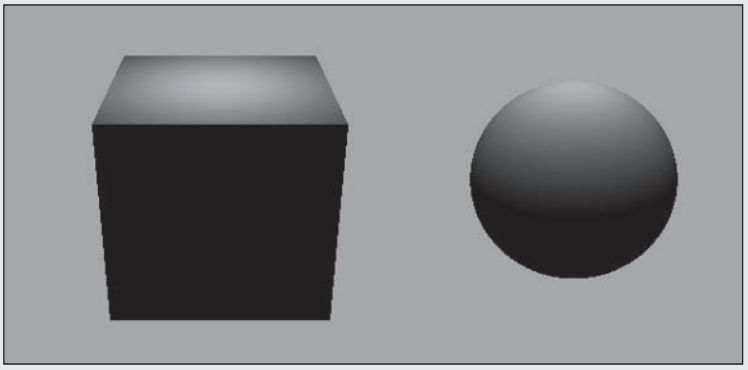


Figure 6.9. Per-pixel lighting

Figure 6.10. A Warn lookup texture

The following is a snippet of shader code that uses this texture to find a final value of the Warn light. Notice that the shader only uses the red channel of the texture. Other functions could be encoded into the other three channels.

```
vector  LightVector  : register(c0);
float   WarnExponent : register(c1);
sampler PowerTex     : register(s0);
...
float2 WarnLookUp;
WarnLookUp.x = dot(LightToVertex, LightVector);
WarnLookUp.y = WarnExponent;

float Value = tex2D(PowerTex, WarnLookUp).x
```

This approach can be convenient because this one texture can be used for several Warn lights with different exponent values. However, the downside is that you are limited by the precision and range of the texture. If the texture is of a low resolution and you use a high value for the exponent, you could create a situation where there is almost no gradient between unlit and fully lit objects. If you look at the bottom rows of Figure 6.10 you can see there are not many individual values available for light intensity.

Going forward, you can rely more and more on real math instructions to find complex solutions. However, it is still worthwhile to keep lookup textures in mind. For instance, imagine you want to simulate the highlight rings visible with some flashlights. You could probably derive a mathematical solution that simulates this, but it might be just as easy to create a lookup texture that incorporates brighter bands at

certain angles. It might be worthwhile to solve the base lighting with full precision instructions and then use a relatively low resolution texture to add the highlight bands.

In any case, the point is that going forward, lookup textures will be less necessary for approximation of basic mathematical functions and more interesting as a means of encoding other special features and effects.

Conclusion

Typical games deal with directional, point, and spotlights. In this chapter, I've only shown how to implement one light, and it wasn't even one of the most common. There are a couple of reasons for this. First, there are a large number of great online and print resources (some of which are listed below) for common shaders. Secondly, the Warn light was good for highlighting several points that apply to several light types. Here are a couple of things to remember when writing your own shaders or adapting shaders from other resources:

- Deal with object space light vectors whenever you can. It will save you a few operations for every vertex.
- Higher level shading languages greatly simplify shader creation. Often, the final shader will look very similar to the pseudocode you might find in a book or other resource. This is a dramatic change from cryptic assembly shaders.
- In many cases, complex lights on simple objects might necessitate per-pixel lighting.
- With more advanced pixel shaders, per-pixel techniques are nearly identical to per-vertex techniques in terms of approach and final code.
- When using per-pixel techniques, remember to account for the way that vector values are interpolated across polygons and normalize when you need to.
- Lookup textures, which were commonly used to approximate mathematical functions in early shaders, might not be as necessary in new shader versions. However, they are still useful for other effects.

These points apply across light types, graphics APIs, and shader paradigms. The Warn example highlights many ideas that you can apply to almost any other type of light.

References

[1] "Normalization Heuristics," http://developer.nvidia.com/object/normalization_heuristics.html.

Other Resources

Use the following references to get more information about other types of lights and their associated shaders.

Sander, P., "A Fixed Function Shader in HLSL," http://www2.ati.com/misc/samples/dx9/FixedFuncShader.pdf

Fernando, R. and M. Kilgard, "The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics," http://developer.nvidia.com/object/cg_tutorial_home.html

Engel, W., "Advanced Shader Programming: Diffuse & Specular Lighting with Pixel Shaders," http://www.gamedev.net/reference/articles/article1849.asp

Dempski, K., *Real-Time Rendering Tricks and Techniques in DirectX*, Indianapolis: Premier Press, 2002.

Chapter 7

Implementing BRDFs in Shaders

Introduction

The previous chapter focused on implementing lights, and therefore used the simplest material possible for the lit objects. This chapter takes the opposite approach. I use the simplest light possible (a single directional light) and show you how to implement the reflection models that you saw in Chapter 5.

Some of the concepts shown here will be familiar from the last chapter, such as transforming a light vector to object space, but now you will see how those concepts are used with reflection models. Also, as with the last chapter, I cover both per-vertex and per-pixel approaches and discuss the differences. I focus on implementation, so you might want to review the equations shown in Chapter 5 before I cover the following material.

- Diffuse materials
- Phong and Blinn-Phong specular materials
- Oren-Nayar materials
- Minnaert materials
- Ward materials (isotropic and anisotropic)
- Schlick materials
- Cook-Torrance materials

NOTE:

The vertex shader code has been built to compile vs1.1 vertex shaders so that readers with older hardware can compile and use the shaders. Readers with newer hardware might want to change the application code to target VS_2_0 or above. The shaders themselves do not need to be changed. All pixel shaders are aimed at PS2.0 or better.

7.1 Basic Setup and Diffuse Materials

The previous chapter implicitly covered simple diffuse materials, but I want to explicitly cover them here to establish a base technique against which all of the subsequent techniques will be compared. Also, this is a convenient time to talk about the application setup before getting into the more complex reflection models.

7.1.1 Basic Application Code

The code for this chapter can be found on the CD in the following directories: \Code\Chapter 07 - BRDFs with VS\ for the vertex shaders and \Code\Chapter 07 - BRDFs with PS\ for the pixel shaders. The shaders themselves can be found in the \Code\Shaders\ directory.

This chapter is based on the same code you saw in the previous chapter. The only real change is in the Render function, where you need to send a few extra parameters to the shaders. Not every shader will use all of the supplied parameters. Also, in this chapter, I will be hard-coding shader-specific values into the individual shaders rather than supplying them through the application as constants. This probably isn't the best approach if you are building your own application, but it allows me to present each shader as a self-contained unit without referring back to different parts of the application code. If you like, you can modify the application to send values to the shaders.

I will begin with the Render function for the vertex shader application. This function assumes that a set of shaders has already been created when the application initialized. The purpose of the Render function is simply to pass a set of constants to the shaders and render the current model.

```
void CLightingApplication::Render()
{
```

These first two lines provide a bit of feedback to you so that you know which shader is currently in use. The "V" key on the keyboard can be used to cycle through the different shaders.

```
    sprintf(m_TextBuffer, "\n%s", m_VertexShaders
       [m_CurrentVertexShader].pName);
    m_pFont->DrawText(NULL, m_TextBuffer, -1,
       &m_TextRect, DT_LEFT, 0xffffffff);
```

The first constant passed to the shader is the light vector. I don't actually change the vector, so this code could be optimized slightly by pulling the first line out of the render loop. Once the vector is set, it is transformed to object space and normalized. Finally, it is sent to the shader. Different shaders might use this vector in different ways.

```
    D3DXVECTOR4 LightVector(0.0f, 1.0f, 0.0f, 0.0f);
    D3DXVec4Transform(&LightVector, &LightVector,
       &m_InverseWorld);
    D3DXVec4Normalize(&LightVector, &LightVector);
    m_pD3DDevice->SetVertexShaderConstantF(4,
       (float *)&LightVector, 1);
```

Next, I send the light position to the shader. It too is transformed to object space, but it wouldn't make sense to normalize the position. Although it is sent to all shaders, some shaders might not use this constant. I don't set a light brightness because all shaders will assume a simple white directional light with no attenuation. In the previous chapter, the material properties were implicitly set to a value of 1.0. Here, the light color will be implicitly set and the material will change.

```
D3DXVECTOR4 LightPosition(0.0f, 50.0f, 0.0f, 0.0f);
D3DXVec4Transform(&LightPosition, &LightPosition,
   &m_InverseWorld);
m_pD3DDevice->SetVertexShaderConstantF(6,
   (float *)&LightPosition, 1);
```

Now that all the light values are set, I will set the eye position and use the position to create a view matrix.

```
D3DXVECTOR4 EyePosition(-100.0f, 50.0f, 0.0f, 0.0f);
D3DXMatrixLookAtLH(&m_View, (D3DXVECTOR3*)
   &EyePosition, &D3DXVECTOR3(0.0f, 0.0f, 0.0f),
   &D3DXVECTOR3(0.0f, 1.0f, 0.0f));
```

Once the view matrix is set, I transform the eye position to object space and pass it to the shader. View-independent shaders will not use this value, but others will.

```
D3DXVec4Transform(&EyePosition, &EyePosition,
   &m_InverseWorld);
m_pD3DDevice->SetVertexShaderConstantF(5,
   (float *)&EyePosition, 1);
```

As usual, I send the concatenated transformation matrix to the shader. This matrix has no part in the BRDF calculations.

```
m_ShaderMatrix = m_World * m_View * m_Projection;
m_pD3DDevice->SetVertexShaderConstantF(0,
   (float *)&m_ShaderMatrix, 4);
```

The remaining lines perform the actual rendering.

```
   m_pD3DDevice->SetVertexDeclaration
      (m_pVertexDeclaration);
   m_pD3DDevice-> SetVertexShader(m_VertexShaders
      [m_CurrentVertexShader].pShader);
   m_pD3DDevice->SetStreamSource(0, m_Models
      [m_CurrentModel].pModelVertices, 0, m_Stride);
   m_pD3DDevice->SetIndices(m_Models
      [m_CurrentModel].pModelIndices);
   m_pD3DDevice->DrawIndexedPrimitive
      (D3DPT_TRIANGLELIST, 0, 0, m_Models
      [m_CurrentModel].NumVertices, 0, m_Models
      [m_CurrentModel].NumFaces);
}
```

In the interest of brevity, I will not outline the complete code for the pixel shader application. In that instance, the Render function is nearly identical except for the fact that all of the constants except for the transformation matrix are passed to the pixel shader instead of the vertex shader. Also, the pixel shader code features the same setup vertex shader seen in the last chapter.

7.1.2 Basic Diffuse Material

Simple diffuse materials are ubiquitous in computer graphics. If you've rendered anything at all, you have probably rendered an object with a diffuse material lit by a directional light. I am going to repeat that here, if only to create a basis for comparison with the other techniques in this chapter.

The vertex shader implementation is very simple. It can be found on the CD as \Code\Shaders\Diffuse_VS.hsl. The inputs and outputs are the same as those you saw in the previous chapter.

```
struct VS_INPUT
{
   float4 Position    : POSITION;
   float3 Normal      : NORMAL;
};

struct VS_OUTPUT
{
   float4 Position    : POSITION;
   float3 Diffuse     : COLOR;
};
```

Only two constants are needed: the transformation matrix and the object space light vector.

```
matrix Transform      : register(c0);
vector LightVector    : register(c4);

VS_OUTPUT main(const VS_INPUT InVertex)
{
```

The first two lines handle the transformation and do not affect the material properties of the object. These lines will be the same in every shader presented in this chapter.

```
   VS_OUTPUT OutVertex;
   OutVertex.Position = mul(Transform, InVertex.Position);
```

Here I set the light color and the material color. This is the only shader in which I will explicitly set the light color. If you want to use a light color/brightness other than pure white, you should either set a value in the shader or pass it as a constant.

```
      float4 LightColor   = {1.0f, 1.0f, 1.0f, 1.0f};
      float4 DiffuseColor = {1.0f, 1.0f, 1.0f, 1.0f};
```

Finally, this one line does the actual work. Here, the final value is comprised of the incoming light brightness, modulated by the cosine term (dot product), modulated by the material color.

```
      OutVertex.Diffuse  = LightColor * DiffuseColor *
         max(0.0f, dot(LightVector, InVertex.Normal));

      return OutVertex;
   }
```

The equivalent pixel shader is very similar. The inputs and outputs are the same as those you saw in the previous chapter. As before, the vertex shader passes the vertex normal to the pixel shader. The interpolated position is not used in this shader.

```
struct PS_OUTPUT
{
   float4 Color       : COLOR0;
};

struct VS_OUTPUT
{
   float4 Position    : POSITION;
   float3 InterPos    : TEXCOORD0;
   float3 Normal      : TEXCOORD1;
};

float3  LightVector   : register(c4);

PS_OUTPUT main(const VS_OUTPUT In)
{
   PS_OUTPUT OutPixel;

   float4 DiffuseColor = {1.0f, 1.0f, 1.0f};
```

At a high level, the pixel shader is nearly identical to the vertex shader. The two main differences are that I have removed the light color term and normalized the interpolated normal vector. The light color was dropped in favor of an implicit multiplication by 1.0, and the vector must be normalized for the reasons mentioned in the previous chapter.

```
   OutPixel.Color  = DiffuseColor * max(0.0f, dot
      (LightVector, normalize(In.Normal)));

   return OutPixel;
}
```

The results of these two shaders are shown in Figure 7.1. As you can see, the difference between the two techniques is very slight, even for a low-resolution model. The difference is nearly nonexistent for denser meshes. When you experiment with the application, note that I have included both high- and low-resolution meshes for each of the three basic shapes. This will be useful in examining the effect of mesh density on the quality of the final results.

Keep in mind that using a pixel shader for a simple diffuse material is, in most cases, dramatic overkill. I only include these shaders so that I can build upon them with the rest of the shaders.

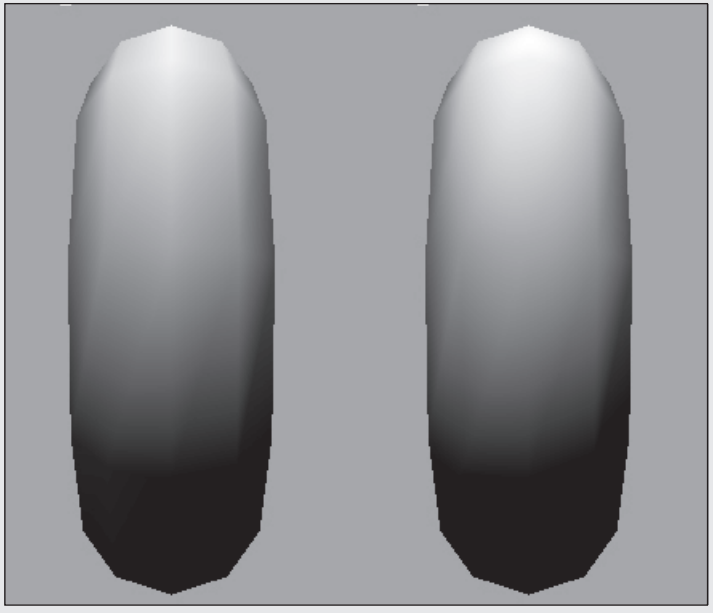


Figure 7.1. Simple diffuse materials shown per-vertex (left) and per-pixel (right)

7.2 Specular Materials

Specular effects reproduce the shininess of an object that is seen when the eye direction approaches the mirror direction of the light reflected off the surface. In section 7.1, you saw that the application passes the object space eye position to the shader, but the preceding shader did not use it. Now, the Phong and Blinn-Phong shaders will use the eye position to generate specular effects.

7.2.1 The Phong Shaders

The Phong shader will compute the specular effects by computing the true relationship between the eye-to-vertex vector and the light-to-vertex reflection vector. The shader is a basic implementation of the equations seen in Chapter 5. It can be found on the CD as \Code\Shaders\Phong_VS.hsl.

The first three lines define the inputs from the application. Unlike the last shader, the Phong shader needs to use the eye position.

```
matrix Transform      : register(c0);
vector LightVector    : register(c4);
vector EyePosition    : register(c5);

VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;
```

The equations are dependent on the vector between the vertex and the eye, so that the vector can be compared to the direction of the light reflecting from the vertex. If you know that the vector varies very little over the entire surface of the object, you can optimize this by finding an object-to-eye direction in the application. This will allow you to compute the vector once and reuse it for all the vertices. This might be appropriate for very small or very faraway objects, but the approximation will begin to be apparent with larger objects.

```
   vector VertexToEye = normalize
      (EyePosition - InVertex.Position);
```

As before, I am defining a set of constants within the shader instead of in the application in order to simplify the application code and to make the shader as clear as possible. Note that the w component of the specular color contains the specular power value.

```
   float4 DiffuseColor  = {1.0f, 1.0f, 1.0f, 1.0f};
   float4 SpecularColor = {1.0f, 1.0f, 1.0f, 7.0f};
```

The following line of code computes the reflection vector as discussed in Chapter 5. At this point, the shader is still computing values that will be used as inputs to the actual specular calculation.

```
   float3 Reflection = normalize(2.0 * InVertex.Normal
      * dot(InVertex.Normal.xyz, LightVector.xyz) –
      LightVector);
```

The diffuse component is exactly the same as in the diffuse shader.

```
float3 Diffuse = DiffuseColor * max(0.0f, dot
   (LightVector, InVertex.Normal));
```

The following line computes the specular component as described in Chapter 5. Here, the specular power in the w component of the specular color controls the specular highlight.

```
float3 Specular = SpecularColor *
   (pow(max(dot(Reflection.xyz, VertexToEye),
   0.0), SpecularColor.w));
```

The final color is the sum of the diffuse and specular components.

```
    OutVertex.Diffuse =  Diffuse + Specular;

    OutVertex.Position = mul(Transform, InVertex.Position);

    return OutVertex;
}
```

Figure 7.2 shows a final rendering based on the Phong vertex shader. It also shows the image decomposed into separate diffuse and specular components.

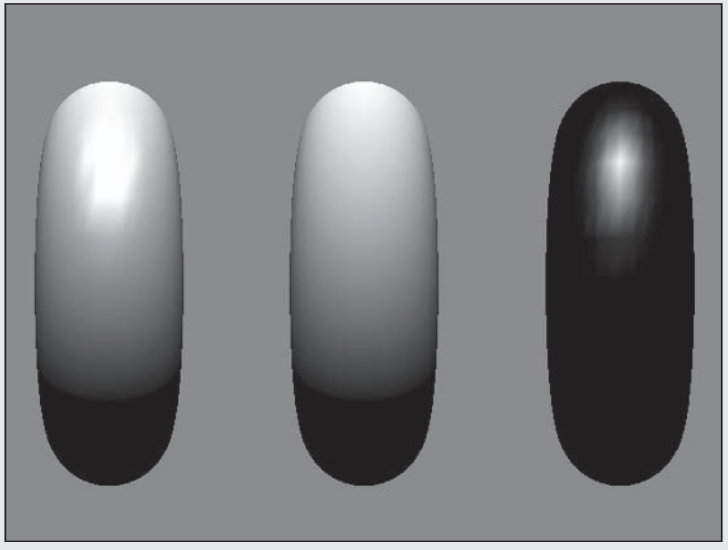


Figure 7.2. Object rendered with diffuse and specular components

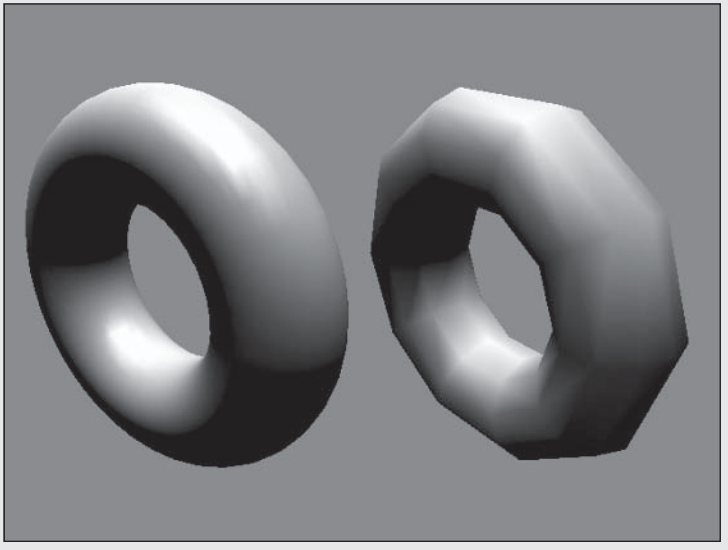


Figure 7.3. Phong with different mesh resolutions

Figure 7.3 shows the difference between a high-resolution mesh and a low-resolution mesh rendered with the Phong shader. Here, the resolution of the mesh can have a fairly dramatic effect on the shape of the resulting highlight.

The pixel shader version is again very similar to the vertex shader. It can be found on the CD as \Code\Shaders\Phong_PS.hsl. Here, the interpolated pixel position serves as the basis for the reflection vector and the eye-to-pixel vector.

```
struct PS_OUTPUT
{
   float4 Color : COLOR0;
};
```

The diffuse pixel shader did not make use of the interpolated position. Now that data becomes an integral part of the specular calculations.

```
struct VS_OUTPUT
{
   float4 Position    : POSITION;
   float3 InterPos    : TEXCOORD0;
   float3 Normal      : TEXCOORD1;
};
```

The light vector and eye position are passed from the application to the shader. Remember, the light color is assumed to be pure white.

```
float3 LightVector    : register(c4);
vector EyePosition    : register(c5);

PS_OUTPUT main(const VS_OUTPUT In)
{
   PS_OUTPUT OutPixel;
```

Aside from some minor changes to variable types, the following operations are exactly the same as seen in the vertex shader. The only major difference here is that the calculations are based on a pixel-to-eye vector that is itself based on the interpolated object space position of each pixel. Remember to renormalize the incoming interpolated normal vector.

```
   float4 DiffuseColor  = {1.0f, 1.0f, 1.0f, 1.0f};
   float4 SpecularColor = {1.0f, 1.0f, 1.0f, 7.0f};

   float3 PixelToEye = normalize(EyePosition -
      In.InterPos);

   float3 NewNormal  = normalize(In.Normal);

   float3 Reflection = normalize(2.0 * NewNormal *
      dot(NewNormal, LightVector) - LightVector);

   float4 Diffuse = DiffuseColor * max(0.0f, dot
      (LightVector, NewNormal));

   float4 Specular = SpecularColor * (pow(max(dot
      (Reflection.xyz, PixelToEye), 0.0),
      SpecularColor.w));

   OutPixel.Color  = Diffuse + Specular;

   return OutPixel;
}
```

Figures 7.4 and 7.5 show the same view as seen in Figures 7.2 and 7.3, only with a pixel shader. These figures demonstrate two main points. The first is that the per-pixel specular highlight is much smoother than the per-vertex highlight, even for low-resolution meshes. However, the second point is that per-pixel effects are not necessarily a magic solution that makes everything perfect. The interpolated normals and positions are still dependent on the vertex data.

There is, in some cases, a way to make the per-pixel solution better, even with low-resolution meshes. One can map the object with a texture that contains higher resolution normal and/or position data and use those values in the calculations. This would improve the quality of the object's surface, but the silhouette of the object would still be dependent on the vertex resolution.

Creating these textures can be nontrivial, but there are tools such as ATI's NormalMapper [1] that can be very helpful.

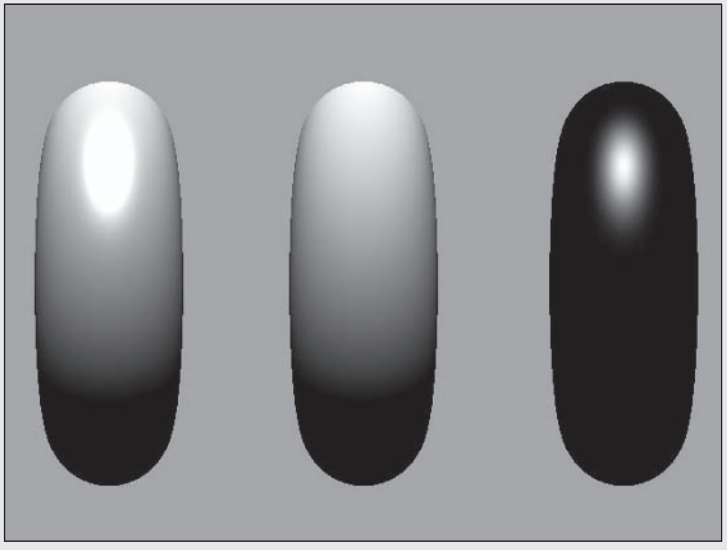


Figure 7.4. Object rendered with diffuse and specular components

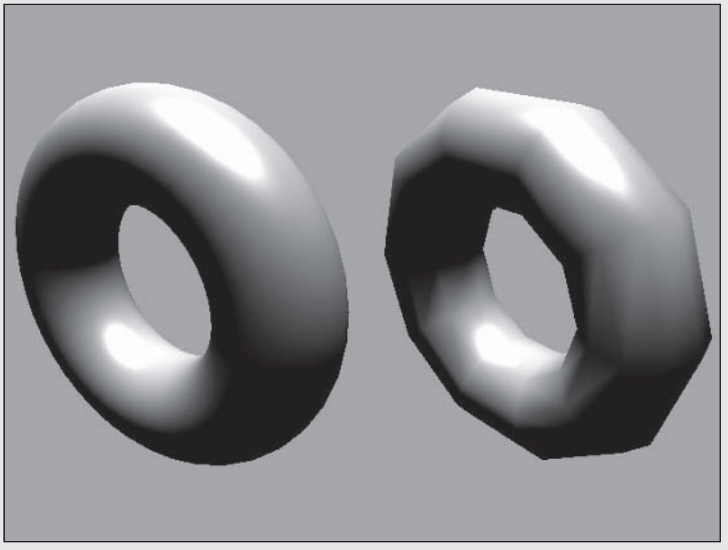


Figure 7.5. Per-vertex Phong shader with different mesh resolutions

7.2.2 The Blinn-Phong Shaders

As discussed in Chapter 5, because the reflection calculations can be cumbersome, Blinn introduced the half vector approximation to streamline the calculations. The result is a solution that is visually similar but requires fewer shader instructions. The Blinn-Phong vertex shader can be found on the CD as \Code\Shaders\Blinn_Phong_VS.hsl. It is very similar to the previous vertex shader.

```
matrix Transform      : register(c0);
vector LightVector    : register(c4);
vector EyePosition    : register(c5);

VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;

   vector VertexToEye = normalize(EyePosition -
      InVertex.Position);
```

The biggest difference is that this technique is based on the half vector rather than a complete reflection calculation. The following line of code computes the normalized half vector.

```
   float3 HalfVector = normalize(LightVector.xyz +
      VertexToEye.xyz);

   float4 DiffuseColor  = {1.0f, 1.0f, 1.0f, 1.0f};
   float4 SpecularColor = {1.0f, 1.0f, 1.0f, 7.0f};

   float3 Diffuse = DiffuseColor * max(0.0f,
      dot(LightVector, InVertex.Normal));
```

Find N dot H and raise this value to the specular power to find the final specular value. Once you have that, the final result is the sum of the diffuse and specular components.

```
   float NdotH = max(0.0f, dot(InVertex.Normal,
      HalfVector));
   float3 Specular = SpecularColor * (pow(NdotH,
      SpecularColor.w));

   OutVertex.Diffuse =  Diffuse + Specular;

   OutVertex.Position = mul(Transform,
      InVertex.Position);

   return OutVertex;
}
```

Figure 7.6 shows the result of this vertex shader (left) and compares it to the Phong result with the same input values (right). As you can see, the approximation changes the size of the highlight but not the character of the overall effect.

In the end, the Blinn-Phong version uses 20 instructions versus 22 instructions for the Phong solution. These shaders are not optimized, so it's possible that the gap could be widened, but there isn't a huge difference between the techniques. However, one could argue that even small instruction savings could be important for vertex shaders that are executed millions of times per frame.

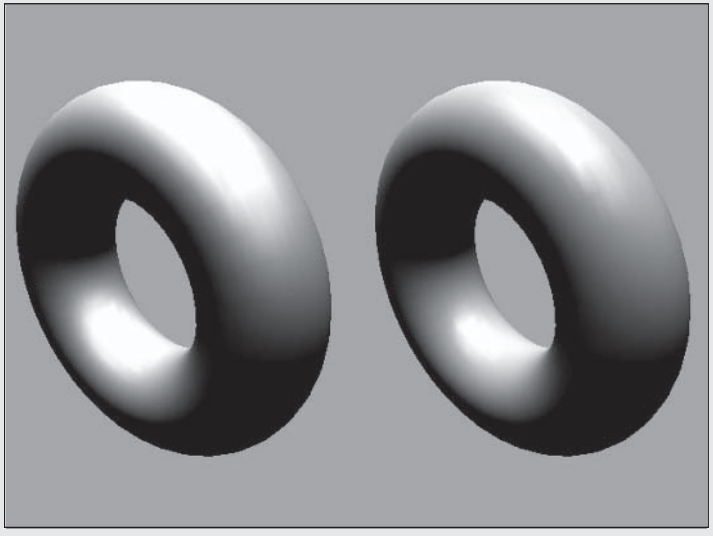


Figure 7.6. Blinn-Phong (left) and Phong (right) renderings based on the same input data

NOTE:

For a more in-depth discussion of instruction counts and how they might change, see section 7.6 on Schlick shaders. Section 7.6 rounds out the discussion on Phong variants and also discusses how comparing shader counts can sometimes be a less than simple task.

The Blinn-Phong pixel shader is very similar and can be found on the CD as \Code\Shaders\Blinn_Phong_PS.hsl.

```
float3 LightVector    : register(c4);
vector EyePosition    : register(c5);

PS_OUTPUT main(const VS_OUTPUT In)
{
   PS_OUTPUT OutPixel;

   float4 DiffuseColor = {1.0f, 1.0f, 1.0f, 1.0f};
   float4 SpecularColor = {1.0f, 1.0f, 1.0f, 7.0f};
```

Other than the fact that these calculations involve a pixel-to-eye vector in place of a vertex-to-eye vector, the operations are exactly the same.

```
   float3 PixelToEye = normalize(EyePosition -
      In.InterPos);
   float3 HalfVector = normalize(LightVector.xyz +
      PixelToEye.xyz);

   float3 NewNormal = normalize(In.Normal);
   float4 Diffuse = DiffuseColor * max(0.0f,
      dot(LightVector, NewNormal));

   float NdotH = max(0.0f, dot(NewNormal,
      HalfVector));
   float4 Specular = SpecularColor * (pow(NdotH,
      SpecularColor.w));

   OutPixel.Color  = Diffuse + Specular;

   return OutPixel;
}
```

Figure 7.7 shows the output of this pixel shader. As you can see, the results are similar to the Phong pixel shader, with changes to the size of the specular highlight. For pixel shaders, the number of instructions comes to 19 for Blinn-Phong versus 21 for Phong. Given the frequency of pixel shader executions, the difference might be noticeable.

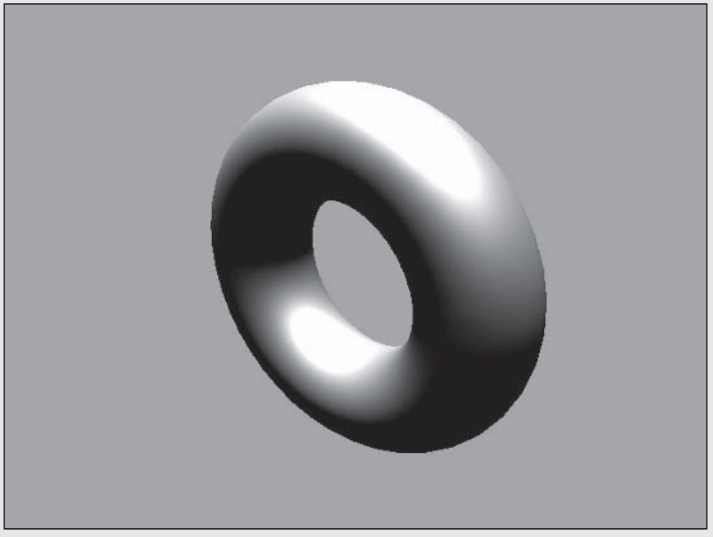


Figure 7.7. Blinn-Phong per-pixel effect

7.3 Oren-Nayar Materials

Oren-Nayar materials are diffuse, but they are also view-dependent, making them similar to specular materials. The following vertex shader implements the equations shown in Chapter 5. It can be found on the CD as \Code\Shaders\Oren-Nayar_VS.hsl. One of the important things to note about this shader is that it makes much greater use of the new HLSL functions such as acos() and tan().

```
matrix Transform      : register(c0);
vector LightVector    : register(c4);
vector EyePosition    : register(c5);

VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;
```

For Oren-Nayar, the vertex-to-eye vector will be used to determine the effects of retroreflection.

```
    vector VertexToEye = normalize(EyePosition -
       InVertex.Position);

    float VdotN = dot(VertexToEye, InVertex.Normal);
    float LdotN = dot(LightVector, InVertex.Normal);
```

L dot N is useful later in the function, but I also want the clamped value for an irradiance value.

```
    float Irradiance = max(0.0f, LdotN);
```

The following values are explained in greater depth in Chapter 5.

```
    float AngleViewNormal  = acos(VdotN);
    float AngleLightNormal = acos(LdotN);

    float AngleDifference = max (0.0f,
       dot(normalize(VertexToEye - InVertex.Normal *
       VdotN), normalize(LightVector -
       InVertex.Normal * LdotN)));

    float Alpha = max(AngleViewNormal,
       AngleLightNormal);
    float Beta  = min(AngleViewNormal,
       AngleLightNormal);

    float4 DiffuseColor  = {1.0f, 1.0f, 1.0f, 1.0f};
```

Since the equation is really a function of roughness squared, I have set that value directly. It would be more optimal to set the roughness values in the application and pass them to the shader, thereby saving a few redundant instructions. In this case, I have placed them in the shader so that it is more self-contained.

```
    float RoughnessSquared = 0.5f;
    float A = 1.0f - (0.5f * RoughnessSquared) /
       (RoughnessSquared + 0.33f);
    float B = (0.45f * RoughnessSquared) /
       (RoughnessSquared + 0.09f);
```

The next line computes the final Oren-Nayar value as discussed in Chapter 5.

```
    OutVertex.Diffuse  = DiffuseColor *
       (A + B * AngleDifference * sin(Alpha) *
       tan(Beta)) * Irradiance;
    OutVertex.Position = mul(Transform,
       InVertex.Position);

    return OutVertex;
}
```

Figure 7.8 shows the final output. Notice that the result is a dustier or softer look than the images shown earlier. Experiment with light direction in the code to see more or less dramatic effects. The effect is dependent on the relationship between the view vector and the light direction.

The pixel shader is a different story. Up to this point, the pixel shaders have all been slightly reworked vertex shaders. However, the Oren-Nayar shader uses too many arithmetic instructions to be a PS2.0 shader. The reworked vertex shader amounts to 70 arithmetic pixel shader instructions compared to a maximum number of 64. It's possible that some optimizations and a little cleverness could bring the

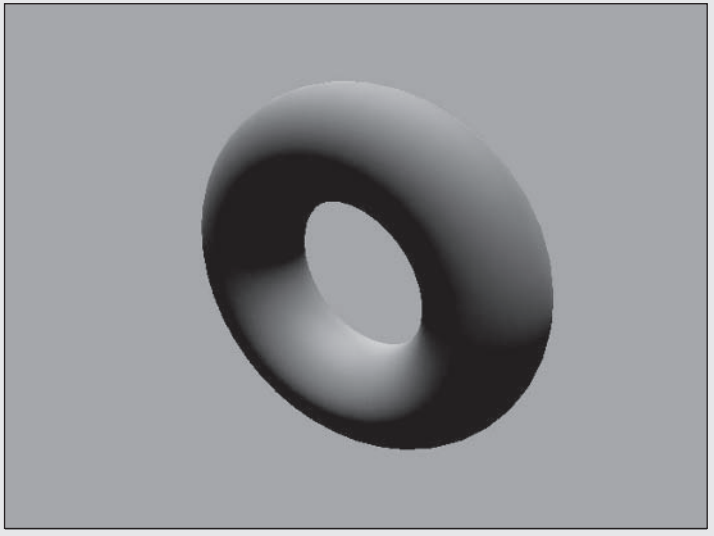


Figure 7.8. Per-vertex Oren-Nayar

count down to 64, but this seems like an excellent time to talk about the texture instructions (32 for PS2.0).

A large number of the instructions come from the alpha and beta values of the Oren-Nayar equation, so a texture operation that can quickly find these values based on some set of input values would be very useful. A close look at Chapter 5 and the vertex shader code can lead one to the following equation.

$$I_0 = D * (N \bullet L) * \left(A + B * f\left((N \bullet L),(N \bullet V)\right) * MAX\left(0, \cos(C)\right)\right) * I_1$$

$$A = 1 - 0.5\frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45\frac{\sigma^2}{\sigma^2 + 0.09}$$

C = the azimuth angle between the light vector and the view vector

This means that the instructions that use alpha and beta can be boiled down to one function with two input values and one output value. This seems like a perfect use for a 2D texture. Somehow, you need to get the results of the equation above into a 2D texture. Once you have that, you can use the texture lookup in the same way that you would use a function with two inputs.

I will outline how to create the texture using DirectX, but the technique can be mapped easily to OpenGL. I'm really only interested in one output value, so I'll create a texture that holds what is essentially a 2D array of 32-bit floating-point values. In DirectX, the code to do that looks like this:

```
D3DXCreateTexture(m_pD3DDevice, 64, 64, 1, 0,
   D3DFMT_R32F, D3DPOOL_MANAGED,
   &m_pOrenNayarTexture);
```

OpenGL has similar texture creation functions. In either case, once you have the texture, you need to fill it with values. One way to do this would be to lock the contents of the texture, compute the values, and write the values to the texels. This should be fairly easy in either API. However,

there are other options. You already have shader code that computes the final output value of the function, so it might be nice to reuse that. This is where D3DXFillTextureTX comes in handy. This utility function allows you to fill a texture with the results of a shader function. So, I simply cut and pasted some of the vertex shader into a new file called \Code\Shaders\Fill_OrenNayar_Texture.hsl. After some minor reworking, the shader is as follows.

The function expects the first input to be a set of 2D texture coordinates. These coordinates will be used to compute the correct result for each texel. The second parameter is not used. The return value is a single floating-point value.

```
float FillOrenNayar(float2 DotProducts : POSITION,
   float2 NotUsed : PSIZE) : COLOR
{
```

The function should scale and bias the texture coordinates so that values are in the range from –1 to 1.

```
    float VdotN = 2.0f * DotProducts.x - 1.0f;
    float LdotN = 2.0f * DotProducts.y - 1.0f;
```

The rest of the code is copied directly from the vertex shader.

```
    float AngleViewNormal  = acos(VdotN);
    float AngleLightNormal = acos(LdotN);

    float Alpha = max(AngleViewNormal, AngleLightNormal);
    float Beta  = min(AngleViewNormal, AngleLightNormal);
```

The final return value is the product of the two terms. This value is written into each texel of the texture.

```
   return sin(Alpha) * tan(Beta);
};
```

The following code uses the shader and the D3DX functions to fill the texture.

```
LPD3DXBUFFER pShaderCode = NULL;
D3DXCompileShaderFromFile("..\\Shaders\\Fill_OrenNayar_
   Texture.hsl", NULL, NULL, "FillOrenNayar",
   "tx_1_0", NULL, &pShaderCode, NULL, NULL);

D3DXFillTextureTX(m_pOrenNayarTexture,
   (DWORD*)pShaderCode->GetBufferPointer(), NULL, 0);
```

Doing the same with straight C++ code would be very easy because the function doesn't use anything that is specific to the hardware or the API. However, I wanted to show the D3DX utility because it can simplify things, as well as make it easier to reuse existing shader code.

Once the texture is filled, it represents a set of precomputed outputs to the function shown above. This lets you greatly simplify the pixel shader by replacing those instructions with one 2D texture lookup. The resulting pixel shader can be found on the CD as \Code\Shaders\OrenNayar_PS.hsl.

All of the inputs are the same, but now the texture of precomputed values is used as an input texture.

```
float3  LightVector   : register(c4);
vector  EyePosition   : register(c5);
sampler AngleTex      : register(s0);

PS_OUTPUT main(const VS_OUTPUT In)
{
   PS_OUTPUT OutPixel;
```

```
    float4 DiffuseColor = {1.0f, 1.0f, 1.0f, 1.0f};
    float3 PixelToEye = normalize(EyePosition -
       In.InterPos);
    float3 NewNormal = normalize(In.Normal);
```

I have collapsed the two dot products into a single set of the two values to facilitate passing them to the texture lookup function.

```
    float2 DotProducts;
    DotProducts.x = dot(PixelToEye, NewNormal);
    DotProducts.y = dot(LightVector, NewNormal);
    float Irradiance = max(0.0f, DotProducts.y);

    float AngleDifference = max (0.0f, dot
       (normalize(PixelToEye - NewNormal *
       DotProducts.x), normalize(LightVector -
       NewNormal * DotProducts.y)));
```

At this point, the vertex shader would have calculated the alpha and beta values. The pixel shader replaces these instructions with the 2D texture lookup.

```
    float AngleTerm = tex2D(AngleTex, 0.5f *
       DotProducts + 0.5f);

    float RoughnessSquared = 0.5f;
    float A = 1.0f - (0.5f * RoughnessSquared) /
       (RoughnessSquared + 0.33f);
    float B = (0.45f * RoughnessSquared) /
       (RoughnessSquared + 0.09f);
```

The final output color includes the value that was retrieved from the texture.

```
    OutPixel.Color  = DiffuseColor * (A + B *
       AngleDifference * AngleTerm) * Irradiance;

    return OutPixel;
}
```

In the end, the number of pixel shader instructions drops from 70 to 27. This suggests that even if you could do everything with arithmetic instructions, you might not want to. Figure 7.9 shows that the visual result of the pixel shader is basically the same as the vertex shader. On lower-resolution meshes, the difference would be greater.

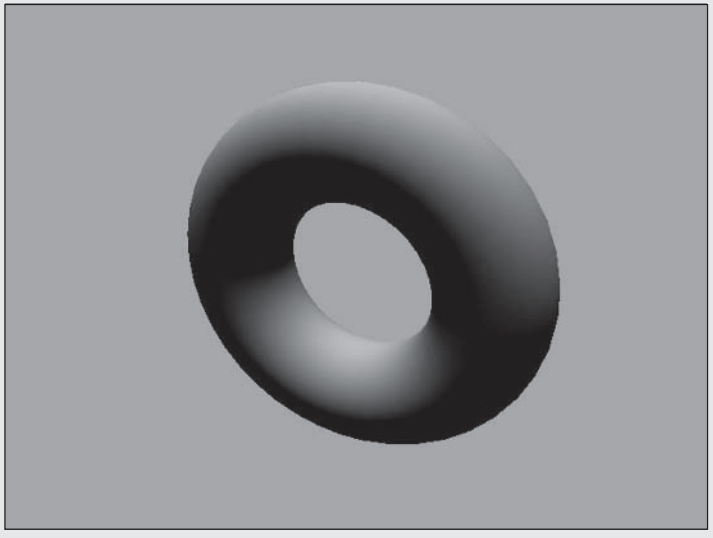


Figure 7.9. Per-pixel Oren-Nayar

7.4 Minnaert Materials

Minnaert materials have the same inputs and some of the same building blocks as Oren-Nayar, but the final equation uses the blocks differently. The Minnaert vertex shader is on the companion CD as \Code\Shaders\Minneart_VS.hsl and it is also shown below.

```
matrix Transform     : register(c0);
vector LightVector   : register(c4);
vector EyePosition   : register(c5);

VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;
```

The first set of instructions is basically the same as seen in the previous shaders. This might convince someone to rewrite the shaders as a main function that does initial setup and then a set of different functions that each calculate different end results. This could be interesting or beneficial, but I am trying to keep the shaders as simple and clear as possible.

```
   vector VertexToEye = normalize(EyePosition -
      InVertex.Position);

   float VdotN = dot(VertexToEye, InVertex.Normal);
   float LdotN = dot(LightVector, InVertex.Normal);
   float Irradiance = max(0.0f, LdotN);
```

I've introduced a Minnaert roughness parameter. Like most of these constants, a real application would probably pass this to the shader as a constant.

```
   float4 DiffuseColor = {1.0f, 1.0f, 1.0f, 1.0f};
      float  Power     = 0.8f;
```

The final calculation mirrors the equation found in Chapter 5.

```
      OutVertex.Diffuse  = DiffuseColor *
         pow(VdotN * LdotN, Power) * Irradiance;
      OutVertex.Position = mul(Transform,
         InVertex.Position);

      return OutVertex;
   }
```

Figure 7.10 shows a Minnaert shaded object with two different values for the power parameter.

This pixel shader doesn't require a lookup texture, so the HLSL code is essentially the same.

```
float3  LightVector    : register(c4);
vector  EyePosition    : register(c5);

PS_OUTPUT main(const VS_OUTPUT In)
{
   PS_OUTPUT OutPixel;

   float3 PixelToEye = normalize(EyePosition -
      In.InterPos);
   float3 NewNormal  = normalize(In.Normal);

   float VdotN = dot(PixelToEye,  NewNormal);
   float LdotN = dot(LightVector, NewNormal);
   float Irradiance = max(0.0f, LdotN);
   float4 DiffuseColor = {1.0f, 1.0f, 1.0f, 1.0f};
```

```
    float  Power        = 0.8f;

    OutPixel.Color  = DiffuseColor *
       pow(VdotN * LdotN, Power) * Irradiance;

    return OutPixel;
}
```

Figure 7.11 shows a side-by-side comparison of the per-vertex Minnaert material (left) and the per-pixel version (right). The differences in the printed version might be harder to see, but the per-pixel version is smoother, especially in the interior of the torus.

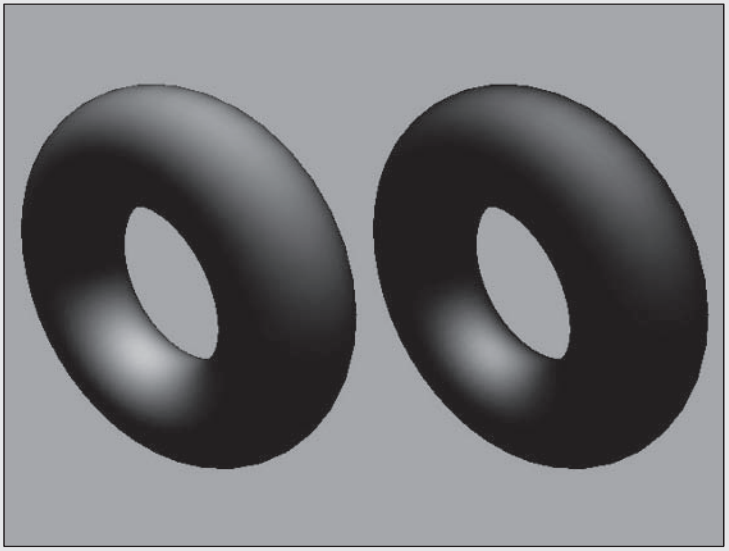


Figure 7.10. Per-vertex Minnaert with values of 0.75 (left) and 1.5 (right)

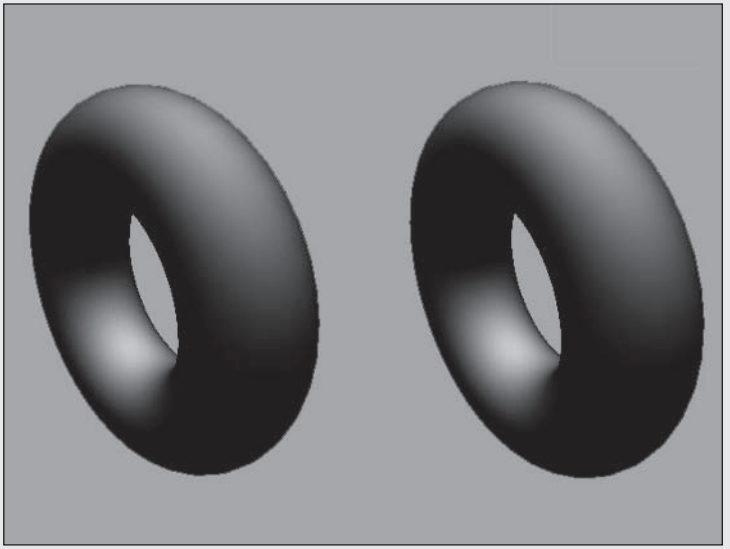


Figure 7.11. Comparing per-vertex Minnaert with the per-pixel version

7.5 Ward Materials

In Chapter 5, I discussed the rationale behind Ward materials, as well as the isotropic and anisotropic versions. In this section, I'll cover the shader implementations of both.

7.5.1 Isotropic Ward Materials

By this point, you should be pretty adept at translating the equations from Chapter 5 into HLSL shader code. The Ward equations are not very pretty, but the implementation is fairly straightforward. The isotropic vertex shader can be found on the CD as \Code\Shaders\Ward_Isotropic_VS.hsl.

```
VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;

   vector VertexToEye = normalize(EyePosition -
      InVertex.Position);
   float3 HalfVector  = normalize(LightVector +
      VertexToEye);
```

The majority of the work involves the trigonometric functions. As usual, the roughness parameter would probably be an input constant in a real application.

```
   float Roughness = 0.25f;

   float tan2NH    = -pow(tan(acos(dot
      (InVertex.Normal, HalfVector))), 2);

   float RMS2      = pow(Roughness, 2);
   float Den       = 6.28 * RMS2;

   float FirstTerm = exp(tan2NH / RMS2) / Den;

   float CosTheta = dot(InVertex.Normal, LightVector);
   float CosDelta = dot(InVertex.Normal, VertexToEye);

   float SecondTerm = 1.0f / sqrt(CosTheta * CosDelta);
```

The CosTheta variable is really L dot N, so it is reused in the irradiance calculation.

```
   float3 Irradiance = max(0.0f, CosTheta);

   float4 DiffuseColor  = {1.0f, 1.0f, 1.0f, 1.0f};
   float4 SpecularColor = {1.0f, 1.0f, 1.0f, 1.0f};
```

Put all the terms together to get the final value.

```
   float3 SpecularTerm = SpecularColor * FirstTerm *
      SecondTerm;

   OutVertex.Diffuse  = (DiffuseColor + SpecularTerm)
      * Irradiance;
   OutVertex.Position = mul(Transform,
      InVertex.Position);

   return OutVertex;
}
```

Figure 7.12 shows Phong and Ward objects side by side. It is difficult to find exact matching specular values, but the effects are similar. There are slight differences in the shape and size of the highlights.

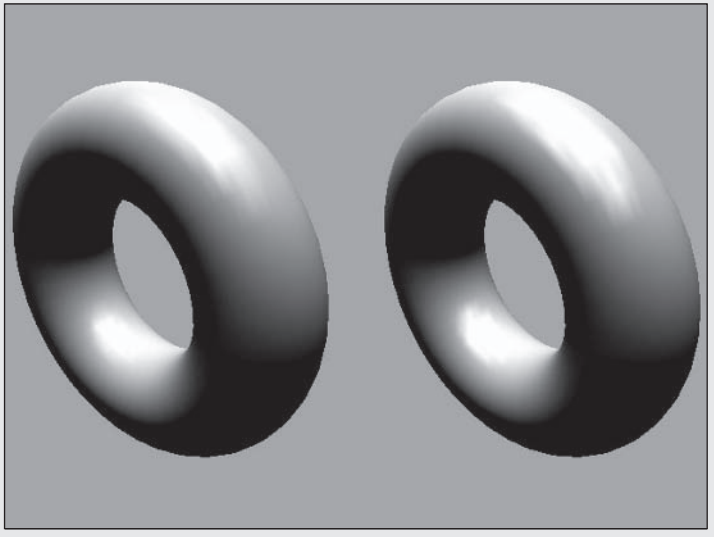


Figure 7.12. Comparing Phong (left) with Ward (right)

As usual, the pixel shader version is essentially the same and is on the CD as \Code\Shaders\Ward_Isotropic_PS.hsl. I haven't included the code here, but I still want to point out a couple of interesting features. As I have written the shader, the total instruction count comes to 46 instructions. This is a fairly long pixel shader. Also, the fact that the roughness value is a constant lets the shader compiler precompute a handful of values. If the roughness is changed to an input constant, the number of instructions jumps to 51. This is starting to be more of a concern. If you apply what you learned with Oren-Nayar shaders, you might find that you can boil several terms down to the following equation:

$$I_o = (N \bullet L) * \left(D + S * f\left((N \bullet L), (N \bullet V), (N \bullet H), r\right)\right)$$

The described function is 4D and therefore not easily encoded into a texture. However, if you think only of the dot product terms, it becomes a 3D function, which can be encoded into a volume texture (assuming your hardware supports them well).

Once you have created a volume texture (the algorithm to do this is included in the source code), the steps needed to fill the texture are the same as those you saw in the Oren-Nayar section. In this case, the shader function is in the file \Code\Shaders\Fill_WardIsotropic_Texture.hsl and is shown below.

```
float4 FillWardIsotropic(float3 DotProducts : POSITION,
                         float2 NotUsed : PSIZE) : COLOR
{
```

For this example, I used a set value for the roughness, which means that you would have to precompute new texture values every time you needed a new roughness parameter. However, I am only using one of the four channels in the volume texture, so you could just as easily encode the results for three more roughness values. On the pixel shader side, a constant could be used to mask out the desired value and to interpolate between values. That exercise is left to the interested reader, but the volume texture creation parameters are currently set for a four-channel color.

```
    float Roughness = 0.25f;
```

The remaining code is basically the same as you have seen in the vertex shader.

```
    float tan2NH    = -pow(tan(acos(DotProducts.z)), 2);
    float RMS2      = pow(Roughness, 2);
    float FirstTerm = exp(tan2NH / RMS2) / (6.28 * RMS2);

    float SecondTerm = 1.0f / sqrt(DotProducts.x *
       DotProducts.y);

    return FirstTerm * SecondTerm;
};
```

Next, you need a pixel shader that does the 3D texture lookup to extract these values. That shader is in the file \Code\Shaders\Ward_Isotropic_PS_Tex.hsl and is shown below.

First, you must tell the shader which texture to use.

```
float3  LightVector    : register(c4);
vector  EyePosition    : register(c5);
sampler WardTex        : register(s1);

PS_OUTPUT main(const VS_OUTPUT In)
{
   PS_OUTPUT OutPixel;

   float3 PixelToEye = normalize(EyePosition -
      In.InterPos);
   float3 NewNormal  = normalize(In.Normal);
   float3 HalfVector = normalize(LightVector +
      PixelToEye);
```

The three dot product results are packed into a single vector that will be used for the volume texture lookup.

```
    float3 DotProducts;
    DotProducts.x = dot(LightVector, NewNormal);
    DotProducts.y = dot(PixelToEye, NewNormal);
    DotProducts.z = dot(HalfVector, NewNormal);

    float4 Irradiance = max(0.0f, DotProducts.x);

    float4 DiffuseColor  = {1.0f, 1.0f, 1.0f, 1.0f};
    float4 SpecularColor = {1.0f, 1.0f, 1.0f, 1.0f};
```

Finally, the roughness and trigonometric functions are collapsed to the single 3D texture lookup and the result is used to find the specular term.

```
    float4 SpecularTerm = SpecularColor *
       tex3D(WardTex, DotProducts);

    OutPixel.Color  = (DiffuseColor + SpecularTerm) *
       Irradiance;

    return OutPixel;
}
```

The resulting shader uses 19 shader instructions versus 51. However, the volume texture could be quite memory intensive or not a viable solution on some hardware. Figure 7.13 compares (from left to right) the vertex shader version, the pixel shader version, and the pixel shader with the volume texture version.

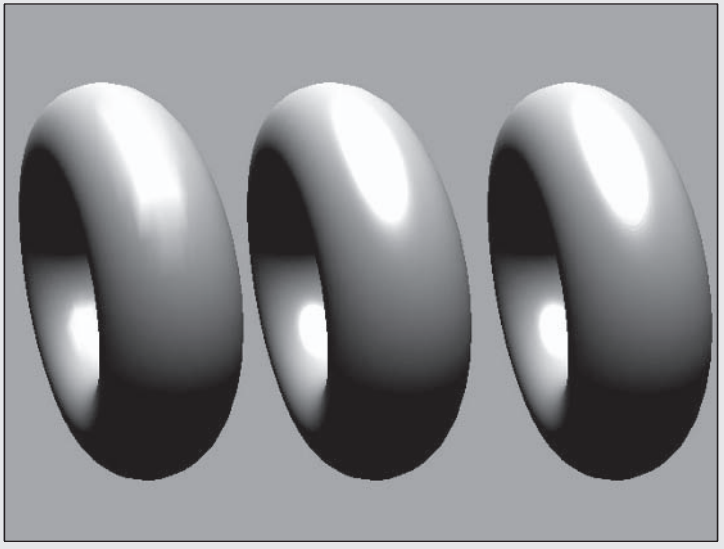


Figure 7.13. Comparing per-vertex and per-pixel Ward shaders

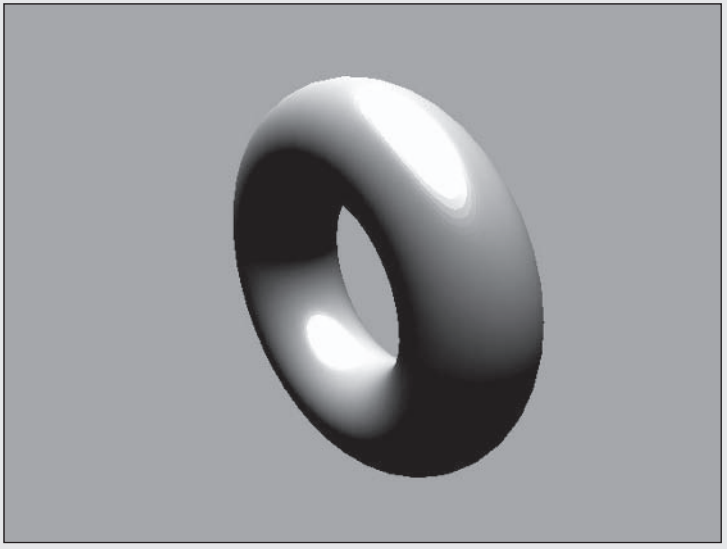


Figure 7.14. Banding with a lower resolution volume texture

The pixel shader versions are smoother than the vertex shader version. However, the quality of the volume textured version is a function of the dimensions of the volume texture. For Figure 7.13, I chose a 128x128x128 texture, which is quite large. A 64x64x64 texture is much smaller, but can cause banding as seen in Figure 7.14.

All things considered, the volume texture might not be the best approach here, especially when you consider lower cost alternatives for the same overall effect. However, I wanted to show you another texture lookup technique because it might be very useful for other materials.

7.5.2 Anisotropic Ward Materials

The isotropic Ward equation is fairly straightforward. The anisotropic version is a more complicated beast. Chapter 5 showed the equation, but I will reformulate it here to ease implementation. The two representations are functionally equivalent, but the version seen below is easy to break up into manageable chunks.

The X and Y terms are tangent to the object and are determined by the direction of anisotropy. The H term is the familiar half vector.

The vertex shader is on the CD as \Code\Shaders\Ward_Anisotropic_VS.hsl and is explained below. As usual, keep in mind that many of the variables contained in the shader could be passed in as constants.

```
VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;
```

```
   vector VertexToEye = normalize(EyePosition -
      InVertex.Position);
   float3 HalfVector  = normalize(LightVector +
      VertexToEye);
```

The first term is very easy to compute and is reminiscent of the second term in the isotropic implementation.

```
   float CosTheta   = dot(InVertex.Normal, LightVector);
   float CosDelta   = dot(InVertex.Normal, VertexToEye);

   float4 FirstTerm = 1.0f / sqrt(CosTheta * CosDelta);
```

The second term is also easy to compute. If the roughness parameters were passed in as constants, it would probably be good to compute the second term in the application and pass that in as well.

```
   float2 Roughness = {0.9f, 0.1f};
   float4 SecondTerm = 1.0f / (12.56 * Roughness.x *
      Roughness.y);
```

$$I_0 = \left(D + S * \frac{1}{4\pi\sigma_x\sigma_y} * \frac{1}{\sqrt{(N \bullet L)(N \bullet V)}} * e^{-2\left(\frac{\left(\frac{H \bullet X}{\sigma_x}\right)^2 + \left(\frac{H \bullet Y}{\sigma_y}\right)^2}{1 + H \bullet N} \right)} \right) I_i (N \bullet L)$$

Anisotropic Ward equation

This block of code determines two perpendicular tangent vectors on the object's surface. One could save several shader instructions by packing this data into the vertices themselves. I didn't do that here in order to minimize the amount of customization needed for any one particular shader. You could change the direction of anisotropy by either changing the roughness values above or by tweaking the direction below. These new vectors, along with their relationships with the half vector, eventually play a key role in the third term of the Ward equation.

```
    float3 Direction = {0.0f, 0.0f, 1.0};
    float3 X = normalize(cross(InVertex.Normal, Direction));
    float3 Y = normalize(cross(InVertex.Normal, X));

    float HdotX = dot(HalfVector, X);
    float HdotY = dot(HalfVector, Y);
    float HdotN = dot(HalfVector, InVertex.Normal);

    float A = -2.0f * (pow((HdotX / Roughness.x), 2) +
       pow((HdotY / Roughness.y), 2));
    float B = 1.0f + HdotN;

    float4 ThirdTerm = exp(A / B);

    float4 Irradiance = max(0.0f, CosTheta);

    float4 DiffuseColor  = {1.0f, 1.0f, 1.0f, 1.0f};
    float4 SpecularColor = {1.0f, 1.0f, 1.0f, 1.0f};
```

The remainder of the shader brings together all three terms to compute the final color value.

```
    float4 Specular = SpecularColor * FirstTerm *
       SecondTerm * ThirdTerm;

    OutVertex.Diffuse  = (Specular + DiffuseColor) *
       Irradiance;
    OutVertex.Position = mul(Transform, InVertex.Position);

    return OutVertex;
}
```

Figure 7.15 shows anisotropic objects with highlights oriented vertically and horizontally (with respect to the image).

The pixel shader is again very similar to the vertex shader. I have not included the code here in the text, but you can find it on the CD as \Code\Shaders\Ward_Anisotropic_PS.hsl. The shader comes out to a fairly hefty 43 instructions (as written), and there isn't necessarily any good way to collapse several of the vector operations down to a simple texture lookup. However, the shader could be greatly simplified if the tangent vectors were passed from the vertex shader or if they were contained in a texture map. Another advantage of using a texture map is that you can make the direction of anisotropy change over the surface of the object. If you'd like to explore these possibilities, there are many good anisotropic samples at ATI's [1] and NVIDIA's [2] developer sites, as well as other sites.

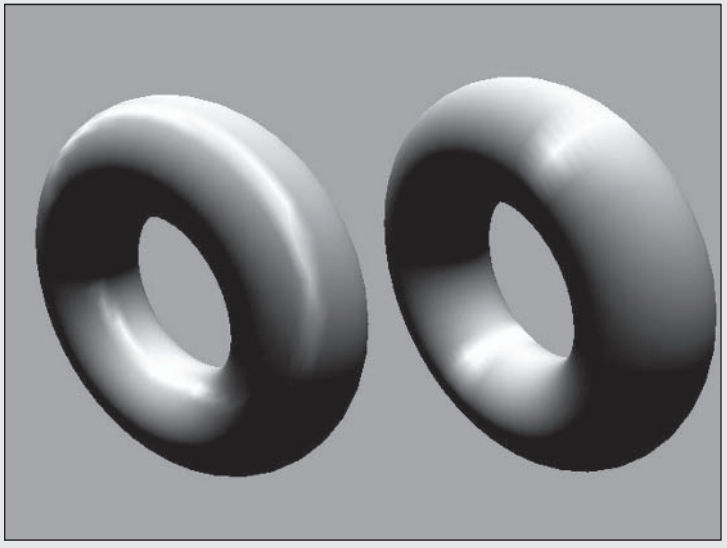


Figure 7.15. Anisotropic objects

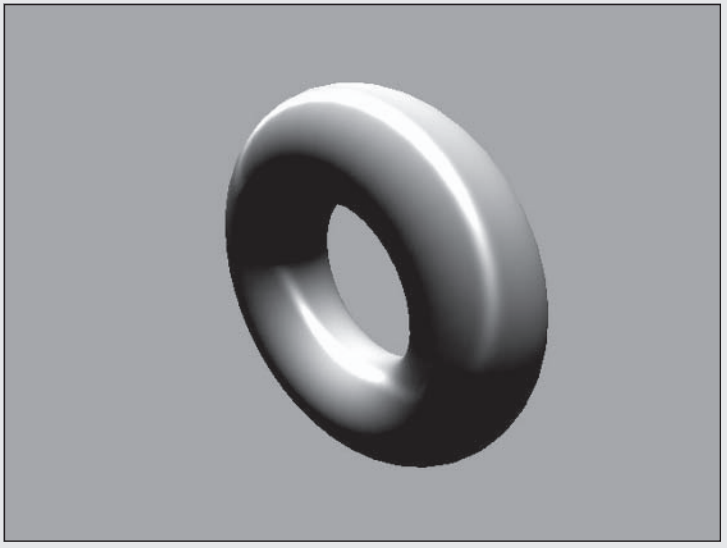


Figure 7.16. Per-pixel anisotropic shading

Figure 7.16 shows the result of the per-pixel technique. As usual, the result is smoother than the per-vertex version. Experiment with offloading some of the processing to the application, the vertex shader, or textures to lower the overall instruction count of the pixel shader.

7.6 Schlick Materials

In Chapter 5, the Schlick model was introduced as a way to circumvent the exponential factor in the Phong model. Implementations will shed some light on the real differences between the models. In fairness, remember that nearly all of the shaders shown here could probably be optimized to save an instruction or two.

The Schlick vertex shader is included on the CD as \Code\Shaders\Schlick_VS.hsl. The listing is included below.

```
VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;

   vector VertexToEye = normalize(EyePosition -
      InVertex.Position);
```

Schlick is a function of the real reflection vector, so the first thing one needs to do is compute that vector and find its relationship to the view vector.

```
   float3 Reflection = normalize(2.0 *
      InVertex.Normal.xyz * dot(InVertex.Normal.xyz,
      LightVector.xyz) - LightVector.xyz);

   float RdotV = max(dot(Reflection.xyz, VertexToEye),
      0.0);

   float4 DiffuseColor  = {1.0f, 1.0f, 1.0f, 1.0f};
   float4 SpecularColor = {1.0f, 1.0f, 1.0f, 7.0f};
      //Power in w
```

The Schlick specular term eliminates the exponential component found in the Phong model. Once you compute the Schlick term, the rest of the shader is very simple.

```
   float SchlickTerm = RdotV / (SpecularColor.w -
      (SpecularColor.w * RdotV) + RdotV);

   float3 Diffuse = DiffuseColor * max(0.0f,
      dot(LightVector, InVertex.Normal));

   float3 Specular = SpecularColor.xyz * SchlickTerm;

   OutVertex.Diffuse =  Diffuse + Specular;

   OutVertex.Position = mul(Transform,
      InVertex.Position);

   return OutVertex;
}
```

The difference in instruction counts between the Phong shader and the Schlick shader is not just a matter of compiling and counting the instructions. In the simplest case, the HLSL shader compiles the pow() function to a series of multiplies if the exponent is an integer. For lower power values, the Schlick shader might be longer than the equivalent Phong shader. For higher power values, the Phong shader starts to be a couple of instructions longer. However, things really fall apart if the power is not an integer or if it is passed in as a float constant (where the compiler cannot assume it's an integer). In this case, the Phong shader can expand to

nearly double the number of instructions of the Schlick shader.

There are a couple of points to take away from this. The first is that blind optimization isn't always worthwhile. If you had a low specular power that never changed, the "more expensive" Phong shader might be better. The second point is that small changes to HLSL shaders can produce dramatically different final shaders. Earlier is this chapter, I mentioned that the Phong shader as described in section 7.2.2 required 21 shader instruction slots. If you change that shader so that the specular power can be input as a constant, the number jumps to 37. While HLSL dramatically eases shader development, it is sometimes worthwhile to take a peek at the resulting assembly code to see what's being produced.

NOTE:

You can quickly get shader statistics by using the command-line shader compiler and writing the results to an output file. For example, I use the following line to see the assembly code for the Blinn-Phong vertex shader:

```
fxc /T vs_1_1 /Fc BPVS_out.txt Blinn_Phong_VS.hsl
```

Figure 7.17 shows the Schlick (left) and Phong (right) results side by side. The results are very similar and the difference in cost could be substantial.

The pixel shader version is included on the CD as \Code\Shaders\Schlick_PS.hsl, but is not included in the text as it is basically the same as the vertex shader. The performance trade-offs described with the vertex shader apply to the pixel shader, but there are other considerations. For both Schlick and Phong, the specular term is a function of the specular power and R dot V. In both cases, these functions could be implemented as a 2D texture lookup, meaning that the two pixel shaders would be the same and only the shader functions used to fill the textures would be different. In that case, Phong or Ward might be better options unless the Schlick model was attractive to you for other reasons.

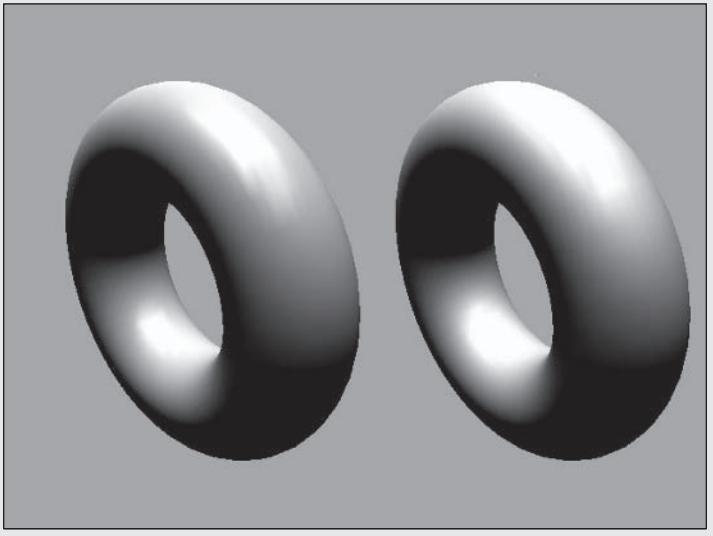


Figure 7.17. Schlick (left) and Phong (right) shaded models

7.7 Cook-Torrance Materials

The final shader implementation focuses on the Cook-Torrance model. I discussed the model itself in Chapter 5, and the implementation is fairly straightforward. However, in the absence of real values for the Fresnel function, I am using the approximation described in [3]. While it might not be physically accurate, it should produce decent results. The approximation equation is shown below.

$$F = R + (1 - R) * (1 - (N \bullet L))^5$$

The complete shader can be found on the CD as \Code\Shaders\CookTorrance_VS.hsl. The listing is shown below.

```
VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;

   vector VertexToEye = normalize(EyePosition -
      InVertex.Position);
   float3 HalfVector  = normalize(LightVector.xyz +
      VertexToEye.xyz);
   float  NdotH       = max(0.0f, dot(InVertex.Normal,
      HalfVector));
```

The three roughness parameters are used for the roughness term and the Fresnel approximation. If you change the shader to pass in these values, make sure that the Fresnel value is precomputed properly as described in the equation above.

```
   float3 RoughnessParams = {0.5f, 0.5f, 0.5f};
```

In this case, I am using the simpler Blinn function for the distribution term.

```
   float Alpha = acos(NdotH);
   float C     = RoughnessParams.x;
   float m     = RoughnessParams.y;
   float3 D    = C * exp(-(pow(Alpha / m, 2.0f)));
```

The geometric term is a fairly basic implementation of the equation seen in Chapter 5.

```
   float NdotV = dot(InVertex.Normal, VertexToEye);
   float VdotH = dot(HalfVector, VertexToEye);
   float NdotL = dot(LightVector, InVertex.Normal);
   float3 G1 = 2 * NdotH * NdotV / NdotH;
   float3 G2 = 2 * NdotH * NdotL / NdotH;
   float3 G = min(1.0f, max(0.0f, min(G1, G2)));
```

This is the Fresnel approximation described above and in [3].

```
   float R0 = RoughnessParams.z;
   float3 F = R0 + (1.0f - R0) * pow(1.0f - NdotL, 5.0);
```

Once all the terms are ready, the only thing left to do is put them together.

```
   float4 DiffuseColor = {1.0f, 1.0f, 1.0f, 1.0f};

   OutVertex.Diffuse   = DiffuseColor * F * D * G /
      (NdotL * NdotV);
   OutVertex.Position  = mul(Transform, InVertex.Position);
   return OutVertex;
}
```

Figure 7.18 shows the obligatory screen shot of the per-vertex Cook-Torrance technique. Notice the difference in overall character between this figure and the earlier Phong variants. Here, the differences between "plastic" and "metal" are quite apparent.

As usual, the equivalent pixel shader (\Code\Shaders\CookTorrance_PS.hsl) is basically the same and omitted here in the text. However, as a purely arithmetic shader, the final code weighs in at roughly 45 instructions. Any or all of the three terms could be encapsulated into 2D or 3D textures, depending on how many variables you wanted to encode directly into the textures. You could probably cut the number of instructions down to less than 30 if you wanted to use more textures. I will leave that as an exercise for the reader, but everything you need to know has been shown working for other shaders in this chapter.

Figure 7.19 shows a screen shot of the per-pixel effect. As usual, it is much smoother than the per-vertex version.

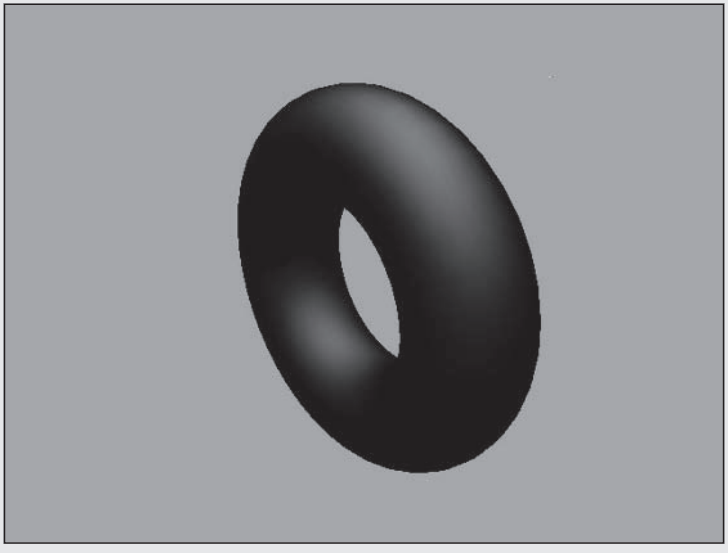


Figure 7.18. Per-vertex Cook-Torrance

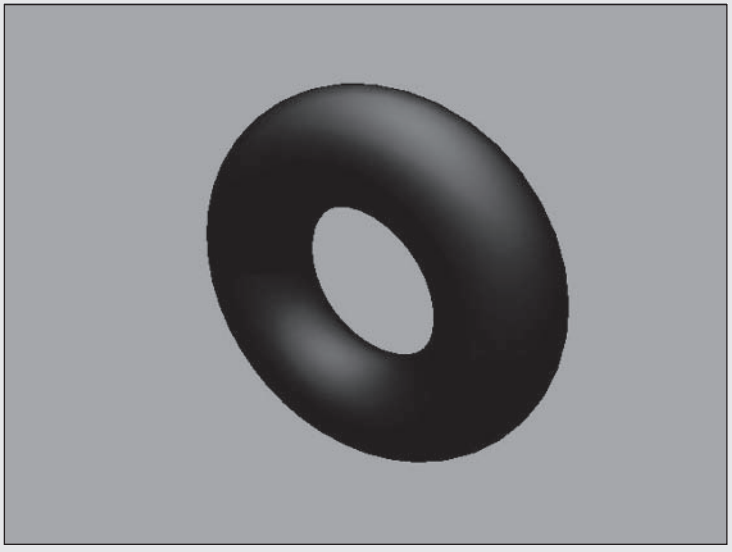


Figure 7.19. Per-pixel Cook-Torrance

Conclusion

On one level, this chapter is simply a collection of vertex and pixel shaders based on the equations you saw in Chapter 5. However, I'd like you to also look at it as a set of examples of how easily those equations can be implemented with relatively simple higher level shader code. This is very different from the DirectX 8 class of shaders, which are implemented using relatively cryptic code with few higher level instructions.

Also, although every reflectance model was presented in both vertex and pixel shader form, it's clear that pixel shaders aren't necessarily better in some cases. Also, some of my approaches (from using purely arithmetic shaders to using resource-heavy volume textures) might not be the best approaches for all situations. It's ultimately up to you to decide how you want to handle the trade-offs between quality, performance, and the distribution of work across different segments of the pipeline. The purpose of this chapter was to show a collection of approaches rather than to show the "best" approach.

Most of these shaders could probably be streamlined with more careful coding, handcrafted assembly code, or a different approach. Interested readers might want to spend some time changing the shaders to see if they can improve performance.

Finally, although this book tries to present both DirectX and OpenGL implementations, this chapter has presented only DirectX shaders, mainly because the shaders herein can be ported to cg and other languages with very few changes (and in most cases, no changes at all).

References

[1] ATI Developer Tools, http://www.ati.com/developer/tools.html

[2] NVIDIA developer site, http://developer.nvidia.com/

[3] "Fresnel Reflection," (NVIDIA), http://developer.nvidia.com/object/fresnel_wp.html

Chapter 8

Spherical Harmonic Lighting

Introduction

In previous chapters, you saw different techniques and approaches used to render realistic scenes in real time. This ranged from different mapping techniques, to procedural texturing, to the use of different BRDFs to achieve a greater level of realism.

In this chapter, I talk about an advanced topic referred to as spherical harmonic lighting to generate realistic real-time renderings of 3D scenes. More precisely, I present this recent technique (at least when applied in the field of computer graphics; see [1] for the initial paper introducing this techinque) that takes into account subtle effects through three types of diffuse lighting models: unshadowed, shadowed, and inter-reflected. The end goal is to take into account and render the global illumination of scenes lit by any type of area light sources. As you will see, even though it involves quite a lot of mathematics, the technique is elegant and simple enough to be considered for implementation in any game engine or other real-time rendering applications.

This is quite a long chapter, as there is a lot of fairly complex material to discuss. The chapter is broken up as follows.

- Understanding the basic ideas behind spherical harmonics (SH)
- The mathematics of SH
- Full implementation details in OpenGL
- DirectX implementation using D3DX helper functions

8.1 Understanding the Need for Spherical Harmonics

This chapter includes what is perhaps the most complex math in the entire book. At times, the math can be a bit daunting, so I would like to spend a few pages talking about the basic ideas behind the math. Once you can see why the technique is useful, it will be easier to understand how the different pieces fit together. I will frame these explanations in terms of how they apply to spherical harmonics, but keep in mind that some of the problems that I discuss can be solved using other techniques and tools. The reason I focus on SH as a solution is that it is currently one of the best solutions, and it is of course the subject of this chapter.

8.1.1 Hemispheres of Light

In Chapter 5 and other chapters, I show that you can think of a point on a surface as being surrounded by a hemisphere of light. This lets you map XYZ positions of point lights to spherical coordinates, providing a convenient basis for BRDFs and other techniques. In most real-time techniques, the idea of the hemisphere of light is just a conceptual basis for the math you see in BRDFs and their related shaders; you really don't account for a full hemisphere of light acting on the surface. This lets you write fast, simple shaders, but it's not very realistic because it doesn't provide good support for area lights and reflected light from surrounding objects.

Consider Figure 8.1. In this scene, I have created a very simple room with three ceiling lights (the center light has a decorative grating for reasons that will be clear later). There is a small box on the floor of the room. In this simple scene, I'd like to compute the amount of light that reaches the top of the small box.

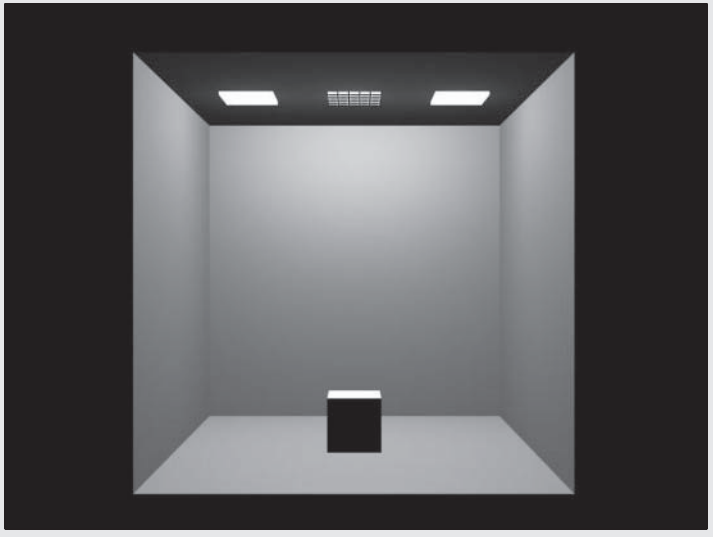


Figure 8.1. A very simple scene

Now, imagine lying on the top of the box and looking up at the incoming light. You would probably see something similar to Figure 8.2. If you were a point on the top of the box, you'd receive energy from the three ceiling lights.

If you wanted to render this scene, you might set it up with three point lights and the appropriate shaders. Figure 8.3 shows this scenario. Here, the view from the top of the box shows three small point lights.

If you rendered this scene, you'd get something that looks fairly nice. Or would you? Compare Figure 8.3 to Figure 8.2. In Figure 8.3, the top of the box is lit from three discrete points on the hemisphere. In Figure 8.2, it is lit from every point on the hemisphere (the actual lights and reflections from the walls). The lights themselves are much larger than points and the walls reflect light to the surface of the box. When you put the two side by side, you can see that the point light representation is not a good representation of reality at all.

However, the advantage of the point light representation is that it's very fast and easy to work with. It also preserves the most essential components of the scene, although it discards many other components. Ideally, you want to preserve more detail than that of Figure 8.3, but including everything from Figure 8.2 would be too slow. That's the crux of the matter and the point of this chapter. In the ideal case, you want a way to represent the light in a scene that preserves as much

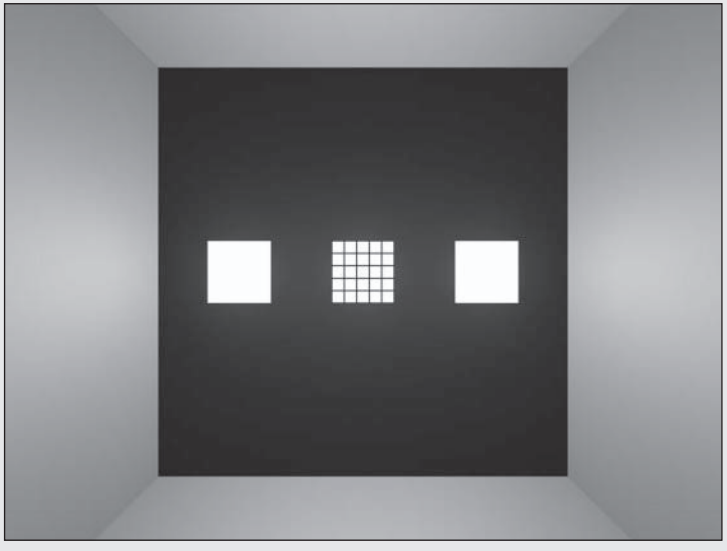


Figure 8.2. A view from the top of the box

Figure 8.3. Point lights as seen from the top of the box

detail as possible and is very fast. Point lights are fast, but sparse in detail. Raytracing preserves detail, but is very slow. The trick is to find something in the middle.

8.1.2 Representations of Light

You might not think of it this way, but the techniques you choose to render something are all based on different representations of different phenomena. For example, you might draw a scene with a planar area light using a two-dimensional array of point lights. You might draw the same scene with a backward raytracer that checks to see when a ray intersects a rectangle. In both cases, the scene is the same, but your representations are different and each has different advantages. To give away the punch line, spherical harmonics give you a different way of representing light around a point, but it will take a little bit more explanation to show why SH is interesting.

Two important features of a good representation are that it's compact in terms of data and that it can be used with relatively low performance cost. Assume that Figure 8.2 contains all the data needed to compute the amount of light that reaches the top of the box. You need a way to compress that data dramatically so that it can be easily stored for each point or object. Also, the method of compression must support very fast computations. SH provides both.

8.1.3 Compressing Data Signals

To understand how one might compress a hemisphere of incoming light, take a look at the data in that hemisphere. Because the page is 2D, it's easier to look at a single 2D slice of the hemisphere. Figure 8.4 shows a view of a center slice from the hemisphere shown in Figure 8.2. For reference, this slice would be the center row of pixels from Figure 8.2.

Now, I will explain things in terms of this 2D slice, but the concepts apply to the 3D hemisphere as well. Figure 8.4 shows the data as dark and light values, but you can represent the same data as a graph as shown in Figure 8.5.

Figure 8.4. Looking at a slice of the hemisphere

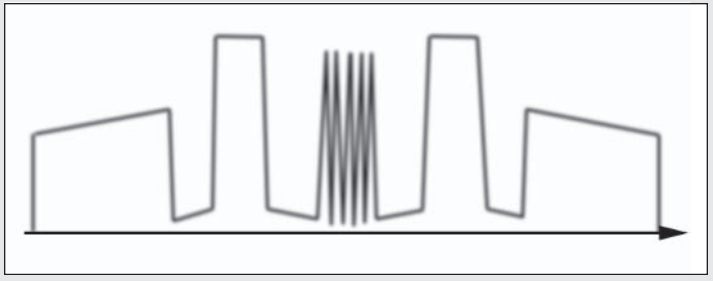


Figure 8.5. Brightness values as a graph

This looks similar to an audio signal or any other form of data signal. If you want to compress this, you can look at how people in other domains compress similar signals. One could write several books on the topic of signal compression, so I'm going to talk about the simplest cases. Many compression techniques are based on the idea of using different combinations of different basis functions to provide a more compact representation of a digital signal. The choice of basis functions is dependent on the task and the properties you need, but in all cases, the basic ideas are the same.

Figure 8.6 shows some arbitrary input signal comprised of 20 data points. These could be readings from a microphone, a seismograph, or anything else.

If you wanted to store this signal, you'd have to store 20 data points. If you wanted to manipulate this signal, you'd have to perform calculations on 20 points. A real-world signal might have millions of points, so there's clearly an advantage to finding a way to compress the data.

In this simple case, the answer is fairly clear. The data looks amazingly like a sine wave. If you can figure out the frequency, amplitude, and phase, you can represent this signal as three values instead of 20. If there were a million similar points, three values would still suffice. These three values are *coefficients* that scale your basis function to match the data. So, for this simple explanation, I will use a sine wave for my basis function.

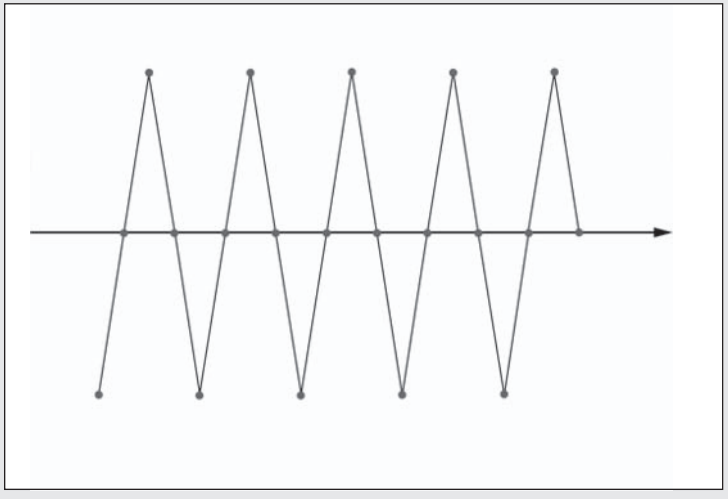


Figure 8.6. An input signal

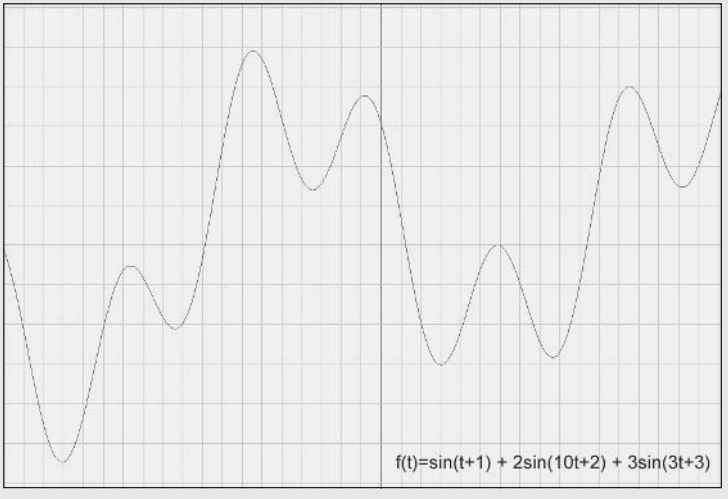


Figure 8.7. A more complex input signal

More complex signals can be represented as combinations of different basis functions. Figure 8.7 shows a signal with N data points that can be represented with six values.

Not only are these smaller to store, but they can be cheaper to manipulate. Changing the amplitude could be a matter of changing a few coefficients rather than scaling all of the individual data points.

Obviously, the signals shown in Figures 8.6 and 8.7 are contrived to make a point. Interesting, real-world signals are usually not that clean and straightforward. In many cases, the number of coefficients needed to accurately represent a signal might be as high as the number of original data points. However, this points to another form of compression.

Figure 8.8 shows something that looks a little bit closer to something from the real world. Accurately representing this signal as a set of sine waves could yield a lot of coefficients.

In these cases, the people using the signal start asking themselves what features they think are most important. For example, if this were an audio signal, some of the features of the signal might be noise or features that could never be reproduced on a speaker. Those features could be stripped out of the compressed signal by simply ignoring the coefficients needed to reproduce them. In the end, the compressed signal might look something like Figure 8.9.

This signal would be much more compressed than the original and, assuming it preserves the important features, it

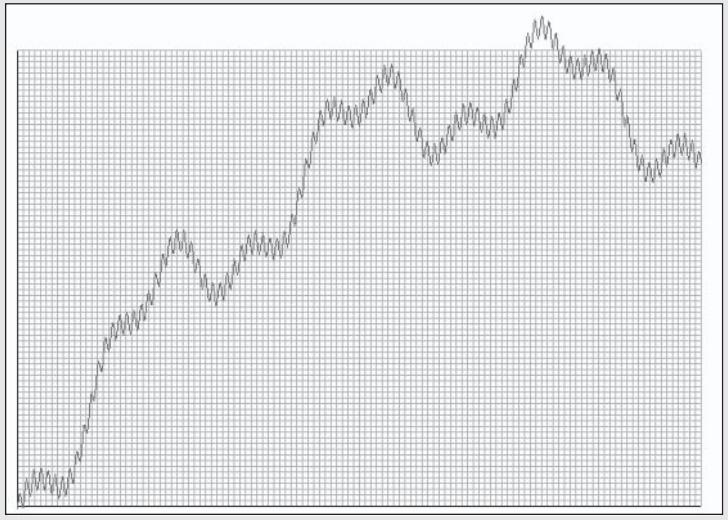


Figure 8.8. A real-world input signal

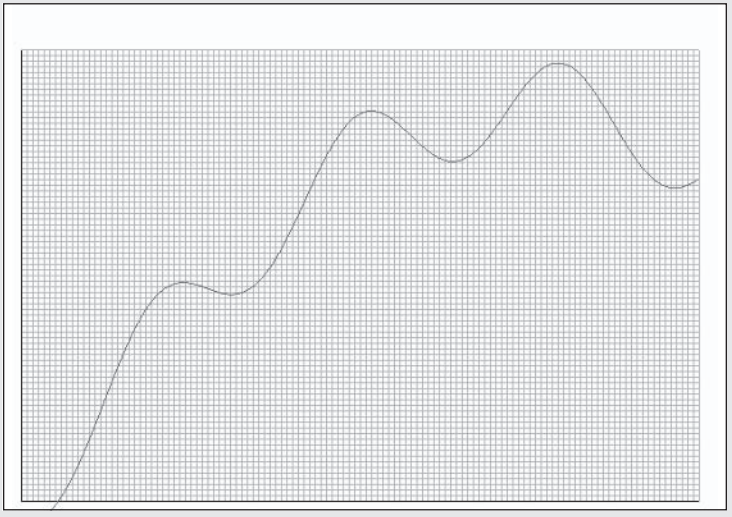


Figure 8.9. The compressed signal

will be just as good for whatever task it is needed. Figure 8.9 shows an example of a signal that has been run through a low-pass filter. The high-frequency noise was stripped out and only the lower frequency components were preserved. Many compression formats such as JPEG and MP3 employ a similar technique.

So, basis functions can be used as a set of building blocks for compressed signals. Once you have these blocks, you can start to throw out the less important ones for even more compression. Now, I'll talk about how that applies to light.

8.1.4 Compressing Hemispheres of Light

Looking again at Figure 8.5, you can see that the slice of light contains low-frequency contributions from the walls and side lights and some high-frequency data that is caused by the grating on the second light. (Yes, this was added so that I could talk about high frequencies.) In the remainder of this chapter, you'll see how to use SH to represent this signal in a compact form. However, remember that you can make the signal more compact if you make decisions about what's really important. For example, this chapter uses SH to provide solutions for diffuse lighting. For diffuse lighting, the high-frequency component of light will probably get "washed out" and not be highly visible in the final rendered scene. So, you can probably generate a more compact representation if you ignore it. In the end, your signal might look more like the one shown in Figure 8.10.

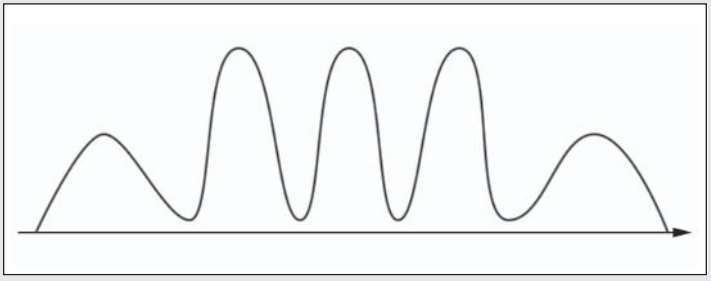


Figure 8.10. Preserving only the lower frequencies in the "light slice"

The remainder of this chapter talks about the SH basis functions, how to generate the coefficients, and how to use them. If you find yourself lost in the math, return to these basic concepts. The math you're about to see might be complex, but it is based on the simple concepts you've seen in this first section.

8.2 Spherical Harmonics Theory

In this section, I am going to walk you through the complex mathematics involved in SH lighting. Don't be afraid though, when you start applying these mathematical entities to lighting and computer graphics, things will get better!

NOTE:

Readers who find themselves uncomfortable with the math in this section will want to keep in mind that C language versions of the equations can be found in section 8.3.

8.2.1 Definition

Spherical harmonics are basis functions that can be used to reconstruct any function. Where the Fourier transform works over the unit circle for one-dimensional functions, spherical harmonics work over the unit sphere for two-dimensional functions.

The term "basis function" comes from the fact that this type of function can be scaled and combined to approximate any mathematical function, as you saw in the first section. The more basis functions you use, the better the approximation. The scaling factors used to combine the basis functions are usually referred to as *coefficients*. Take the example of a function $f(x)$ that you would like to approximate by the family of basis functions $B_i(x)$. The coefficients c_i are calculated by integrating these basis functions $B_i(x)$ with the function $f(x)$, over $f(x)$'s domain of definition D, as shown in Equation 1.

This is how, for instance, the Fourier Transform process works in one dimension to approximate a function of time $f(t)$ (e.g., an audio signal) by its frequencies (the c_i coefficients).

$$f(x) = \lim_{n\to\infty} \sum_{i=0}^{n} c_i B_i(x)$$

where

$$c_i = \int_D f(x) B_i(x) dx$$

Equation 1. Basis functions in one dimension

Spherical harmonics are basis functions defined over the surface of the unit sphere. In this case, you need to work in spherical coordinates, defined in Equation 2.

$$\begin{cases} x = \sin\theta \cos\varphi \\ y = \sin\theta \sin\varphi \\ z = \cos\theta \end{cases}$$

Equation 2. Spherical coordinates

The general form of spherical harmonics is defined using complex numbers, as shown in Equation 3. I explain the different terms on the right of the equation in more detail following Equation 5.

$$Y_l^m(\theta,\varphi) = K_l^m e^{im\varphi} P_l^{|m|}(\cos\theta)$$
$$\text{where}$$
$$l \in \mathbb{N}, -l \le m \le l$$

Equation 3. General form of spherical harmonics

Since I will be using spherical harmonics to approximate real functions, the real form is sufficient. Equation 4 gives the definition of real spherical harmonic functions that I will be using for the remainder of this chapter.

$$y_l^m(\theta,\varphi) = \begin{cases} \sqrt{2}\,\mathrm{Re}(Y_l^m), m > 0 \\ \sqrt{2}\,\mathrm{Im}(Y_l^m), m < 0 \\ Y_l^0, m = 0 \end{cases}$$

Equation 4. Real form of spherical harmonics

The final form of the spherical harmonics is shown in Equation 5, where $l \in \mathbb{N}, -l \le m \le l$.

$$y_l^m(\theta,\varphi) = \begin{cases} \sqrt{2}K_l^m \cos(m\varphi)P_l^m(\cos\theta), m > 0 \\ \sqrt{2}K_l^m \sin(-m\varphi)P_l^{-m}(\cos\theta), m < 0 \\ K_l^0 P_l^0(\cos\theta), m = 0 \end{cases}$$

Equation 5. Spherical harmonics definition

Now let's take a look at the different terms in these equations. P_l^m are the associated Legendre polynomials. They are defined over the range [–1, 1] by the recursive relations given in Equation 6.

$$(l-m)P_l^m(x) = x(2l-1)P_{l-1}^m(x) - (l+m-1)P_{l-2}^m(x)$$
$$P_m^m(x) = (-1)^m(2m-1)!!(l-x^2)^{m/2}$$
$$P_{m+1}^m(x) = x(2m+1)P_m^m(x)$$

Equation 6. Associated Legendre polynomials recursive formulae

Note that the *x!!* is the double factorial of *x*; that is, the product: *x (x – 2) (x – 4) (x – 6)…*

The associated Legendre polynomials present the following interesting properties:

- They are orthonormal polynomials: When you integrate the product of two of them, the value is either 1 if they are the same or 0 if they are different:

$$\int_{-1}^{+1} P_m(x)P_n(x)dx = \begin{cases} 1, if: m = n \\ 0, if: m \ne n \end{cases}$$

- They are defined over the range [–1, 1].

The arguments are defined as follows:

- $l \in \mathbb{N}, 0 \le m \le l$
- l is often called the *band* index

Figure 8.11 shows the plotting of the first three bands of the associated Legendre polynomials, i.e., P_0^0, P_1^0 and P_1^1, and P_2^0, P_2^1, and P_2^2.

K_l^m are the scaling factors used to normalize the functions. Their definition is given by Equation 7.

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

Equation 7. Spherical harmonics normalization constants

The *band* index l takes positive integer values. *m* takes signed integer values between –l and l. In mathematical terms: $l \in \mathbb{N}, -l \le m \le l$. It is also convenient to use the following form of basis functions, as you will see later:

$$y_l^m(\theta,\varphi) = y_i(\theta,\varphi)$$
$$where$$
$$i = l(l+1) + m$$

Equation 8. Single-indexed form of spherical harmonics

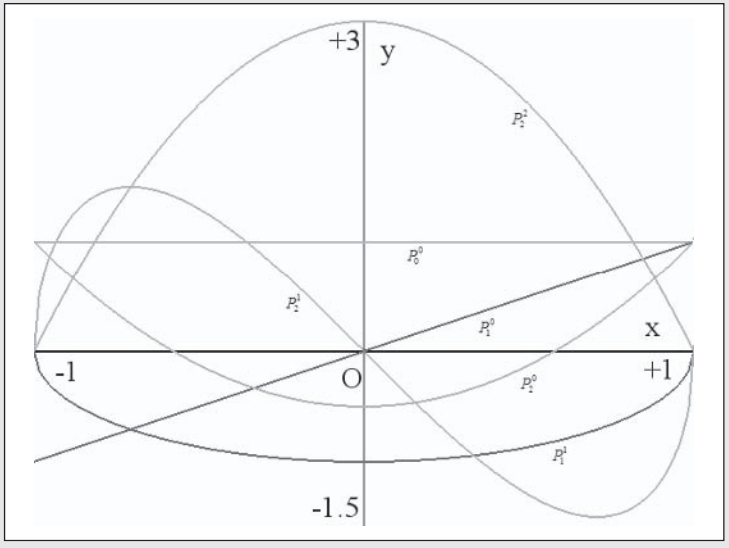


Figure 8.11. The first three bands of associated Legendre polynomials

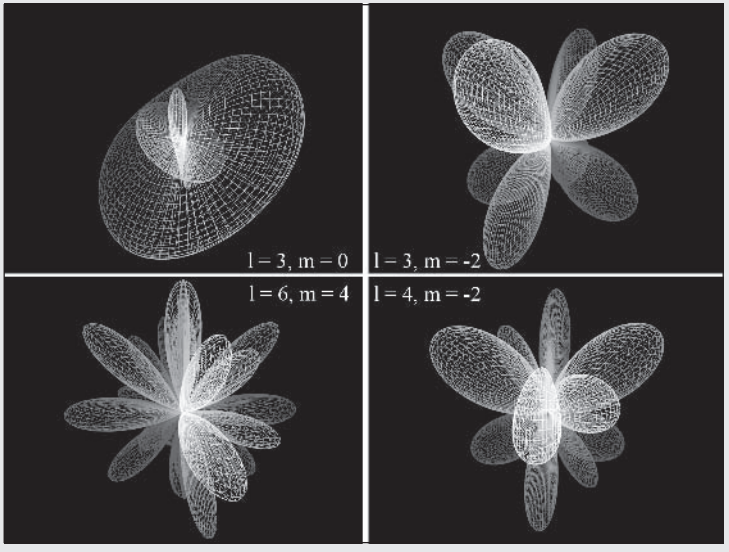


Figure 8.12. SH basis as spherical functions

In summary, Equations 5 through 8 define a set of basis functions that can be used over the unit sphere to approximate a two-dimensional function.

Spherical harmonic basis functions can be plotted as spherical functions (i.e., where the distance from the origin is a function of theta and phi). Figure 8.12 shows a few examples of such functions.

8.2.2 Projection and Reconstruction

If you go back to the example of the one-dimensional function f that I talked about previously, and now take a two-dimensional function $f(s)$ defined over the unit sphere s, the coefficients c_i become c_l^m, the spherical harmonics coefficients, and the basis function B_i becomes $y_l^m(s)$, the spherical harmonics. You then have the nth order approximated function of f, called $\tilde{f}$, reconstructed by Equation 9.

$$\tilde{f}(s) = \sum_{l=0}^{n} \sum_{m=-l}^{l} c_l^m y_l^m(s)$$

Equation 9. Approximated function using spherical harmonic projection

The c_l^m coefficients are given by Equation 10.

$$c_l^m = \int_s f(s) y_l^m(s) ds$$

Equation 10. Spherical harmonics coefficient over the unit sphere s

Using the more convenient form of spherical harmonics defined in Equation 8, we finally get the general form we will use later, as given by Equation 11. As you can see, the n^{th} order approximated function $\tilde{f}$ requires n^2 coefficients to be calculated.

$$\tilde{f}(s) = \sum_{i=0}^{n^2} c_i y_i(s)$$
$$where$$
$$i = l(l+1) + m$$

Equation 11. n^{th} order approximated function using spherical harmonics

The $\tilde{f}$ function is a *band-limited* approximation of the original function f, which in turn can be fully reconstructed by using Equation 12.

$$f(s) = \sum_{i=0}^{\infty} c_i y_i(s)$$

Equation 12. Function full reconstruction with spherical harmonics

Before going through the main properties of spherical harmonics, I will give an example of reconstruction of a function defined over the unit sphere. I will choose the same function as in [2], given by Equation 13, where θ is in the range $[0; \pi]$ and φ is in the range $[0; 2\pi]$.

$$f(\theta,\varphi) = \max(0,\cos\theta - 4) + \max(0,-4\sin(\theta - \pi)\cos(\varphi - 2.5) - 3)$$

Equation 13. Spherical function example

To calculate the c_i coefficients, you use Equation 10 where the domain s is the unit sphere. Equation 14 gives the formula for the c_i coefficients.

$$c_i = \int_0^{2\pi} \int_0^{\pi} f(\theta,\varphi) y_i(\theta,\varphi) \sin\theta \, d\theta\varphi$$

Equation 14. SH coefficient for a spherical function

Of course, you can use symbolic integration to resolve this equation, but you can also use another method that can be applied to the integration of any function: Monte Carlo integration. The advantage is that with Monte Carlo integration, you can numerically resolve any integral without actually resolving the equation itself! As you may remember from Chapter 3, Monte Carlo integration is based on stochastic sampling. In this case, it means that you need to take a number of samples evenly distributed on the unit sphere. Just divide the unit sphere in a regular grid of n x n cells. Now, instead of using a regular sampling scheme, where each sample is at the center of each cell in the grid, simply pick a random point inside each cell. You may remember from Chapter 3 that this technique is called *jittering* (used, for instance, in antialiasing techniques). Many other sampling techniques could be used, but this one will suffice.

Now that you have chosen a sampling scheme, you still need to numerically resolve the integral in Equation 14. The important thing to remember from Monte Carlo integration is the Monte Carlo estimator. It allows estimating the integral of any function f by summing the product of the value of that function at sample points by the value of the probability density function p (PDF) at the same sample points.

$$E[f(x)] = \int f(x)dx = \int \frac{f(x)}{p(x)} p(x)dx \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$

If the distribution of the sample points is uniform over the set of integration, we can simply take out the inverse of the PDF (also known as the weighting function), and the approximation of the integral of the function is just the product of the function evaluated at these sample points, divided by a weight.

$$\int f(x)dx \approx \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)} \approx \frac{1}{N} \sum_{i=1}^{N} f(x_i) w(x_i)$$

Now, going back to the rendering equation, the set of integration is the surface of the unit sphere, so the sample points need to be uniformly distributed over the unit sphere. As a result, the weighting function is constant and is the value of the surface of the unit sphere (i.e., 4π). In other words, we now have:

$$\int f(x)dx \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)w(x_i) = \frac{4\pi}{N}\sum_{i=1}^{N} f(x_i)$$

Equation 15. Monte Carlo estimator

Finally, the SH coefficient can be approximated as shown by Equation 16.

$$c_i = \frac{4\pi}{N}\sum_{j=1}^{N} f(x_j)y_i(x_j)$$

Equation 16. Monte Carlo integration for SH coefficients

Going back to the function that you would like to reconstruct using spherical harmonics, all you need to do is to construct a set of n x n samples over the unit sphere. Then, for each SH coefficient to calculate, you just loop through all these samples, evaluating the product of the function and the corresponding SH basis. At the end of the process you have the expression of a function as a set of SH coefficients.

Figure 8.13 shows the spherical plotting of the function given by Equation 13.

The 25 SH coefficients for the first five bands can be calculated by Equation 16. The results are the following: 0.399249, –0.210748, 0.286867, 0.282772, –0.315302, –0.000398, 0.131591, 0.000977, 0.093585, –0.249934, –0.000721, 0.122896, 0.304576, –0.164276, –0.000615, –0.091265, –0.101271, –0.000139, 0.132715, –0.000166, 0.344469, 0.001542, –0.041332, –0.000069, –0.153648.

Figure 8.14 shows the spherical plotting of the SH approximation of that same function, using only five bands (i.e., 25 SH coefficients), and a grid of 100 x 100 samples (i.e., 10000 samples).

You will find the program used to generate these figures on the companion CD, and explained in detail in the last section of this chapter.

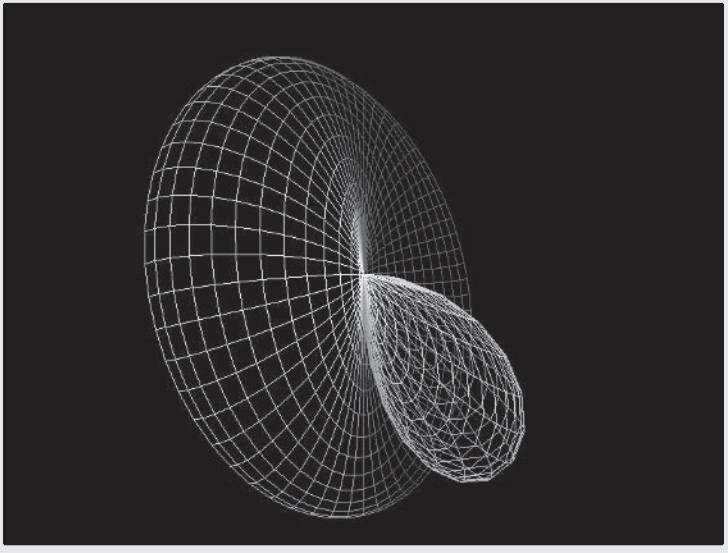


Figure 8.13. Spherical function plotted

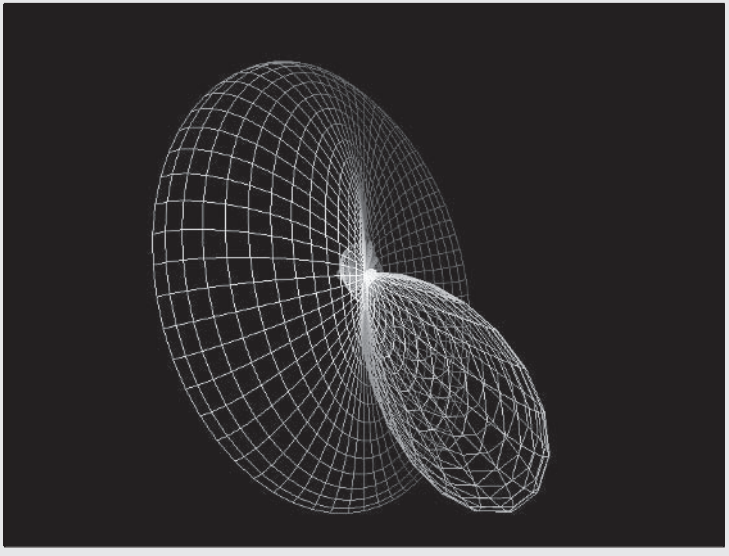


Figure 8.14. SH reconstructed function

8.2.3 Main Properties

Now that I have defined what spherical harmonics are and how to reconstruct a function using them, I am going to list their main properties.

First and foremost, they are *orthonormal*. This simply means that, as you saw earlier with the associated Legendre polynomials in one dimension, integrating two spherical harmonics that are different will give 0, whereas integrating two identical spherical harmonics will give 1, as shown in Equation 17.

$$\int_s y_i(s) y_j(s) ds = \begin{cases} 1, if : i = j \\ 0, if : i \neq j \end{cases}$$

Equation 17. Spherical harmonics are orthonormal functions.

The second property of spherical harmonics is also very interesting: Integrating the product of two band-limited functions, say f and g, over the unit sphere s is the same as

calculating the dot product of the spherical harmonics coefficients of these functions, thanks to orthonormality.

$$\int_s \tilde{f}(s)\,\tilde{g}(s)ds = \int_s \left[\sum_{i=0}^{n^2} c_i y_i(s) \sum_{i=0}^{n^2} d_i y_i(s)\right]$$

$$ds = \sum_{i=0}^{n^2} c_i d_i \int_s y_i^2(s)\,ds = \sum_{i=0}^{n^2} c_i d_i$$

Equation 18. Integration of the product of two band-limited functions

In other words, you go from a complex integral of two functions over the unit sphere to the simple addition and multiplication of numbers.

Spherical harmonics are also invariant by rotations over the unit sphere. It means that if the arbitrary rotation R transforms the function f in the function g over s, then the same rotation will transform the band-limited function $\tilde{f}$ into the function $\tilde{g}$. In mathematical terms, if:

$$g(s) = f(R(s))$$

then:

$$\tilde{g}(s) = \tilde{f}(R(s))$$

This property is crucial when using spherical harmonics with lighting and animation. Unfortunately, although very interesting, this property turns out to be very complex to implement. Our chemist friends have studied SH rotations extensively (see [3] and [4]).

8.2.4 The Spherical Harmonic Lighting Technique

In this section I explain why these complex mathematical objects are so interesting for our real-time photorealistic rendering purposes.

8.2.4.1 The Rendering Equation

In computer graphics, a single equation can describe the complete distribution of light in a scene: the rendering equation. The full equation was first introduced by Kajiya in [5] in 1986, and has been used since then to implement all sorts of global illumination algorithms. The common formulation of the rendering equation without participating media is as follows:

$$L_r(x,\vec{\omega}) = \int_\Omega f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}')(\vec{\omega}' \bullet \vec{n})d\omega'$$

This equation simply says that the outgoing radiance at point x on a surface in the outgoing direction ω is the sum of the emitted radiance and the radiance reflected from this point in this direction. The reflected radiance can be expressed as the integral over the hemisphere of incoming directions at point x of the product of the BRDF and the incident radiance (in other words, the incoming light).

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + L_r(x,\vec{\omega})$$

By using differential solid angle, the rendering equation can be rewritten as follows (see [6] or [7] for more details):

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_s f_r(x,x' \to x,\vec{\omega}) L_i(x' \to x) V(x,x') G(x,x') dA'$$

This equation expresses the reflected radiance as an integral over the set of all points in the scene S of the incident radiance leaving x' toward x and V, the visibility function between the x and x', and G, the geometry term. V is a simple test that returns 1 if x and x' can see each other, or 0 if not. The geometry term depends on the geometric relationship between the surfaces at point x and x'.

As you can see, the intensity of light leaving a given point on a surface can be expressed as an integral. This integral cannot be calculated in real time nor in symbolic terms (at least for the time being when it comes to real-time calculation). Therefore the idea behind spherical harmonic lighting is to compute this integral as a preprocessing step and project the different functions on the spherical harmonic basis. At run time, the program will use Equation 18 at all vertices to evaluate the rendering equation with the precalculated SH coefficients of the different functions involved. In other words, the preprocessing steps can be decomposed as follows:

1. Project the incoming light onto the SH basis. The incoming light needs to be expressed as a spherical function, either numerically (e.g., using HDR image light probes, where light intensity is represented as a spherical map) or symbolically (as you saw earlier in the chapter).
2. For all points (vertices) of the objects in the scene, project on the SH basis the product of the BRDF, the visibility function, and the geometric term. This product is also referred to as the transfer function. This transfer function can be simplified by making some assumptions and taking into account or not the visibility function and the geometric term. The next sections explain this step in more detail.

The run-time integral evaluation is then simply performed by computing the dot product of the SH coefficients of the transfer function and those of the incoming light. The technique to calculate the SH coefficients has already been described earlier in this chapter. You saw that it comes down to a simple sum of the product of the function to project and the spherical harmonic basis function, evaluated at sample points.

$$c_i = \frac{4\pi}{N}\sum_{j=1}^{N} f(x_j)y_i(x_j)$$

You also saw that the sample points were distributed on an n x n square grid, jittered, and mapped into spherical coordinates, by using the following equation:

$$\theta = 2\cos^{-1}(\sqrt{1-x})$$
$$\varphi = 2\pi y$$

All we need to do now is simplify the rendering equation to calculate the amount of light at the different points on the surfaces of the scene. Since we are not considering self-emitting objects, the emitted radiance is not taken into account. Also, since we are only considering Lambertian diffuse lighting — light is equally scattered in any direction, whatever the incident direction — the BRDF becomes just a

constant and can be taken out of the integral. [8] tells us that the BRDF is the ratio of the reflectance and π. We finally get:

$$L_o = \frac{\rho_d}{\pi} \int_s L_i(s) V(s) G(s) dS$$

The remainder of this section is all about using spherical harmonics to approximate this integral with different levels of simplification for the visibility and geometric terms inside this equation.

8.2.4.2 SH Diffuse Lighting

In the simplest diffuse lighting mode, the visibility term is not taken into account. In other words, you don't get shadowing, just standard diffuse lighting, but the strength is that it works the same with any type of light, not just point lights! The geometric term is the cosine term, i.e., the dot product of the normal at the point and the light source. This touches on a key point in SH lighting: The normal at a given vertex becomes encoded into the SH coefficients. There is therefore no need to store it, and any 3D API that supports Gouraud shading (i.e., interpolation of colors between the vertices of a triangle) is sufficient for SH lighting. It also means that SH diffuse lighting can be faster than most modern API shading models (e.g., Phong shading).

In diffuse unshadowed mode, the diffuse light at a given point is therefore given by the following equation:

$$L_o = \frac{\rho_d}{\pi} \int_s L_i(s) \max(\vec{n}.\vec{\omega}, 0) dS$$

The max function allows clamping values between 0 and 1. In other words, you don't get lighting at the back of a surface, and you don't create "light."

At this stage, you only need to SH project the incoming light, then SH project the cosine term (the transfer function); both use the Monte Carlo integration technique explained above. All of this is performed during a preprocessing step. At run time, the light at a given vertex is then simply calculated using the equation above and by performing the dot product of both functions' SH coefficients.

The code snippet on the following page shows how to SH project the cosine term for a given vertex, using Monte Carlo integration (see section 8.3.6 for more details on the different structures used).

You will see this code in action and explained in section 8.3.6. Figure 8.15 shows a scene in SH diffuse unshadowed mode, using the St. Peters light probe.

The program that generated this image can be found on the companion CD, and its code is explained in section 8.3.6. This scene looks similar to scenes rendered using modern API standard shading models. This is because you only consider the cosine term when SH projecting the geometric term of the rendering equation. It can nevertheless be rendered more quickly because, at run time, the normal at each vertex is not manipulated — it has been encoded in the SH coefficients. So what you get with this technique is visually similar to what you get with OpenGL and Direct3D; however, it is faster and you can use any type of light source you

want — in our case, lighting captured in a real environment (here, St. Peter's Cathedral).

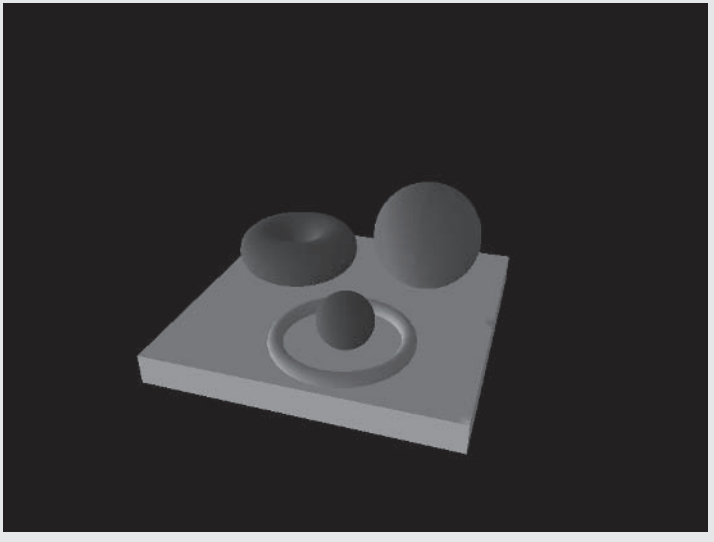


Figure 8.15. Simple scene displayed using SH lighting diffuse unshadowed mode

```
    /* Loop through all the spherical samples for current /*
    /* vertex */
    for (i = 0; i < numSamples; i++)
    {
       /* The transfer function is the cosine term */
       VecDot(sphericalSamples[i].vec, normal, H);

       /* calculate the albedo */
       RGBScale(color, ONE_OVER_PI, albedo);

       /* Add only if cosine positive */
       if (H > 0.0)
       {
          /* Calculate the coefficients */
          for (n = 0; n < numCoeffs; n++)
          {
             value = H * sphericalSamples [i].coeff[n];

             result[n].r += albedo.r * value;
             result[n].g += albedo.g * value;
             result[n].b += albedo.b * value;
          }
       }
    }

    /* Scale the SH coefficients */
    for (i = 0; i < numCoeffs; i++)
    {
       result[i].r = result[i].r * 4.0 * PI / numSamples;
       result[i].g = result[i].g * 4.0 * PI / numSamples;
       result[i].b = result[i].b * 4.0 * PI / numSamples;
    }

}
```

8.2.4.3 SH Diffuse Shadowed Lighting

In this lighting mode, the simplified rendering equation is reused, but this time the visibility term is taken into account, thus:

$$L_o = \frac{\rho_d}{\pi} \int_s L_i(s) V(s) G(s) dS$$

becomes:

$$L_o = \frac{\rho_d}{\pi} \int_s L_i(s) V(s) \max(\vec{n}.\vec{\omega}, 0) dS$$

The visibility term is going to give shadowing at the given vertex. The strength of SH lighting is that the transfer function becomes a bit more complex, but the number of SH coefficients stays the same. In other words, the level of performance will stay the same as with SH diffuse unshadowed mode, but the level of realism will increase greatly.

The new transfer function is therefore:

$$V(s) \max(\vec{n}.\vec{\omega}, 0)$$

The cosine term is kept, and $V(s)$ is a "simple" test that returns 1 if light can be seen for the current vertex or 0 if not. This is where you need to trace rays from the current vertex out into the scene toward the different triangles of the scene. If the ray intersects at least one triangle, the light is blocked. Note that you only need to know that there is one intersection; you don't really mind if there are more.

The following code snippet shows how to SH project this new transfer function for a given vertex using Monte Carlo integration.

```
/* The origin of the ray is the current vertex */
VecCopy(point, ray.origin);

/* Loop through all the spherical samples */
for (i = 0; i < numSamples; i++)
{
   /* Calculate the cosine term */
   VecDot(sphericalSamples[i].vec, normal, H);

   /* calculate the albedo */
   RGBScale(color, ONE_OVER_PI, albedo);

   /* Add only if cosine positive */
   if (H > 0.0)
   {
      /* Ray direction is the spherical sample */
      /* direction */
      VecCopy(sphericalSamples[i].vec, ray.direction);

      /* Add only if not in shadow */
      if (!intersectScene(&ray,aScene,
         faceIndex,meshIndex))
      {
         /* Calculate the coefficients */
         for (n = 0; n < numCoeffs; n++)
         {
            value = H * sphericalSamples[i].coeff[n];

            result[n].r += albedo.r * value;
            result[n].g += albedo.g * value;
            result[n].b += albedo.b * value;
         }
      }
```

```
      }
   }

   /* Scale the SH coefficients */
   for (i = 0; i < numCoeffs; i++)
   {
      result[i].r = result[i].r * 4.0 * PI / numSamples;
      result[i].g = result[i].g * 4.0 * PI / numSamples;
      result[i].b = result[i].b * 4.0 * PI / numSamples;
   }
```

This portion of code is very similar to that used in SH diffuse unshadowed mode. The intersectScene() function simply returns 1 if an intersection is found. Since you are testing for intersection with the rest of the scene for all vertices, it is important to use an acceleration technique (see Chapter 3 for examples of such techniques).

Figure 8.16 is the same scene as before with the same lighting but using SH diffuse shadowed lighting mode.

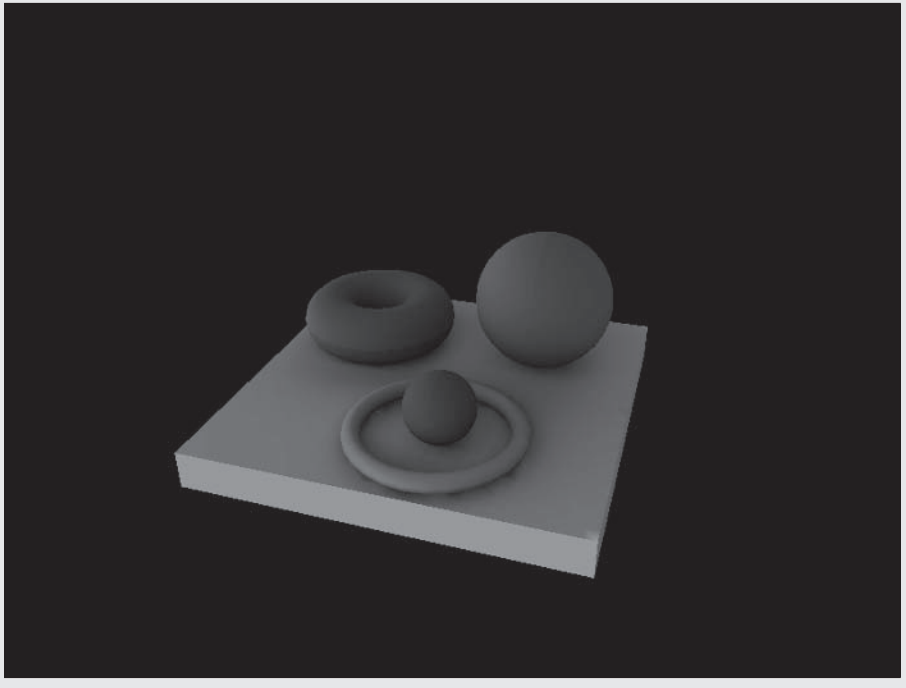


Figure 8.16. Simple scene displayed using SH lighting diffuse shadowed mode

Figure 8.17 shows the same scene rendered in the two SH lighting modes you have seen so far.

As you can see, the shadowed scene on the right is much more visually interesting and realistic, and costs the same in terms of performance.

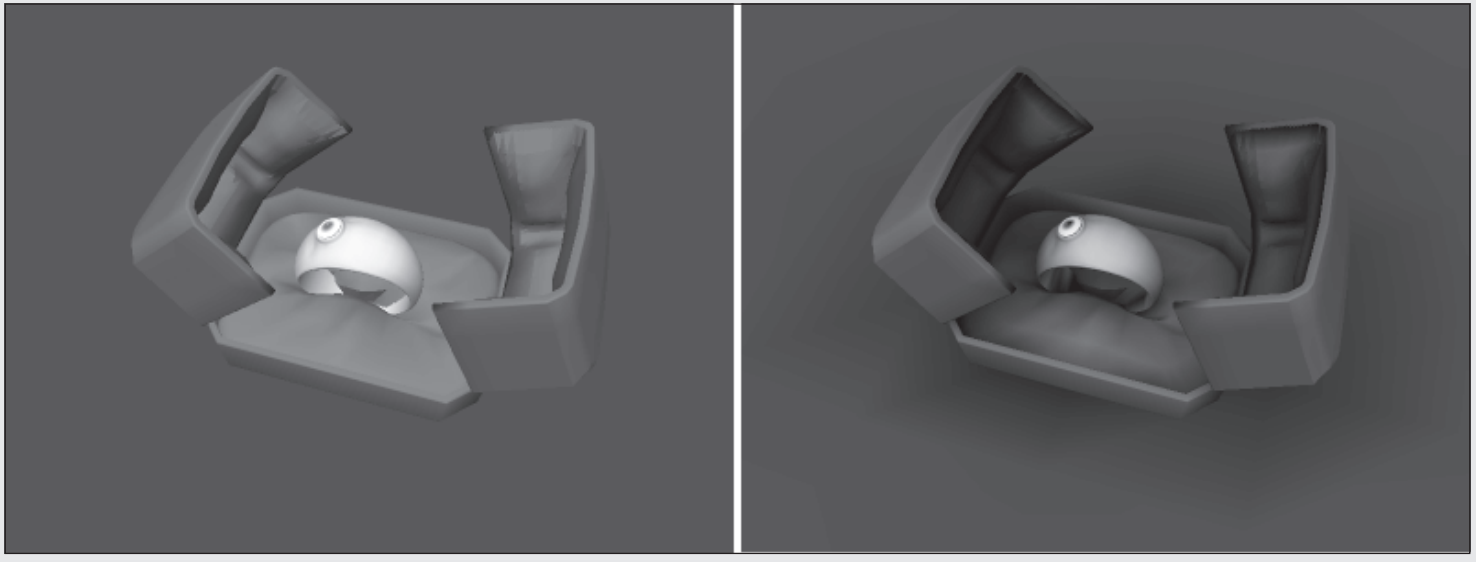


Figure 8.17. Same scene rendered in two different SH lighting modes, unshadowed (left) and shadowed (right)

8.2.4.4 SH Diffuse Shadowed Inter-Reflected Lighting

The last SH lighting mode goes even further than the two you have seen thus far. Now, you not only take into account light coming from the light sources in the rendering equation (and therefore taking into account shadowing), but you also add the contribution of light that has bounced around in the scene before reaching the current vertex. In effect, you take into account the global lighting of the scene, not just the direct lighting. This leads to a greater level of realism.

The simplified rendering equation becomes thus:

$$L_o = L_{DS} + \frac{\rho_d}{\pi}\int_S L(s)(1-V(s))\max(\vec{n}.\vec{\omega},0)dS$$

In this equation, the lighting at a vertex is the sum of the direct lighting (diffuse shadowed lighting that you saw before) and the inter-reflected lighting that comes from all other triangles in the scene and that can be seen from the current vertex. This is very similar to the radiosity technique that you will see in Chapter 10.

The inter-reflected lighting transfer function is slightly more complex to project onto the SH basis.

1. The first step calculates the diffuse shadowed coefficients, as explained before.
2. For all vertices, you trace rays until a first hit is found.
3. If the hit is found, the SH coefficients of each vertex of the triangle intersected are combined using the barycentric coordinates of the hit. This represents the contribution of light from the intersected triangle.
4. Now, the light contribution from the intersected triangle is scaled by the dot product of the ray's direction and the intersected triangle's normal. The resulting SH coefficients are stored in a temporary buffer. The cosine term is simply applied to indirect lighting.
5. Once all rays have been traced, the resulting SH coefficients are scaled using the number of samples and the Monte Carlo weighting term you saw earlier.
6. When all vertices have been processed in this manner, you get the first bounce of inter-diffuse lighting. This step can be repeated using the same technique but with the calculated SH coefficients as the new light contribution for all vertices. The process can be repeated a given number of times (or bounces).
7. The diffuse inter-reflected SH coefficients are then the sum of all bounces' SH coefficients, in addition to the diffuse shadowed coefficients (direct lighting).

This technique is even more computing intensive than the ones you saw before, but the results are visually even more realistic.

8.3 Sample Implementations in OpenGL

In this section I explain how to implement a program that illustrates spherical harmonic lighting. The OpenGL implementation features all of the necessary steps to create an SH solution. Direct3D users have access to a set of D3DX helper functions that can be used to help with SH lighting, but a Direct3D user might want to review the following code to gain a better understanding of what the D3DX functions are doing "under the hood."

8.3.1 Introduction

For this chapter, several programs were implemented to generate the different figures you've seen so far. For this section, I have chosen the C programming language for implementation, as well as OpenGL for 3D display. The different programs will compile as is provided that you have the following libraries installed on your system (note that some of these libraries are also provided on the companion CD):

- OpenGL 1.1 and above. OpenGL is bundled with most modern systems. Alternatively, the Mesa 3D library can be used (open source implementation of OpenGL). Mesa is available at [9].
- GLUT 3.6 and above. GLUT is used to handle the user interface (mouse, keyboard, window creation, etc.). GLUT is available for free for most platforms where OpenGL is present. GLUT is available at [10].
- Lib3ds. Lib3ds is used to easily load 3DS files. It is an open-source library that can be found at [11].

The goal of this section is to walk you through all the different aspects of spherical harmonic lighting, step by step, as follows:

1. The first program will plot in two dimensions the associated Legendre polynomials that are at the core of spherical harmonic calculations.
2. The second program will use the different mathematical functions from the previous program to plot in three dimensions and in spherical coordinates (that is, the distance from the origin as a function of the spherical angles θ and φ) the spherical harmonic basis functions. In this program you will see how simple it is to implement such basis functions in the C programming language.
3. With the third program, I will show how to plot a given function in three dimensions and in spherical coordinates, and the spherical harmonic approximation of that function.
4. In the fourth program, I will use C functions to load and display high-definition range images (HDR images). The functions are freely available at [12]. Here again, OpenGL will be used along with GLUT to display such images in two dimensions, with some restrictions.

5. The final program will build on top of previous programs to provide a sample implementation of spherical harmonic lighting. The lib3ds library (available at [11]) will be used to load 3DS files. The program will use HDR images to illuminate the scenes. This program will make use of intersection routines that have been developed for the sample raytracer implementation found in Appendix B, more precisely the ray/triangle and ray/box intersection routines.

It is important to note that all those programs have been implemented with simplicity and readability in mind. For instance, all mathematical functions have been coded in the simplest way possible (almost as direct translation of their corresponding mathematical formulae).

NOTE:

Applications written solely to help visualize SH components have not been ported to Direct3D.

8.3.2 Associated Legendre Polynomials 2D Display

The first program that we are going to develop for this chapter simply plots associated Legendre polynomials in two dimensions. You will find the full source code, project files, and precompiled binaries on the CD in the following directory: Code\Chapter 08 - SH Lighting LegendrePolynomials.

8.3.2.1 Design

The design of this program is very simple. It consists of a single source file containing all the code. The core of the program is the display routine (display()), which computes and plots the associated Legendre polynomials by calling the drawALPStd() function. This function, in turn, computes and draws a given associated Legendre polynomial by calling the ALPStd() function several times with appropriate parameters. The drawLine() function is used to draw the vertical and horizontal axes. The range of display along the x-axis (i.e., [–1;1]) is divided into 128 sample points for which the polynomials are calculated. All successive points calculated are then simply connected by a line. The core of the routine is therefore simply a loop over the horizontal interval of display as shown in the following code snippet:

```
glColor3f(r, v, b);
glBegin(GL_LINE_STRIP);
for (i = 0; i <= samples; i++)
{
     glVertex2f(x, ALPStd(x, l, m));
     x += step;
}
glEnd();
```

For the sake of simplicity, I have limited the display to the first four bands. It is trivial to change that, or better yet, make the program flexible enough to accept the band level as an input parameter.

Figure 8.18 summarizes the different functions, global variables, and macros of the program.

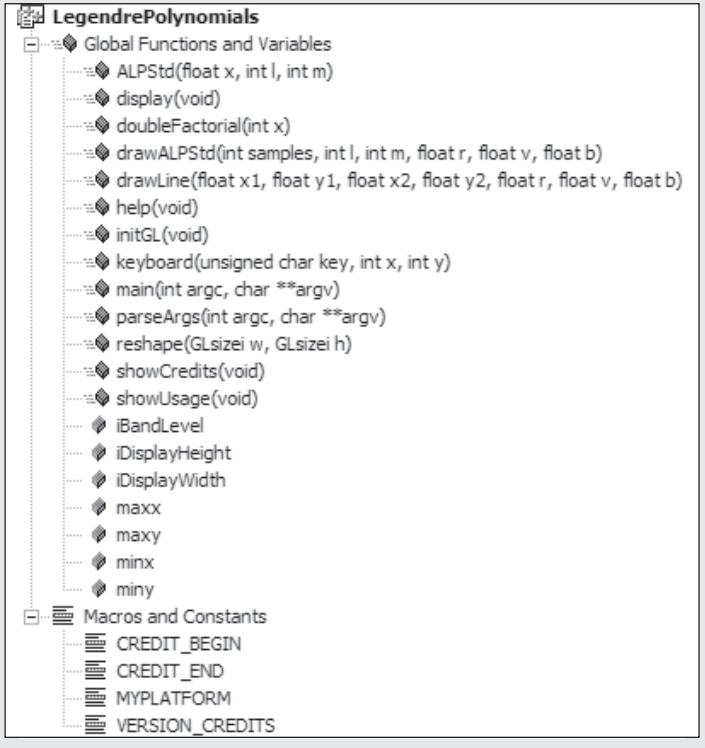


Figure 8.18. LegendrePolynomials program functions, variables, and macros

8.3.2.2 Implementation

As with any other program in this section, the implementation uses OpenGL for display, as well as GLUT for the creation of the user interface and to handle keyboard input. The user interface has been kept to the bare minimum though: A simple window displays curves in 2D, with only one keyboard input to exit the program. The mouse can of course be used to resize and move the window as desired.

In this program, a core spherical harmonics calculation routine has been implemented: the function that calculates the value of the associated Legendre polynomial for a given value, a given band index l, and a given index m (where $l \in \mathbb{N}, 0 \leq m \leq l$). In this implementation, I have simply used the recursive formulae given in the first section of this chapter. Equation 19 gives these formulae:

$$(l-m)P_l^m(x) = x(2l-1)P_{l-1}^m(x) - (l+m-1)P_{l-2}^m(x)$$
$$P_m^m(x) = (-1)^m(2m-1)!!(1-x^2)^{m/2}$$
$$P_{m+1}^m(x) = x(2m+1)P_m^m(x)$$

Equation 19. Associated Legendre polynomial recursive formulae

The following code snippet is the C implementation of these formulae.

```
double ALPStd(float x, int l, int m)
{
      if (l == m) return (pow(-1, m) * doubleFactorial(2
           * m - 1) * pow(sqrt(1 - x * x), m));

      if (l == m + 1) return (x * (2 * m + 1) * ALPStd(x,
           m, m));

      return ((x * (2 * l - 1) * ALPStd(x, l - 1, m) -
           (l + m - 1) * ALPStd(x, l - 2, m)) / (l - m));
}
```

Note that you can easily find faster nonrecursive versions of this code. The doubleFactorial() function calculates the double factorial of a given integer; what follows is an implementation of this function.

```
int doubleFactorial(int x)
{
      int result;

      if (x == 0 || x == -1) return (1);

      result = x;
      while ((x -= 2) > 0) result *= x;

      return result;
}
```

These functions will be used later for the implementation of the spherical harmonic functions. As I said earlier, the code is pretty simple and not fully optimized.

8.3.2.3 Command-line Parameters

The LegendrePolynomials program accepts different command-line parameters, which are summarized in the following table.

Switch	*Parameter(s)*	*Description*
-r	float float	Min and max values displayed on the y-axis
--resolution	float float	Min and max values displayed on the y-axis
-h	integer	Height of the display window
--height	integer	Height of the display window
-w	integer	Width of the display window
--width	integer	Width of the display window
-?	none	Display a help message
--help	none	Display a help message

8.3.2.4 Results

Figure 8.19 shows a screen shot of the LegendrePolynomials application plotting the first four bands of associated Legendre polynomials using the following command line:

```
LegendrePolynomials.exe -r -15 6.
```

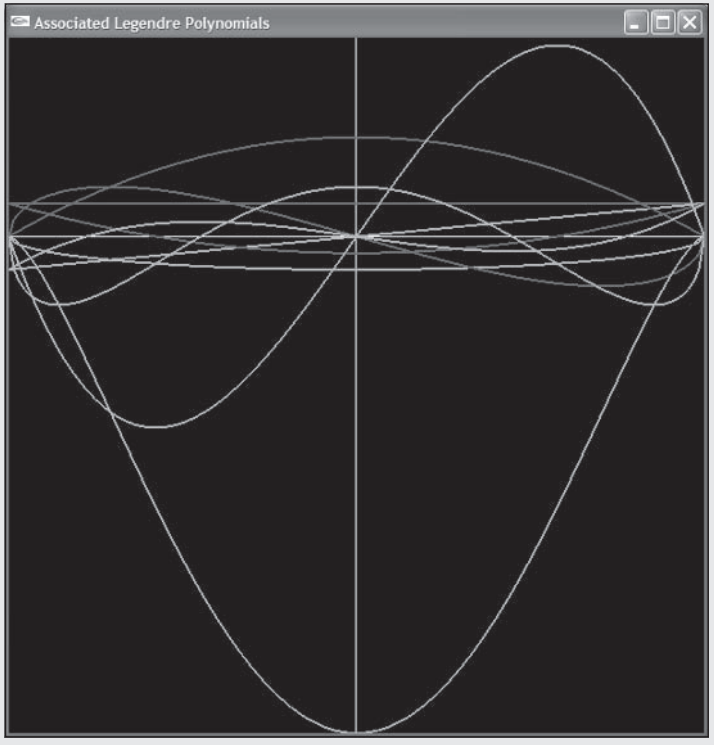


Figure 8.19. LegendrePolynomials application running

8.3.3 Spherical Harmonics 3D Display

The second program that we are going to develop for this chapter will plot the spherical harmonic basis functions in 3D. The plotting is done using spherical coordinates (i.e., spherical plotting where the distance from the origin is a function of the spherical angle θ and φ). This program uses the core associated Legendre polynomial calculation routines from the previous program.

You will find the full source code, project files, and precompiled binaries on the CD in the following directory: Code\Chapter 08 - SH Lighting SimpleSH.

8.3.3.1 Design

For this program, I have put all the spherical harmonic related functions in a single file, called sh.c, with its corresponding header file sh.h. The logic of the program itself is now coded in another file, called SimpleSH.c, with its corresponding header file SimpleSH.h. This separation becomes necessary as we are going to keep adding functions related to spherical harmonic calculations.

The associated Legendre polynomial calculation routines have been taken as is from the previous project. However, the following core spherical harmonic routines have been added:

- factorial(): This function simply computes the factorial of a given integer. It is used in the computation of the scaling factor K from the spherical harmonics equation (see Equation 5). The factorial of the integer x is simply:

$$x! = x \times (x-1) \times (x-2) \times \ldots \times 2 \times 1$$

- evaluateK(): This function computes the value of the scaling factor K for the given band index l and index m. The formula used is given in Equation 5.
- evaluateSH(): This is the core function for computing the value of a spherical harmonic. The input parameters are the band index l and index m, as well as the two spherical angles θ and φ. This function uses the formulae given in the first section of this chapter.

The program uses the keyboard to change the way spherical harmonic functions are plotted in 3D. It is possible to scale the function currently plotted, change the way it is plotted (wireframe, dots, or plain surface), or even change the number of triangles used to plot that function.

The core of the program is the display routine (displayScene()), which sets up the view in 3D and calls the renderSH() function. The renderSH() function then computes and plots a given spherical harmonic basis function by calling the evaluateSH() function. For the sake of simplicity, I haven't used any OpenGL optimization, such as display lists, as this would have made the code a bit more obscure. In any case, the program runs reasonably well on first-generation 3D cards such as NVIDIA TNT 16.

Figure 8.20 summarizes the different functions and global variables of the program.

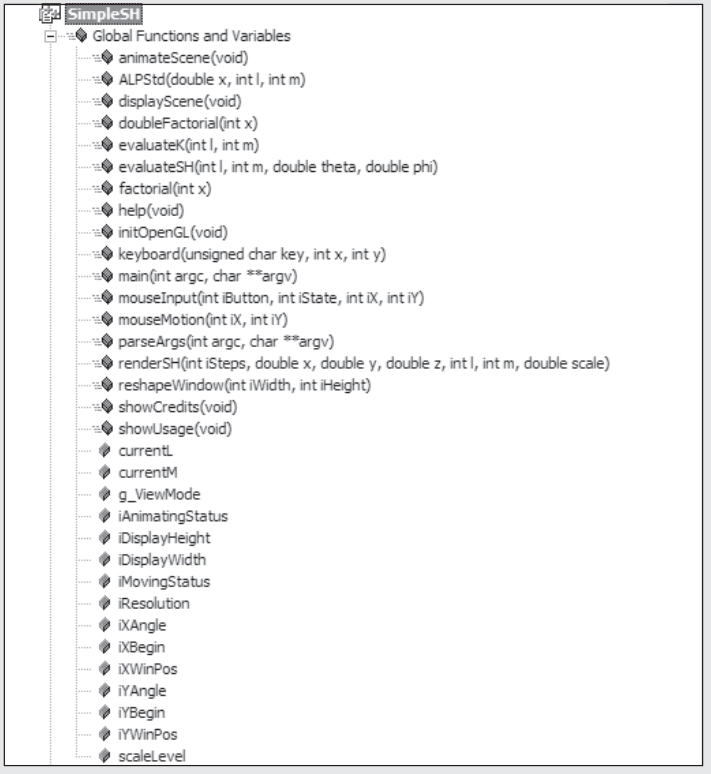


Figure 8.20. SimpleSH functions and global variables

8.3.3.2 Implementation

GLUT is used here as well to handle the user interface. Unlike the previous program, the mouse and keyboard are now used to change the view, rotate the camera around the function being plotted, change the resolution of the plotting, zoom in and out, etc.

The computation of a given spherical harmonic function is split into two functions: evaluateK() and evaluateSH(). The following C code shows how simple it is to implement spherical harmonic calculation.

```
double evaluateSH(int l, int m, double theta, double phi)
{
      double SH = 0.0;

      if (m == 0)
      {
            SH = evaluateK(l, 0) * ALPStd(cos(theta),
                  l, 0);
      }
      else if (m > 0)
      {
            SH = SQRT2 * evaluateK(l, m) * cos(m * phi) *
                  ALPStd(cos(theta), l, m);
      }
      else
      {
            SH = SQRT2 * evaluateK(l, -m) * sin(-m * phi)
                  * ALPStd(cos(theta), l, -m);
      }

      return SH;
}
```

The code to calculate the scaling factor K is also very easy to implement.

```
double evaluateK(int l, int m)
{
      double result;

      result = ((2.0 * l + 1.0) * factorial(l - m)) /
            (4 * PI * factorial(l + m));

      return sqrt(result);
}
```

Finally, here is a nonrecursive implementation to calculate the factorial of an integer.

```
int factorial(int x)
{
      int result;

      if (x == 0 || x == -1) return (1);

      result = x;
      while ((x -= 1) > 0) result *= x;

      return result;
}
```

As you can see, the complex mathematic formulae behind spherical harmonics can be easily implemented in C (or any other language). Note that it is always possible to further optimize this type of code.

The core rendering routine is renderSH(). It simply loops through the spherical angles θ and φ, and calculates the value of the spherical harmonic function for those angles. It then uses quadrilaterals (polygons made of four vertices) to plot the function.

8.3.3.3 Command-line Parameters

The SimpleSH program accepts different command-line parameters, which are summarized in the following table.

Switch	*Parameter(s)*	*Description*
-l	integer	Band level l
--ll	integer	Band level l
-m	integer	Value of $m\left(l \in \mathbb{N}, 0 \le m \le l\right)$
--mm	integer	Value of $m\left(l \in \mathbb{N}, 0 \le m \le l\right)$
-h	integer	Height of the display window
--height	integer	Height of the display window
-w	integer	Width of the display window
--width	integer	Width of the display window
-?	none	Display a help message
--help	none	Display a help message

8.3.3.4 Keyboard Mapping and Mouse Usage

The keyboard mapping and mouse usage of this program are summarized in the following tables.

Key	*Description*
l	Decrease the value of the band index l
L	Increase the value of the band index l
m	Decrease the value of the m index
M	Increase the value of the m index
r	Decrease the resolution of the model (the number of triangles used in the plotting); minimum value is 16
R	Increase the resolution of the model
z	Decrease the scale of the model; minimum value is 0.1
Z	Increase the scale of the model
a/A	Toggle animation on/off
y/Y	Switch between the different display modes: point, triangle, and fill
q/Q	Exit the program

Mouse Action	*Description*
Left-click and drag	Rotates the camera around the function, thus giving the impression of rotating the function
Right-click	Toggles animation on/off

8.3.3.5 Results

Figure 8.21 shows a screen shot of the SimpleSH application plotting the spherical harmonic for $m = 2$ and $l = 2$, using the following command line:

```
SimpleSH -l 2 -m 2.
```

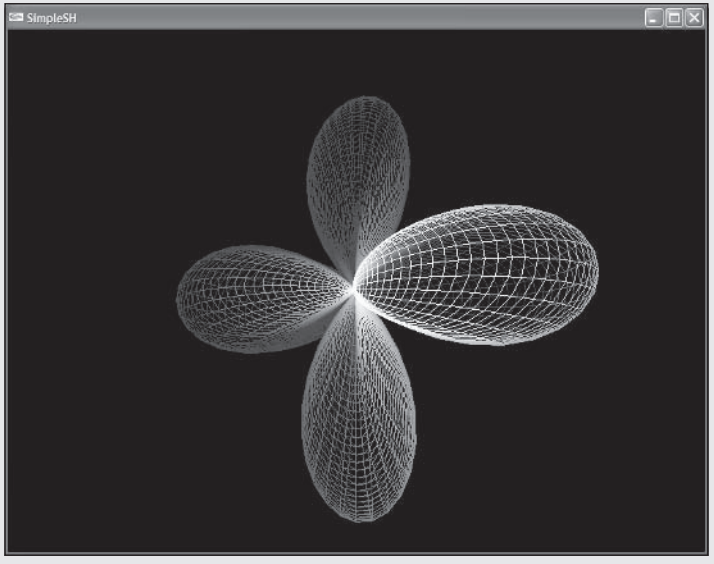


Figure 8.21. SimpleSH application running

8.3.4 Function Approximation and Reconstruction Using Spherical Harmonics

Now that we have built the core of the spherical harmonic basis functions computation, let's go a bit further and use them to reconstruct a given mathematical function. The goal here is to compare a given spherical function (e.g., a function describing the spherical distribution of a light source) and its low-frequency spherical harmonic approximation.

For this program, you will also find the full source code, project files, and precompiled binaries on the CD, in the following directory: Code\Chapter 08 - SH Lighting SimpleSHLight.

8.3.4.1 Design

This program plots functions in three dimensions using spherical coordinates. As a result, a lot of the code from the previous program has been reused. The spherical harmonic functions from sh.c have been reused for this project, and two new functions have been added — one to calculate spherical jittered samples (stratified sampling) and another to project a given function on the spherical harmonic basis:

- sphericalStratifiedSampling() performs spherical stratified sampling as explained previously in section 8.2.2.
- SHProjectSphericalFunction() calculates the spherical harmonic coefficients of the input functions.

As we will have to use vectors and other mathematical entities, it is now time to define more advanced data types for easier data manipulation. First, I define a structure

holding the x, y, and z Cartesian coordinates of a vector. I'll simply call this structure SHVector3d. I will use this type of entity to manipulate both 3D vectors and 3D points. I also define a new data type corresponding to the stratified samples used in all spherical harmonic calculations. This structure contains the spherical coordinates on the unit sphere (i.e., $(\theta, \varphi, r = 1)$), the direction of the sample (i.e., the corresponding Cartesian coordinates (x, y, z)), and the spherical harmonic coefficients of that sample.

The program plots in 3D a function and its low-frequency spherical harmonic approximation for any band index *l*, as well as the stratified samples that have been calculated and used in all computations. As with the previous program, it is also possible to use the keyboard to scale the function plotted, change the way it is plotted (wireframe, dots, or plain surface), or even change the number of triangles used to plot that function.

The program first initializes the stratified samples that will be used in all spherical harmonic calculations by calling the sphericalStratifiedSampling() function. It then calculates the spherical harmonic coefficients of the mathematical spherical function coded in the testLight() function. The actual display is coded in the display routine (displayScene()), which sets up the view in 3D and calls the appropriate function to render the selected item: the original spherical function (renderLightFunction()), its spherical harmonic approximation (renderSHApprox()), or the stratified samples (renderStratSamples()). Here again, for the sake of simplicity, I haven't used any OpenGL optimization and tried to keep the program's code easy to read.

Figure 8.22 summarizes the different functions and global variables of the program.

Figure 8.22. SimpleSHLight functions and global variables

8.3.4.2 Implementation

GLUT is used to handle the user interface in the same way as with the previous programs.

The following C code snippet shows the declaration of the SHVector3d structure.

```
/* Vector structure */
typedef struct
{
   double x, y, z;
} SHVector3d;
```

The following code snippet shows the declaration of the SHSample structure.

```
/* SH Sample structure */
typedef struct
{
   SHVector3d sph;
   SHVector3d vec;
   double *coeff;
} SHSample;
```

The coeff member of this structure is a pointer to double-precision floats so that memory can be dynamically allocated based on the number of bands needed.

The routine that initializes the SH samples used in all computations of the program is sphericalStratifiedSampling(). It has been implemented following the algorithm explained in section 8.2.2.

```
void sphericalStratifiedSampling(SHSample *samples,
   int sqrtNumSamples, int nBands)
{
   /* Indexes */
   int a, b, index, l, m;

   /* Loop index */
   int i = 0;

   /* Inverse of the number of samples */
   double invNumSamples = 1.0 / sqrtNumSamples;

   /* Cartesian and spherical coordinates */
   double x, y, theta, phi;

   /* Loop on width of grid */
   for (a = 0; a < sqrtNumSamples; a++)
   {
      /* Loop on height of grid */
      for (b = 0; b < sqrtNumSamples; b++)
      {
         /* Jitter center of current cell */
         x = (a + rnd()) * invNumSamples;
         y = (b + rnd()) * invNumSamples;

         /* Calculate corresponding spherical angles*/
         theta = 2.0 * acos(sqrt(1.0 - x));
         phi = 2.0 * PI * y;

         /* Sample spherical coordinates */
         samples[i].sph.x = theta;
         samples[i].sph.y = phi;
         samples[i].sph.z = 1.0;

         /* Sample normal */
         samples[i].vec.x = sin(theta) * cos(phi);
         samples[i].vec.y = sin(theta) * sin(phi);
```

```
            samples[i].vec.z = cos(theta);

            /* Calculate SH coefficients of current sample */
            for (l = 0; l < nBands; ++l)
            {
               for (m = -l; m <= l; ++m)
               {
                  index = l * (l + 1) + m;
                  samples[i].coeff[index] = evaluateSH(l, m,
                                                theta, phi);
               }
            }

            ++i;
         }
      }
   }
```

The spherical function used in this program is coded in the testLight() function, which computes the value of that function based on the input angles θ and φ.

```
/* Computes the value of the function based on the input
   angles */
double testLight(double theta, double phi)
{
   return (MAX(0, 5 * cos(theta) - 4) + (MAX(0,
      -4*sin(theta - PI) * cos(phi-2.5)-3)));
}
```

I chose the same function as in [2] (given by Equation 13) to test that my own implementation of SH computation routines were giving good results!

The core rendering routines are renderLightFunction() (for the original function) and renderSHApprox() (for the approximated function). These functions are very similar — they simply loop through the spherical angles θ and φ, and calculate the value of the original function or the approximated function for those angles. They both use quadrilaterals for plotting.

The function that renders the spherical samples is renderStratSamples(). It loops through all the spherical harmonic samples and uses their vec member for plotting.

```
void renderStratSamples(double x, double y, double z,
   double scale)
{
   int i;

   glPushMatrix();
   glTranslatef(x, y, z);
   glScalef(scale, scale, scale);

   glBegin(GL_POINTS);

   for (i = 0; i < iNumSamples; i++)
   {
   glNormal3f(samples[i].vec.x, samples[i].vec.y,
      samples[i].vec.z);
   glVertex3f(samples[i].vec.x, samples[i].vec.y,
      samples[i].vec.z);
   }

   glEnd();
   glPopMatrix();
}
```

The function that projects a given mathematical function onto the spherical harmonic basis, as explained in detail in the previous section of this chapter, is SHProjectSphericalFunction(). It takes the mathematical function to project as an input parameter and returns its spherical harmonic coefficients. This input mathematical function is simply a pointer to the C routine that implements the actual computation. It is therefore easy to extend the program to approximate any other mathematical functions by "pointing" to the right routine! The following code is how this routine was implemented in the program; the resulting SH coefficients are stored in the result array of double-precision floats.

```
void SHProjectSphericalFunction(SphericalFunction mySPFunc,
   SHSample *samples, double *result, int numSamples,
   int numCoeffs)
{
   /* Weighting factor */
   double dWeight = 4.0 * PI;
   double factor = dWeight / numSamples;

   /* Spherical angles */
   double theta, phi;

   /* Loop indexes */
   int i, n;

   /* Loop through all samples */
   for (i = 0; i < numSamples; i++)
   {
      /* Get current sample spherical coordinates */
      theta = samples[i].sph.x;
      phi = samples[i].sph.y;

      /* Update SH coefficients */
      for (n = 0; n < numCoeffs; n++)
      {
      result[n] += mySPFunc(theta, phi) *
         samples[i].coeff[n];
      }
   }

   /* Scale SH coefficients */
   for (i = 0; i < numCoeffs; i++)
   {
      result[i] = result[i] * factor;
   }
}
```

The SphericalFunction function type is defined as follows:

```
typedef double (*SphericalFunction)(double theta,
   double phi);
```

8.3.4.3 Command-line Parameters

The SimpleSHLight program accepts different command-line parameters, which are summarized in the following table.

Switch	*Parameter(s)*	*Description*
-s	integer	The size of the grid of stratified samples
--samples	integer	The size of the grid of stratified samples
-b	integer	The number of bands used to approximate the spherical function
--bands	integer	The number of bands used to approximate the spherical function
-h	integer	Height of the display window
--height	integer	Height of the display window
-w	integer	Width of the display window
--width	integer	Width of the display window
-?	none	Display a help message
--help	none	Display a help message

8.3.4.4 Keyboard Mapping and Mouse Usage

The keyboard mapping and mouse usage of this program are summarized in the following tables.

Key	*Description*
r	Decrease the resolution of the model; minimum value is 16
R	Increase the resolution of the model
z	Decrease the scale of the model; minimum value is 0.1
Z	Increase the scale of the model
t/T	Cycle through the different rendering modes: the original function, its spherical harmonic approximation, and the stratified samples
a/A	Toggle animation on/off
y/Y	Cycle through the different display modes: point, triangle, and fill
q/Q	Exit the program

Mouse Action	*Description*
Left-click and drag	Rotates the camera around the function, thus giving the impression of rotating the function
Right-click	Toggles animation on/off

8.3.4.5 Results

Figure 8.23 shows a screen shot of the SimpleSHLight application plotting 10,000 spherical samples.

Figure 8.24 shows a screen shot of the application plotting the original spherical function.

Finally, Figure 8.25 shows a screen shot of the application plotting the spherical harmonic approximation of that same function using only five bands (25 coefficients).

As you can see, by only using five bands (and thus only 25 coefficients), the low-frequency approximated function is pretty close to the original function. We can notice some slight "perturbations" below the main lobe of the approximated function. Using more bands tend to reduce these "perturbations."

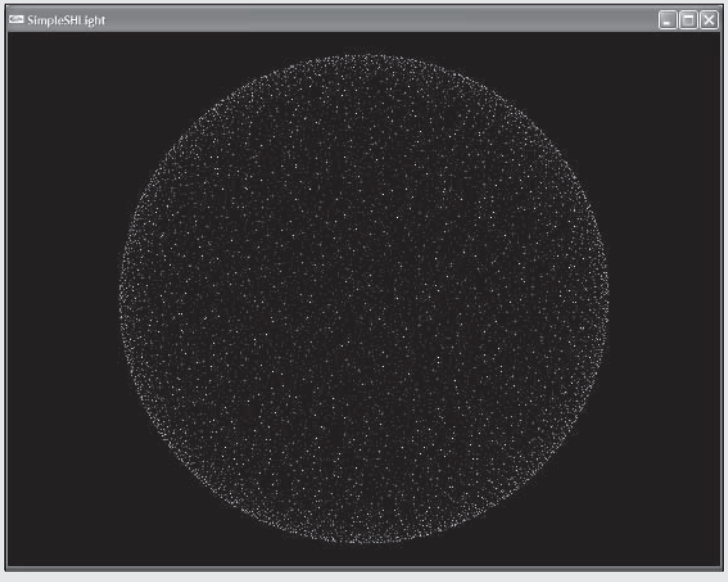


Figure 8.23. SimpleSHLight application plotting 10,000 spherical samples

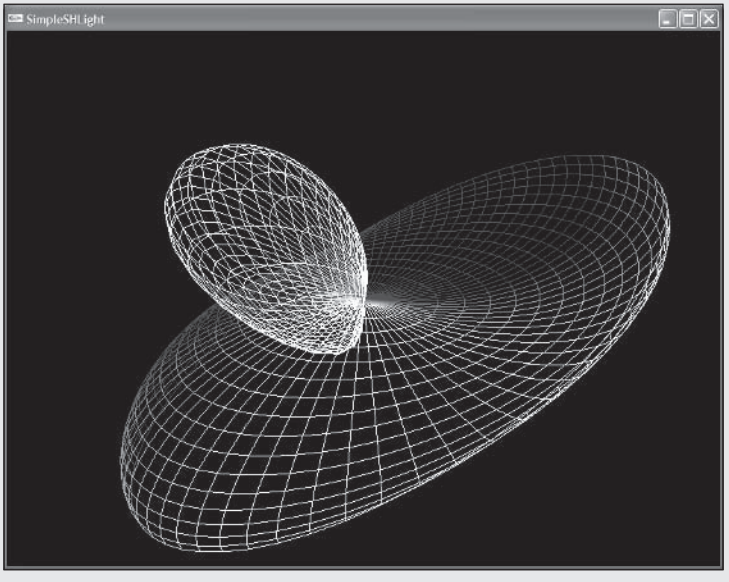


Figure 8.24. SimpleSHLight application plotting the original function

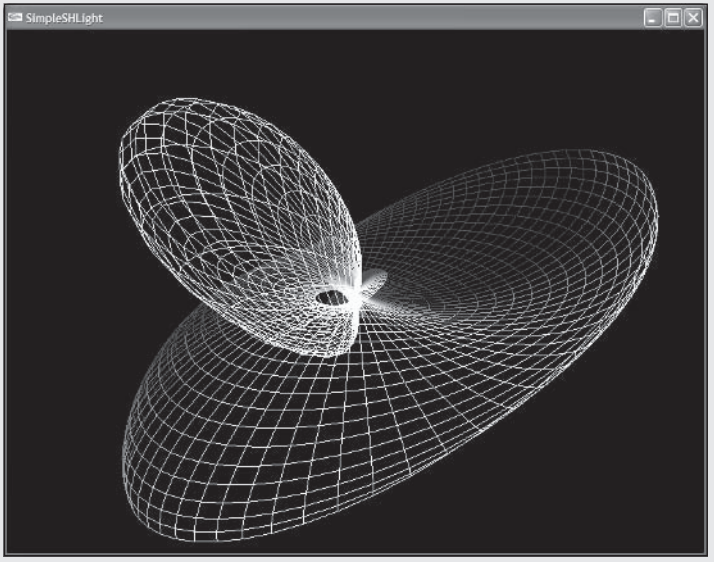


Figure 8.25. SimpleSHLight application plotting an SH approximation of a function

As you will notice, when executing the program, the approximated function is much slower to plot than the original function. This is because the original function plotting uses only one function call (testLight()) with a simple calculation, whereas the approximated function plotting needs to reconstruct the function by computing the dot product between its coefficients and the spherical harmonic basis functions. The goal of the program is to show that any function can be approximated efficiently by using very few coefficients. You will see in the next section that spherical harmonics are really efficient in the context of real-time rendering with arbitrary lighting.

8.3.5 HDR Images Loading and Display

We have now implemented most of what is needed for spherical harmonic lighting. Before going into details on how to actually implement it, I will show you how to load and display, with some restrictions, HDR images in the four-byte RGBE file format (originally developed by Greg Ward). The goal is to experiment with this file format. I highly recommend that you download an HDR image manipulation program such as HDRView or HDRShop, which can be found at [13].

In the next section, HDR images will be used as the numerical data set of the incoming light in the environment. They will be SH projected as explained in section 8.2.2, and the rendering equation will be evaluated at run time with their SH coefficients.

For this program, you will find the full source code, project files, and precompiled binaries on the CD, in the following directory: Code\Chapter 08 - SH Lighting HDRISHLighting. You will also find some HDR light probes from [13] on the CD, in the \Code\Probes directory.

8.3.5.1 Design

This program is fairly simple: It loads an HDR image in the four-byte RGBE file format (i.e., with the .hdr file extension), converts this float image into an OpenGL texture by clamping the values in the range [0,1] and converting them into bytes, and then displays this texture in two dimensions. I

have used C routines to load .hdr files that are available for free at [12].

The program consists of two files: the main file, GLHDRViewer.c, which contains the logic of the program, and rgbe.c, which contains the routines to manipulate HDR image files. These files will be used in the next program to load different light probes. On startup, the program reads the command-line parameters including, most importantly, the name of the image to display. It then calls the makeTexture-Image() function to load the file and create the OpenGL texture buffer. After setting up the two-dimensional view, the program simply displays the image by using a textured quadrilateral.

Because OpenGL texture dimensions have to be a power of 2, this program will only display HDR images whose dimensions are a power of 2; for other dimensions, the texture won't appear on the screen but will load. There are many resources on the Internet that show how to load images as OpenGL textures whose dimensions are not a power of 2. Look at [14] for such resources.

Figure 8.26 summarizes the different functions and global variables of the GLHDRViewer program.

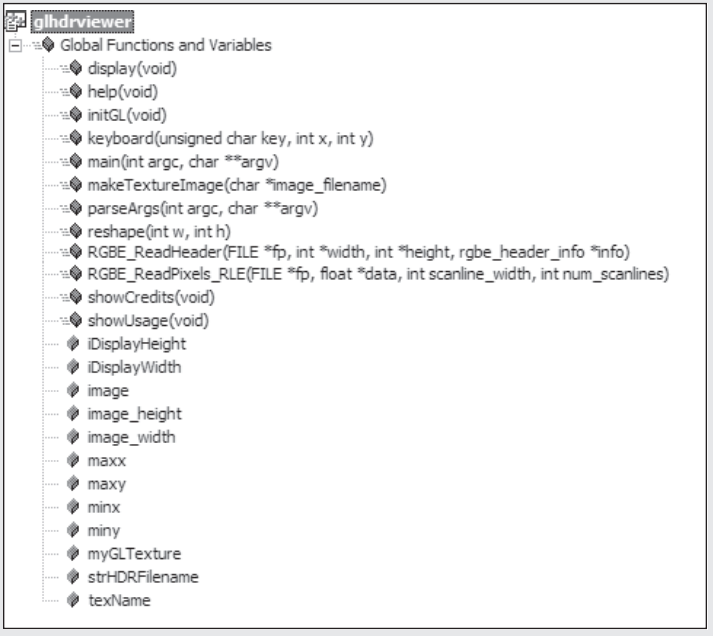


Figure 8.26. GLHDRViewer project functions and global variables

8.3.5.2 Implementation

The implementation of this program is pretty straightforward. OpenGL is used to draw a textured quadrilateral in two dimensions using the following C code snippet:

```
void display(void)
{
   /* Clear screen */
   glClear(GL_COLOR_BUFFER_BIT);

   /* Draw a textured quad between [-1,1]x[-1,1] */
   glBindTexture(GL_TEXTURE_2D, texName[0]);
   glBegin(GL_QUADS);
   glTexCoord2f(0.0, 1.0); glVertex3f(-1.0, -1.0, 0.0);
   glTexCoord2f(0.0, 0.0); glVertex3f(-1.0, 1.0, 0.0);
   glTexCoord2f(1.0, 0.0); glVertex3f(1.0, 1.0, 0.0);
   glTexCoord2f(1.0, 1.0); glVertex3f(1.0, -1.0, 0.0);
   glEnd();
   glFlush();

   /* Swap front and back buffers */
   glutSwapBuffers();
}
```

The core HDR image loading routine, makeTextureImage(), calls RGBE_ReadHeader() and RGBE_ReadPixels_RLE() from the rgbe.c file. I have modified the RGBE_ReadHeader() routine to allow it to read all light probes from [13].

8.3.5.3 Command-line Parameters

The GLHDRViewer program accepts different command-line parameters, which are summarized in the following table.

Switch	*Parameter(s)*	*Description*
-i	filename	The name of the HDR image to display
--input	filename	The name of the HDR image to display
-h	integer	Height of the display window
--height	integer	Height of the display window
-w	integer	Width of the display window
--width	integer	Width of the display window
-?	none	Display a help message
--help	none	Display a help message

8.3.5.4 Results

Figure 8.27 shows a screen shot of the GLHDRViewer application displaying the rnl_probe.hdr image resized to 512x512 using the HDRShop program.

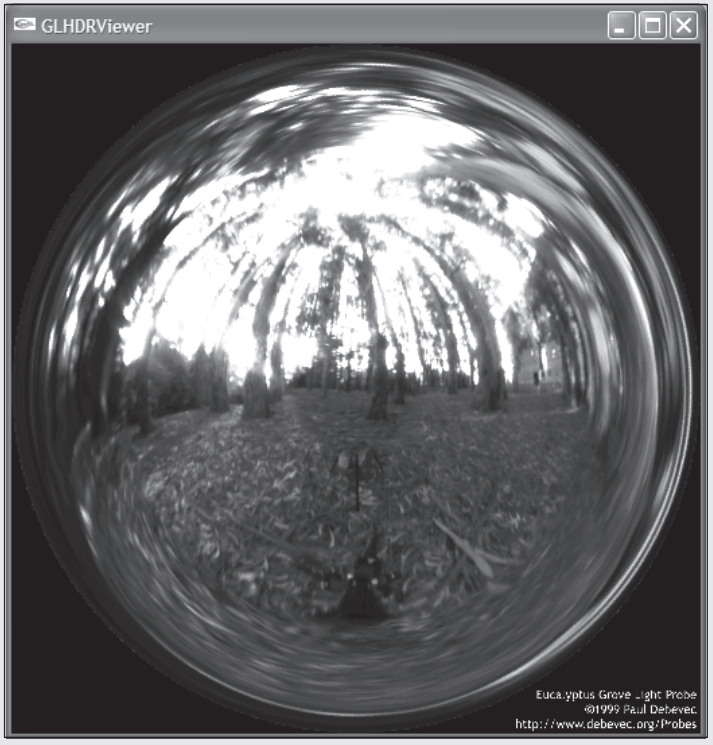


Figure 8.27. GLHDRViewer program displaying the rnl_probe.hdr file

8.3.6 Spherical Harmonic Lighting Program

At this stage, you have all that is needed to actually write an interactive program displaying three-dimensional scenes with spherical harmonic lighting using HDR light probes as light sources. In this section, I explain the implementation of an SH preprocessor and lighting program. This implementation should be considered as a basis for future work. It hasn't been optimized, and some features haven't been implemented (you'll see which ones and why).

The full source code, project files, and precompiled binaries are on the CD, in the following directory: Code\Chapter 08 - SH Lighting HDRISHLighting.

8.3.6.1 Design

This program is the synthesis of the previous programs: It uses the spherical harmonic calculation routines and the HDR image loading routines, and it also reuses the ray/object intersection routines that have been implemented for the sample raytracing program from Chapter 3.

The 3D vector and SH sample structures introduced will be reused as is. I introduce new structures to easily manipulate different entities:

- SHRay: Structure representing a ray with two members — its origin and its direction — both being of type SHVector3d.
- SHRGBColor: Structure representing an RGB color whose three members are double-precision floats.
- SHCoeff: Structure that holds SH coefficients for each color channel. This is useful for colored lighting.

- SHMaterial: This structure contains the basic properties of a surface for use with the Phong shading model: ambient, diffuse, and specular colors, and a specular coefficient. The last member is a unique identifier referenced by 3D objects to which this surface is attached (see the SHFace3d discussion below). Note that for SH lighting, only the diffuse color property of the surface is needed. The other properties will be used to render objects with the standard OpenGL shading model.
- SHMaterialLookup: This structure is a simple lookup table for material structures. This is necessary because the 3DS file format identifies materials by name, not by a unique identifier (e.g., using an integer). The structure has two members: a string representing the name of a material and an integer representing its unique identifier that corresponds to the unique identifier member of the SHMaterial structure. This identifier will serve as an index for the array of materials of the scene.
- SHFace3d: This structure holds the characteristics of a given triangle in the scene. Its members are:
  - A material identifier referencing the material attached to this triangle.
  - A vector representing the normal of the triangle.
  - An array that contains the three indices of the three points making the triangle. These indices are used to get the points from the array of points of the object to which this triangle is attached.
  - The constant of the embedding plane used to test intersection with rays efficiently.
  - The two indices of the axis determining the plane on which the triangle is projected during intersection tests with rays.
- SHMesh3d: This structure holds the different characteristics of a given object of the scene. Its members are:
  - The number of points of this object.
  - The number of triangles of this object.
  - Two vectors representing the extents of the bounding box of the object. This is used to speed up the ray/object intersection tests with the object.
  - An array of triangles. A pointer is used to dynamically allocate the necessary memory.
  - An array of the vertices of the object. A pointer is used as well.
  - An array of normal vectors at the different vertices. A pointer is used here again.
  - An array for the SH coefficients for each transfer functions: unshadowed, shadowed, and inter-reflected.
- SHCamera3d: This structure holds the different properties of a pinhole camera. Its members are the position and the target points, both being of type SHVector3d, and the field of view, which is of type float.
- SHLight3d: This structure is not used in the SH lighting mode, but has been declared to easily extend the OpenGL rendering mode of the scene to load and display.

- SHScene3d: This is the main structure used in the program. It contains everything needed to display a 3D scene:
  - The number of cameras in the scene
  - The number of lights
  - The number of materials
  - The number of objects
  - The total number of triangles
  - An array containing all the objects in the scene
  - A default material that is used when no material has been assigned to a given object
  - A lookup table for the materials of the scene
  - A background color
  - An array of cameras
  - An array of lights

Figure 8.28 (shown on the following page) summarizes the structures and their relationships in a UML class diagram.

It should be pretty easy to add support for other 3D file formats. All you need to do is write a set of routines that actually load the file and populate the different structures listed above. I have chosen the 3DS file format to load scenes as it is a de facto standard for 3D files. In doing so, I have also chosen the lib3ds library for its portability and the fact that it's open-source software.

The following is a list of the different C files of the project and a brief description of their contents.

- sh.c contains all the routines that have been implemented in the previous sections. This is where we'll add a few other routines such as SH projection and ray/object intersection functions.
- rgbe.c contains the different routines to manipulate HDR images. Nothing will be added to this file.
- 3ds.c contains the routines that use the lib3ds library to load a given 3DS file into the program's internal scene structure.
- HDRISHLighting.c contains the logic of the program and the GLUT callback functions to handle the user's input.

The following list summarizes the main functionalities of the program:

- 3DS file loading: As I already mentioned, this is achieved with the help of lib3ds. The program will load any 3DS file provided that it has at least one camera.
- HDR RGBE image loading using the code developed for the previous program.
- Switching between different rendering modes (standard OpenGL-shaded display, SH diffuse unshadowed, and SH diffuse shadowed)
- Automatic caching of computed SH coefficients for each scene (this can be overridden with a command-line switch; see section 8.3.6.3)
- Switching between different display modes (points, wireframe, and plain)

- Manual adjustment of the intensity of the lighting in SH diffuse unshadowed and SH diffuse shadowed rendering modes
- Use of OpenGL display lists for better performances (except when adjusting lighting intensity)
- Automatic capture of the scene to BMP images

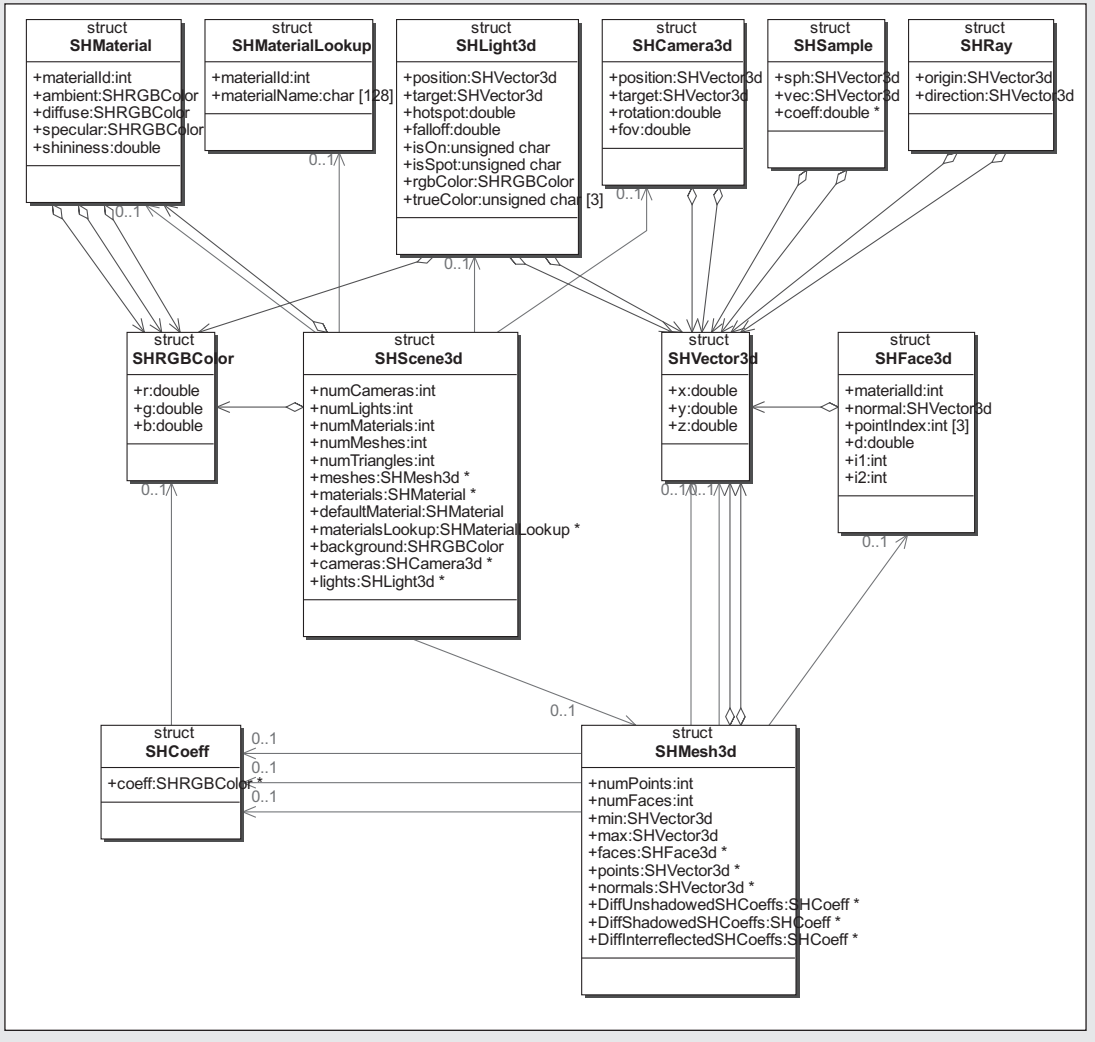


Figure 8.28. HDRISHLighting data structures diagram

8.3.6.2 Implementation

To calculate the SH diffuse unshadowed coefficients, the algorithm from section 8.2.2 has been implemented in the SHProjectDiffuseUnshadowedVertex() routine as follows:

```
void SHProjectDiffuseUnshadowedVertex(SHVector3d point,
   SHVector3d normal, SHRGBColor color, SHSample
   *sphericalSamples,
   SHRGBColor *result, int numSamples, int numCoeffs)
{
   /* The albedo */
   SHRGBColor albedo;

   /* Dot product and scale coefficient */
   double H, value;

   /* Weighting factor */
   double dWeight = 4.0 * PI;
   double factor = dWeight / numSamples;

   /* Indexes */
   int i, n;

   /* Loop through all the spherical samples */
   for (i = 0; i < numSamples; i++)
   {
      /* The transfer function is the cosine term */
      VecDot(sphericalSamples[i].vec, normal, H);

      /* calculate the albedo */
      RGBScale(color, ONE_OVER_PI, albedo);

      /* Calculate only if cosine positive */
      if (H > 0.0)
      {
         /* Calculate the coefficients */
         for (n = 0; n < numCoeffs; n++)
         {
            value = H * sphericalSamples[i].coeff[n];

            result[n].r += albedo.r * value;
            result[n].g += albedo.g * value;
            result[n].b += albedo.b * value;
         }
      }
   }

   /* Scale the SH coefficients */
   for (i = 0; i < numCoeffs; i++)
   {
      result[i].r = result[i].r * factor;
      result[i].g = result[i].g * factor;
      result[i].b = result[i].b * factor;
   }

}
```

The parameters are as follows:

- point: The current vertex for which SH coefficients are computed
- normal: The normal at this vertex
- color: The color at this vertex
- sphericalSamples: The spherical samples used in all SH computations
- result: The resulting SH coefficients
- numSamples and numCoeffs: The number of SH samples and the number of coefficients to calculate

For SH diffuse shadowed coefficients calculation, the transfer function includes the visibility term. This means that you need to check for intersection with all other triangles in the scene. The implementation reuses the ray/object intersection routines from the raytracer sample implementation. This raytracer was implemented with simplicity in mind; as a result, none of the acceleration techniques listed in Chapter 3 were implemented. This means that it takes a very long time to render scenes with lots of objects! For this program, it means that using this brute-force method for SH diffuse shadowed coefficients calculations would also take a very long time. I have therefore used a quick acceleration technique (or trick): For a given vertex, instead of testing intersection with all triangles of all objects in the scene, I first check the intersection with the bounding box of each object. If an intersection is found, I then test for intersection with all the triangles of the object intersected. This quick acceleration trick is still slow, but faster than the brute-force calculations, and very easy to implement. It works best with well-distributed scenes (i.e., scenes where objects are approximately the same size (occupy a similar volume in space) and have, on average, the same number of triangles).

It is also possible to speed up the program in its current state. Indeed, instead of looping through every single vertex of the scene, the program loops through every single triangle, then processes its vertices. This means that for triangles that share vertices, the SH coefficients of the same vertex are calculated several times. This has been implemented in this way because the lib3ds library doesn't allow accessing vertices directly. By looping directly through the vertices of the geometry, the program could easily be sped up.

The implementation uses several routines: the main routine SHProjectDiffuseShadowedVertex(), the main intersection routine intersectScene(), and the two ray/object intersection routines boxIntersect () and triangleIntersect(). The SHProjectDiffuseShadowedVertex() routine is very similar to the SHProjectDiffuseUnshadowedVertex() routine that calculates the SH diffuse unshadowed coefficients; it only adds the visibility term through a call to intersectScene().

```
void SHProjectDiffuseShadowedVertex(SHScene3d *aScene,
   int meshIndex, int faceIndex, SHVector3d point,
   SHVector3d normal, SHRGBColor color, SHSample
   *sphericalSamples, SHRGBColor *result, int numSamples,
   int numCoeffs)
{
   /* The albedo */
   SHRGBColor albedo;

   /* Dot product and scale coefficient */
   double H, value;

   /* Weighting factor */
   double dWeight = 4.0 * PI;
   double factor = dWeight / numSamples;

   /* Indexes */
   int i, n;

   /* Ray used for the visibility term */
   SHRay ray;

   /* Origin of the ray is the current vertex */
   VecCopy(point, ray.origin);
```

```
/* Loop through all the spherical samples */
for (i = 0; i < numSamples; i++)
{
   /* The transfer function is the cosine term */
   VecDot(sphericalSamples[i].vec, normal, H);

   /* calculate the albedo */
   RGBScale(color, ONE_OVER_PI, albedo);

   /* Calculate only if cosine positive */
   if (H > 0.0)
   {
      /* The direction is the spherical sample */
      /* direction */
      VecCopy(sphericalSamples[i].vec, ray.direction);

      /* Determine the visibility for shadowing */
      if (!intersectScene(&ray,aScene,faceIndex,
          meshIndex))
      {
         /* Calculate the coefficients */
         for (n = 0; n < numCoeffs; n++)
         {
            value = H * sphericalSamples[i].coeff[n];

            result[n].r += albedo.r * value;
            result[n].g += albedo.g * value;
            result[n].b += albedo.b * value;
         }
      }
   }
}

/* Scale the SH coefficients */
for (i = 0; i < numCoeffs; i++)
{
      result[i].r = result[i].r * factor;
      result[i].g = result[i].g * factor;
      result[i].b = result[i].b * factor;
   }
}
```

The parameters for this routine are the same as in the routine for SH diffuse unshadowed coefficients, with the following additions:

- aScene: The scene being lit
- meshIndex: The index of the mesh to which the current vertex belongs
- faceIndex: The index of the triangle to which the current vertex belongs

The visibility term uses the main intersection routine intersectScene().

```
int intersectScene(SHRay *ray, SHScene3d *aScene,
   int faceIndex, int meshIndex)
{
   /* The current mesh */
   SHMesh3d *mesh;

   /* Indexes */
   int i, j;

   /* Go through each object of the model */
   for(i = 0; i < aScene->numMeshes; i++)
   {
      /* Assign current mesh */
      mesh = &aScene->meshes[i];

      /* Check if ray intersects the mesh's bounding box */
```

```
        if (!boxIntersect(ray, mesh->min, mesh->max))
            continue;

        /* Go through all of the faces of the object */
        for(j = 0; j < mesh->numFaces; j++)
        {
           /* Skip triangle from which the ray originated */
           if ((i == meshIndex) && (j == faceIndex))
               continue;

           /* Test intersection, returns if intersection */
           /*  found */
           if(triangleIntersect(&mesh->faces[j],
              mesh->points[mesh->faces[j].pointIndex[0]],
              mesh->points[mesh->faces[j].pointIndex[1]],
              mesh->points[mesh->faces[j].pointIndex[2]],
                 ray))
              return 1;
        }
     }

     /* Returns 0 if no intersection found */
     return 0;
  }
```

The ray/triangle intersection routine is based on an algorithm found in [15]. The ray/object intersection routine is based on an algorithm found in [16]. I decided not to implement the SH diffuse inter-reflected mode in this program because it is very slow, even when using the acceleration trick described above. The algorithm to calculate the SH coefficients in this mode is given in section 8.2.2 of this chapter.

Two other routines have been added in the sh.c file: one that is used to retrieve the color of an HDR image based on a direction, getHDRIColor(), and one to use SH coefficients at run time to light the scene, SHLighting(). getHDRIColor() follows the explanations found at [17]. SHLighting() is the simple implementation (a dot product!) of the algorithm presented in section 8.2.2 of this chapter:

```
SHRGBColor SHLighting(SHRGBColor *light, SHRGBColor
   *vertexSH, int numCoeffs, double dScale)
{
   /* The color returned */
   SHRGBColor result;

   /* Index */
   int i;

   /* Initialize the color */
   result.r = result.g = result.b = 0.0;

   /* Perform the dot product of the SH coefficients */
   for (i = 0; i < numCoeffs; i++)
   {
      result.r += light[i].r * vertexSH[i].r * dScale;
      result.g += light[i].g * vertexSH[i].g * dScale;
      result.b += light[i].b * vertexSH[i].b * dScale;
   }

   /* Return the color */
   return result;
}
```

The program uses two methods to render a scene: one that uses OpenGL display lists and one where primitives are rendered directly. The method using display lists is faster than direct rendering, but the scene cannot be altered once it has been created as display lists. This method is not suitable

when one wants to modify the scene; with SH lighting rendering modes, for instance, it cannot be used if you want to increase or decrease the intensity of the light source (the HDR light probe). I have therefore decided to allow for the choice of whether or not to use OpenGL display lists by introducing a command-line switch (see section 8.3.6.3). This allows for best performances with large models, as well as the possibility to alter the light intensity interactively. For the three rendering modes (OpenGL, SH diffuse unshadowed, and SH diffuse shadowed), I have created three corresponding routines: createGLModelList(), createSHDUModelList(), and createSHDSModelList(). They can be called once to create a display list for their corresponding rendering mode (hence their respective names!) or called directly each time the screen needs to be refreshed. Both createSHDUModelList() and createSHDSModelList() call SHLighting() with the right SH coefficients and use Gouraud shading (interpolation of color between vertices), whereas createGLModelList() simply renders the scene with the OpenGL shading model (interpolation of normal between vertices).

8.3.6.3 Command-line Parameters

The HDRISHLighting program accepts different command-line parameters, which are summarized in the following table.

Switch	*Parameter(s)*	*Description*
-s	integer	The size of the grid of stratified samples
--samples	integer	The size of the grid of stratified samples
-b	integer	The number of bands used to approximate the spherical function
--bands	integer	The number of bands used to approximate the spherical function
-h	integer	Height of the display window
--height	integer	Height of the display window
-w	integer	Width of the display window
--width	integer	Width of the display window
-hdr	filename	Name of the HDR light probe to load
--hdr	filename	Name of the HDR light probe to load
-3ds	filename	Name of the 3DS scene to load
--3ds	filename	Name of the 3DS scene to load
-hs	float	Scale the intensity of the input HDR image
--hdrscale	float	Scale the intensity of the input HDR image
-dm	none	Force direct mode rendering to allow contrast adjustment of the lighting intensity

Switch	Parameter(s)	Description
--directmode	none	Force direct mode rendering to allow contrast adjustment of the lighting intensity
-nd	none	Only calculate SH coefficients (no display) and exit
--nodisplay	none	Only calculate SH coefficients (no display) and exit
-nc	none	Do not try to read cached SH coefficients from file, and force SH coefficients calculation
--nocache	none	Do not try to read cached SH coefficients from file, and force SH coefficients calculation
-v	none	Verbose mode
--verbose	none	Verbose mode
-?	None	Display a help message
--help	none	Display a help message

8.3.6.4 Keyboard Mapping and Mouse Usage

The keyboard mapping and mouse usage of this program are summarized in the following tables.

Key	Description
f/F	Switch to full screen display
n/N	Switch to windowed display
D	Switch to the next rendering mode (order: GL, SH unshadowed, SH shadowed)
d	Switch to the previous rendering mode (order: GL, SH shadowed, SH unshadowed)
S	Scale up the lighting intensity
s	Scale down the lighting intensity
>	Toggle between the different cameras (forward)
<	Toggle between the different cameras (backward)
b/B	Capture the content of the window to a BMP file named HDRISHLightingXXXX.bmp
a/A	Toggle animation on/off
y/Y	Cycle through the different display modes: point, triangle, and fill
q/Q	Exit the program

Mouse Action	Description
Left-click and drag	Rotates the camera around the function, thus giving the impression of rotating the function
Right-click	Toggles animation on/off

8.3.6.5 Results

Figures 8.29, 8.30, and 8.31 show several screen shots of the HDRISHLighting application displaying the same scene in diffuse shadowed mode with only four bands (16 coefficients), under different viewpoints and with different HDR light probes.

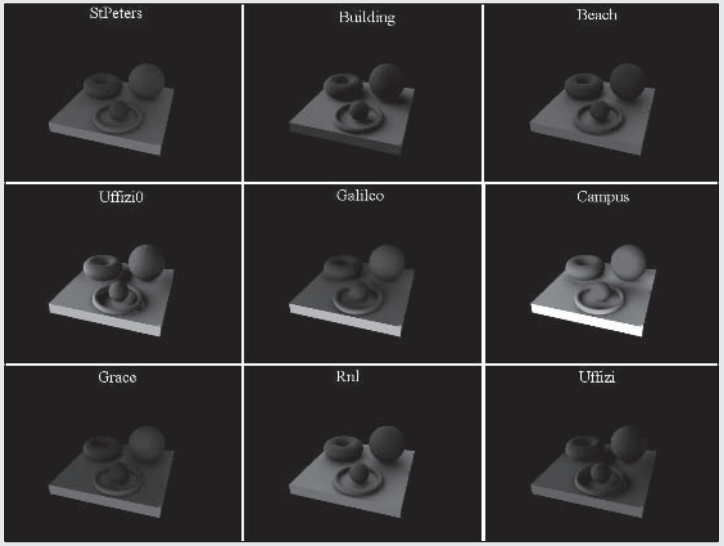


Figure 8.29. SH scene lit with different HDR light probes (1/3)

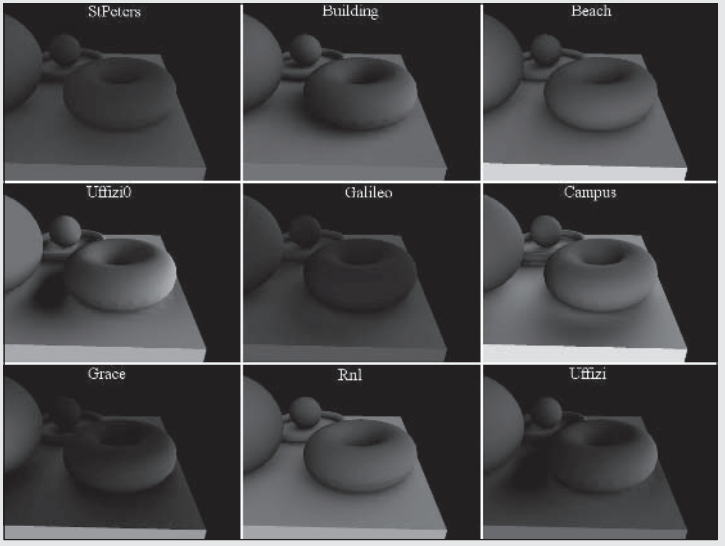


Figure 8.30. SH scene lit with different HDR light probes (2/3)

Figure 8.32 shows several versions of the HDRISHLighting application displaying the same scene with the Kitchen HDR light probe in the different display modes (SH modes using only four bands, i.e., 16 SH coefficients), along with the same scene rendered in 3D Studio MAX with default settings (scanline rendering).

Finally, you will find on the companion CD a version of this program that uses a simple raytracing acceleration technique based on bounding volumes hierarchies. The code has been adapted from [18].

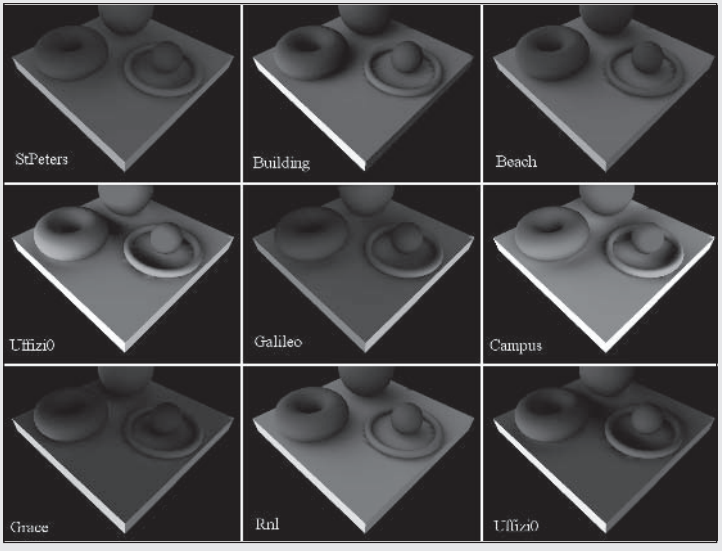


Figure 8.31. SH scene lit with different HDR light probes (3/3)

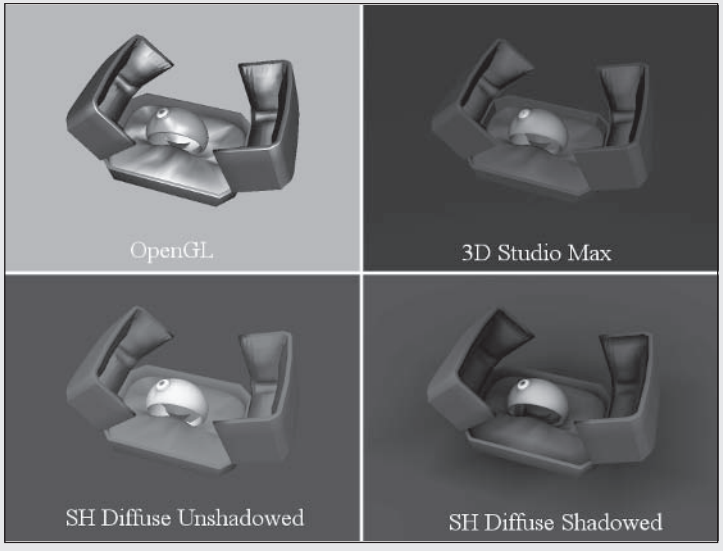


Figure 8.32. Scene rendered in different modes

Conclusion and Further Reading

In this chapter, I have shown how spherical harmonics can be used to perform real-time lighting of complex scenes using precalculated coefficients. This technique can be applied to any type of light source: analytical, empirical, or numerical. It is very fast as it only relies on Gouraud shading (color interpolation between vertices). In its simplest form, diffuse unshadowed rendering, spherical harmonic lighting is similar to the Lambert law that you already saw previously; the only difference here is that it is easy to calculate the diffuse color at vertices for any type of light source. With the second method, diffuse shadowed rendering, it is possible to take into account the visibility term in the rendering equation, and thus take into account shadowing. Finally, the third and most complex form, inter-diffuse rendering, makes it possible to render the global illumination of a scene in real time. The three forms are equally fast to render for a given scene; however, the preprocessing steps will take more time as the complexity of the transfer function increases. With all these wonderful properties, spherical harmonic lighting still presents some drawbacks. It is not for the faint-of-heart when it comes to mathematics! Rotating the 3D model or the light is possible but not both. Also, the scene needs to be static; objects cannot move relative to each other. Finally, SH lighting is a low-frequency approximation of light sources, and as with any other approximation technique, it can lead to artifacts. A similar technique has been developed to get around this problem. Instead of projecting onto the SH basis, lighting is approximated using the wavelet basis. This technique allows taking into account all frequencies, and is faster at run time (see [19] for more details).

References

[1] Sloan, P., J. Kautz, and J. Snyder, "Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments," *ACM Transactions on Graphics* 21, no. 3 (2002), pp. 527-536.

[2] Green, R., "Spherical Harmonic Lighting: The Gritty Details," *GDC 2003 Proceedings*.

[3] Blanco, M., M. Florez, and M. Bermejo, "Evaluation of the rotation matrices in the basis of real spherical harmonics," *Journal of Molecular Structure*, http://wwwl.elsevier.com/homepage/saa/eccc3/paper48/eccc3.html

[4] Ivanic, J. and K. Ruedenberg, "Rotation Matrices for Real Spherical Harmonics: Direct Determination by Recursion," *The Journal of Chemical Physics*, 1996, Vol. 100, pp. 6342-6347.

[5] Kajiya, J. T., "The Rendering Equation," *SIGGRAPH '86 Proceedings*.

[6] Dutré, P., P. Bekaert, and K. Bala, *Advanced Global Illumination*, Natick, PA: A.K. Peters, Ltd., 2003, p. 41.

[7] Jensen, H. W., *Realistic Image Synthesis Using Photon Mapping*, Natick, PA: A.K. Peters, Ltd., 2001, p. 28.

[8] Cohen, M. F. and J. R. Wallace, *Radiosity and Realistic Image Synthesis*, Boston: Academic Press Professional, 1993, p. 32.

[9] Mesa Library, OpenGL open-source implementation, http://www.mesa3d.org/

[10] GLUT Library, cross-platform UI library for OpenGL, http://www.sgi.com/software/opengl/glut.html

[11] Lib3ds library, cross-platform to work with 3DS files, http://lib3ds.sourceforge.net/

[12] Walter, B., HDR image manipulation routines, http://www.graphics.cornell.edu/~bjw/

[13] Debevec, P., HDR image manipulation programs, http://www.debevec.org/

[14] OpenGL community, developer resources, http://www.opengl.org/

[15] Badouel, D., "An Efficient Ray-Polygon Intersection," *Graphics Gems*, Academic Press, 1990, p. 390.

[16] Haines, E., "Ray/Box Intersection: An Introduction to Raytracing," *Graphic Gems*, Academic Press, 1990, p. 65.

[17] Debevec, P., "Light Probes," http://www.debevec.org/Probes, Light Probes section.

[18] Watkins, D., S. Coy, and M. Finlay, *Photorealism and Raytracing in C*, M&T Books, 1992.

[19] Hanrahan, P., R. Ramamoorthi, and R. Ng, "All-Frequency Shadows Using Non-Linear Wavelet Lighting Approximation," http://graphics.stanford.edu/papers/allfreq/

Chapter 9

Spherical Harmonics in DirectX

Introduction

The previous chapter introduced spherical harmonics as a method of working with precomputed radiance information. It focused on using a simple raytracer to precompute all of the SH data needed for later real-time rendering. Because OpenGL does not provide any "native" functionality for dealing with SH, the implementation details were provided in OpenGL and included everything from the raytracer to the SH equations.

In this chapter, I discuss the SH functions that are built into DirectX. However, OpenGL users are strongly encouraged to read this chapter in order to see how the SH computations can be improved using clustered principal component analysis (CPCA) [1]. This chapter also includes many other SH features that are not in the previous chapter, but are still very interesting for OpenGL readers. In fact, readers who demand a cross-platform solution might see opportunities to use DirectX utility functions for offline computation of data that can be used in any OpenGL application. The new features discussed in this chapter include:

- SH solution generation using D3DX
- SH rendering with vertex shaders
- Subsurface scattering effects with SH
- CPCA compression of SH coefficients
- SH using cube maps
- Using HDRI with DirectX
- A simple specular hack for SH scenes

9.1 Per-Vertex SH Data Generation with D3DX

The previous chapter discussed the theory behind SH, so I will jump directly into the implementation details. If you haven't read the previous chapter, this might be a good time to give it a skim. If you have read the previous chapter, you will see that D3DX encapsulates much of the basic SH functionality into a few handy functions. There are pros and cons to using D3DX as opposed to a do-it-yourself approach. The greatest advantage is probably the fact that the functions are written, tested, and maintained by the D3DX team. This offloads much of the work from you. The greatest disadvantage is arguably the fact that the D3DX functions are black boxes. You can't get inside and modify them (although some would list this in the advantages column). Also, a cross-platform raytracer makes it possible to use your teraflop Linux cluster for SH computations.

Ultimately, you have to choose the implementation that's right for you. However, one thing to note is that the data generated by the D3DX functions is not DirectX specific, but rather a set of SH coefficients that can be used anywhere. OpenGL users might want to explore ways to generate SH coefficients using D3DX and use that data on other platforms. This is especially interesting for features such as subsurface scattering or CPCA compression. Both are techniques that are supported by D3DX and nontrivial to implement yourself. Going forward, this chapter will discuss DirectX, with explicit and implicit mentions of OpenGL uses.

The sample applications for this chapter can be found on the CD in the \Code\Chapter 09 directories. The main application is in the \SH in DX directory, with other projects building off of that basic code. The samples are rarefied versions of the DirectX 9.0 SDK samples. They forgo the UI and error handling of the samples in favor of very explicit usage of the key functions. I tried to keep my samples structurally similar to the SDK samples so that you could see the simple versions and then refer back to the SDK samples to see how to add UI features and other niceties. Therefore, I encourage you to refer to the SDK samples once you understand the basics.

9.1.1 The Main SH Simulator

Much of the SH coefficient generation functionality seen in the previous chapter is encapsulated into one D3DX function that handles raytracing and SH projection. This function (for per-vertex SH coefficients) is D3DXSHPRTSimulation.

```
HRESULT WINAPI D3DXSHPRTSimulation(UINT Order, LPD3DXMESH
    *ppMeshArray, D3DXSHMATERIAL **ppMaterials,
    UINT NumRays, UINT NumBounces, BOOL
    EnableSubSurfaceScattering, FLOAT LengthScale,
    BOOL EnableSpectral, LPD3DXBUFFER *ppResultsBuffer,
    LPD3DXSHPRTSIMCB pProgressCallback);
```

Dissecting the name of the function, this D3DX function runs the simulation (basically, a raytracer with a few added features) and generates the SH coefficients for precomputed radiance transfer (PRT). The parameters of this function are mostly self explanatory. You can specify the order of the SH

approximation as discussed in the previous chapter. The objects are given as an array of pointers to D3DXMESH objects, and you must supply a set of materials. Any set of vertices and corresponding indices can be used to generate a D3DXMESH object; you aren't constrained to .x files.

The material parameter defines the diffuse color of the meshes and also includes parameters needed for subsurface effects. I will talk more about the D3DXSHMATERIAL structure when I talk about subsurface effects later in this chapter. If you are not using subsurface effects, you can get away with setting only the diffuse color of the material.

The next five parameters drive the raytracing simulation. You can set the number of rays "shot" from each vertex and the number of bounces. You can also enable the simulation of subsurface scattering effects and define the scale factor (more on this later). Finally, you can request a solution that only returns intensity information or a full color solution. The full color solution will encode more chromatic effects, but it will also require more storage and processing. All of the samples in this chapter assume a full color solution. See the SDK samples to see how the code can be trimmed down for monochrome lighting.

The last two parameters allow the function to communicate with the rest of the application. The result buffer contains the final SH coefficients. The size of the buffer is dependent on the order, the number of vertices, and whether or not you asked for spectral processing. The callback function provides a way to get simulation status messages when using the function asynchronously (multithreaded). This isn't a requirement, but it certainly makes for a nicer application, because you can continue rendering and responding to UI messages while the simulation is running.

The following chunk of code is one example of how this function might be called. In this example, the color of the mesh is white, there is no subsurface scattering, and there is no second-order reflection (there are no bounces). In this case, the function is being called from a separate thread, and D3DX uses StaticSHCallback to communicate status back to the user. When the simulation is complete, the data buffer contains the SH coefficients.

```
SetColorValue(&m_SHMaterial.Diffuse, 1.0f, 1.0f, 1.0f,
   1.0f);
D3DXSHMATERIAL *pSHMaterial = &m_SHMaterial;

D3DXSHPRTSimulation(ORDER, 1, &(m_Models[0].pModel),
   &pSHMaterial, NUM_RAYS, 0, FALSE, 1.0f, TRUE,
   &m_pSHDataBuffer, StaticSHCallback);
```

9.1.2 Parameters and Performance Implications

D3DXSHPRTSimulation is an offline function; it should not be called during any time-critical operations. However, it's still interesting to understand exactly what the performance implications are for the different parameters.

9.1.2.1 Vertex Count

Obviously, performance is proportional to vertex count, and smooth SH lighting is dependent on a fairly high amount of tessellation in your model. Figure 9.1 shows three models with increasing vertex counts (each model is roughly two to

four times more dense than the last). Comparing the first model to the last, you can clearly see that a higher vertex count translates into a smoother solution. However, comparing the last two highlights some subtleties.

If you look at the differences between the shapes of the second and third meshes, you see that the increased vertex count produces almost no visual benefit in terms of the shape. If you look at the shadow, you will see that the vertex count produces a "noisier" shadow. This would imply that you might need to over-tessellate your mesh to produce the cleanest shadows. The downside is that the increased vertex count definitely affects both your simulation and real-time performance. In the end, the middle result might be quite acceptable and quicker to compute. As with most things, finding a compromise between quality and processing time (offline and real time) is a balancing act.

9.1.2.2 Ray Count

The speed of the simulation is also closely related to the number of rays. Unlike the vertex count, the number of rays only affects the simulation time; the real-time processing cost is independent of ray count. This means that you can use very high ray counts to get good solutions, and the only downside is that you have to spend a lot of time waiting on the offline calculations.

That's fine, but there are points of diminishing returns. For some models, larger and larger ray counts don't necessarily produce better and better results. Consider Figure 9.2, which shows three different meshes with three different ray counts.

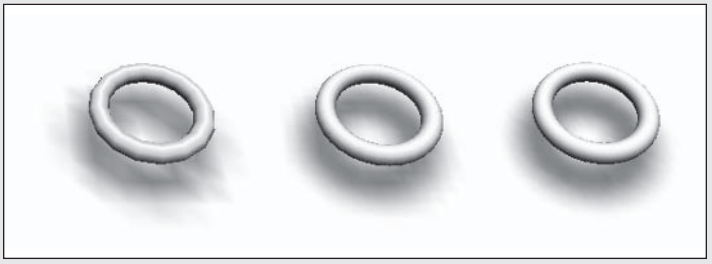


Figure 9.1. SH solutions with increasing vertex counts

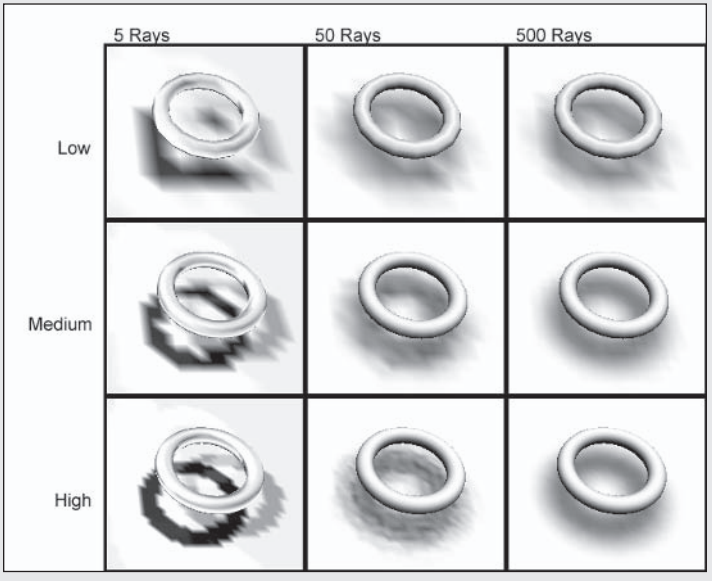


Figure 9.2. SH solutions with increasing ray counts

The low-resolution mesh gains almost nothing when the ray count changes from 50 to 500. On the other hand, both the medium- and high-resolution meshes show a substantial difference in quality. However, you can see that the solution for the high-resolution mesh probably wouldn't change dramatically with ray counts higher than 500.

So, the thing to take away from this is that experimentation is your friend. Blindly picking an arbitrarily high ray count will probably cost you a lot of needless waiting time. On the other hand, picking a low ray count might produce artifacts in places you didn't expect. Spend some time experimenting to understand optimal values. When inevitable slight changes to your models necessitate reprocessing, you'll be glad you have the right value.

9.1.2.3 Bounce Count

The number of bounces also affects both the final look and feel of the solution and the processing time. "Bounces" account for higher order reflections between different parts of the scene. Figure 9.3 shows a perfect example of this. The leftmost image has no interreflection between objects. The underside of the ball is not exposed to the light and is therefore black. The middle image shows the effect of a single bounce. Here, reflections from the plane light the underside of the ball. Finally, the rightmost image shows what happens when that light is reflected (in a second bounce) back to the plane. The effect might be subtle on the printed page, but the shadow is lighter and softer than the first two images.

Higher order reflections are much more realistic than no bounces at all, but they do affect your processing time

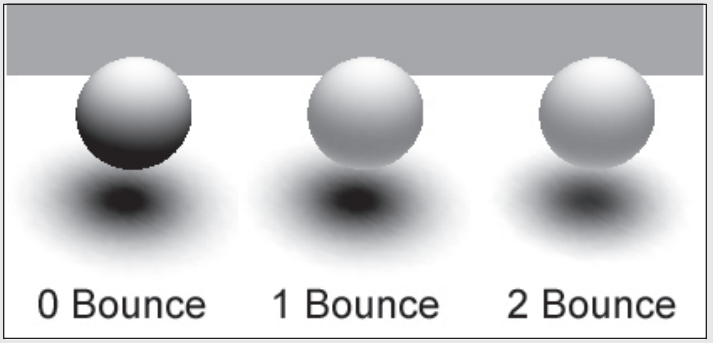


Figure 9.3. SH solutions with increasing bounce counts

(although not rendering time). Also, multiple bounces make more sense for shiny materials like metals than they do for highly absorbent materials. As always, experiment to find a solution that looks good and processes in an acceptable amount of time.

9.1.2.4 Order

The final parameter to discuss (for now) is the order of the final SH solution. The ray count and the bounce count determine the overall solution (per vertex), but the order determines the quality of the approximated solution. In other words, remember that any solution can be reproduced exactly with a very high-order SH solution. The problem is, higher-order SH solutions require more coefficients, which require more processing. Therefore, you're constrained to low-order solutions if you want to render in real time. Still, there is some room to experiment.

As in the last chapter, higher-order SH solutions preserve more high-frequency components. In other words, they preserve detail. Shadows look sharper and so on. On the other hand, they require more processing and more vertex shader constant storage (assuming you are rendering with a shader, as I will discuss soon). Figure 9.4 shows the difference between a low-order solution (order = 3) and a higher-order solution (order = 6). The low-order solution features very soft shadows, while the higher-order solution sharpens the shadows and produces an effect that is more accurate (for the given single overhead directional light).

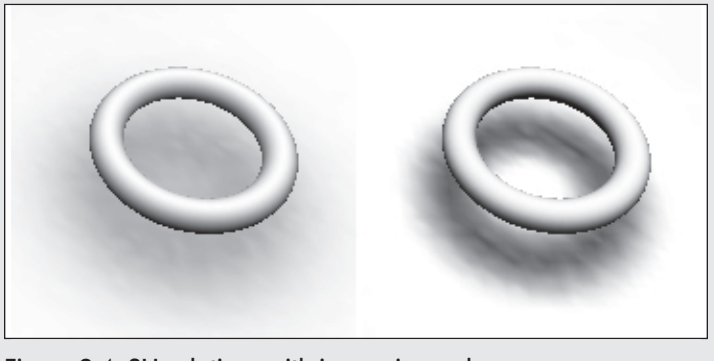


Figure 9.4. SH solutions with increasing order

Still, accuracy is somewhat in the eye of the beholder, or at least the programmer. In some cases, the very soft effect might be preferable and cost less in terms of storage and processing. Again, experimentation is your friend.

After you have chosen your values and run the simulation, D3DXSHPRTSimulation will return a buffer of SH coefficients. You can use these coefficients as described in the previous chapter, or you can do more preprocessing.

9.1.3 Compressed SH Coefficients

At this point, you have a set of SH coefficients for each vertex. However, the number of coefficients might be quite large and therefore difficult to use efficiently. For example, all of the figures shown so far were generated with a fifth-order SH solution and all color channels enabled. This means that each vertex has 75 SH coefficients (5 squared times 3 color channels). Dealing with 75 numbers per vertex is much easier than raytracing or other techniques, but that's a lot of data to deal with.

There is a way to compress the data further, making it both smaller and easier to deal with. At a very high level, the approach is basically the same as was discussed at the beginning of the previous chapter. Choose a basis that allows you to encode the features of your "signal" and then approximate your solution using only the most important features and a processing method that exploits the new encoding.

9.1.3.1 Generating Compressed SH Coefficients

The technique used by D3DX is called Clustered Principal Component Analysis (CPCA) [1]. To understand how this works, consider the conceptual example in Figure 9.5. Here, a set of arrows represents sets of SH coefficients. In reality, a set of SH coefficients can be thought of as a multidimensional

vector, but anything greater than 2D is very difficult to show on a page, so this is a conceptual view only.

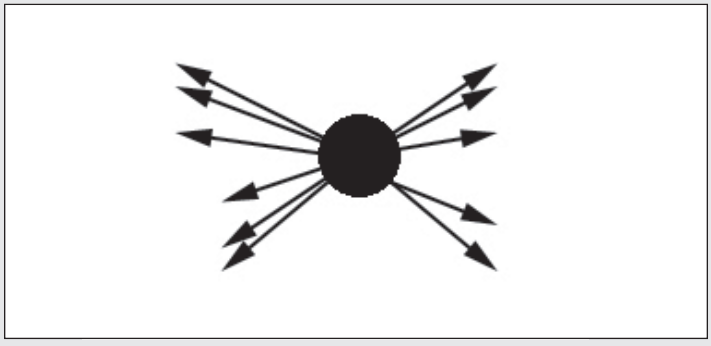


Figure 9.5. A conceptual 2D view of the SH solution vectors

If you were to take a look at the SH coefficients for a model with many vertices, you'd see that sets of coefficients would tend to fall into clusters. For example, adjacent vertices on a flat plane with few surrounding objects would have very similar sets of coefficients. Adjacent vertices under an object would be similar to each other, but different from the first two. Figure 9.6 divides the SH vectors into clusters.

Identifying the clusters alone doesn't really do much for you, so the first step is to find the mean (average) vector for each cluster. Figure 9.7 shows the mean vector of each cluster. If you wanted, you could stop here and represent each vertex's SH coefficients with the mean vector of the appropriate cluster. This would give you fewer values, but it wouldn't look very good.

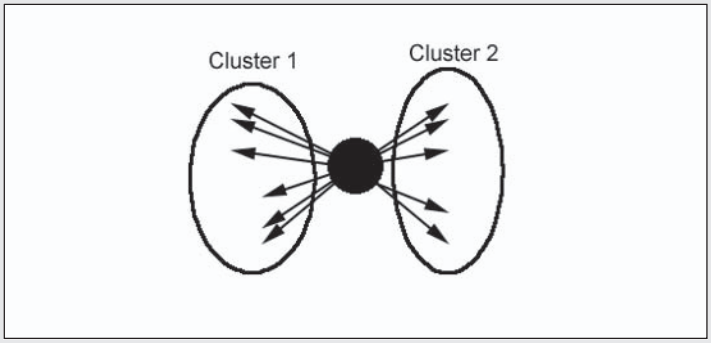


Figure 9.6. Clustering the SH solution vectors

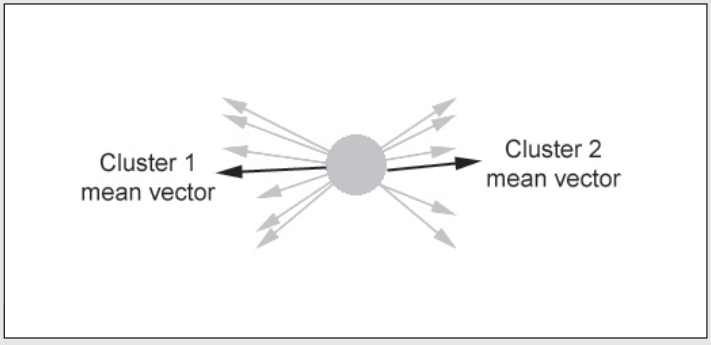


Figure 9.7. Finding the cluster mean vectors

You can go one step further. You want to use more data than the cluster mean, but less than the complete set of all of the SH coefficients. Therefore, identify a set of n vectors that are representative of the cluster. Basically, find the vectors that preserve the most important features. Because the

technique used is called Principal Component Analysis, the resulting vectors will be called PCA vectors. Figure 9.8 shows an example of this for $n = 2$.

Finally, each "real" SH solution vector can be approximated as a linear combination of the n new vectors. In other words, the solution for any vertex can be approximated as the weighted sum of the PCA vectors of a given cluster. This is shown in Figure 9.9.

In the previous chapter, you saw that you could compute the radiance from a given vertex by finding the dot product of the vertex SH coefficients and those of the incoming light. The same basic idea applies here, but now you can find the dot product of the light and the cluster mean and PCA vectors. More formally, the rendering equation for precomputed radiance using compressed PCA vectors can be expressed as:

$$I = (M_k \bullet L) + \sum_{j=1}^{N} w_j (B_{kj} \bullet L)$$

Equation 9.1. Finding precomputed radiance with compressed coefficients

In the above equation, the mean vectors (M) and the PCA basis vectors for each cluster (B) are constant across the mesh. Therefore, the per-vertex radiance is dependent on per-vertex weight values and a set of batch computed values. More on that later.

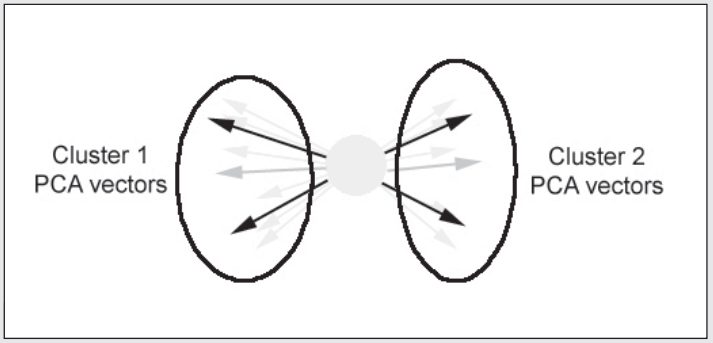


Figure 9.8. Finding a set of representative PCA vectors

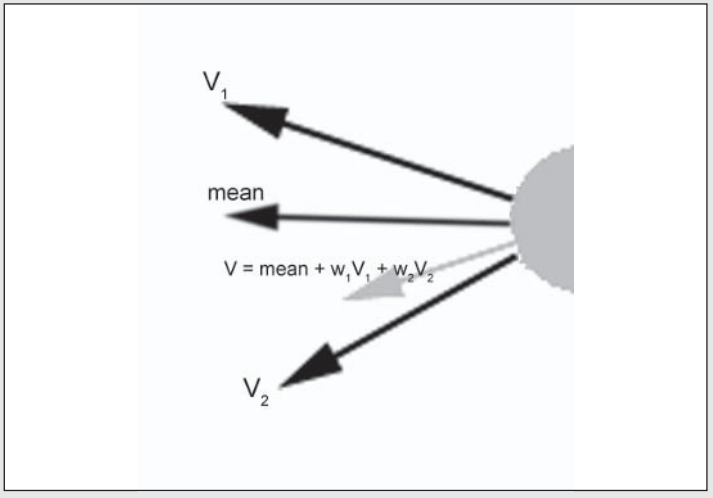


Figure 9.9. An actual solution vector approximated with weighted PCA vectors

In the end, the use of CPCA allows you to represent a set of SH coefficients of each vertex with *n* weight values that will be used with a fixed set of PCA vectors. The PCA vectors themselves still contain order squared coefficients; they aren't compressed. Instead, the amount of data stored per-vertex is much smaller; you only need a cluster index and a set of weights to take advantage of a larger set of coefficients in the form of PCA vectors. Furthermore, you will see later that the dot product terms can be evaluated outside of the vertex shader, making the above equation very fast at render time. Luckily, you don't have to derive the CPCA values yourself. D3DX includes a function that will do it for you. Typically, the input values will be the values generated by D3DXSHPRTSimulation, but they could just as easily come from your own raytracer as discussed in the previous chapter. In either case, you can compress SH coefficients with D3DXSHPRTCompress.

```
HRESULT WINAPI D3DXSHPRTCompress(UINT Order,
   LPD3DXBUFFER pSHCoefficients,
   D3DXCOMPRESSQUALITYTYPE Quality,
   UINT NumClusters, UINT NumPCAVectors,
   LPD3DXBUFFER *ppResults);
```

The parameters are straightforward. The function takes the order of the SH coefficients, the coefficients themselves, a compression quality flag, the requested number of clusters, and the requested number of PCA vectors. Finally, it creates and fills a buffer containing the compressed results.

9.1.3.2 Using Compressed SH Coefficients

The buffer of compressed results stores a set of PCA vectors for each cluster. The buffer also contains a set of weight values for each vertex. In order to make use of that data, the weight values need to be "attached" to the corresponding vertices. Therefore, before moving on, I'd like to talk about the vertex structure and declaration.

Like the SDK sample, I have limited the number of PCA weights to a maximum of 24. This works well in terms of not inflating your vertex size too much, and it also reduces the number of vertex shader instructions you need to process. With that constraint in place, the vertex structure is as follows:

```
struct MODEL_VERTEX
{
   float x, y, z;
   float nx, ny, nz;
   float ClusterID;
   float PCAData[24];
};
```

This structure stores position and normal data as well as a single float for the cluster ID and 24 floats for the PCA weights. This will allow the vertex shader to index into the proper cluster and then apply the weights to the corresponding set of PCA vectors.

The vertex declaration defines the semantics of the vertex structure. Here, the cluster ID and 24 PCA weights are given the blend weight semantic. This is an arbitrary choice, but it's convenient because blending animation cannot be used with SH (the mesh must remain rigid). Notice that the 24

weights are distributed between six four-component blend weights.

```
D3DVERTEXELEMENT9 Declaration[] =
{
   {0,   0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_POSITION,    0},
   {0,  12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_NORMAL,      0},
   {0,  24, D3DDECLTYPE_FLOAT1, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_BLENDWEIGHT, 0},
   {0,  28, D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_BLENDWEIGHT, 1},
   {0,  44, D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_BLENDWEIGHT, 2},
   {0,  60, D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_BLENDWEIGHT, 3},
   {0,  76, D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_BLENDWEIGHT, 4},
   {0,  92, D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_BLENDWEIGHT, 5},
   {0, 108, D3DDECLTYPE_FLOAT4, D3DDECLMETHOD_DEFAULT,
      D3DDECLUSAGE_BLENDWEIGHT, 6},
   D3DDECL_END()
};
```

With the vertex format in place, it is now possible to show how the SH coefficients are compressed and the PCA weights are applied to the vertices. Also, the PCA vectors themselves must be extracted and stored for later use. The application calls the following function once the SH coefficients are generated or loaded from a file.

```
HRESULT CLightingApplication::CompressDataAndSetVertices()
{
```

The D3DXSHPRTExtractDesc function provides a simple way to query the attributes of a set of SH coefficients. In this simple application, all of the attributes are either already defined constants or can be derived from the mesh. However, in a real-world application, you might want to use this function to make sure that data loaded from a file matches the mesh and that any other assumptions you might have about the SH data are true.

```
    D3DXSHPRTBUFFER_DESC BufferDescription;
    D3DXSHPRTExtractDesc(m_pSHDataBuffer,
       &BufferDescription);
```

Here, the SH coefficients are compressed using the method described in the previous section.

```
    ID3DXBuffer* pCompressedBuffer = NULL;
    D3DXSHPRTCompress(BufferDescription.Order,
       m_pSHDataBuffer, D3DXSHCQUAL_FASTLOWQUALITY,
       NUM_CLUSTERS, NUM_PCAVECTORS,
       &pCompressedBuffer);
```

The compressed buffer now contains everything you need to render. The remainder of this function formats the data for easier use. The first step is to extract the cluster IDs and attach them to the correct vertices. D3DXSHPRTCompExtractClusterIDs extracts the IDs into an array of integer values. Those values are not useful themselves; instead, they are used to generate a value that will be used as an offset into a set of PCA vectors.

```
    UINT *pClusterIDs = new UINT
       [BufferDescription.NumSamples];
    D3DXSHPRTCompExtractClusterIDs(pCompressedBuffer,
       pClusterIDs);

    MODEL_VERTEX *pV = NULL;
    m_Models[0].pModel->LockVertexBuffer(0, (void**)&pV);

    for(UINT CurrentVertex = 0;
       CurrentVertex < BufferDescription.NumSamples;
       CurrentVertex++)
    {
```

For each vertex, use the cluster ID to compute the offset into a set of PCA vectors. The index value is the cluster ID multiplied by the size of each cluster. In this case, the size of each cluster is 1 (for the cluster mean) plus 3 (for the number of color channels) times the number of PCA vectors (which are packed into sets of four when placed in vertex shader constant registers). In the shader, this value will be used to index into a much larger set of constants.

```
       float CurrentClusterIndex = (float)
          (pClusterIDs[CurrentVertex] *
          (1 + 3 * (NUM_PCAVECTORS / 4)));

       pV[CurrentVertex].ClusterID =
          CurrentClusterIndex;
    }

    m_Models[0].pModel->UnlockVertexBuffer();

    SAFE_DELETE_ARRAY(pClusterIDs);
```

To place the PCA weights into the vertices, you can let D3DX do the work for you. D3DXSHPRTCompExtractToMesh takes a mesh, a compressed buffer, a number of vectors, and the semantic of the first vertex element wherein you want the weight data.

```
    D3DXSHPRTCompExtractToMesh(NUM_PCAVECTORS,
       m_Models[0].pModel, pCompressedBuffer,
       D3DDECLUSAGE_BLENDWEIGHT, 1);
```

At this point, you have processed all of the vertex data. You have set the PCA weights for each vertex and you have set a cluster index that will allow the shader to index into a large set of PCA vectors. Finally, you must extract the cluster basis vectors. Each cluster is large enough to store a full set of SH coefficients for each vector and each color channel, with one additional vector for the cluster mean. The m_ClusterBases variable comes into play during rendering.

```
    long SizeOfCluster = ORDER * ORDER * 3 *
       (NUM_PCAVECTORS + 1);
    m_ClusterBases = new float[SizeOfCluster *
       NUM_CLUSTERS];

    for(long CurrentCluster = 0;
       CurrentCluster < NUM_CLUSTERS;
       CurrentCluster++)
    {
       D3DXSHPRTCompExtractBasis(CurrentCluster,
          pCompressedBuffer, &m_ClusterBases
          [CurrentCluster * SizeOfCluster]);
    }
```

The m_ClusteredPCA member is not filled at this point. I am simply allocating a large array that will be filled with the dot products of the PCA vectors and the lighting SH coefficients.

```
m_ClusteredPCA = new float[NUM_CLUSTERS*(4 +
   3*NUM_PCAVECTORS)];
```

At this point, there is no reason to keep the compressed buffer. Even if you wanted to cache the results in a file, you might be better off saving the formatted vertices and the array of cluster bases.

```
    pCompressedBuffer->Release();

    return S_OK;
}
```

From this point on, all shader and rendering discussion will assume that there is PCA data in the vertices and in m_ClusterBases. Before moving on, remember that CPCA can be used for other rendering tasks as well. [2] shows how CPCA can be used to encode more information to support specularity and arbitrary BRDFs. I chose not to cover that technique because rendering rates are still a bit too low for games, but the theory behind the technique (BRDFs, SH, CPCA, and other acronyms) is the same as can be found in this book.

9.2 Rendering the Per-Vertex SH Solution

The actual Render function of the sample application is very similar to any other render function. The only new functionality is encapsulated by the SetSHShaderConstants function. Like every other sample, the user can loop through the shaders with the F1 key. If the SH shader is active, the Render function will call SetSHShaderConstants and pass in the object space direction of a single light.

9.2.1 Encoding Lights for SH Rendering

Referring to the previous chapter or to Equation 9.1, you know that you can only find a final SH lighting solution if you have projected the scene's lights into a set of SH coefficients. D3DX provides another set of helper functions that allow you to easily generate SH coefficients for directional, cone, spherical, and hemispherical lights. Each light type behaves in basically the same way, so this section concentrates on directional and cone lights.

This instantiation of SetSHShaderConstants takes an object space direction vector for the light. This direction vector can be used for either the cone light or the directional light, depending on which you enable.

```
HRESULT CLightingApplication::SetSHShaderConstants(
   D3DXVECTOR3 *pObjectSpaceLightVector)
{
```

The following array will hold the SH projected light values. The SDK sample projects the light and sets the vertex shader constants only when the light direction changes. For simplicity, this example changes it every frame. While not the most optimal, this example represents a worst case although the performance is decent even with lower-end graphics cards.

```
        float LightValues[3][ORDER*ORDER];
```

The example can be set to use either the directional light or a cone light. For more visual interest, the width of the cone changes over time. Screen shots and a discussion of this effect can be found following the explanation of this function.

```
#ifdef USE_CONE_LIGHT
   float Radius = (0.51f + 0.5f * sin((float)
      GetTickCount() / 1000.0f)) * 3.0f;
   D3DXSHEvalConeLight(ORDER, &(D3DXVECTOR3(0.0f,
      1.0f, 0.0f)),Radius, 1.0f, 1.0f, 1.0f,
      LightValues[0], LightValues[1],
      LightValues[2]);
#else
   D3DXSHEvalDirectionalLight(ORDER,
      pObjectSpaceLightVector, 1.0f, 1.0f, 1.0f,
      LightValues[0], LightValues[1],
      LightValues[2]);
#endif
```

Here begins the most important part of the function. Equation 9.1 can be evaluated very quickly in a vertex shader if you preprocess the dot products between the cluster mean and lights and the PCA vectors and lights. The following loop does exactly that.

```
    for (long Cluster = 0; Cluster < NUM_CLUSTERS;
       Cluster++)
    {
```

The loop must index into both the array that holds the raw cluster information and the computed values that will go to the shader. The PCAIndex variable indexes into the final values, and the BasisIndex variable indexes into the raw cluster data. Hence the difference in multipliers.

```
        long PCAIndex  = Cluster * (4 + 3 *
           NUM_PCAVECTORS);
        long BasisIndex= Cluster * ORDER * ORDER * 3 *
           (NUM_PCAVECTORS + 1);
```

The following block of code computes the dot product of the light coefficients and the cluster mean. This creates the first term of Equation 9.1.

```
        m_ClusteredPCA[PCAIndex + 0] =
           D3DXSHDot(ORDER, &m_ClusterBases
           [BasisIndex+0 * ORDER * ORDER],
           LightValues[0]);
        m_ClusteredPCA[PCAIndex + 1] =
           D3DXSHDot(ORDER, &m_ClusterBases
           [BasisIndex+1 * ORDER * ORDER],
           LightValues[1]);
        m_ClusteredPCA[PCAIndex + 2] =
           D3DXSHDot(ORDER, &m_ClusterBases
           [BasisIndex+2 * ORDER * ORDER],
           LightValues[2]);
        m_ClusteredPCA[PCAIndex + 3] = 0.0f;
```

Now, you must find the dot product between the light solution and the individual PCA vectors. Here, pPCAVector is introduced to help index into the clustered PCA array. When setting values of pPCAVector, you are actually setting data in m_ClusteredPCA.

```
        float *pPCAVector = &m_ClusteredPCA[Cluster *
           (4 + 3 * NUM_PCAVECTORS) + 4];

        for(long CurrentPCA = 0;
           CurrentPCA < NUM_PCAVECTORS;
           CurrentPCA++)
        {
```

Find the dot product between each PCA vector and the light solution. These values will be used in the shader to compute the second term of Equation 9.1. These values are the B dot L values that are weighted differently for each vertex.

```
           long Offset = BasisIndex + (CurrentPCA + 1) *
              ORDER * ORDER * 3;

           pPCAVector[0 * NUM_PCAVECTORS + CurrentPCA] =
              D3DXSHDot(ORDER, &m_ClusterBases[Offset
              + 0 * ORDER * ORDER], LightValues[0]);
           pPCAVector[1 * NUM_PCAVECTORS + CurrentPCA] =
              D3DXSHDot(ORDER, &m_ClusterBases[Offset
              + 1 * ORDER * ORDER], LightValues[1]);
           pPCAVector[2 * NUM_PCAVECTORS + CurrentPCA] =
              D3DXSHDot(ORDER, &m_ClusterBases[Offset
              + 2 * ORDER * ORDER], LightValues[2]);
           }
        }
```

When all the values have been computed, pass the entire array to the vertex shader. The shader uses this data to calculate the radiance for each vertex.

```
        m_pD3DDevice->SetVertexShaderConstantF(4,
           m_ClusteredPCA, (NUM_CLUSTERS * (4 + 3 *
           NUM_PCAVECTORS)) / 4);

        return S_OK;
     }
```

9.2.2 The Basic SH Vertex Shader

Last, but certainly not least, you need the vertex shader to bring all of the pieces of Equation 9.1 together. The shader can be found on the CD as \Code\Shaders\SH_VS.hsl. The shader used for the SDK sample is used with the DirectX effect framework, and includes the ability to easily change various parameters. This example assumes a set number of 24 PCA vectors. Also, it runs with three channels of color data. You could trim quite a few instructions off the shader if you use fewer PCA vectors and/or fewer color channels.

The vertex input structure matches the vertex declaration seen in the actual code. Again, this assumes a maximum of 24 PCA vectors. The output vertex structure is very simple — the position and color. This shader could also set and output other features, such as texture coordinates, but this example includes only the basic SH functionality.

```
struct VS_INPUT
{
   float4 Position   : POSITION;
   float3 Normal     : NORMAL;
   float  ClusterID  : BLENDWEIGHT0;
   float4 Weights[6] : BLENDWEIGHT1;
};

struct VS_OUTPUT
{
   float4 Position  : POSITION;
   float4 Diffuse   : COLOR0;
};
```

Shader constant data is comprised of the basic transformation matrix and the PCA vector data. The array is set to be large enough for 200 clusters and 24 PCA vectors. You can experiment with larger values (to a point), but remember that you will need to change these numbers.

```
matrix Transform                       : register(c0);
float4 ClusteredPCA[200*(1+3*(24/4))] : register(c4);


VS_OUTPUT main(const VS_INPUT InVertex)
{
   VS_OUTPUT OutVertex;
```

Like any other shader, this one must transform the vertex to clip space. This operation is the same as you've seen in every other shader.

```
   OutVertex.Position = mul(Transform,
      InVertex.Position);

   float4 R = float4(0,0,0,0);
   float4 G = float4(0,0,0,0);
   float4 B = float4(0,0,0,0);
```

The rest of the shader is surprisingly simple. The remaining lines are a straightforward implementation of Equation 9.1. The constants contain the dot products and the PCA vectors. The vertex contains the weight values. The shader just needs to weight the PCA vectors and add everything together. This loop computes the second term of Equation 9.1. Remember that it assumes 24 PCA vectors. The operations work on sets of four values at a time.

```
    for (int i = 0; i < 6; i++)
    {
       float Offset = InVertex.ClusterID + i + 1;

       R += InVertex.Weights[i] *
          ClusteredPCA[Offset];
       G += InVertex.Weights[i] * ClusteredPCA
          [Offset + 6];
       B += InVertex.Weights[i] * ClusteredPCA
          [Offset + 12];
    }
```

The diffuse color is set with the dot product of the light and the cluster mean. This is the first term of Equation 9.1.

```
    float4 Diffuse = ClusteredPCA[InVertex.ClusterID];
```

At this point, R, G, and B are comprised of four different values. Each of those values represents a quarter of the sum of the weighted PCA values. Therefore, the final step is to sum these four values into one final value and add that to the value already in Diffuse. The dot product provides a convenient and compact way to do this. The dot product of 1 and a set of values is the sum of the set.

```
    Diffuse.r += dot(R, 1);
    Diffuse.g += dot(G, 1);
    Diffuse.b += dot(B, 1);
```

Diffuse contains the final output color. Copy it to the output vertex and you're done.

```
    OutVertex.Diffuse = Diffuse;

    return OutVertex;
}
```

The shader is the final step. The last stop is the screen. Figures 9.10 and 9.11 highlight the differences between a simple directional light and the same light with SH precomputed radiance. The screen shots speak for themselves.

Figures 9.12 and 12.13 show the result of a cone light with a variable radius. In both figures, only the radius of the cone light has changed; everything else is exactly the same.

You might want to experiment with spherical and hemispherical light sources. Each light will create a different effect, and the rendering cost (once the light has been evaluated) will be the same for any light type.

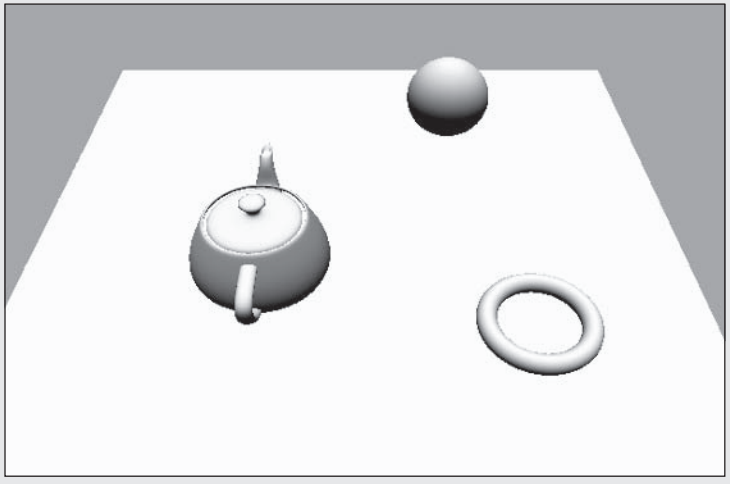


Figure 9.10. A scene with diffuse directional lighting

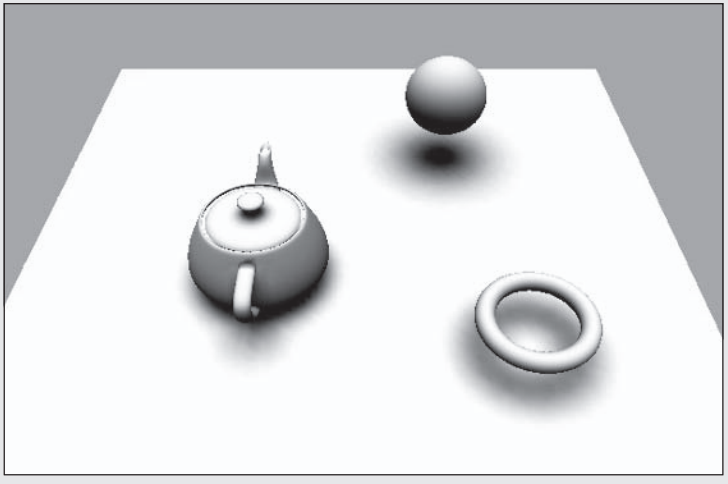


Figure 9.11. A scene with diffuse SH lighting

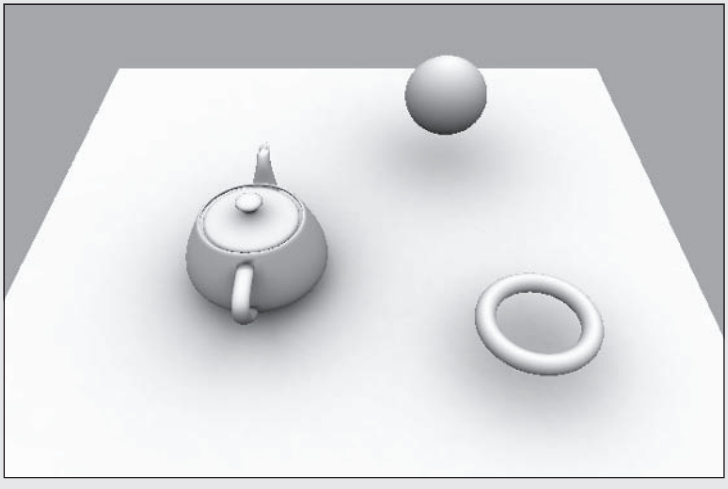


Figure 9.12. A cone light with a smaller radius

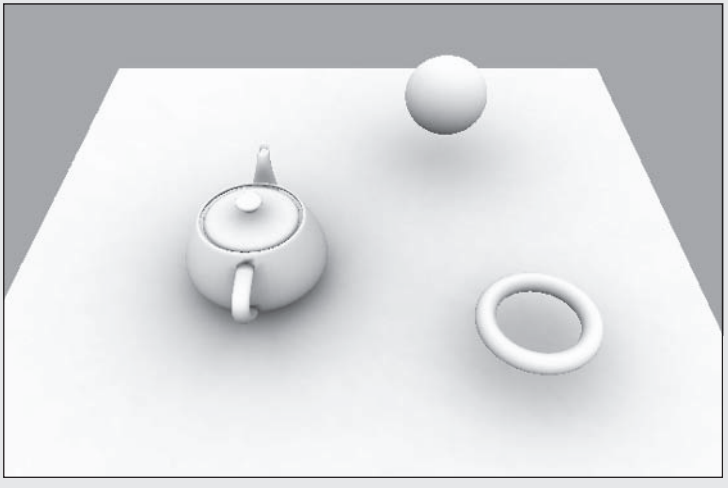


Figure 9.13. A cone light with a larger radius

9.3 SH with Cube Maps

In addition to a number of basic light types, D3DX also provides the functions necessary to generate an SH solution from a cube map. You've seen cube maps used to support environmental effects such as reflection and refraction, but those effects tend to be highly specular and don't recreate the smooth effects of highly diffuse scenes. You can create these subtle diffuse environmental effects by projecting the cube map into a set of SH coefficients. The result only preserves the low-frequency components of the cube map, which is what you want for diffuse lighting.

Example code demonstrating the use of cube maps for SH can be found on the CD in the \Code\Chapter 09 - SH in DX With CubeMaps directory. The sample is very similar to the basic SH sample, with one key exception. To simplify the sample even more, the cube map sample assumes that an SH solution has already been calculated and saved to a file. Therefore, use the first SH example application to generate data for any scenes you want to use in the cube map example application.

There are just a few new features in the new application. First, you have to create the actual cube map. This can be done using the basic D3DX cube map creation functions.

```
D3DXCreateCubeTextureFromFile(m_pD3DDevice,
   "..\\Textures\\cubemap.dds", &m_pCubeMap);
```

I load my cube map from a file, but you can use any of the D3DXCreateCubeTexture functions. At this point, it's useful to note that the cube map is just like any other cube map. You don't need to do anything different for SH. Next, you can use D3DX to project the cube map into a set of SH coefficients.

```
D3DXSHProjectCubeMap(ORDER, m_pCubeMap, m_LightValues[0],
   m_LightValues[1], m_LightValues[2]);
```

In the basic example, I created new lighting coefficients in every frame. I did that to simplify the code and because the operation was fairly inexpensive. Projecting cube maps can be quite a bit more expensive, so I call D3DXSHProjectCubeMap during initialization and save the results to a member variable that will be reused with every frame.

Once the light coefficients have been created, you no longer need the cube map (assuming you're not using it for something else). You could release it immediately. In any case, m_LightValues stores the SH light coefficients for use during rendering.

The final difference between the cube map sample and the basic sample can be found in the SetSHShaderConstants function. Here, the code rotates the coefficients and writes the rotated values to LightValues.

```
D3DXMATRIX Rotation;
D3DXMatrixRotationZ(&Rotation, (float)GetTickCount() /
   1000.0f);
D3DXSHRotate(LightValues[0], ORDER, &Rotation,
   m_LightValues[0]);
D3DXSHRotate(LightValues[1], ORDER, &Rotation,
   m_LightValues[1]);
D3DXSHRotate(LightValues[2], ORDER, &Rotation,
   m_LightValues[2]);
```

The rest of the function (and the shader) is exactly the same as in the previous example. There are a few things to note here. I arbitrarily chose rotation about the z-axis; however, any rotation matrix will work. Also, the previous example transformed the directional light before projecting the light into SH. Here, I rotate the lights that are already projected. In the case of a directional light, rotation before projection is probably the least expensive order of events. For a cube map, projection and then rotation is most likely far better than the alternatives. The point to remember is that you can rotate either the raw lights or the SH lights, but the most optimal choice will depend on the type of light. Finally, note that the m_LightValues data remains unchanged.

Typically, scenes lit with a cube map have more subtle shadows and coloring because the cube map encodes all of the direct and reflected light from the surrounding environment. As a result, the images tend to be richer and more realistic. Figures 9.14 and 9.15 are two examples of these effects. In Figure 9.14, most of the light is coming from an overhead source, resulting in dark shadows. In Figure 9.15, the light is more evenly distributed throughout the environment, resulting in softer shadows and other subtleties. In both cases, the full color version is more visually interesting; spend a bit of time experimenting with the sample code.

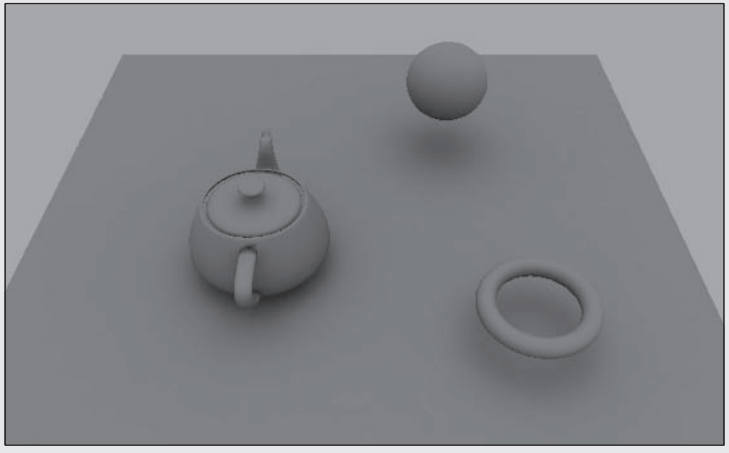


Figure 9.14. A scene lit by a cube map with primarily overhead light

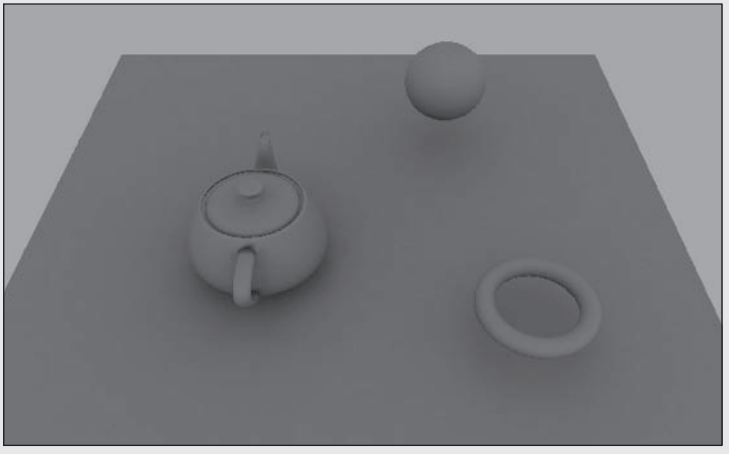


Figure 9.15. A scene lit by a cube map with more distributed light

9.4 DX SH with HDRI

Cube maps are an easy way to get a realistic lighting solution but, in the last chapter, you saw that you could also use HDR light probes to achieve similar effects. D3DX does not have native support for HDR files, but it is very easy to add your own code to use HDR data in a DirectX app.

The sample code for this can be found on the CD in the \Code\Chapter 09 - SH with HDRI directory. Like the cube map sample, this is a simplified sample that assumes that the SH data already exists for your mesh. Also, like the cube map example, this sample preloads and projects the HDR image and only preserves the resulting SH coefficients for later use.

The bulk of the new material can be found in the new LoadAndProjectHDRI function. This function has been written to minimize the changes to the HDR code from the previous chapter. Once you are familiar with the code, you can probably streamline it further.

```
BOOL CLightingApplication::LoadAndProjectHDRI(void)
{
```

These variables are only needed for the initial HDR loading and projection. They will not be needed later in the application.

```
    int         HDRIWidth;
    int         HDRIHeight;
    float      *pHDRImage;
    SHRGBColor *HDRISHCoeffs;
    SHSample   *sphericalSamples;
    int         SQRTNumSamples = 100;
```

The previous chapter discussed HDR loading in more depth. I reused the loading function with one minor change — here, I pass a pointer to a pointer to an array of float values. The function will allocate that pointer and pass back the HDR data in that array.

```
    loadHDRImage(m_pHDRIFileName, &HDRIWidth,
       &HDRIHeight, &pHDRImage);
```

Next, allocate and initialize the data needed for both the SH coefficients and the spherical sampling functions.

```
    HDRISHCoeffs = (SHRGBColor *) malloc(ORDER * ORDER
       * sizeof(SHRGBColor));

    memset(HDRISHCoeffs, 0, ORDER * ORDER * sizeof
       (SHRGBColor));

    sphericalSamples = (SHSample *) malloc
       (SQRTNumSamples * SQRTNumSamples *
       sizeof(SHSample));

    memset(sphericalSamples, 0, SQRTNumSamples *
       SQRTNumSamples * sizeof(SHSample));

    for (long i = 0; i < SQRTNumSamples *
       SQRTNumSamples; i++)
    {
       sphericalSamples[i].coeff = (double *)
          malloc(ORDER * ORDER * sizeof(double));

       memset(sphericalSamples[i].coeff, 0, ORDER *
          ORDER * sizeof(double));
    }
```

Generate a set of spherical samples as described in the previous chapter. Here, I am using the same function with one minor but very important change — the OpenGL version uses a different coordinate system. For the DirectX version, I have swapped the Y and Z components of the sample vectors.

```
sphericalStratifiedSampling(sphericalSamples,
   SQRTNumSamples, ORDER);
```

Use the function from the previous chapter to generate a set of SH coefficients for the HDR image. (See Chapter 8 for a more in-depth explanation.)

```
SHProjectHDRImage(pHDRImage, HDRIWidth, HDRIHeight,
   sphericalSamples, HDRISHCoeffs,
   SQRTNumSamples * SQRTNumSamples,
   ORDER * ORDER, 0.35f);
```

SHProjectHDRImage returns the SH coefficients as a set of doubles, but SetSHShaderConstants works with a set of floats. There are many ways to deal with this, but here I simply copy the doubles to a corresponding set of floats that will be used later in SetSHShaderConstants.

```
for (i = 0; i < ORDER * ORDER; i++)
{
   m_HDRICoeffs[2][i] = (float)HDRISHCoeffs[i].r;
   m_HDRICoeffs[1][i] = (float)HDRISHCoeffs[i].g;
   m_HDRICoeffs[0][i] = (float)HDRISHCoeffs[i].b;
}
```

Finally, free all of the intermediate data. You only need the final SH coefficients as stored in m_HDRICoeffs.

```
   free(HDRISHCoeffs);

   for (i = 0; i < SQRTNumSamples * SQRTNumSamples;
      i++) free(sphericalSamples[i].coeff);

   free(sphericalSamples);
   free(pHDRImage);

   return TRUE;
}
```

The resulting set of m_HDRICoeffs is used in exactly the same way as the cube map coefficients from the previous section.

```
D3DXMATRIX Rotation;
D3DXMatrixRotationY(&Rotation, (float)GetTickCount() /
   1000.0f);
D3DXSHRotate(LightValues[0], ORDER, &Rotation,
   m_HDRICoeffs[0]);
D3DXSHRotate(LightValues[1], ORDER, &Rotation,
   m_HDRICoeffs[1]);
D3DXSHRotate(LightValues[2], ORDER, &Rotation,
   m_HDRICoeffs[2]);
```

Again, the original coefficients are unchanged, and the rotated coefficients are used in the rest of the function as seen in the first example. Figure 9.16 shows an example of the scene lit by an HDR image. Again, you should experiment with the sample and different probes to get a better impression of full color effects.

So far, you have seen how one shader and the same basic code can be used to render scenes with several very different light sources. The same underlying techniques can be used to render very different materials as well.

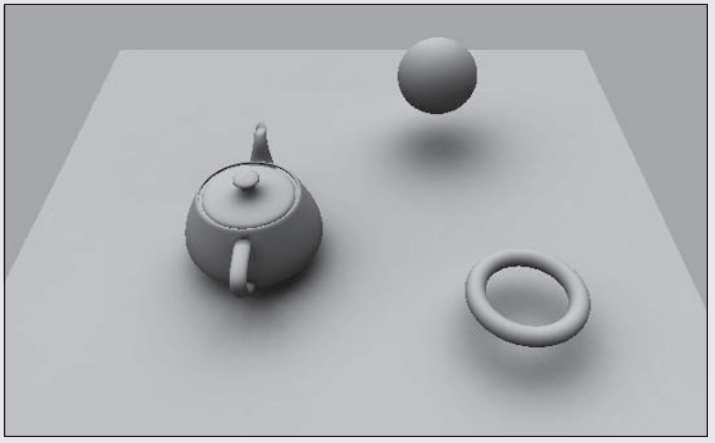


Figure 9.16. A scene lit by a cube HDR probe

9.5 Multiple Meshes/Materials

The last several examples have concentrated on changing the lighting conditions. This section will concentrate on rendering scenes with multiple materials. The major change here is that you need to send multiple meshes and a matching set of materials to the simulation function. Once you generate and set the correct coefficients in all your meshes, lighting and shading operations are the same as in the previous sections. The example code for this section can be found in the \Code\Chapter 09 - SH in DX More Materials directory.

To begin, I'll create a set of meshes and materials that will eventually be passed to the simulator. The framework already handles arrays of meshes, so the only major change here was that I had to go from a single material to an array of materials.

```
m_Models[0].pFileName = "..\\Models\\plane.x";
m_Models[1].pFileName = "..\\Models\\ball.x";
m_Models[2].pFileName = "..\\Models\\teapot.x";
m_Models[3].pFileName = "..\\Models\\torus.x";
ZeroMemory(&m_SHMaterials, sizeof(D3DXSHMATERIAL) *
   NUM_MODELS);
```

```
SetColorValue(&m_SHMaterials[0].Diffuse, 1.0f, 1.0f, 0.0f,
   0.0f);
SetColorValue(&m_SHMaterials[1].Diffuse, 1.0f, 1.0f, 1.0f,
   1.0f);
SetColorValue(&m_SHMaterials[2].Diffuse, 1.0f, 0.0f, 1.0f,
   0.0f);
SetColorValue(&m_SHMaterials[3].Diffuse, 1.0f, 0.0f, 0.0f,
   1.0f);
m_NumMaterials = 4;
```

The final line above is there so I can choose the number of meshes and materials I want to use while testing. I assume that material 0 will be used with mesh 0 and so on. The material counter variable is important for later data generation and extraction.

The next major change can be seen inside the thread where I call D3DXSHPRTSimulation. When using multiple meshes, the function takes an array of mesh pointers and an array of materials. Therefore, I will copy the pointers of each to a new pair of arrays. There are several ways I could have handled this, but this way fits well with the existing framework arrays.

```
for (long i = 0; i < m_NumMaterials; i++)
{
   m_pMeshes[i]              = m_Models[i].pModel;
   m_pSHMaterialPointers[i] = &m_SHMaterials[i];
}
```

The two arrays are arrays of mesh and material *pointers*. I pass these arrays to the simulation function just as I did the single mesh and material pointers. The rest of the parameters can be whatever you want. However, I recommend setting the bounce count greater than 1 so that you can see the reflected colors between objects.

```
if (FAILED(D3DXSHPRTSimulation(ORDER, m_NumMaterials,
   m_pMeshes, m_pSHMaterialPointers, NUM_RAYS,
   NumBounces, SubSurface, LengthScale, Spectral,
   &m_pSHDataBuffer, StaticSHCallback)))
      return 0;
```

The simulator will run as normal and generate a set of SH coefficients equal to the total number of vertices in the scene. You can compress the data as you did in the previous sections, but extraction can be a bit trickier because the PCA weights and cluster IDs need to be assigned to vertices in several meshes.

To handle this, I have changed some of the code in CompressDataAndSetVertices. Once you have the compressed results from D3DXSHPRTCompress, you can extract the cluster IDs as before. However, you also need to extract the PCA weights.

```
float *pPCAWeights = new float[BufferDescription.NumSamples
   * NUM_PCAVECTORS];
D3DXSHPRTCompExtractPCA(0, NUM_PCAVECTORS,
   pCompressedBuffer, pPCAWeights);
```

Next, the cluster IDs and PCA weights need to be assigned to all of the meshes. The ID and weight arrays are serialized across multiple sets of vertices. As long as you run through your mesh array consistently (in both the simulator and this code), you can easily set the right values to the correct vertices. To help, I create a variable (LastOffset) that will help me walk through the arrays.

```
MODEL_VERTEX *pV = NULL;
long LastOffset = 0;
```

Walk through each mesh.

```
for (long i = 0; i < m_NumMaterials; i++)
{
```

Lock each vertex buffer so you can set the individual vertices.

```
m_Models[i].pModel->LockVertexBuffer(0, (void**)&pV);
   for(UINT CurrentVertex = 0;
      CurrentVertex < m_Models[i].NumVertices;
      CurrentVertex++)
   {
```

The code to set the cluster index hasn't changed that much. The only difference is that LastOffset keeps track of the total vertex count so that you can iterate through each vertex but match it up against each value in the buffers that contain data for all vertices.

```
      float CurrentClusterIndex = (float)
         (pClusterIDs[CurrentVertex + LastOffset]
         * (1 + 3 * (NUM_PCAVECTORS / 4)));

      pV[CurrentVertex].ClusterID =
         CurrentClusterIndex;
```

In the previous samples, D3DXSHPRTCompExtractToMesh greatly simplified setting the PCA weights in a single mesh. Now you must set the PCA weights for individual meshes yourself. A memcpy works nicely, copying all of the PCA values to a vertex in one shot.

```
      memcpy(pV[CurrentVertex].PCAData, &pPCAWeights
         [(CurrentVertex + LastOffset) *
         NUM_PCAVECTORS], NUM_PCAVECTORS *
         sizeof(float));
   }
```

Finally, close this mesh and get ready to move to the next one. Increment LastOffset so that you properly index into the compressed data arrays.

```
   m_Models[i].pModel->UnlockVertexBuffer();
   LastOffset += m_Models[i].NumVertices;
}
```

Don't forget to delete the new data that you extracted with D3DXSHPRTCompExtractPCA.

```
SAFE_DELETE_ARRAY(pPCAWeights);
```

In the end, the effect is better in color, but Figure 9.17 shows several objects with different materials. In the color picture, the plane is red and the sphere is white. The light reflected from the plane casts a red light on the bottom of the sphere, highlighting the advantages of interreflections between colored objects.

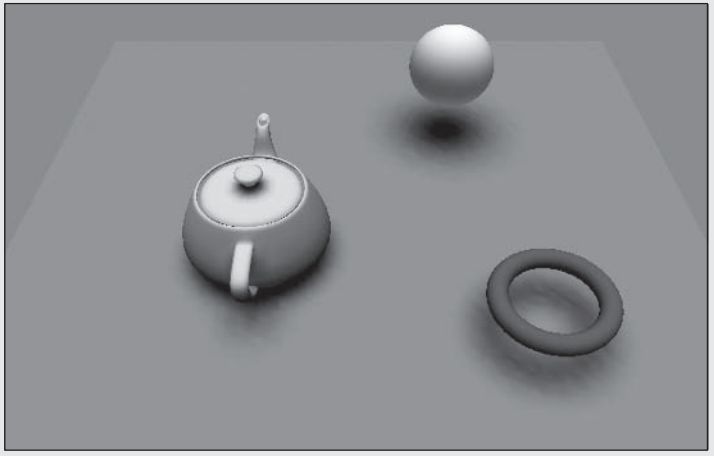


Figure 9.17. A heterogeneous scene with multiple materials

9.6 Subsurface Scattering

D3DX also implements a subsurface scattering model, based on the one described by Jensen, et al. [3]. The specifics of Jensen's model are beyond the scope of this book, but the high-level idea is that the subsurface scattering model accounts for light traveling into and through the object. Light is refracted, absorbed, and scattered along the way, before it leaves the object and becomes part of the outgoing radiance. The model can be quite complex to implement, but D3DX will do it for you with a few changes to your materials and a call to D3DXSHPRTSimulation.

The code for this can be found on the CD in the \Code\Chapter 09 - SH in DX with SS directory. It is based on the previous example, which allows you to set different materials for different meshes. In this case, I am going to set up the second material with subsurface scattering parameters. A description and rationale for each of the parameters can be found in [3]. Even if you are not completely comfortable with the math, there is a table in the paper that lists different parameters for common materials, which should help to explain the parameters by example. For instance, whole milk is a denser liquid than skim milk and has a

correspondingly higher absorption coefficient. Similarly, ketchup scatters more red than the other channels and absorbs less red. The values below are set for skim milk.

```
SetColorValue(&m_SHMaterials[1].Diffuse, 1.0f, 0.81f,
   0.81f, 0.69f);
m_SHMaterials[1].bSubSurf                  = true;
m_SHMaterials[1].RelativeIndexOfRefraction = 1.3f;
SetColorValue(&m_SHMaterials[1].Absorption, 1.0f, 0.70f,
   1.22f, 1.90f);
SetColorValue(&m_SHMaterials[1].ReducedScattering, 1.0f,
   0.0014f, 0.0025f, 0.0142f);
```

Once the material is set, the only other change is to tell D3DXSHPRTSimulation to include the subsurface scattering model in its solution. This is simply a matter of setting the corresponding flag. Also, the subsurface model works in units of length, so you need to set the length scale parameter to make sense with the scale of your model. Since I am using a very artificial scene, I remained consistent with the SDK sample with a value of 25.0, but you might want to set different values for meshes of "real" objects.

```
    float LengthScale  = 25.0f;
    bool  SubSurface   = true;

    if (FAILED(D3DXSHPRTSimulation(ORDER,
       m_NumMaterials, m_pMeshes,
       m_pSHMaterialPointers, NUM_RAYS,
       NumBounces, SubSurface, LengthScale, Spectral,
       &m_pSHDataBuffer, StaticSHCallback)))
       return 0;
```

Once D3DX computes a solution, working with the data is the same as any other example. From the perspective of the rendering equation, the subsurface scattering data is just another contributor to the exiting radiance.

I have, however, changed one small part of the lighting for the purposes of this example. I have added a second light to the environment because the subsurface scattering effect is best highlighted by a light moving behind the object. In this sample, the main light is effectively orbiting around the model and a second light is shining straight down. In addition to lighting the object from two directions, it also allows me to show the SH addition helper function in D3DX.

D3DXSHAdd can be helpful when adding two different SH approximations. This can be useful for adding a second light to the scene or another light to a cube map solution. In any case, you simply pass an output array, an order, and both sets of input coefficients. The following code computes and then adds two lighting solutions for the overhead and main light.

```
float LightValues[3][ORDER*ORDER];
float LightValues1[3][ORDER*ORDER];
float LightValues2[3][ORDER*ORDER];



D3DXSHEvalDirectionalLight(ORDER, pObjectSpaceLightVector,
   1.0f, 1.0f, 1.0f, LightValues1[0],
   LightValues1[1], LightValues1[2]);
D3DXSHEvalDirectionalLight(ORDER, &(D3DXVECTOR3(0.0f,
   1.0f, 0.0f)), 1.0f, 1.0f, 1.0f, LightValues2[0],
   LightValues2[1], LightValues2[2]);
```

```
D3DXSHAdd(LightValues[0], ORDER, LightValues1[0],
   LightValues2[0]);
D3DXSHAdd(LightValues[1], ORDER, LightValues1[1],
   LightValues2[1]);
D3DXSHAdd(LightValues[2], ORDER, LightValues1[2],
   LightValues2[2]);
```

Finally, Figure 9.18 shows a scene where all the objects are rendered with subsurface scattering In this shot, the light is coming from behind the objects, and you get the effect that some light is shining through. Overall, this creates a softer feel that is much more lifelike for materials such as marble. The printed figure doesn't convey all of the subtleties, especially for colored objects. Spend a bit of time experimenting with different materials either in this sample or in the samples provided with the SDK.

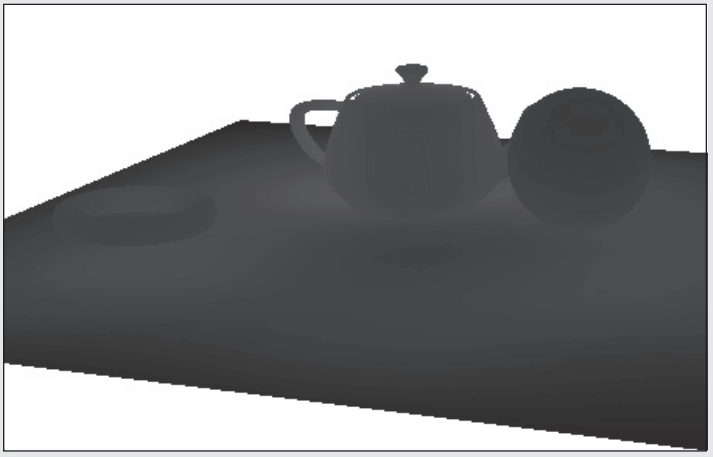


Figure 9.18. A scene with subsurface scattering

9.7 Simple Specular Highlights

For one last technique, I would like to address one of the limitations of SH — mainly that the material presented thus far is really only good for diffuse lighting. There has been work done in encoding arbitrary materials with SH [2], but these techniques can be fairly slow. While I highly recommend reading the papers listed in the reference section, I would like to demonstrate a very simple way to add specular highlights to an SH lit scene.

9.7.1 The Basic Idea

Low-order SH solutions are suited for diffuse lighting in that they encode the low-frequency aspects of direct and reflected light in an environment. This makes them less suitable for specular reflection, where objects might reflect relatively high-frequency lighting components back to the viewer. So, you need a more detailed version of the environment, but that version only needs to preserve components that would produce specular highlights. If you store that in a cube map and calculate basic environmental mapping, you will get the

specular contribution for each vertex. From there, you can produce an effect that closely follows Phong if you simply add the mapped specular component to the SH diffuse component. First, though, you have to generate a "specular view" of the environment.

While this might not be entirely correct for all situations, assume that the only light that will produce a specular highlight will be light coming directly from a light source and perhaps highly reflective surfaces. Also, the specular view of the world will be dependent on the material properties of the object. A perfectly matte object will not have specular highlights, so the specular view is, for all intents and purposes, black. Moderately shiny objects will have less tightly defined specular highlights. Their specular view of the environment will have soft edges. With those assumptions, consider the light probe in Figure 9.19.

Most specular highlights come from the overhead light, the windows, and other direct light sources. Therefore, we want to generate another picture that preserves only those components. There are several ways to do this, but one way is to convert the image to grayscale and lower both the brightness and the contrast. This does two things. First, it eliminates everything but the brightest components. Second, it darkens the image overall. This can be important because the Phong-like technique is additive. If the specular view is too bright, the highlights will saturate the color of the pixels and tend to produce very hard edged highlights. Figure 9.20 shows the resulting image.

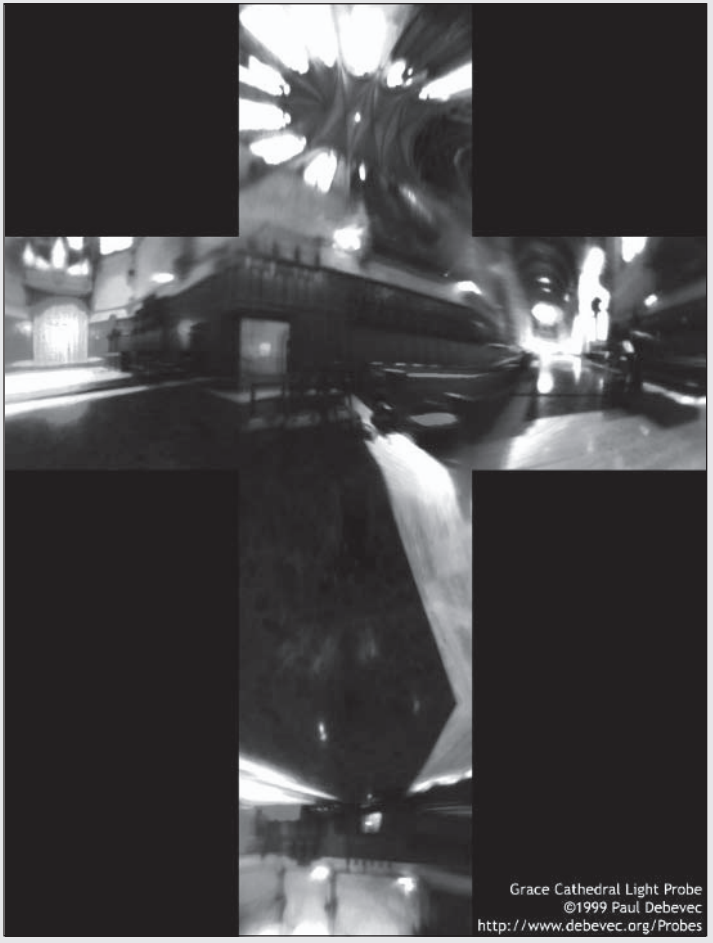


Figure 9.19. The cathedral light probe in the cross configuration

Figure 9.20. The same probe with only specular contributors

If each of these two probes were converted to cube maps, the original probe could be converted to SH coefficients and used for diffuse lighting, while the specular view could be used as a cube map for standard environment mapping. If the two were brought together, the result would be a Phong-like effect as shown in Figure 9.22.

Also, different levels of shininess and specular power can be simulated by preprocessing the specular view. Figure 9.21 shows how a blurred cube map can produce a less shiny appearance.

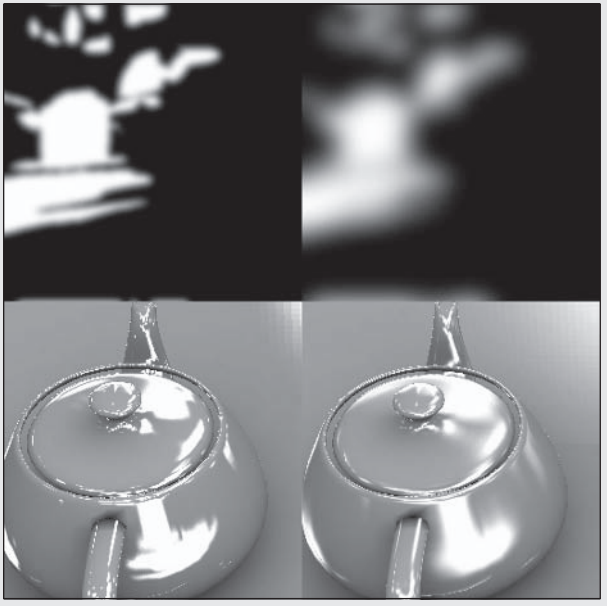


Figure 9.21. Adjusting specular properties by blurring the cube map

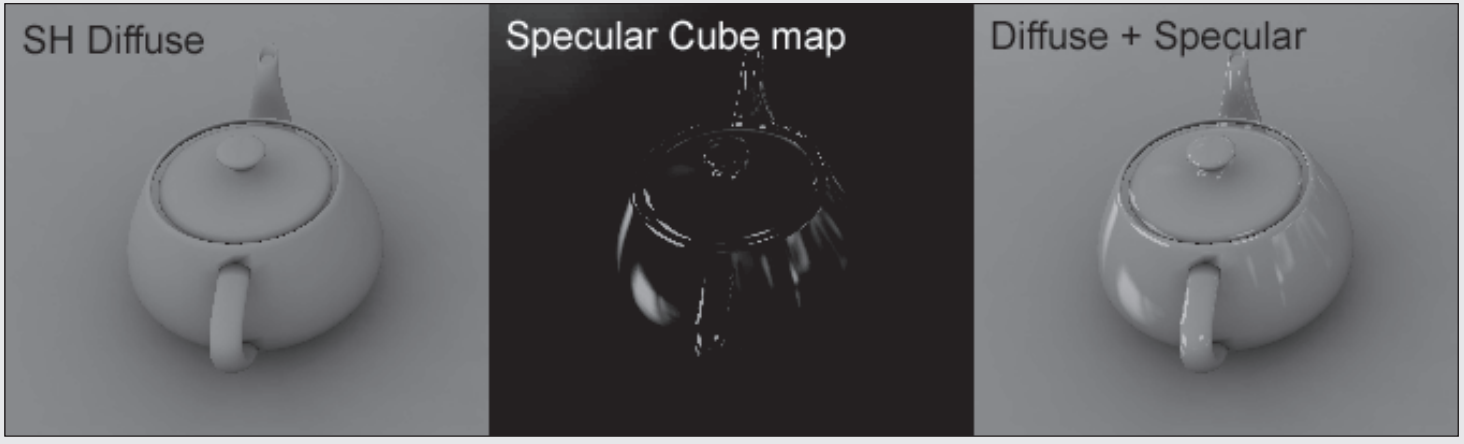


Figure 9.22. Adding specular and diffuse components

9.7.2 The Implementation

This effect is very simple to implement. The sample code is on the CD in the \Code\Chapter 09 - SH in DX with Specular directory. The diffuse component is exactly the same as the cube map example, with one notable change. Because there are a few more shader constants to set for the environment mapping, the SH cluster constants begin at c9 instead of c4. Both the application code and the shader must be changed.

Like the previous samples, this application rotates the lighting coefficients. However, in the previous samples, the rotation was added to create a more interesting effect. In this sample, the rotation serves a more specific purpose.

As part of the general framework, the left and right arrow keys change the world angle, spinning the model. Therefore, the lighting effect I'll create is one where the environment is stationary and the model turns within that static environment. However, the SH coefficients are in object space. As the world turns, so does the diffuse lighting. To prevent that, I will rotate the SH coefficients in the opposite direction.

```
D3DXMATRIX Rotation;
D3DXMatrixRotationY(&Rotation, -m_WorldAngle);
D3DXSHRotate(LightValues[0], ORDER, &Rotation,
   m_LightValues[0]);
D3DXSHRotate(LightValues[1], ORDER, &Rotation,
   m_LightValues[1]);
D3DXSHRotate(LightValues[2], ORDER, &Rotation,
   m_LightValues[2]);
```

The end result of this is that the scene rotates, but the lighting environment stays fixed. With a fixed diffuse environment, you don't need to worry about rotating the specular environment. With those two caveats noted, the diffuse lighting component is rendered in exactly the same way you've seen throughout this chapter.

The specular component requires a few new pieces. Most of the real work happens in the shader, but reflection calculations require an eye position and a world transformation. Therefore, you need to pass those two items to the shader as constants. Note the register numbers.

```
m_pD3DDevice->SetVertexShaderConstantF(4, (float
   *)&m_EyePosition, 1);
m_pD3DDevice->SetVertexShaderConstantF(5, (float
   *)&m_World, 4);
```

Finally, you need to set the specular cube map texture (which is actually used as a texture rather than a raw material for SH coefficients) and a color operation. Normally, the default color operation is a multiplication of the color and the texel value. Instead, set the color operation to add. The output value will then be the sum of the diffuse SH value and the specular highlight from the cube map texture.

```
m_pD3DDevice->SetTexture(0, m_pSpecMap);
m_pD3DDevice->SetTextureStageState(0, D3DTSS_COLOROP,
   D3DTOP_ADD);
```

This could also be done in a pixel shader, and a pixel shader might make more sense if you were adding more effects. For this simple example, a pixel shader isn't necessary. Setting the texture state allows the sample to run on lower-end video cards that don't support pixel shaders.

At this point, everything is in place. The application has set everything up for the vertex shader. The shader can be found on the CD as \Code\Shaders\SH_VS_Specular.hsl. The core of the shader is the same as what you've seen before, with a few changes and additions.

The incoming vertex data is unchanged, but you need to output a three-component texture coordinate to use the cube map. The new lines in the shader are all directed toward computing a final value for SpecTex.

```
struct VS_OUTPUT
{
   float4 Position  : POSITION;
   float4 Diffuse   : COLOR0;
   float3 SpecTex   : TEXCOORD0;
};
```

In addition to the main transform, the shader now needs a separate world transform and an eye position. Note the new register number for the PCA data. Aside from the new register position, the diffuse SH shader code is unchanged.

```
matrix Transform                      : register(c0);
float3 EyePosition                    : register(c4);
matrix WorldTransform                 : register(c5);
float4 ClusteredPCA[200*(1+3*(24/4))] : register(c9);
```

The shader begins with the diffuse SH calculations. Once it determines the diffuse output value, the shader moves on to calculating the texture coordinates for the specular component. First, it calculates the world space position of the

vertex along with the world space normal. Next, the position is used to find the vector between the vertex and the eye. Finally, the specular texture coordinate is found as the reflection vector based on the normal and the view vector.

```
float3 Position    = mul(WorldTransform,
   InVertex.Position);
float3 Normal      = normalize(mul(WorldTransform,
   InVertex.Normal));
float3 EyeToVertex = normalize(EyePosition - Position);
OutVertex.SpecTex  = 2.0f*Normal*dot(Normal,EyeToVertex)
   -EyeToVertex;
```

Figure 9.23 shows the final result. Compare this image to the purely diffuse images earlier in the chapter. While the effect might not be entirely accurate, it is visually pleasing.

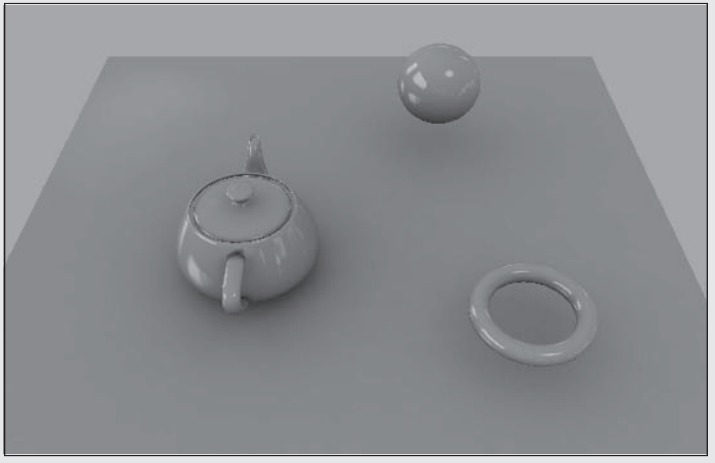


Figure 9.23. The final Phong-like scene

Conclusion

The first application in this chapter was a greatly simplified version of the SHPRTVertex sample in the DX SDK. I wanted to highlight the key SH components without getting bogged down in the UI or niceties like error checking. Once you understand the basic ideas here, I strongly recommend looking at the real sample in more depth. Also, the nicer UI in the sample makes it easier to experiment with different SH and PCA parameters.

Additionally, I made the decision not to cover per-pixel spherical harmonics. The relatively steep requirements of a PS2.0-based technique make it less attractive than the vertex shader version for the purpose of teaching concepts. If you're interested in the per-pixel version, the SDK sample is very similar to the SDK's vertex shader example with a few minor changes. Once you understand the basic concepts, other instantiations are easy to follow.

Also, I alluded to several other techniques that make use of SH and similar technologies. Those techniques are featured in the papers listed in the reference section. I recommend taking a look at the papers. Even if you don't plan on implementing the techniques, it can still be very interesting to see all of the different ways a technology can be used.

References

[1] Sloan, Peter-Pike, Jesse Hall, John Hart, and John Snyder, "Clustered Principal Components for Precomputed Radiance Transfer," *ACM Transactions on Graphics (TOG)*, Vol. 22, Issue 3, July 2003 (SIGGRAPH), pp. 382-391.

[2] Kautz, Jan, Peter-Pike Sloan, and John Snyder, "Fast, Arbitrary BRDF Shading for Low-Frequency Lighting Using Spherical Harmonics," *Eurographics Rendering Workshop 2002*, June 2002.

[3] Jensen, Henrik Wann, Steve Marschner, Marc Levoy, and Pat Hanrahan, "A Practical Model for Subsurface Light Transport," *Proceedings of SIGGRAPH 2001*, August 2001, pp. 511-518.

This page intentionally left blank.

Chapter 10

Toward Real-Time Radiosity

Introduction

So far, you have seen several lighting techniques that could be applied to render 3D scenes in real time (such as spherical harmonic lighting) or not (such as raytracing — although modern GPUs and programmability could soon change that). Some of these techniques were aimed at taking into account specular lighting effects (reflection and refraction, for instance, with raytracing) or diffuse lighting (as with spherical harmonic lighting). In this chapter, I explore another popular field of computer graphics in general, and rendering techniques in particular: radiosity. This rendering method's primary purpose is to take into account the global lighting of a scene regardless of the point of view from which it is rendered. I will first explain the background of the method, the theory behind it, and a bit of its history and evolution. I will then explain the different methods to calculate the radiosity solution of a scene. This section will cover the matrix solution, including the progressive refinement method that allows calculating the radiosity solution of a scene over multiple iterations, with the ability to visually check the result on the screen after each iteration. This is where you will see how radiosity solutions can be calculated in near real time. Finally, I will point to an implementation of a radiosity processor using the OpenGL APIs and 3D hardware acceleration.

10.1 Radiosity Background and Theory

In this section, I describe the main goals and background of the radiosity method, as well as its theory. I will build on previous chapters to explain the theory and conclude with the formulation of a key aspect of radiosity: form factors.

10.1.1 What Radiosity Tries to Achieve

You saw in Chapter 3 that the raytracing algorithm, in its simplest form, is very good at reproducing specular reflections and specular refractions. In other words, raytracing is very good at simulating local direct illumination and specular effects at a given point on the surface of an object. After its introduction, it became clear that it needed to be extended to take into account effects due to global illumination such as color bleeding and global diffuse lighting. As you saw in Chapter 3, various techniques such as Monte Carlo raytracing and photon mapping have been developed with these goals in mind. In parallel, another method to achieve the same goals, but in a different way, was introduced in the mid-'80s and was called radiosity. The name comes from the radiometric term the method computes.

The radiosity methods try to account for global illumination effects by using a different approach than that of the raytracing algorithm. Instead of tracing rays out into the world from the camera's position, it divides the geometric primitives and the lights of the scene into small elements (called patches), and then computes an approximation of the rendering equation for each of these patches, regardless of the position of the camera in the virtual scene. As such, the method calculates the lighting of a scene globally, and renderings of the same scene can be created from any point of view. In other words, radiosity is a view-independent technique. Using 3D hardware or even a simple software renderer with Gouraud shading, it is even possible to render successive frames in real time once the solution has been computed, provided that the lights and geometry of the scene don't change. In the mid-'90s, games started using this technique to achieve levels of realism unsurpassed at that time.

10.1.2 Historical Background and Evolution

Radiosity techniques were first introduced in the '50s by engineers to solve problems in thermometry. The techniques were applied in the field of computer graphics in the mid-'80s. In the first applications, the radiosity method expressed the balance of lighting distribution in the scene by a set of linear equations to solve. Also, the surfaces on which to apply the techniques were limited to perfectly diffuse ones.

One limitation of the first radiosity methods was aliasing artifacts appearing where the meshing of the scene was not detailed enough in regions presenting high variations of lighting. Hierarchical radiosity methods provided solutions to this problem. Instead of subdividing the primitives of the scene in grids of patches of the same size, these methods adaptively added patches where necessary and skipped expensive calculations in regions where lighting was relatively flat by

dynamically lowering the number of patches. The methods not only gained in performance, but also in the quality of the solution computed, and thus the higher level of realism achieved.

The basic radiosity methods, as you will see later in this chapter, can be expressed as a set of linear equations to solve. These equations establish the interaction between a patch in the scene and every other patch. For n patches, this leads to a matrix of n^2 elements (n x n matrix). As you can see, as the number of polygons grows in the scene (as does the number of patches subdividing the polygons), the matrix to solve becomes even larger. If you take the example of a scene with 6,000 polygons subdivided into 9 patches each, the resulting matrix is made of 2,916,000,000 elements (6,000 x 6,000 x 9 x 9); using 32-bit floating-point values, the matrix would require 11,664,000,000 bytes of storage! Storing this matrix in memory is one thing; solving it is another. The other major drawback of earlier radiosity methods is the fact that you need to fully compute the radiosity solution before displaying the results.

A few years after the introduction of radiosity applied to computer graphics, a major advancement was achieved when the progressive refinement method was introduced. Instead of fully solving the radiosity equations and then displaying the result from any angle, this method progressively distributes (or radiates) the light in the environment from the brightest patches to the other patches in the scene. The first step of the method consists of calculating the total energy in the scene. Then, all patches in the scene are initialized with their corresponding energy (i.e., no energy for surfaces, light energy for light sources). For each iteration, a patch with the highest energy is chosen. It distributes the energy over all the other patches and decreases the total energy to distribute in the scene accordingly. The results of an iteration can be displayed after it has finished. The results will be visually dark at first (only the lights will appear), but the scene will become brighter as the energy is distributed after each iteration. A radiosity solution can therefore be quickly shown to the end user, and is refined over the successive iterations. With this approach, you no longer need to store the radiosity matrix, and early results can be displayed interactively.

10.1.3 Near Real-Time Radiosity

Since the introduction of the progressive refinement approach, radiosity has taken advantage of the use of 3D hardware acceleration. Indeed, as you will see later, the way the energy is distributed from one patch to the rest of the scene can be accelerated greatly by the Z-buffer algorithm (present in all 3D hardware) to determine visibility. The recent introduction of programmable GPUs even allows off-loading the CPU to calculate a radiosity solution solely on 3D hardware. I give references at the end of this chapter to recent research papers where this capability has been exploited.

10.2 Radiosity Theory and Methods

In this section, I explain the theory behind radiosity. In doing so, I use the different definitions of radiometry that you have seen in various sections of this book, and I also formulate the radiosity equation derived from the rendering equation that was introduced in Chapter 8. Some mathematics is necessary to cover this subject, so I recommend reading section 10.2.2.

The next topic of this section centers on a key concept in radiosity: form factors. You will see what is hidden behind this term. I conclude this section by describing two main methods to solve the rendering equation, and show how 3D hardware can help.

10.2.1 Definitions

Before formalizing the radiosity equation, let's go over some definitions and the underlying terminology of global illumination.

The first definition needed in radiosity is that of the solid angle. The solid angle is the extension to 3D of the "standard" angle that you already know in 2D. Instead of working on the unit circle in two dimensions, you calculate the projection of a given object onto the unit sphere. Figure 10.1 shows the solid angle ω of a shape as seen from a given point.

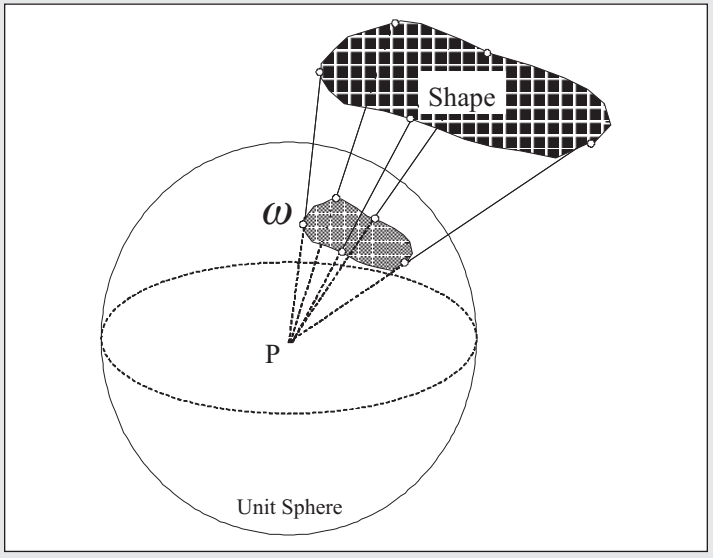


Figure 10.1. Solid angle

A solid angle is expressed in steradians (*sr*). The solid angle of the whole unit sphere is 4π steradians (its surface is $4\pi r^2$ with $r = 1$).

For small areas, the differential solid angle is used. Take a point on the unit sphere. The spherical coordinates of this point are (r, θ, φ). The area of a small differential surface dA (see Figure 10.2) at this point is given by:

$$dA = r^2 \sin\theta \, d\theta \, d\varphi$$

The differential solid angle, $d\omega$, is given by:

$$d\omega = \frac{dA}{r^2} = \sin\theta \, d\theta \, d\varphi$$

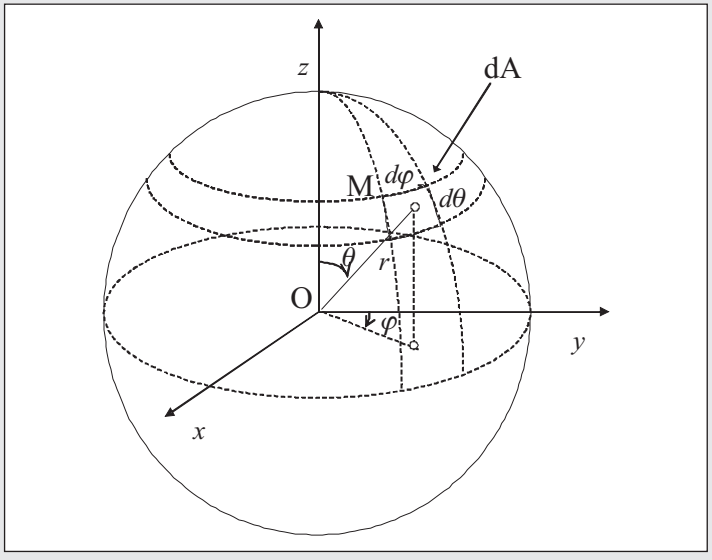


Figure 10.2. Solid angle and spherical coordinates

A very important quantity in radiosity is the radiance. It is the flux emitted in a given direction per unit solid angle per unit area. It is expressed in *W/(sr.m²)*. Take a small area, *dA*, on a surface around a given point *M*, and take a ray of light leaving this surface at this point with an angle of θ (not to be confused with the polar angle in spherical coordinates I used previously) with the normal of the surface. Call $d\omega$ the differential solid angle in the direction of the ray. Call Φ the radiant power (or radiant flux). The radiance L is given by the following formula:

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

The $dA.cos\theta$ term is simply the projection of the area along the ray's direction. Figure 10.3 summarizes the notion of radiance.

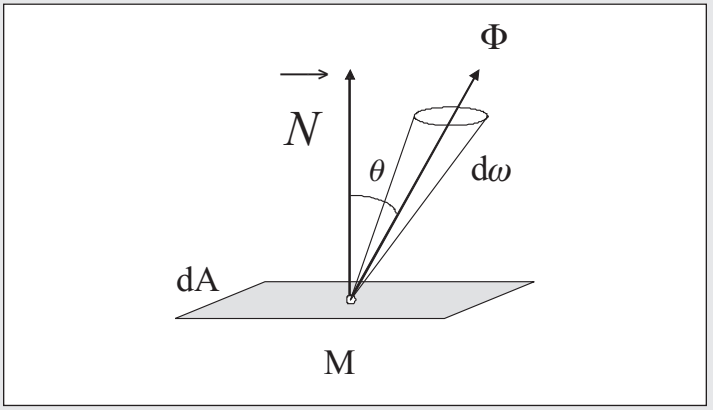


Figure 10.3. Radiance

The radiance presents some interesting properties. The most interesting one is that it is not attenuated by distance. In other words, it remains constant as it propagates in space (provided that there is no interaction with intervening media,

such as scattering or absorption). If you take two points in space that are visible to each other (i.e., not blocked by any shape), the radiance coming from the first point in the direction of the second point will be the same as the radiance coming from the second point in the direction of the first point. As you will see later, this property will help in simplifying the radiosity equation.

Another very important quantity in radiosity is… radiosity! The real name of the radiosity quantity is *radiant exitance*. It is defined as the amount of energy leaving a surface per unit area. The radiosity B at a given point on a surface is given by the following formula:

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

By using the previous equation of the outgoing radiance (*outgoing* radiance because the energy *leaving* the surface is concerned) and integrating it over the hemisphere Ω, you get the following formula:

$$B = \int_{\Omega} L_o \cos\theta \, d\omega$$

Similarly for light sources, the exitance or self-emitted radiosity is the amount of energy *emitted* by the surface of the light per unit area. It is denoted as E and is given as an integral over the hemisphere Ω of the *emitted* radiance by the following formula:

$$E = \int_{\Omega} L_e \cos\theta \, d\omega$$

Of course, all these different quantities will be used in the radiosity equation.

Finally, before introducing the equation that is at the core of all radiosity methods (the radiosity equation), now is a good time to revisit the rendering equation that you have already seen in various places throughout this book. The most common formulation of the rendering equation (without any participating media) is as follows:

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + L_r(x,\vec{\omega})$$

It can be translated as follows: The outgoing radiance at point x on a surface in the outgoing direction ω is the sum of the emitted radiance and the radiance reflected from this point in this direction. The emitted radiance is only present when the surface radiates light on its own; in other words, it exists only in the case of light sources (and also for glowing objects that can be approximated as light sources). The reflected radiance can be expressed as the integral over the hemisphere of incoming directions at point x of the product of the BRDF and the incident radiance (the incoming light).

$$L_r(x,\vec{\omega}) = \int_{\Omega} f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}')(\vec{\omega}' \bullet \vec{n}) d\omega'$$

The rendering equation can then be formulated as follows:

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \int_{\Omega} f_r(x,\vec{\omega}',\vec{\omega}) L_i(x,\vec{\omega}')(\vec{\omega}' \bullet \vec{n}) d\omega'$$

As you may have guessed, this equation cannot be resolved analytically as it has unknowns on both sides. This is why numerical methods have to be used in this case. The radiosity method (or set of methods) is a numerical approximation of this general equation. As you have seen in other chapters of this book, several methods can be used to solve the equation, one of which is the method often referred to as spherical harmonic lighting.

10.2.2 The Radiosity Equation

The radiosity equation is derived from the rendering equation that you saw previously. In order to simplify the general case equation, several assumptions and simplifications have to be made.

The first and most important assumption made in the radiosity method is to take into account only ideal diffuse surfaces, or *Lambertian* reflectors or emitters. A Lambertian surface has a constant radiance that is independent of the viewing direction. In other words, such surfaces emit or reflect light with a distribution that is independent of the incoming direction. The radiance of Lambertian surfaces only depends on the position of the point of concern on such surfaces, and no longer on the direction (since light is equally reflected or emitted in space). In addition, the BRDF is constant, and can be taken out of the integral on the right side of the rendering equation. It can be proved easily that, in this case, the BRDF is the ratio of the reflectance (defined as the ratio of the reflected energy and the incident power) and π.

Going back to the rendering equation, and taking into account this assumption and this simplification, you get the following equation:

$$L_o(x,\vec{\omega}) = L_e(x,\vec{\omega}) + \frac{\rho(x)}{\pi}\int_\Omega L_i(x,\vec{\omega}')(\vec{\omega}'\bullet\vec{n})d\omega'$$

The assumption of perfect diffuse reflectors or emitters also leads to another interesting property: Radiance and radiosity are proportional. Since radiance is only a function of the position x (i.e., no longer a function of the position and direction ω), the following equation can be derived from the definitions you saw earlier:

$$B(x) = \int_\Omega L_o \cos\theta\, d\omega = L_o(x)\int_\Omega \cos\theta\, d\omega = \pi L_o(x)$$

In the same way, the exitance can be expressed as follows:

$$E(x) = \pi L_e(x)$$

Multiplying the simplified rendering equation above by π, you get:

$$B(x) = E(x) + \rho(x)\int_\Omega L_i(x,\vec{\omega}')(\vec{\omega}'\bullet\vec{n})d\omega'$$

The integral on the right side of this equation can be further transformed. If you recall, the key property of radiance is that it doesn't change between two mutually visible points. This means the incoming radiance is equal to that coming from another visible point in the scene. For y, another point visible from x, the incoming radiance is:

$$L_i(x) = L_o(y) = \frac{B(y)}{\pi}$$

By introducing the visibility function $V(x,y)$, that is equal to 0 if the two points are not mutually visible and 1 otherwise, the following general expression of the incoming radiance can then be formulated:

$$L_i(x) = \frac{B(y)}{\pi} V(x,y)$$

By putting it all together and using the expression of the differential solid angle, you obtain the following modified rendering equation:

$$B(x) = E(x) + \rho(x) \int_S B(y) V(x,y) \frac{\cos\theta \cos\theta'}{\pi r^2} dA_y$$

Figure 10.4 gives a visual description of the different terms in this equation.

The idea of the radiosity method, as I briefly touched on earlier, is to split the environment into little geometrical elements called patches (that can be triangles or quadrilaterals). The integral over all surfaces of the scene in the simplified rendering equation above can then be changed into a sum over the different patches making the scene. The assumption is that each patch will present a uniform radiosity across its shape (and thus can be taken out of the integral on the right side of the equation). This leads to:

$$B(x) = E(x) + \rho(x) \sum_j B_j \int_{y \in P_j} V(x,y) \frac{\cos\theta \cos\theta'}{\pi r^2} dA_y$$

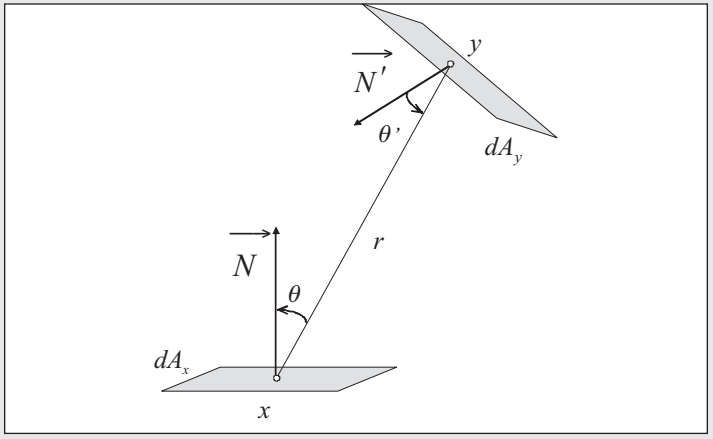


Figure 10.4. Two visible points and their geometric relationships

For a given patch, the constant radiosity is given by:

$$B_i = \frac{1}{A_i} \int_{x \in P_i} B(x) dA_x$$

Similarly, the constant exitance of a patch of light source is:

$$E_i = \frac{1}{A_i} \int_{x \in P_i} E(x) dA_x$$

The previous formulation of the rendering equation is for a given point on a given surface. If you now consider the same equation, but for a patch, you need to integrate both sides of the equation over the points of that patch. In other terms:

$$\frac{1}{A_i}\int_{x\in P_i} B(x)dA_x = \frac{1}{A_i}\int_{x\in P_i} E(x)dA_x +$$
$$\frac{1}{A_i}\int_{x\in P_i} \rho(x)\sum_j B_j \int_{y\in P_j} V(x,y)\frac{\cos\theta\cos\theta'}{\pi r^2}dA_y dA_x$$

Since the reflectance ρ is constant for a given patch, this equation can be transformed into:

$$B_i = E_i + \rho_i \sum_j \frac{B_j}{A_i}\int_{x\in P_i}\int_{x\in P_j} V(x,y)\frac{\cos\theta\cos\theta'}{\pi r^2}dA_y dA_x$$

Or more simply:

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

This is the radiosity equation. It expresses the relationship between the radiosity of a given patch, the exitance of that patch, and the radiosity of all other patches present in the scene. The form factor F is given by the following equation:

$$F_{ij} = \frac{1}{A_i}\int_{x\in P_i}\int_{y\in P_j} V(x,y)\frac{\cos\theta\cos\theta'}{\pi r^2}dA_y dA_x$$

The radiosity equation and the form factors are at the core of all radiosity methods. More precisely, radiosity methods are all about efficiently computing the form factors in a scene and then solving the radiosity equation.

10.2.3 Form Factors

Let's now see some of the core properties of form factors and how they can be used to more easily determine form factors.

10.2.3.1 Properties

A way to look at a form factor is to consider it as the fraction of energy leaving a patch and being received by another patch. Its mathematical expression shows that it solely depends on the geometry of the patches involved (geometry and relative positions). It is also the most complex quantity (four-dimensional integral) to compute in radiosity, and the most time-consuming task.

Form factors present several interesting properties that will be used to compute them more easily:

- Form factors are positive or null. A form factor is null when the two patches involved are not mutually visible:

$$\forall i, \forall j, F_{ij} \geq 0$$

- Form factors present a reciprocity relation. Indeed it can be proved that:

$$A_i F_{ij} = A_j F_{ji}$$

- In a closed environment, it can also be proved that:

$$\forall i, \sum_j F_{ij} = 1$$

10.2.3.2 Determination

As stated earlier, form factor determination is at the core of all radiosity methods. The general expression of a form factor between two patches is given by the following equation:

$$F_{ij} = \frac{1}{A_i}\int_{x\in P_i}\int_{y\in P_j} V(x,y)\frac{\cos\theta\,\cos\theta'}{\pi r^2}dA_y dA_x$$

Generally speaking, this expression cannot be solved directly. It can only be solved analytically or numerically, the latter method being easier.

Several analytical methods exist, ranging from direct integration to contour integration. These methods are not practical for implementation, and I will not describe them here. Interested readers can refer to [1] and [2] for more details on the subject.

Numerical methods are more useful for determining form factors. More precisely, projection methods can be used. Two main projection methods have been developed: the Nusselt's analogy (projection of a patch onto a unit sphere) and the hemicube algorithm (projection of a patch onto a unit half-cube, or hemicube). These methods take into account the visibility function in the computation of a form factor. In both cases, the assumption made is that the computation of the form factor between two patches can be simplified into the computation of a form factor between a point and a patch. This assumption is valid when the distance between the two patches is much greater than their relative size. If this is not the case, the receiving patch can be split into smaller elements (small differential areas) to apply the same simplification.

In the case of the Nusselt's analogy method, a unit hemisphere is centered around the receiving patch; the emitting patch is first projected onto this hemisphere, and then on the base plane (the tangent plane to the hemisphere), as shown in Figure 10.5.

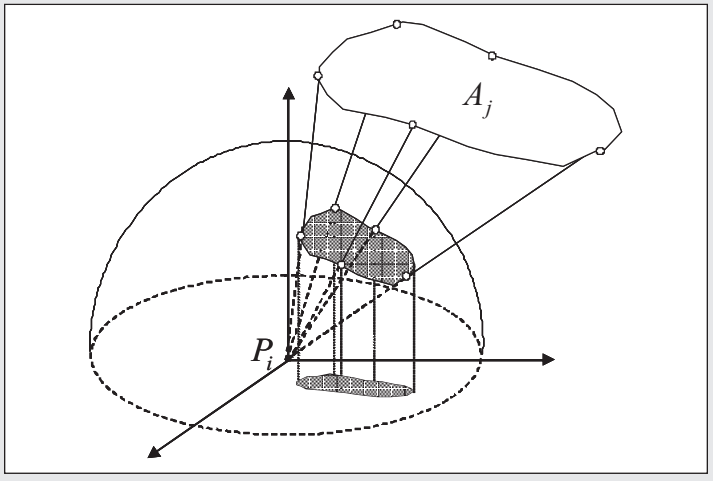


Figure 10.5. Nusselt's analogy projection method

It can be proved that the form factor is equal to the double projection of the patch onto the base plane (see [3] for more details). The main advantage of this method is that it can be applied to any patch. However, projecting a patch onto the unit hemisphere and then onto its base plane is not a simple task. Adding the projection of the visible parts of the patch

increases the complexity of such a task. Also, the unit hemisphere needs to be divided into a grid of regular cells; a form factor is calculated for each small differential (called the delta form factor), and the final form factor is simply calculated as the sum of all delta form factors for each cell. Several methods have been developed to project polygons and calculate their form factors using the Nusselt's analogy (see [3] for more details).

The hemicube algorithm is the most widely used numerical projection method to calculate form factors. Instead of using a unit hemisphere around the point of interest, a unit hemicube is used. The hemicube is divided uniformly into a grid of rectangular cells. The patches of the scene are then projected onto all five sides of the hemicube (the top face and the four surrounding half faces). For each cell of the grid, a delta form factor is calculated. Once a patch has been projected onto each face of the hemicube, the form factor is simply the sum of all delta form factors. Figure 10.6 shows an illustration of the hemicube.

The main advantage of this projection method is that it is very similar to a 3D viewing system projection method. Indeed, the point (differential area) for which you want to calculate the form factor is similar to the position of the synthetic camera. Similarly, any of the hemicube faces can be considered as the image plane (the base of the viewing pyramid), and a hemicube cell as a pixel. The field of view of the hemicube "camera" is 90 degrees. The visibility function for each patch in the scene is very similar to the perspective projection coupled with the hidden surface removal technique of standard 3D libraries. In other words, using a

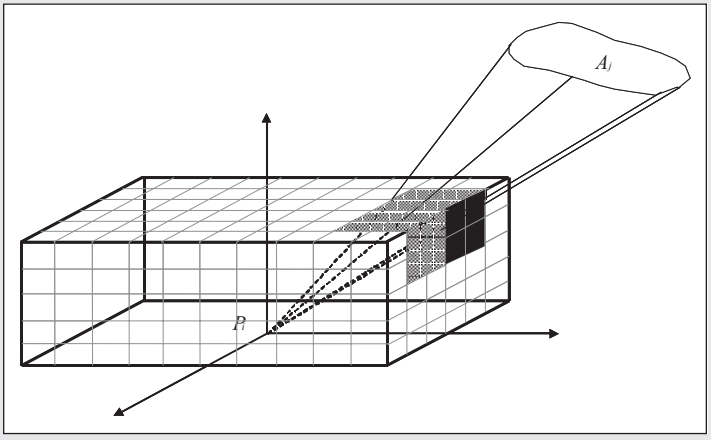


Figure 10.6. Hemicube projection method

modified version of the Z-buffer algorithm, you can calculate the form factor of a given differential area. Instead of storing the color of the closest polygon for a given pixel, you store the identifier of the corresponding closest polygon for this pixel (cell); a delta factor is calculated for each pixel (cell) of the grid of the current image (hemicube face). For the current differential area, all form factors are then calculated at once by adding the delta form factors of each polygon identified.

The advantage of the hemicube method is that it can make use of 3D hardware and modern 3D libraries. In this case, the graphics card will be used to compute form factors, and the CPU will be used to resolve the radiosity equation at each vertex. This method is very well suited to radiosity

calculations when combined with the progressive refinement method that I introduce in section 10.2.5. An important parameter of the hemicube method is the resolution of the grid of cells: The higher the resolution, the better the results, at the cost of longer computation times of form factors. For lower resolutions, aliasing artifacts can appear.

As a final note on form factor calculations, it is important to note that this is a vast field of the radiosity literature. Several variations of the methods briefly presented here have been developed throughout the years. Interested readers can look online at [4] and [5] for an almost complete listing of radiosity books and research papers.

10.2.4 The Classic Radiosity Method

As you saw earlier, the radiosity equation is formulated as follows:

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

This equation can also be expressed as a set of linear equations, with one equation per patch present in the scene, as follows:

$$\begin{cases} B_1 = E_1 + \rho_1 B_1 F_{11} + \rho_1 B_2 F_{12} + \ldots + \rho_1 B_n F_{1n} \\ B_2 = E_2 + \rho_2 B_1 F_{21} + \rho_2 B_2 F_{22} + \ldots + \rho_2 B_n F_{2n} \\ \vdots \\ B_n = E_n + \rho_n B_1 F_{n1} + \rho_n B_2 F_{n2} + \ldots + \rho_n B_n F_{nn} \end{cases}$$

You will recognize a vector-matrix product:

$$\begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix} + \begin{bmatrix} \rho_1 F_{11} & \rho_1 F_{12} & \cdots & \rho_1 F_{1n} \\ \rho_2 F_{21} & \rho_2 F_{22} & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \rho_n F_{n1} & \cdots & \cdots & \rho n F_{nn} \end{bmatrix} \times \begin{pmatrix} B_1 \\ B_2 \\ \cdots \\ B_n \end{pmatrix}$$

By rearranging the radiosity terms from both sides of the equations, you finally get the following product:

$$\begin{bmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ -\rho_n F_{n1} & \cdots & \cdots & 1-\rho n F_{nn} \end{bmatrix} \times \begin{pmatrix} B_1 \\ B_2 \\ \cdots \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{pmatrix}$$

The exitance vector **E** is known (since the lights of the scene are known); therefore to resolve the radiosity value for each patch, you only need to invert the matrix on the left side of the equation.

$$M \bullet B = E$$

Each term of this matrix can be expressed using the Kronecker symbol which, for two integers, is 1 if these two integers are equal and 0 otherwise:

$$M_{ij} = \delta_{ij} - \rho_i F_{ij}$$

Since the reflectance is known for each patch in the scene (it is a surface property), all that is needed to solve the set of linear equations is to determine the form factors and then invert the matrix.

The classic radiosity method works exactly like this:

1. Subdivide the environment (lights and geometry) into patches.
2. Compute form factors for all pairs of patches.
3. Compute the radiosity equation for all patches by resolving the set of linear equations.
4. Display the resulting scene lit with the computed radiosities. This is possible because, as I showed earlier, the radiance is proportional to radiosity: $B=\pi.L$.

In the previous section, I mentioned different methods for determining the form factors. I will describe here how the system of linear equations can be solved using numerical methods.

Two main numerical iterative methods can be used to solve the set of linear equations expressing the radiosity equation: the Jacobi method and the Gauss-Seidel method. Both methods use the main characteristic of the matrix **M** to ensure that the numerical solution will converge: the *diagonal dominance* (the absolute value of a diagonal term of **M** is strictly greater than the sum of the absolute values of the terms of the corresponding row). In other words:

$$|M_{ii}| > \sum_{\substack{j=1 \\ j\neq 1}}^{j=n} |M_{ij}|$$

The iterative Jacobi method works as follows:

1. A first value of the vector **B** is arbitrarily chosen:

$$B^{(0)} = \begin{pmatrix} B_1^{(0)} \\ B_n^{(0)} \\ \vdots \\ B_n^{(0)} \end{pmatrix}$$

2. This first value of the vector is refined over several iterations by using the following formula:

$$B_i^{(k+1)} = \frac{E_i}{M_{ii}} - \sum_{j=1}^{i-1} \frac{M_{ij}}{M_{ii}} B_j^{(k)} - \sum_{j=i+1}^{n} \frac{M_{ij}}{M_{ii}} B_j^{(k)}$$

As you can see, this method is easy to implement. As a first approximation, **B** can be initialized with the value of the exitance vector **E**. The main problem with the Jacobi method is that it converges very slowly. A faster convergence can be obtained by using the Gauss-Seidel method. In this case, the $(k+1)^{th}$ approximation of the first (i–1) components of the vector **B** are used to approximate the $(k+1)^{th}$ approximation of the i^{th} component of **B**; in mathematical terms the following formula is used (note the difference from the Jacobi iteration formula in the second term from the right side of the equation):

$$B_i^{(k+1)} = \frac{E_i}{M_{ii}} - \sum_{j=1}^{i-1} \frac{M_{ij}}{M_{ii}} B_j^{(k+1)} - \sum_{j=i+1}^{n} \frac{M_{ij}}{M_{ii}} B_j^{(k)}$$

For both methods, two criteria need to be chosen to stop the iterative process at some stage. The first criterion is simply a maximum number of iterations. The second criterion uses a parameter, often referred to as the relaxation parameter, which greatly improves the convergence of the algorithm if well chosen. It can be proved that the relaxation parameter needs to be in the range]0;2[. The following C pseudocode shows an implementation of the Gauss-Seidel relaxation (see [6] for more details):

```
double[n] GaussSeidel(int size, int maxIter, double
   matrix[n][n], double vector[n], double epsilon,
   double relaxParam)
{
   /* The resulting vector */
   double resultVector[n];

   /* Initialize the input matrix and vector*/
   for(i = 0; i < n; i++)
   {
      double dTemp = matrix[i][i];
      for(j = 0; j < n; j++) matrix[i][j] = matrix[i][j]
           / dTemp;
      vector[i] = vector[i] / dTemp;
   }

   /* Initialize the convergence criterion */
   epsilonT = 1.1 * epsilon;

   /* Main loop that calculates the resulting vector */
   while ((Iter < maxIter) && (epsilonT > epsilon))
   {
      Iter++;

      for (i = 0; i < n; i++)
      {
         old = x[i];
         sum = c[i];

         for(j=0;j<n;j++) if(i!=j) sum = sum - a[i,j]
             * x[j];

         x[i] = relaxParam * sum + (1 - relaxParam) * old;

         if(x[i]!=0.0) epsilonT=abs((x[i] - old)/x[i])
            * 100;
      }
   }

   return resultVector;
}
```

The Jacobi and Gauss-Seidel methods can be easily implemented in a radiosity processor. However, as I mentioned earlier, the classic radiosity method is not the optimal method to render images with global illumination effects. As the number of patches increase in the scene, the memory requirements to solve the linear equations can become a real problem even for modern systems equipped with a lot of RAM. The other major drawback of the classic radiosity method is that you need to compute all form factors first and then solve the set of linear equations before being able to visualize the end result. In 1988, Cohen, Wallace, Chen, and Greenberg introduced a new way to solve the radiosity equation: the progressive refinement method. This method makes it possible to iteratively solve the radiosity equation while displaying intermediate results (or approximations rather). It is presented in the next section.

10.2.5 The Progressive Refinement Method

The progressive refinement method was a major advance in radiosity. In a nutshell, and as already hinted at previously, it allows you to preview the radiosity solution of a scene progressively. A direct advantage of this technique is, for example, to quickly see if the light settings of a scene are well chosen or if the lights and the geometry of the scene are well positioned in the virtual environment. The main advantage of this method is also that it converges rapidly. The progressive refinement method works like this:

1. Subdivide the environment (lights and geometry) into patches. This step is the same as with the classic radiosity method. The radiosity and delta radiosity (energy to be distributed to the other patches) are both initialized to the exitance of the current patch.
2. Calculate the total energy to be distributed in the scene.
3. Select the current brightest patch in the scene, i.e., the patch with the highest energy (maximum value of the area multiplied by the current delta radiosity estimate for this patch). For the first iterations this will be the patches of the lights of the scene, and the patches close to the lights.
4. Distribute the energy (delta radiosity) of this patch to the other patches of the scene. The radiosity and delta radiosity received by each patch in the scene are updated. Form factors of the current patch and the other patches in the scene are computed and used during this step. The hemicube method is very well suited here.
5. Set the energy to distribute (delta radiosity) for this patch to 0. Reduce the total energy to be distributed in the scene by the current energy radiated.
6. Display the resulting scene lit with the computed (and updated) radiosities. This is possible because, as I showed earlier, the radiance is proportional to radiosity: $B=\pi.L$.
7. Return to step 3. This iterative process can last until a given percentage of the total energy to distribute in the scene has been reached.

You may wonder how this method can be mathematically correct. Indeed, it may be seen as a natural way of simulating the radiation of energy in a scene, but nothing proves that it is physically and mathematically correct. Actually, the progressive refinement method is a relaxation method, just like the Jacobi and Gauss-Seidel methods you saw previously in the classic radiosity method section. It is mathematically known as the Southwell relaxation technique and is thoroughly described in [1] and [2]. The two important formulae to calculate the radiosity and delta radiosity for a given patch are:

$$B_j = B_j + \rho_j F_{ij} \frac{A_i}{A_j} \Delta B_i$$

and:

$$\Delta B_j = \Delta B_j + \rho_j F_{ij} \frac{A_i}{A_j} \Delta B_i$$

Both formulae can be simplified using the reciprocity property of form factors that you saw previously.

$$A_i F_{ij} = A_j F_{ji}$$

This gives:

$$B_j = B_j + \rho_j F_{ji} \Delta B_i$$

and:

$$\Delta B_j = \Delta B_j + \rho_j F_{ji} \Delta B_i$$

The use of either version of these formulae depends on the implementation and method chosen to calculate the form factors.

The following C pseudocode summarizes the progressive refinement method.

```
void progressiveRefinement(double someFractionTotalEnergy)
{
   /* Initialization */
   for (j = 0; j < numPatches; j++)
   {
      remainingEnergy += patches[j].exitance;
      patches[j].Rad = patches[j].exitance;
      patches[j].deltaRad = patches[j].exitance;
   }

   /* Main loop that calculates the resulting vector */
   while (remainingEnergy > someFractionTotalEnergy)
   {
      /* Select a patch if the most energy to radiate */
      /* i.e., maximum(deltaRad * area) */
      i = indexOfBrightestPatch();

      /* Loop through all patches in the scene */
      for (j = 0; j < numPatches; j++)
      {
         /* Calculate the form factor for patch i and j */
         ff = calculateFormFactor(i, j);

         /* Calculate the delta radiosity */
         deltaRad = patches[j].reflectance *
               patches[i].deltaRad * ff *
               (patches[i].area / patches[j].area);

         /* Update the delta radiosity and radiosity */
         patches[j].deltaRad += deltaRad;
         patches[j].Rad += deltaRad;
      }

      /* Decrease the total amount of enery to radiate */
      remainingEnergy -= patches[i].deltaRad;

      /* The energy of this patch has been radiated */
      patches[i].deltaRad = 0.0;

      /* Display the intermediate results */
      displayScene();
   }
}
```

10.2.6 Radiosity and Subdivision

All the methods you have seen so far have been described in the context of an environment uniformly subdivided into patches. As with many other techniques of computer graphics, uniform subdivision yields to the undesired effect of aliasing. In radiosity, the first assumption is to consider that each patch has the same average radiosity value across its surface. When two neighboring patches present a large difference in terms of average radiosity values, visual discontinuities appear. In this case, several approaches can be considered to further subdivide the patches and refine their radiosity values until the differences become visually negligible.

The first approach is simply to increase the number of patches of the surfaces in the scene. This is easy to implement but has the major drawback of greatly increasing both the computation times and the memory storage needs. The other drawback of this method is that it is far from being efficient. In most cases, some areas of the scene will still present visual discontinuities and the subdivision process will have to be applied again, while other areas of the scene will not benefit from the increased number of patches and will nevertheless cost valuable computation time.

Fortunately more clever methods exist! The main method is called adaptive subdivision. The idea is to start with a uniform subdivision of the surfaces of the environment. The radiosity estimates are calculated, for instance, by using the progressive refinement method. However, after each iteration (that is, after the brightest patch has emitted its delta radiosity), the patches of the scene that have received energy from the emitting patch are analyzed. When a group of neighbor patches are found to present a high variation of radiosity values, they are further subdivided, and their respective radiosity contribution received from the emitting patch is cancelled. This group is further subdivided and the energy from the brightest patch selected is redistributed. The subdivision process can be repeated until the neighbor patches present a small difference of radiosity values (a maximum number of subdivisions can be set to avoid too many iterations).

Figure 10.7 shows a scene using radiosity and adaptive subdivision. This scene was rendered using the Luminance Radiosity Studio program by Stéphane Marty that can be found on the companion CD. Note the difference in the subdivision of the different parts of the scene, such as the back wall where the light sources project an important amount of energy.

An important concept for most subdivision techniques in radiosity is that of patches and elements. The patches represent the first level of subdivision and are used as emitters of energy. Elements, on the other hand, represent the subdivision of patches when receiving energy. Figure 10.8 shows the concept of patches and elements.

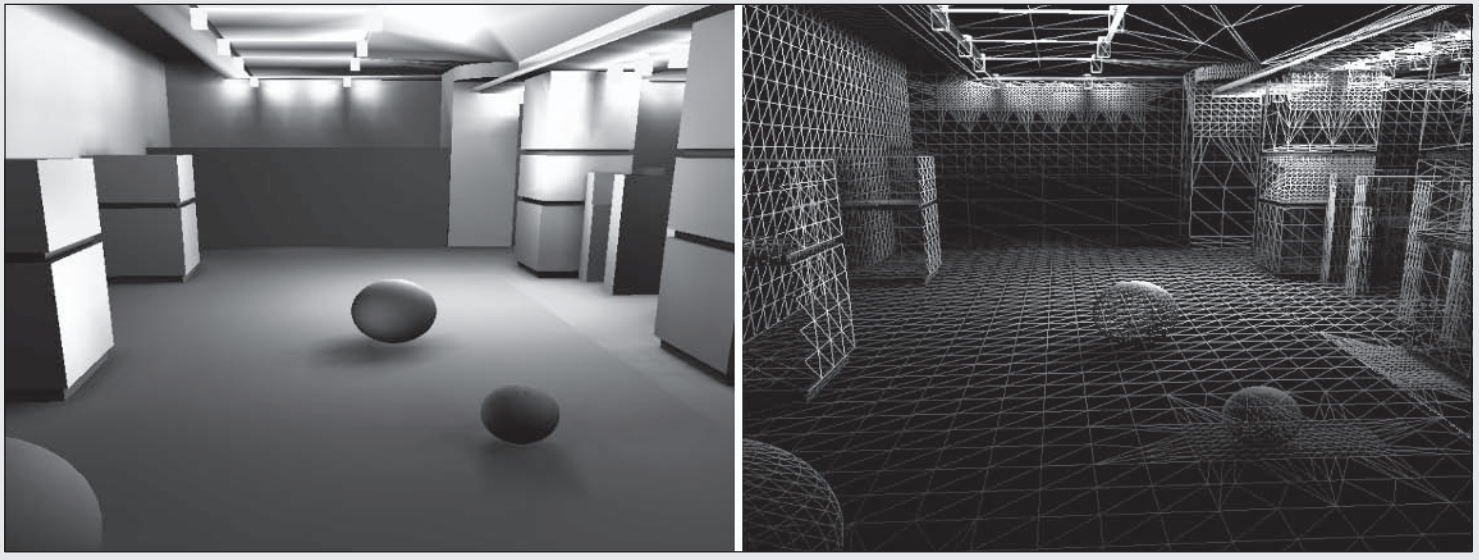


Figure 10.7. A scene rendered using radiosity and adaptive subdivision

As with form factor computation, subdivision techniques are an important and vast field of the radiosity literature. Several adaptive subdivision methods, hierarchical methods, etc., have been developed over the years. More information on this topic can be found in [1] and [2], as well as online at [4] and [5].

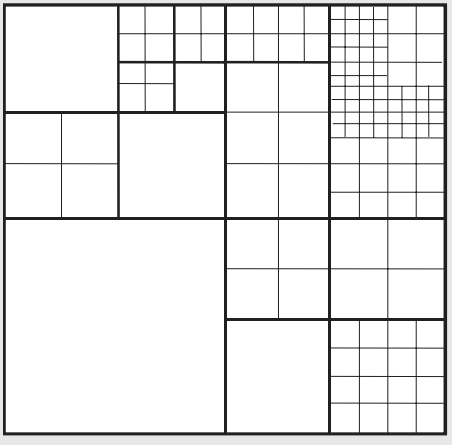


Figure 10.8. Patches, elements, and subdivision

10.3 Sample Implementation in OpenGL

As already stated, radiosity has been around for quite some time. There are many programs that make use of 3D hardware to accelerate the computation of the radiosity method.

Instead of reinventing the wheel, I have listed here a few examples of programs using OpenGL to calculate and render scenes with the radiosity method:

- Radiator available at [7]
- RadiosGL available at [8]
- GLUTRAD. At the time of this writing this project has been taken offline. It has, however, been provided on the companion CD. This program illustrates very well how the OpenGL 3D APIs can be used to calculate the delta form factors along with the progressive refinement method.
- Parallelized GLUTRAD, available at [9], goes further by adding parallelization to the GLUTRAD program. Systems with more than one processor can benefit from this technique, even though the computation of delta form factors is handled by the OpenGL-accelerated 3D hardware.
- Finally, RADical, another radiosity processor and renderer shows similar techniques. It is available at [10].

These are only a few of the resources that can be found on the Internet. There exist many such programs that show how modern and fast 3D-accelerated hardware can be used to render radiosity.

Recent advances in 3D hardware programmability have made it possible to write shaders to compute the full radiosity of a scene. The following list gives references to recent advances in this domain:

- [11] and [12] show how the Cg shading language can be used in the progressive refinement method. For small scenes, the computation time can be less than a second.
- [13] shows how 3D hardware acceleration can be used with large scenes. A new technique (the global cube algorithm) is introduced to achieve this goal.
- [14] gives different implementations and techniques using hardware acceleration.

Conclusion

In this chapter, I presented the radiosity method, or rather a set of radiosity methods that have been developed over the past 20 years. The main purpose of radiosity is to take into account the global lighting of a scene. You saw not only the background of the methods, but also the theory behind them and a brief overview of their history and evolution. I showed how the first method (the matrix solution) can be easily implemented by solving a set of linear equations. I then covered the progressive refinement methods that allow calculating the radiosity solution of a scene over multiple iterations and offer the ability to visually check the result on the screen after each iteration. Progressive refinement gives results iteratively and can be implemented to make use of 3D hardware. Radiosity solutions of simple scenes can be calculated in a very short time, thus giving the impression of real-time processing. Recent advances in hardware-accelerated techniques allow us not only to compute radiosity solutions very quickly (for instance by using the programmability of modern 3D hardware), but also to handle larger scenes.

While the radiosity method presents some advantages, it also has some drawbacks (compared to spherical harmonic lighting, for example):

- The radiosity solution needs to be recomputed if the light sources change.
- It only takes into account the global illumination of a scene. This is, of course, very important in terms of realism; however, in some cases, more simplified effects might be needed. For example, shadows might be needed without global illumination or one might just need diffuse lighting without shadows.
- Radiosity methods are sometimes complex to implement. There exist different techniques to calculate form factors but also to solve the radiosity equation.

References

[1] Cohen, M. F. and J. R. Wallace, *Radiosity and Realistic Image Synthesis*, Boston: Academic Press Professional, 1993.

[2] Sillion, F. X. and C. Puech, *Radiosity and Global Illumination*, Morgan Kaufmann Publishers, Inc., 1994.

[3] Ashdown, I., *Radiosity: A Programmer's Perspective*, Wiley, 1994.

[4] Radiosity bibliography and research papers, http://www.helios32.com/resources.htm

[5] Radiosity bibliography, http://liinwww.ira.uka.de/bibliography/Graphics/rad.html

[6] Stöcker, H., *Taschenbuch mathematischer – Formeln und moderner Verfahren*, Verlag Harri, 2004.

[7] Radiator, http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/ajw/www/software/

[8] RadiosGL, http://hcsoftware.sourceforge.net/RadiosGL/RadiosGL.html

[9] GLUTRAD (parallelization project), http://www.vrac.iastate.edu/~bmila/radiosity/finalpaper/finalpaper.html

[10] RADical, http://www.cse.iitd.ernet.in/~parag/projects/CG2/asign2/report/RADical.shtml

[11] Coombe, G., M. Harris, and A. Lastra, "Radiosity on Graphics Hardware," ACM International Conference Proceedings, 2004.

[12] Radiosity on graphics hardware, http://www.cs.unc.edu/~coombe/research/radiosity/

[13] Hubeli, A., L. Lippert, and M. Gross, "The Global Cube: A Hardware-accelerated Hierarchical Volume Radiosity Technique," Swiss Federal Institute of Technology, 1999.

[14] Radiosity implementations using hardware, http://www.efd.lth.se/~e98mp/thesispage/thesispage.html

Other Resources

Dutré, P., P. Bekaert, and K. Bala, *Advanced Global Illumination*, A.K. Peters Ltd., 2003.

Jensen, H. W., *Realistic Image Synthesis Using Photon Mapping*, A.K. Peters Ltd., 2001.

This page intentionally left blank.

Appendix A

Building the Source Code

Introduction

In this appendix, I explain how to build the different programs developed for this book. The programs have been developed on and for different platforms, using the Direct3D and/or OpenGL libraries. All tools and libraries needed are listed, and I also explain how to build and/or install them where required.

The primary programming languages chosen are C and C++. Both are supported on many platforms (if not all!), with free compilers (like GCC) available as well. All the libraries that we have used throughout this book have API calls in C or C++.

A.1 DirectX/HLSL Programs

In some of the chapters of this book, the programs were developed with the DirectX SDK, in C++, and with the DirectX High Level Shading Language (HLSL) for the accompanying shaders. I explain here how to build these programs and test them with shaders and related data.

A.1.1 Requirements

The following tools are required to build the DirectX/HLSL programs:

- MS DirectX 9: DirectX is used as the primary 3D API for rendering. At the time of publication, the latest version of

Microsoft DirectX is Microsoft DirectX 9.0 SDK Update (Summer 2003), which can be found at [1]. Any version that is more recent should also work, as DirectX is usually backward compatible.

- Visual Studio .NET 2003: This IDE is used to compile the C++ programs. Any other version should work provided that it is compatible with the version of DirectX used. The project files provided on the companion CD will work with Visual Studio .NET only. It is, however, easy to create project files for older versions of Visual C++. It is also possible to use the Microsoft Visual C++ Toolkit 2003 to build the examples of the book, available from [2].

The programs were developed and tested on the following Windows platforms:

- Windows 2000 (Intel) with Visual Studio .NET 2003 and DirectX 9.0 SDK Update (Summer 2003)
- Windows XP (Intel) with Visual Studio .NET 2003 and DirectX 9.0 SDK Update (Summer 2003)

Other Windows versions supporting the IDE and this version of DirectX should work as well.

A.1.2 Building the Programs

The programs provided on the CD have been configured to be compiled with the default installation of DirectX (C:\DX90SDK). However, to change the settings do the following:

1. Under Visual Studio .NET 2003, open a solution file (file with the .sln extension), such as the Sample.sln file in the HLSL Basics directory.
2. Right-click on the Sample project, and select **Properties**. The following window is displayed:

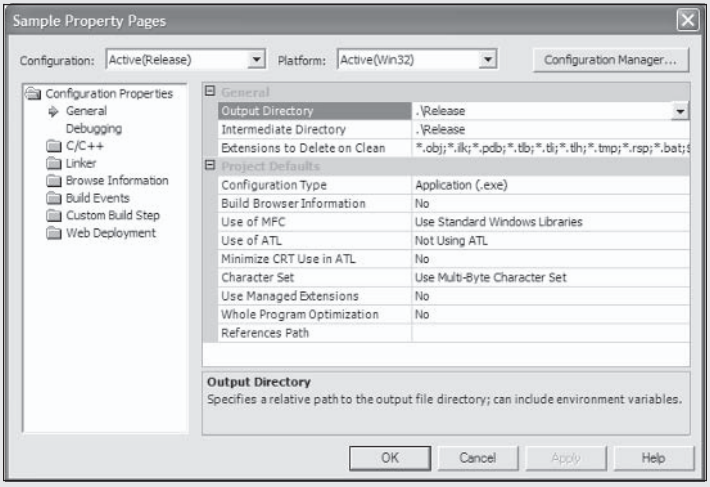


Figure A.1. HLSL Basics main property page

3. Select **C/C++** and **General**.
4. In the Additional Include Directories field, select the directory where the DirectX SDK include directory has been installed (the default location is C:\DX90SDK/Include), as shown in Figure A.2.

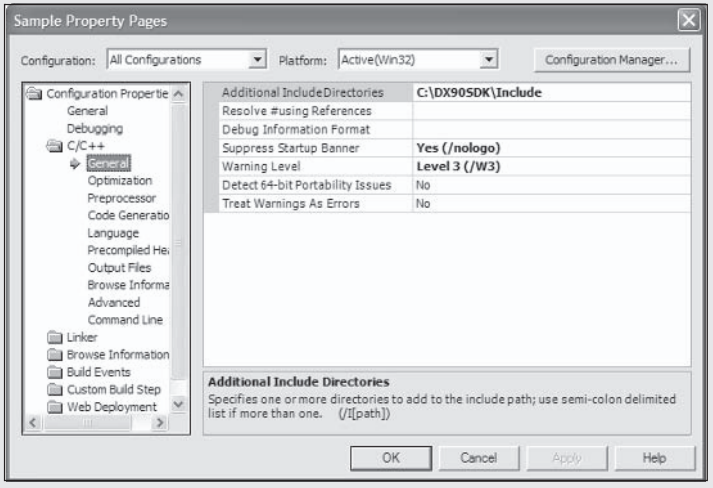


Figure A.2 Project C/C++ General properties page

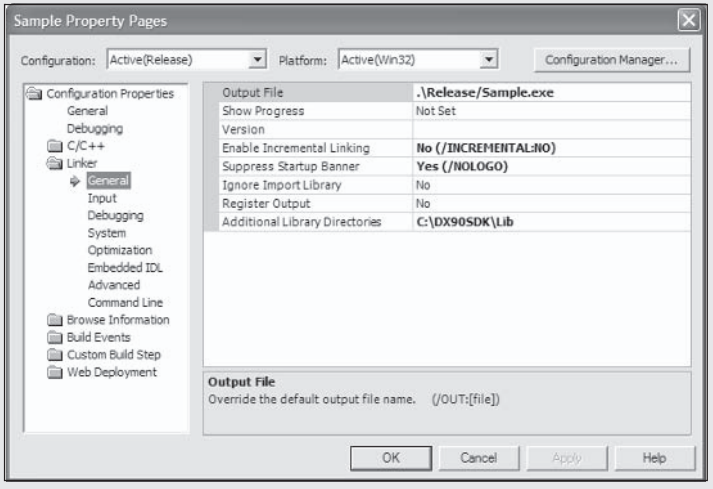


Figure A.3.

5. In the Linker menu, under the General submenu, add the path to the DirectX libraries (the default location is C:\DX90SDK\Lib). See Figure A.3.
6. Make sure the required DirectX libraries are specified for the linker, as shown in Figure A.4.

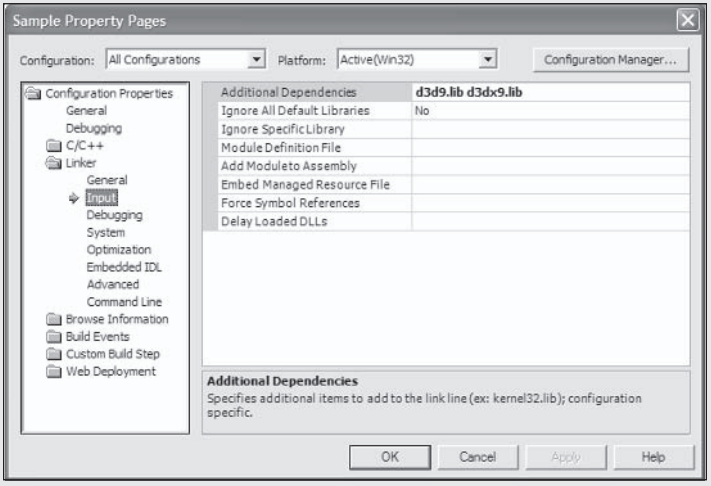


Figure A.4. Project linker properties page

A.1.3 Running and Testing the Programs and Shaders

The DirectX programs and accompanying shaders have been placed in their own directories. For each program, you will find a similar directory structure as explained below.

- Source files: All source files (i.e., .cpp and corresponding .h files) are present at the root of each project directory.
- Visual Studio .NET 2003 solution files: The corresponding solution files (i.e., .sln and .vcproj files) are also located at the root of each project directory.
- Models: All 3D models for a given project are located in the Models directory. The files are in the DirectX 3D file format (with an .x extension).
- Shaders: All shaders for a given project are located in the Shaders directory. The files are in the DirectX HLSL file format (with an .hsl extension).
- The executables for a given project are located in the default Visual Studio build directories: Release for an optimized version of the executable, and Debug for a non-optimized debug version. Both executables will use the same directories for models and shaders.
- Other data files will be located in specific subdirectories. For instance, for spherical harmonic lighting DirectX programs, the SH coefficients will be placed in the SHData directory.

The following figure shows an example of such a directory structure:

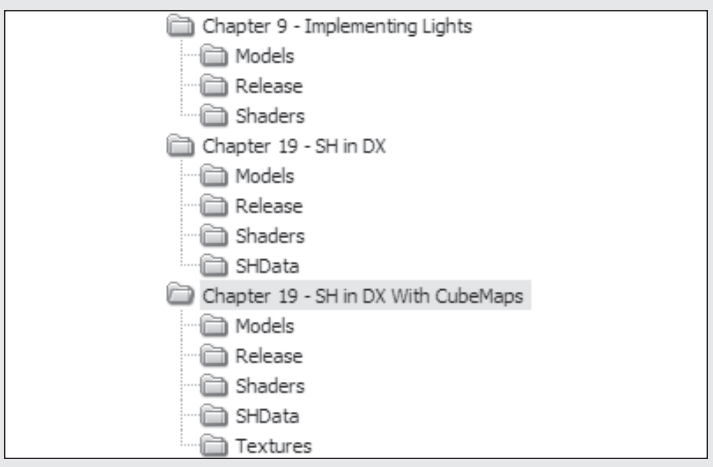


Figure A.5. DirectX programs directory structure

To use these programs, simply run the executable of your choice. The HLSL shaders can be altered using any text editor (e.g., Notepad).

A.2 OpenGL/CG Programs

In this section I explain how to build the OpenGL/CG programs. These programs were developed to allow Linux users to be able to run the different shaders developed throughout the book. They were also developed for OpenGL users to be able to run the different shaders with their favorite 3D API.

A.2.1 Requirements

The following tools are required to build the OpenGL/CG programs:

- OpenGL: OpenGL is used here as the primary 3D API for rendering. Version 1.1 is sufficient; it can be found at [3]. The program will compile and run with any version that is more recent. Mesa is an alternative open-source library implementing the OpenGL APIs.
- Visual Studio .NET 2003: This IDE is used to compile the C++ programs on Windows platforms. Any other version should work. The project files provided on the CD will work with Visual Studio .NET only. It is, however, easy to create project files for older versions of Visual C++.
- GCC: For Linux platforms, GCC is used to compile C++ programs.
- GLUT: In order to develop cross-platform programs with OpenGL, GLUT 3.6 or above is needed. It can be found at [4] and [5].
- Cg: NVIDIA's Cg Toolkit is necessary for the development of vertex and fragment shaders. It can be found for both Linux and Windows at [6].

The programs were developed and tested on the following platforms:

- Windows 2000 (Intel) with Visual Studio .NET 2003 and the Cg Toolkit 1.1
- Windows XP (Intel) with Visual Studio .NET 2003 and the Cg Toolkit 1.1
- Mandrake 9.1 Linux (Intel) with GCC 3.x and the Cg Toolkit 1.1

A.2.2 Building the Programs

Now we'll build the programs using Windows and Linux.

A.2.2.1 Windows Platforms

The programs provided on the CD have been configured to be compiled with the default installation of the Cg Toolkit (C:\Program Files\NVIDIA Corporation\Cg), GLUT, and OpenGL. The OpenGL include files are installed by default under C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\PlatformSDK\Include\gl. It is good practice to place GLUT's main header file, glut.h, in the same directory. Alternatively, place this header file in your OpenGL SDK's include directory. The library file, glut32.lib, can be placed in the

C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\PlatformSDK\Lib directory or in your OpenGL SDK's library directory. To change the settings do the following:

1. Under Visual Studio .NET 2003, open a project file (a file with the .sln extension), such as the OGLCgFramework.sln file in the OGLCgFramework directory.
2. Right-click on the OGLCgFramework project, and select **Properties**. The Property Pages window is displayed, as shown in Figure A.6.
3. Select **C/C++** and **General**.
4. In the Additional Include Directories field, select the directory where the Cg Toolkit has been installed (the default location is C:\Program Files\NVIDIA Corporation\Cg\include), as shown in Figure A.7.
5. In the Linker menu, under the General submenu, add the path to the Cg libraries (the default location is C:\Program Files\NVIDIA Corporation\Cg\lib). See Figure A.8.
6. Make sure the required Cg libraries are specified for the linker, along with GLUT, as shown in Figure A.9.

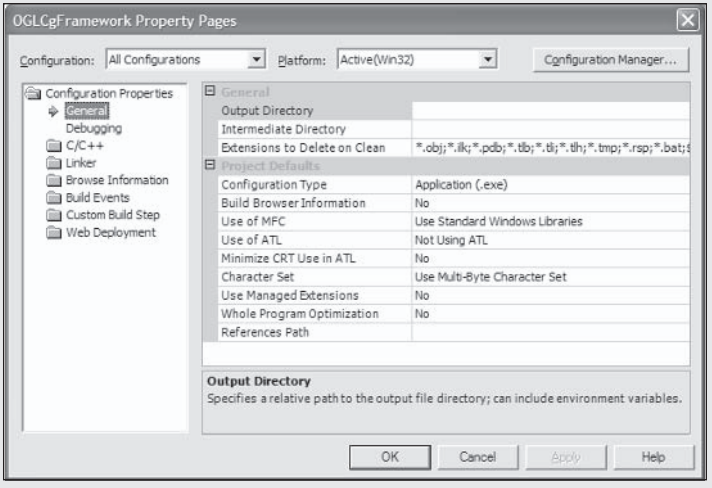


Figure A.6. OGLCgFramework main properties page

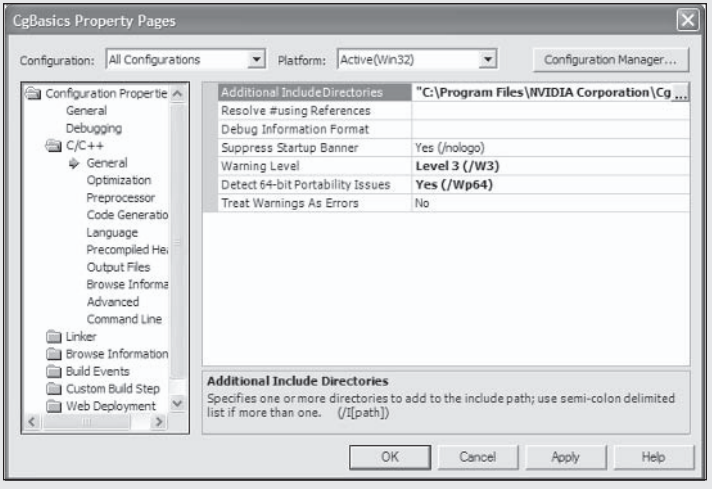


Figure A.7. Project C/C++ General properties page

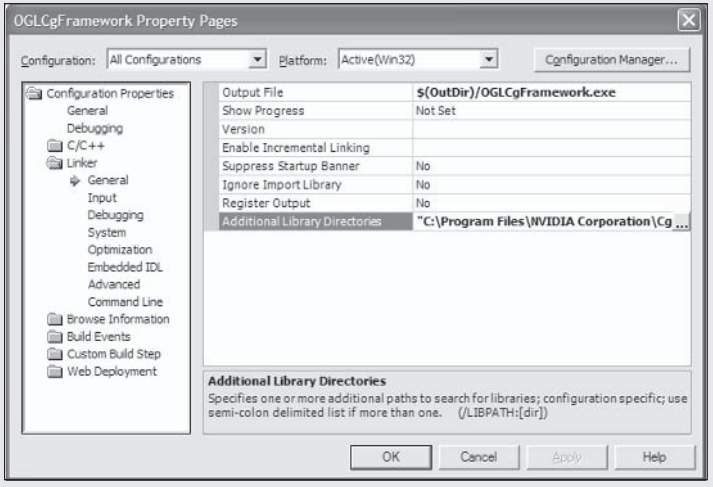


Figure A.8. Linker path properties

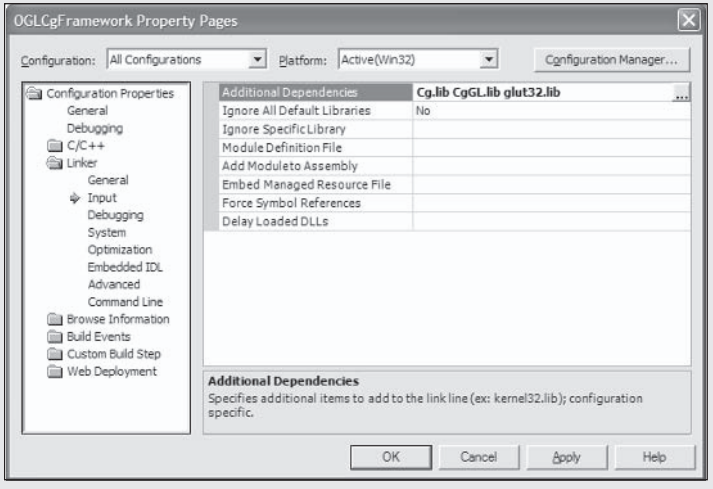


Figure A.9. Project linker properties page

A.2.2.2 Linux Platforms

On Linux Mandrake 9.1, Cg Toolkit files are installed by default in /usr/X11R6/include for the include files and /usr/X11R6/lib for the libraries. OpenGL and GLUT files are installed by default in /usr/X11R6/include for the include files and /usr/X11R6/lib for the libraries. Note that on Linux Mandrake 9.1, you will need to select the libMesaglut3-5.0-3mdk and libMesaglut3-devel-5.0-3mdk packages.

All programs come with source code and make files configured with these directories. If your Linux system comes with different paths for these libraries, or if you install the Cg Toolkit in another directory, you will have to change the make files accordingly.

To build the programs, simply do the following:

1. Uncompress the archive found on the companion CD using gunzip by typing **gunzip -d OGLCgFramework.tar.gz**.
2. Extract the archive using tar by typing **tar -xvf OGLCgFramework.tar**.
3. Go to the directory created: **cd OGLCgFramework**.
4. Type **make** to build the program.
5. If errors occur during the compilation, check that the libraries are all installed correctly. Alternatively, change the provided make file to match your system paths (includes and libraries).
6. Type **make clean** to remove any temporary files.

A.2.3 Running and Testing the Programs and Shaders

The OpenGL/Cg programs and accompanying Cg shaders have been placed in their own directories. In each directory you will find similar structures as explained below.

- Source files: All source files (i.e., .cpp and corresponding .h files) are present at the root of each project directory.
- Project files: On Windows, the Visual Studio .NET 2003 project files (i.e., .sln and .vcproj files) are also located at the root of each project directory. On Linux, the make file is located at the root of each project directory.
- Shaders: All shaders for a given project are located in the Shaders directory. The files are in the Cg file format (with a .cg extension).
- Executables: On Windows, the executables for a given project are located in the default Visual Studio build directories: Release for an optimized version of the executable, and Debug for a non-optimized debug version. Both executables will use the same directories for models and shaders. On Linux, the executable is generated in the bin subdirectory.
- Other data files will be located in specific subdirectories.

To use these programs, simply run the executable of your choice. The Cg shaders can be altered using any text editor (e.g., Notepad on Windows, vi on Linux).

A.3 OpenGL/GLSL Programs

In this section I explain how to build the OpenGL/GLSL programs. These programs were developed to allow users to be able to adapt the different shaders developed throughout the book to their favorite shading language or the emerging GLSL shading language. At the time of publication, only a couple of GLSL SDKs exist, and for the Windows platform only; namely the GLSL SDK from 3Dlabs (available at [7]), and the one from ATI (available at [8]).

A.3.1 Requirements

The following tools are required to build the OpenGL/GLSL programs:

- OpenGL: OpenGL is used here as the core 3D API for rendering. Version 1.1 is sufficient. It can be found at [3]. The program will compile and run with any version that is more recent. More precisely, and to be thorough, GLSL is part of the OpenGL 1.5 standard. However, at the time of publication, this version of OpenGL is not yet available,

and GLSL is only available through two SDKs as mentioned above.

- Visual Studio .NET 2003: This IDE is used to compile the C++ programs on Windows platforms. Any other version should work. The project files provided on the CD will work with Visual Studio .NET only. It is, however, easy to create project files for older versions of Visual C++.
- GLUT: In order to develop cross-platform programs with OpenGL, GLUT 3.6 or above is needed. It can be found at [4] and [5].
- ATI GLSL SDK: This SDK is necessary for the development of GLSL vertex and fragment shaders on ATI hardware. It can be found for Windows at [8].
- 3Dlabs GLSL SDK (also known as the OpenGL 2.0 SDK): This SDK is necessary for the development of GLSL vertex and fragment shaders on 3Dlabs hardware. It can be found for Windows at [7].

The programs were developed and tested on the following platforms:

- Windows XP (Intel) with Visual Studio .NET 2003 and the ATI GLSL SDK beta 1
- Windows XP (Intel) with Visual Studio .NET 2003 and the 3Dlabs GLSL SDK version 6

A.3.2 Building the Programs

The programs provided on the CD have been configured to be compiled with GLUT, OpenGL, and the different GLSL SDKs mentioned above. Theses SDKs can be downloaded from [7] and [8] as ZIP archives, and they can be installed anywhere on the file system. I have chosen the following locations for these SDKs (the program's project files using these locations):

- The ATI GLSL SDK installed in C:\Program Files\ATI Technologies\GLSL-Beta1-sdk
- The 3Dlabs GLSL SDK installed in C:\Program Files\3Dlabs\ogl2sdk6

As was mentioned in the previous section of this appendix, the OpenGL include files are installed by default in C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\PlatformSDK\Include\gl. It is a good practice to place GLUT's main header file, glut.h, in the same directory. Alternatively, place this header file in your OpenGL SDK's include directory. The library file, glut32.lib, can be placed in the C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\PlatformSDK\Lib directory or in your OpenGL SDK's library directory. To change the settings do the following:

1. Under Visual Studio .NET 2003, open a project file (a file with the .sln extension), for instance the OGLGLSL-Framework.sln file in the OGLGLSLFramework directory.
2. Right-click on the **OGLGLSLFramework** project, and select **Properties**. The Property Pages window is displayed, as shown in Figure A.10.
3. Select the configuration to modify (ATI, ATI Debug, 3Dlabs, or 3Dlabs Debug).
4. Select **C/C++** and **General**.
5. In the Additional IncludeDirectories field, select the directory where the GLSL has been installed, as shown in Figure A.11.
6. In the Linker menu, under the General submenu, add the path to the GLSL libraries, as shown in Figure A.12.
7. Make sure the required Cg libraries are specified for the linker, along with GLUT, as shown in Figure A.13.

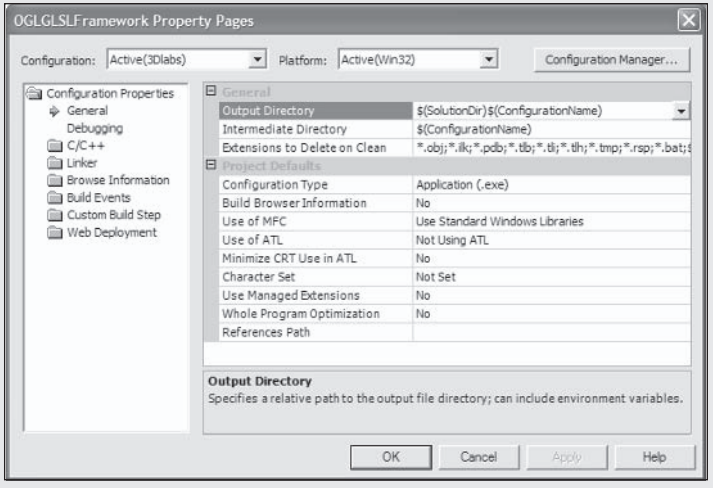


Figure A.10. OGLGLSLFramework main property page

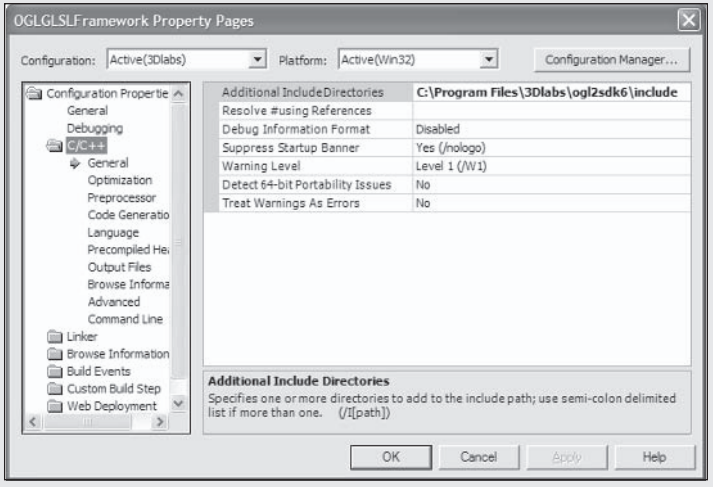


Figure A.11. Project C/C++ General properties page

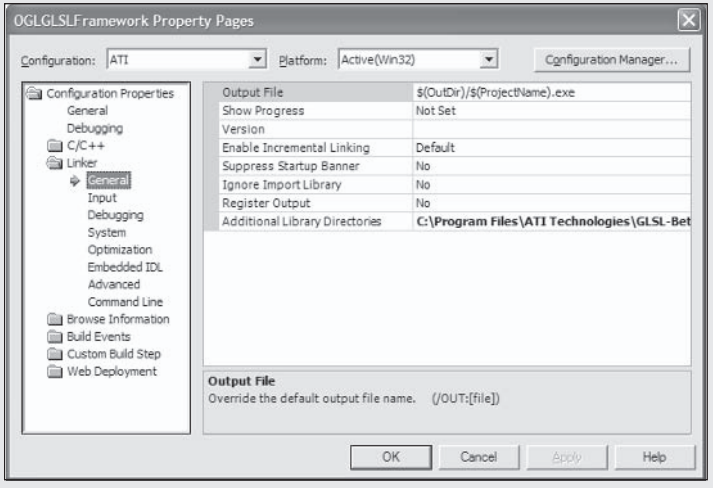


Figure A.12. Linker path properties

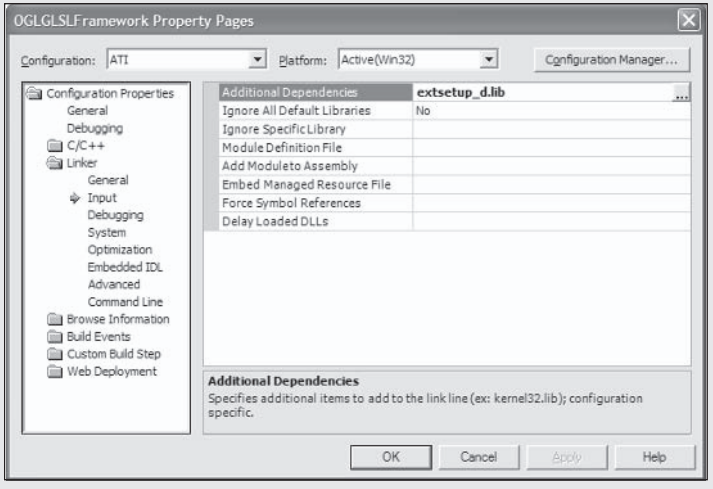


Figure A.13. Project linker properties page

A.3.3 Running and Testing the Programs and Shaders

The OpenGL/GLSL programs and accompanying GLSL shaders have been placed in their own directories. In each directory you will find similar structures as explained below:

- Source files: All source files (i.e., .cpp and corresponding .h files) are present at the root of each project directory.
- Project files: On Windows, the Visual Studio .NET 2003 project files (i.e., .sln and .vcproj files) are also located at the root of each project directory.
- Shaders: All shaders for a given project are located in the Shaders directory. The files are in the GLSL file format (with a .vert extension for vertex shaders and .frag extension for fragment shaders).
- Executables: The executables are located in the following directories:
  - *ATI* for an optimized version of the executable compiled with the ATI GLSL SDK.
  - *ATI Debug* for a non-optimized debug version compiled with the ATI GLSL SDK.
  - *3Dlabs* for an optimized version of the executable compiled with the 3Dlabs GLSL SDK.
  - *3Dlabs Debug* for a non-optimized debug version compiled with the 3Dlabs GLSL SDK.

To use these programs, simply run the executable of your choice. The GLSL shaders can be altered using any text editor (e.g., Notepad).

A.4 OpenGL Programs

In some of the chapters of this book, some programs were developed with the OpenGL library. The sample SH Lighting implementations are examples of such programs. For portability, I chose GLUT and the lib3ds libraries to be able to handle the creation of the UI and to easily load 3D files.

I first list the platforms on which the programs were compiled and tested, as well as the different libraries and respective versions. I then explain how to build the different programs.

A.4.1 Platforms and Tools

The programs were developed and tested on the following platforms:

- Windows 2000 (Intel) with Visual Studio .NET 2003
- Windows XP (Intel) with Visual Studio .NET 2003
- Linux Mandrake 9.1 (Intel) with GCC 3.0
- Irix 6.5 (MIPS) with SGI C Compiler

The following libraries are needed to compile and run the programs:

- OpenGL 1.1 and above (Mesa is an alternative open-source library implementing the OpenGL APIs)
- GLUT 3.6 and above, which can be found at [4] and at [5]
- Lib3ds 1.2 and above, which can be found at [9]

A.4.2 Installing the Libraries

Now we'll discuss how to install the various libraries.

A.4.2.1 GLUT

On Linux Mandrake 9.1, OpenGL and GLUT files are installed by default in /usr/X11R6/include for the include files and /usr/X11R6/lib for the libraries.

On Windows, the OpenGL include files are installed by default in C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\PlatformSDK\Include\gl. It is a good practice to place GLUT's main header file, glut.h, in the same directory. Alternatively, place this header file in your OpenGL SDK's include directory. The library file, glut32.lib, can be placed in the C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\PlatformSDK\Lib directory or in your OpenGL SDK's library directory.

A.4.2.2 Lib3ds

On Linux, lib3ds is not installed by default. It is therefore necessary to build it and install it manually, as follows:

1. Log on as the **root** user.
2. Open a shell terminal.
3. Uncompress the archive found on the companion CD using gunzip: **gunzip -d lib3ds-1.2.0.tar.gz**.
4. Extract the archive using tar: **tar -xvf lib3ds-1.2.0.tar**.

5. Go to the directory created: **cd lib3ds-1.2.0**.
6. Type **./configure** to configure the build process.
7. Type **make** to build the library.
8. Type **make install** to copy the header and library files into the correct directories.
9. Type **make clean** to remove any temporary files.
10. The files will be copied to /usr/local/include and /usr/local/lib.

On Windows, it's up to you to decide where you want to put the different lib3ds files needed. I have decided to place them in the Visual Studio .NET 2003 main directories, as with GLUT files:

1. Extract the lib3s-1.2.0.zip file in a temporary directory, for instance C:\Temp.
2. Create a directory called lib3ds under C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\PlatformSDK\Include\.
3. Copy the header files from C:\Temp\lib3ds-1.2.0\lib3ds to C:\Program Files\Microsoft Visual Studio .NET 2003\Vc7\PlatformSDK\Include\lib3ds.

The programs provided on the CD have been designed to be compiled with this configuration. However, to change the settings do the following:

1. Under Visual Studio .NET 2003, open a project file (a file with the .sln extension), such as the HDRISHLighting.sln file.
2. Right-click on the **HDRISHLighting** project and select **Properties**. The following window is displayed:

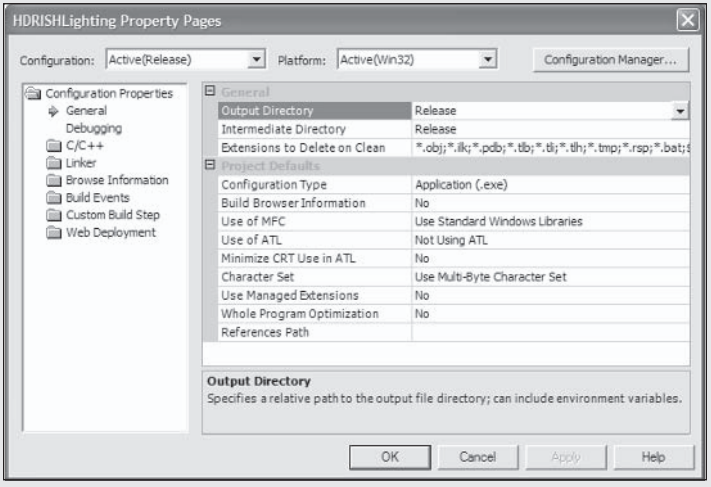


Figure A.14. HDRISHLighting project main property page

3. Select **C/C++** and **General**.
4. In the Additional IncludeDirectories field, select the directory where the lib3ds files have been extracted, as shown in Figure A.15.
5. Make sure the lib3ds library is specified for the linker, as shown in Figure A.16.

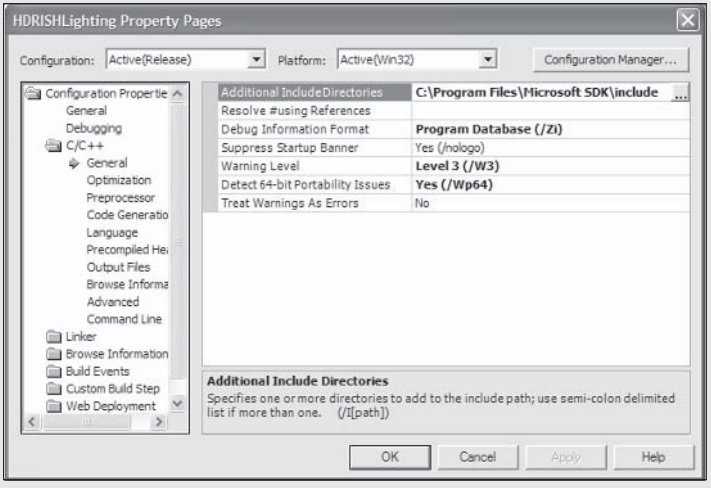


Figure A.15. HDRISHLighting project C/C++ General properties page

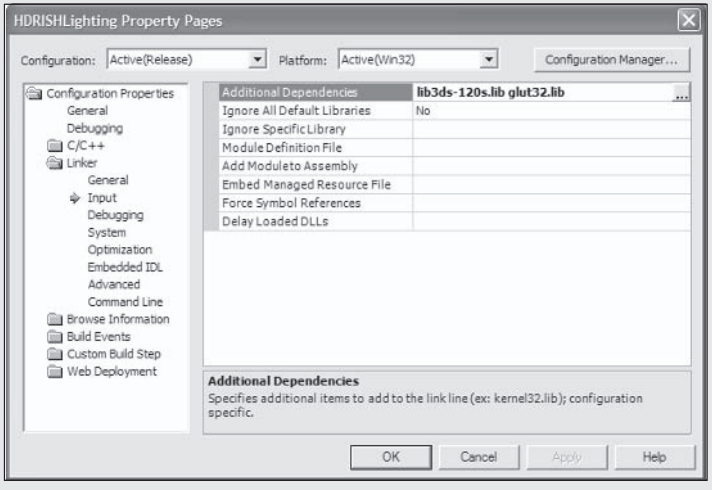


Figure A.16. lib3ds as input for the linker

A.4.3 Building the Programs

Following are instructions for building the programs using Unix and Windows.

A.4.3.1 Unix Platforms

All programs come with full source code and a make file. To build the programs, simply do the following:

1. Uncompress the archive found on the companion CD using gunzip: **gunzip -d LegendrePolynomials.tar.gz**.
2. Extract the archive using tar: **tar -xvf LegendrePolynomials.tar**.
3. Go to the directory created: **cd LegendrePolynomials**.
4. Type **make** to build the program.
5. If errors occur during compilation, check that the libraries are all installed correctly. Alternatively, change the provided make file to match your system paths (includes and libraries).
6. Type **make clean** to remove any temporary files.

A.4.3.2 Windows Platforms

For Windows platforms, the programs come with project files for MS Visual Studio .NET 2003. Simply select your desired target configuration (*Release* for fast production code, *Debug* for debugging) and compile the program. Please refer to the previous section if you need to modify your directory settings. GLUT is needed for all programs, but only some programs need lib3ds.

References

[1] Main DirectX web site, http://www.microsoft.com/directx

[2] Microsoft Visual C++ Toolkit 2003, http://msdn.microsoft.com/visualc/vctoolkit2003/

[3] OpenGL community site, http://www.opengl.org

[4] GLUT main site, http://www.sgi.com/software/opengl/glut.html

[5] GLUT Win32 main site, http://www.pobox.com/~nate/glut.html

[6] Cg main site, http://developer.nvidia.com/view.asp?IO=cg_toolkit

[7] 3Dlabs OpenGL SL site, http://www.3dlabs.com/support/developer/ogl2/index.htm

[8] ATI GLSL SDK, http://www.ati.com/developer/sdk/RadeonSDK/Html/Samples/OpenGL/GLSLSimpleShader.html

[9] Lib3ds main site, http://lib3ds.sourceforge.net

Other Resources

Site to easily find RPMs, http://rpmfind.net

DirectX developer site, http://msdn.microsoft.com/directx

This page intentionally left blank.

Appendix B

Sample Raytracer Implementation

Introduction

In this appendix, I explain how to implement a simple raytracer. As you saw in Chapter 3, and throughout this book, raytracing and its related techniques can be used in many fields of computer graphics.

I first explain the design of a simple raytracer. I then explain how it can be implemented. I conclude with the different results obtained and how to extend the simple implementation provided. Throughout this appendix, you will see that writing a raytracer is a simple task provided that it is properly planned and that the program is properly designed.

B.1 Design

In this section, I explain how to design a raytracer. I first present the different data types. I then explain the different core functions using these data types, and how all this is put together. The last part of this section explains the syntax of the language used to specify the different primitives, light sources, cameras, surfaces, and settings of the scene to render.

B.1.1 Introduction

You may recall from Chapter 3 that raytracing is a rendering algorithm. It takes a 3D scene description as input and produces a 2D image as output. The input will be a simple ASCII file describing the camera of the virtual scene, the different lights, the primitives, materials, and settings of the rendering. What follows is a list of the raytracer's functionalities:

- The raytracer shall read ASCII files as input. The scene file describes the virtual scene to render.
- It shall generate a 2D picture as output. The image file formats supported shall be uncompressed 24-bit BMP, uncompressed 24-bit TGA, and uncompressed 24-bit PPM.
- It shall be easily extensible. For example, one might add acceleration techniques such as those based on bounding volume hierarchies, stochastic sampling, antialiasing techniques, etc.
- It shall be written in C (C compilers are pervasive on modern systems).
- It shall be portable to any platform where a C compiler is provided. Only standard C functions shall be used (functions that are part of the C standard library).
- It shall support the following geometric primitives: plane, polygon, triangle, and sphere.
- It shall support point lights.
- It shall support light intensity attenuation with distance (in $1/d^2$, also known as the Inverse Square attenuation model).
- It shall use the Phong shading model.
- It shall support specular reflection.
- It shall support specular refraction (simple refraction model).
- It shall render environments with fog.

All of these requirements will have an impact on the design of the raytracer.

B.1.2 Data Types

As the chosen language is C, I will use structures to store the different properties of the entities manipulated. C++ could be used to better encapsulate the methods and instance variables of the data types. Here, I'll simply use function pointers inside structures in place of C++ class methods.

At the core of a raytracer are 3D vectors. They can be used to manipulate actual 3D vectors (directions in space) or 3D points (positions in space). The data type used to represent vectors is called RTVector3d. It has only three variables: x, y, and z, corresponding to the three Cartesian coordinates. C macros are used to scale, multiply (dot product), combine, etc., the 3D vectors. Macros are faster than function calls and far simpler to implement, at the cost of data type checking and other such nice features.

The raytracer allows calculating the light intensity at a given point on a surface. The light intensity can be expressed as an RGB color. The second data type is therefore called

RTRGBColor and its variables are r, g, and b, the three components of a color expressed as double-precision float values. Before outputting to the final image, the color is converted into RGB bytes (in the range [0,255]).

The third data type, RTRay, represents a ray. Its two components are of type RTVector3d and represent the direction of the ray and its origin. A camera structure, RTCamera, is also declared to group the pinhole camera parameters (location, target, field of view, etc.). The RTDisplay data structure holds the characteristics of the output image (name, width, height, and type).

The RTSurface structure contains the different parameters related to the shading model implemented: ambient, diffuse, specularity, specularity factor, reflection, refraction, and refraction index. The RTHaze data structure holds the parameters for rendering fog and haze.

The 3D objects to render are grouped in a linked list. Each member of the linked list is identified by its type (plane, polygon, triangle, or sphere). The structures corresponding to the different supported geometric primitives are RTSphere, RTPlane, RTPolygon, and RTTriangle. All lights are placed in a separate linked list. The structure corresponding to the point light source is called RTLight. Finally, an intersection structure, RTIsect, is used to keep track of the different parameters of a given intersection (distance and primitive intersected).

The UML class diagram shown in Figure B.1 summarizes the different data types related to the 3D primitive data types (anObject).

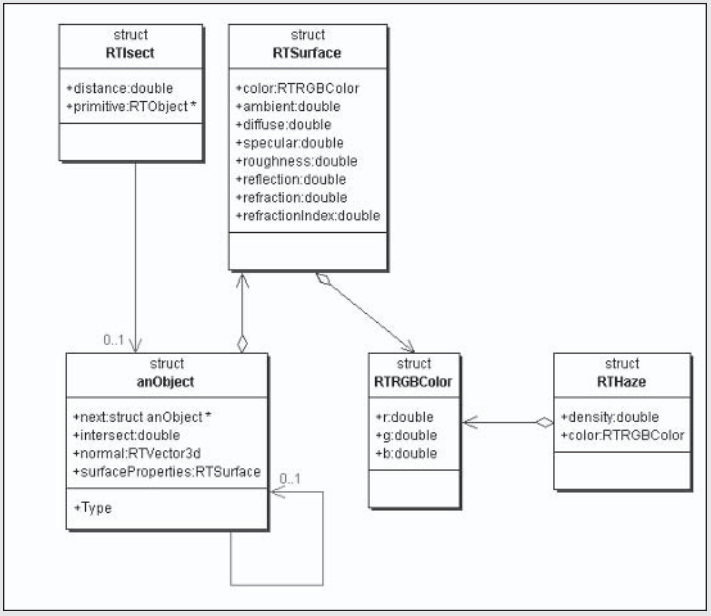


Figure B.1. Raytracer main structures and their relationships

In this diagram, the anObject structure contains a reference (pointer) to the next structure of the same type, thus integrating the concept of a linked list. The intersect and normal members are pointers to functions calculating the intersection of a ray with the current primitive and the normal at a given point on the surface of that primitive. The type member specifies whether the primitive is a sphere, triangle, plane, or polygon.

The UML class diagram shown in Figure B.2 summarizes the different advanced data types involved in the program.

B.1.3 Main Functions and Program Flow

The raytracer has the following functions:

- main(): The main entry point of the program. It sets the different global variables to default values. It parses the input scene file calling the parseFile() function. If no errors occurred during the parsing process, the raytrace() function is called.
- parseFile(): This function parses the input text file containing the scene specifications. It populates the list of lights and the list of primitives with their assigned materials. Upon completion, the function returns to the calling function: main().
- raytrace(): This is the main entry point to the raytracing step. This function opens the output image file to write the results of the rendering (i.e., pixel colors). It initializes the projection (using the pinhole camera model explained in Chapter 3) and the origin of each ray to be traced. It then loops through the image plane, and for each pixel, calculates the direction of the primary ray to be traced. The primary ray is then traced by calling the trace() function. When the trace() function returns from the different recursive calls, the resulting color is written for the current pixel into the output image file. At the end of the loop, the raytrace() function cleans up all allocated memory and closes the output file before returning to the main() function.
- trace(): This is the main function that traces a ray in the scene to determine the color at the first intersection point with a primitive. Since this function is called

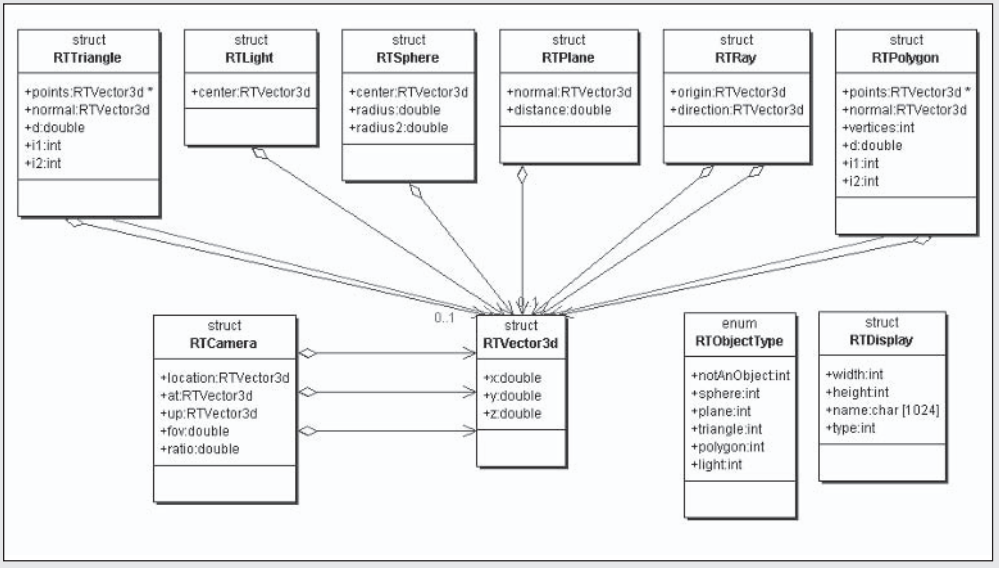


Figure B.2. Data types for a raytracer

recursively, it first checks if the maximum recursion level has been reached, thus avoiding infinite loops. It then checks for an intersection with all primitives present in the scene. If none is found, it returns either the background color or the color of the haze if such a feature is activated, and returns to the calling function. If there is an intersection, it then calculates the point of intersection and the normal at that point, and calls the shade() function before returning to the calling function.

- shade(): This is the main shading function. Its role is to determine the color at a location on a given surface. It first calculates the reflected ray if the object has diffuse or specular reflection properties. It then loops through all the lights in the scene to add their contribution. For each light source, it determines its contribution (i.e., is the object in the shadow of the current light or not?) by tracing a shadow ray, calculates the diffusely reflected light contribution, and finally calculates the specular contribution. It then goes to the next light source in the light list. Once all light contributions have been taken into account, the function traces a reflected ray if the object being shaded has reflective properties by calling the trace() function. It then calls the trace() function with the refracted ray if the object's surface is refractive. Finally, it adds the contribution from the fog, if any. All colors are added and returned to the calling function: trace().

That's about it! Only a few functions are needed in a raytracer. All other functions are auxiliary functions that help in making the code more readable and extensible. It's now time to describe how to implement the raytracer.

B.1.4 Input File Format Specification

As stated above, the raytracer reads simple ASCII files to render 3D scenes. The scene specifications are explained in this section.

B.1.4.1 Basic Data Types

Before describing all the primitives supported, it is important to present the different types of data that are manipulated by the raytracer. The program supports different types of entities that serve as the basis for primitive declarations.

Integers are supported. Integer values can be positive, negative, or null, depending on the context in which they are used. What follows is an example of the use of an integer in a scene script:

```
settings
{
     display 800 600
...
```

This example shows how to define the output format of the image to render. The first integer represents the width of the image; the second integer represents the height of this image (see section B.1.4.2 for more details).

Floats are also supported. Float values can be positive, negative, or null, depending on the context in which they are used. What follows is an example of the use of a float in a scene script:

```
settings
{
     haze_factor 0.345
...
```

This example shows how to define the contribution of haze in a scene (see section B.1.4.2 for more details).

The raytracer makes extensive use of colors in many of its structures. A color is defined by three floating-point components representing (in order of appearance) the red component, the blue component, and the green component. What follows is an example of the use of a color in a scene script:

```
settings
{
     background 0.078 0.361 0.753
...
```

In this example, the background color of the image will have a red component of 0.078, a green component of 0.361, and a blue component of 0.753. Note that for "normal" colors, the components lie between 0.0 and 1.0. When all components are null (0.0), the color specified is black; when they are all equal to 1.0, the color specified is white; and, when the components are all equal, the color specified is a shade of gray. Color components can be greater than 1.0 in the case of lights. As a matter of fact, as stated earlier, light can fade proportionally with distance (like in the real world). If the color specified in the color structure is not sufficient (in this case the color of the light represents the intensity of the light), the scene won't be lit properly. For more details about this effect, see section B.1.4.2.

The program makes extensive use of vectors in many of its structures. A vector is defined by three floating-point components representing (in order of appearance) the "x" component, the "y" component, and the "z" component. What follows is an example of the use of a vector in a scene script:

```
plane
{
     normal <0 0 1>
     ...
```

This example shows how to declare a plane. The "normal" vector represents the normal of the plane (for more details on the plane structure, see section B.1.4.2).

Finally, the raytracer uses strings in different elements of its scripting languages, generally textual directives. Strings are enclosed between double quotes ("). The following is an example of the use of a string in a scene script:

```
settings
{
     output_type "bmp"
     ...
```

This example shows how to specify the BMP file format for the output image. For more details on the output_type directive, see the settings discussion below.

B.1.4.2 Primitives

Now let's see the different structures to declare materials, primitives, lights, cameras, and rendering settings.

Settings — The settings section of the scripting language defines different directives concerning the rendering of the current scene. The syntax is as follows:

```
settings
{
      display              integer integer
      output_type          string
      output_name          string
      jitter               string
      background           color
      haze_color           color
      haze_factor          float
}
```

Here is a description of each parameter of this structure:

- display: Defines the resolution of the final image to render. The keyword is followed by two integers: the width and the height of the image, both in pixels. The display parameter is not mandatory; the default values are 320 (width) x 240 (height).
- output_type: Defines the type of the output. The keyword is followed by a string specifying the format of the output image. The value of this parameter depends on the platform and the support of image file types. See the release notes for more information on the formats supported for a particular platform. Possible values are:
  - "bmp" to specify the BMP file format
  - "tga" to specify the TGA file format
  - "ppm" to specify the PPM file format
- output_name: This keyword is followed by a string that specifies the name of the output image file. The filename specified doesn't need to contain a file extension. The program will automatically append the correct file extension to the name specified.
- jitter: This keyword is followed by a string specifying if jittering should be used during the rendering of the output image. Its value can be either "yes" or "no." This parameter is not mandatory and by default jittering is not activated. Jittering allows the raytracer to introduce pseudorandom values when tracing rays. This feature is useful because it helps reduce the appearance of aliasing effects (see section B.3.1.8) to the human brain. However, jittering shouldn't be used when generating consecutive frames of an animation because pseudorandom values are used when tracing rays and undesired visual effects are generated (noise) when objects move during animation.
- background: This keyword is followed by a value that specifies the background color of the image. If a ray that is traced into the virtual world doesn't hit any primitives, the color returned is that specified by this parameter. This parameter is not mandatory, and its default value is black (0.0 0.0 0.0).
- haze_color: The raytracer can simulate haze effects. This parameter is followed by a value specifying the color of the

haze when this effect is activated. This parameter is not mandatory, and its default value is black (0.0 0.0 0.0).

- haze_factor: This parameter is followed by a float value that specifies the strength of the haze effect in the scene. High values lead to thick haze. A value of 0.0 means that the haze effect is not activated. This parameter is not mandatory, and its default value is 0.0.

The following example shows the declaration of a complete settings structure:

```
settings
{
      display          1024 768
      output_type      "ppm"
      output_name      "myScene"
      jitter           "no"
      background       <0.0 0.0 0.0>
      haze_color       <1.0 1.0 1.0>
      haze_factor      0.05
}
```

Camera — The camera section of the scripting language defines the different parameters of the camera of the scene. The syntax is as follows:

```
camera
{
      position         vector
      look_at          vector
      up               vector
      fov              float
      ratio            float
}
```

Here is a description of each parameter of this structure:

- position: This keyword is followed by a vector that specifies the position of the viewer. This parameter is mandatory and there is no default value.
- look_at: This keyword is followed by a vector that specifies where the viewer is looking in the virtual world. This parameter is mandatory and there is no default value.
- up: This keyword is followed by a vector that defines what vector should be considered as the altitude. For left-handed systems, the value is <0 1 0>; for right-handed systems, it is <0 0 1>. This parameter is not mandatory; its default value is <0 0 1> (the raytracer is right-handed by default).
- fov: This keyword is followed by a float that specifies the field of view in degrees. Small values will lead to an effect of zooming in, while large values will lead to an effect of zooming out. This parameter is not mandatory; its default value is 45.0.
- ratio: This keyword is followed by a float that specifies the ratio of the final image to render. Normal computer screens have a ratio of 4:3. Standard resolutions use this ratio (640 x 480, 1024 x 768, etc.). This parameter is not mandatory; its default value is 4/3, or 1.33333.

The following example shows the declaration of a camera:

```
camera
{
     position        <10 10 10>
     look_at         <0 0 0>
     up              <0 0 1>
     fov             60
     ratio           1.33333
}
```

Light — The light section of the scripting language defines the different parameters of the lights of the scene. The syntax is as follows:

```
light
{
     position        vector
     color           color
}
```

Here is a description of each parameter of this structure:

- position: This keyword is followed by a vector that specifies the position of the light. This parameter is mandatory and there is no default value.
- color: This keyword is followed by a value that specifies the color of the light. The raytracer supports the effect of fading of intensity: The intensity of a light source decreases with the distance from the center of the light source. In this case, the color can be thought of as the intensity of the light. Since this parameter depends on the overall scale of the scene, it needs to be carefully adjusted, and as a result, the color of a light can have components greater than 1.0. This parameter is mandatory and there is no default value.

The following example is a light located at <10 10 10> that is of a white color:

```
light
{
     position        <10 10 10>
     color           1 1 1
}
```

Surface — The surface section of the scripting language defines the different parameters of the surface of an object. Once a surface is declared in a script, it is applied to all the objects that follow in the script. To change a surface, simply declare a new one; it will be used by all following objects within the script.

The syntax is as follows:

```
surface
{
     color           color
     ambient         float
     diffuse         float
     specular        float
     roughness       float
     reflection      float
     refraction      float
     index           float
}
```

Here is a description of each parameter of this structure:

- color: This keyword is followed by a value that specifies the intrinsic color of the object. This color will be used by the raytracing engine along with the other parameters in the structure to determine the exact color at a point on the surface of an object.
- ambient: This keyword is followed by a float that specifies the ambient factor of the object. The ambient term in raytracing is used to simulate the effect of global illumination in a scene. Since standard raytracing cannot determine the exact value of this term, it is simulated by using a constant value. The value can range from 0.0 to 1.0. Typical values range from 0.1 to 0.3. Large values will render the object as if it were glowing.
- diffuse: This keyword is followed by a float (ranging from 0.0 to 1.0) that specifies the amount of light that is reflected off the surface at exactly the same angle it came in. This term is also known as diffuse reflection. A value of 0.5 means that 50 percent of the light seen from the object comes directly from the different light sources of the scene. The rest is absorbed by the object.
- specular: This keyword is followed by a float that controls the way highlights are generated when the light bounces off the object's surface. Simply put, highlights are the bright spots that appear on objects that depend upon viewing angle and illumination angle. The specular term can range from 0.0 to 1.0, 0.0 meaning no highlights, while 1.0 is total saturation of the highlight.
- roughness: This keyword is followed by a float that controls the size of the highlight. Large values will produce large highlights, while small values will give very smooth and small highlights.
- reflection: This keyword is followed by a float that controls the amount of light reflected off the surface of the object. The value can range from 0.0 (no reflection) to 1.0 (total reflection). The light is reflected off the surface at the same angle as the incident light. Note that reflection increases rendering times since new rays must be traced from the intersection point on the surface.
- refraction: This keyword is followed by a float that controls the amount of light refracted by the surface. The value can range from 0.0 to 1.0. It can be thought of as the transparency of the object to which it is applied. Note that refraction increases rendering times since new rays must be traced from the intersection point on the surface. The default value is 0.0 (no refraction).
- index: This keyword is followed by a float that specifies the index of refraction of the object. Typical values are: 1.0 for air, 1.33 for water, 1.5 for glass, and 2.4 for diamond. The default value is 1.0 (air).

The following is an example of a simple surface declaration:

```
surface
{
      color           0.419 0.556 0.137
      ambient         0.1
      diffuse         0.75
      specular        0.3
      roughness       10.0
      reflection      0.25
      refraction      0.0
      index           1.0
}
```

Sphere — The sphere primitive syntax is as follows:

```
sphere
{
      center          vector
      radius          float
}
```

Here is a description of each parameter of this structure:

- center: This keyword is followed by a vector that specifies the center of the sphere. This parameter is mandatory and there is no default value for it.
- radius: This keyword is followed by a float that specifies the value of the radius of the sphere. This parameter is mandatory and there is no default value for it.

The following example shows a sphere centered around the origin (<0 0 0>) and with a 1-unit radius:

```
sphere
{
      center          <0 0 0>
      radius          1
}
```

Plane — The plane primitive syntax is as follows:

```
plane
{
      normal          vector
      distance        float
}
```

Here is a description of the parameters of this structure:

- normal: This keyword is followed by a vector that specifies the normal of the plane. This parameter is mandatory and there is no default value for it.
- distance: This keyword is followed by a float that specifies the value of the distance of the plane from the origin. This parameter is mandatory and there is no default value for it.

The following example shows the declaration of the (x, y) plane at one unit down on the z-axis (z=–1):

```
plane
{
      normal          <0 0 1>
      distance        -1
}
```

Polygon — The polygon primitive syntax is as follows:

```
polygon
{
      integer
      vertex          vector
      vertex          vector
      vertex          vector
      vertex          vector
      ...
}
```

Here is a description of each parameter of this structure:

- The first parameter of the polygon structure specifies the number of vertices. This parameter is mandatory and must be placed as the first entry of the structure; there is no default value for it. The number of vertices must be greater than three. For three vertices, use the triangle structure instead.
- vertex: This keyword is followed by a vector that specifies one of the vertices of the polygon. The vertices must be declared in counterclockwise order.

The following example shows the declaration of a polygon:

```
polygon
{
      4
      vertex          <-1.5 -1.5 0>
      vertex          <1.5 -1.5 0>
      vertex          <2 2 0>
      vertex          <-2 2 0>
}
```

Triangle — The triangle primitive syntax is as follows:

```
triangle
{
      vertex          vector
      vertex          vector
      vertex          vector
}
```

The only parameters in the structure are three vertices using the vertex keyword. The vertices must be declared in counterclockwise order. Only three vertices may be declared. The following example shows the declaration of a triangle:

```
triangle
{
      vertex          <-1.5 -1.5 0>
      vertex          <1.5 -1.5 0>
      vertex          <2 2 0>
}
```

B.2 Implementation

In this section, I explain how to implement a raytracer based on the design that I have just described. I first give an overview of the scene parser. I then explain the implementation of the three main functions: raytrace(), trace(), and shade(). I also describe the ray/primitives intersection routines. Finally, I briefly explain the different auxiliary functions of the raytracer.

B.2.1 Scene Parser Overview

The scene parser is implemented as follows:

- The main entry point is parseFile(). It opens the provided file in ASCII mode and reads keywords until it reaches the end of that file. When a keyword is read, this function tries to recognize it. This is where the keywords for the different structures of the input language are recognized. Only the cameras, settings, surfaces, primitives, and lights are allowed at this stage. Any other keyword will cause the function to exit. Upon recognition of a specific keyword, the function calls one of the following parsing functions: parseCamera(), parseSettings(), parseSurface(), parsePlane(), parseTriangle(), parsePolygon(), parseSphere(), or parseLight(). It continues scanning the file until it reaches the end of the file.
- All other main parsing functions (parseCamera(), parseSettings(), parseSurface(), parsePlane(), parseTriangle(), parsePolygon(), parseSphere(), and parseLight()) have a similar structure: They first check for an open curly bracket. If this character is not found, they return an error and exit. Otherwise, they loop until a closing curly bracket is found. During this loop, they read keywords (the different keywords making their respective structure). For each keyword type, they read a string, integer, real value, vector, or color by calling respectively: readString(), readInteger(), readDouble(), readVector(), or readRGB().
- The core function to read a keyword in the file is readKeyword(). This function scans the input ASCII file using the standard fgetc() to read characters one after another. When a token is recognized, this function exits and returns the keyword type.

The parser has been implemented in the parser.c file. The parsePlane(), parseTriangle(), parsePolygon(), parseSphere(), and parseLight() functions populate the main objects list and the light list, thus providing all necessary data for the raytrace() function.

B.2.2 Core Raytracing Functions

The three core raytracing functions are raytrace(), trace(), and shade(). The first function contains the main loop; the second function traces a given ray, checking for an intersection with primitives in the scene; and the last function returns the color at a given point on the surface of a primitive in the scene.

What follows is the code for the main raytracing loop. Some parts of the code have been removed for clarity (e.g., variable declarations, memory allocation, and deallocation).

```
void raytrace(void)
{
     ...

     /* Open the image for output */
     openImage(myDisplay.name, myDisplay.width, myDisplay.height);

     /* Initialize projection and origin of each ray to be traced */
     initProjection(&ray);

     /* Loop through the image plane */
     for (iY = 0; iY < myDisplay.height; iY++)
     {
          for (iX = 0; iX < myDisplay.width; iX++)
          {
               /* Jitter direction of current ray if needed */
               if (jitter) initRay(iX + RAND(), iY + RAND(), &ray);
               /* Or point ray at the center of the current pixel */
               else initRay(iX + 0.5, iY + 0.5, &ray);

               /* Trace the primary ray for the current pixel */
               trace(&ray, &color, &recurseLevel);

               /* Retrieve the color of the current pixel */
               line[iX * 3] = (BYTE) (255.0 * color.r);
               line[iX * 3 + 1] = (BYTE) (255.0 * color.g);
               line[iX * 3 + 2] = (BYTE) (255.0 * color.b);
          }

          /* Write the current line to the output image */
          writeImageLine(line);
     }
```

```
      /* Close the output image */
      closeImage();
}
```

The trace() function's implementation is as follows (here again, I have removed the variable declaration section):

```
/*
  - Returns the distance between ray's origin and intersection
  - Assigns color based on returned color by shade()
*/
double trace(RTRay *ray, RTRGBColor *color, int *recurseLevel)
{
      ...

      /* Assign the passed color to black */
      color->r = color->g = color->b = 0.0;

      /* Bail if the maximum level of recursion is reached */
      if (*recurseLevel > MAXRECURSIONLEVEL) return 0.0;

      /* Initialize distance to intersection and intersected object */
      dMinT = BIG; minObject = NULL;

      /* Check for intersection */
      intersectAll(ray, &isect);

      /* Retrieve intersection data */
      dMinT = isect.distance; minObject = isect.primitive;

      /* If no intersection is found, return background color */
      if (dMinT == BIG)
      {
            /* Return haze color if density is positive */
            if (myHaze.density > 0.0) myBackGroundColor = myHaze.color;
```

```
            color = myBackGroundColor;

            /* Return distance */
            return dMinT;
      }

      /* If intersection, calculate the point of intersection */
      point = dMinT * ray->direction + ray->origin;

      /* Calculate the normal at point of intersection */
      normal = minObject->normal(minObject, point);

      /* Reverse normal if needed */
      VecDot(normal, ray->direction, dNormalDir);
      if (dNormalDir > 0.0) VecNegate(normal, normal);

      /* Shade at point of intersection */
      shade(minObject, ray, normal, point, color, recurseLevel);

      /* Check for final color components in the range [0,1] */
      color->r = (color->r > 1.0 ) ? 1.0 : color->r;
      color->g = (color->g > 1.0 ) ? 1.0 : color->g;
      color->b = (color->b > 1.0 ) ? 1.0 : color->b;

      /* Return distance */
      return dMinT;
}
```

Finally, the shade() function can be implemented as follows:

```
/*
      Main shading function
      object: the object to shade
      ray: the incoming ray
      normal: the normal at the point of intersection
      point: the point of intersection
      color: the resulting color from the shading
```

```
     recurseLevel: the current level of recursion
*/
void shade(RTObject *object, RTRay *ray, RTVector3d normal, RTVector3d point, RTRGBColor *color, int *recurseLevel)
{
...

     /* Calculate reflected ray if object has diffuse */
     /* or specular reflection properties */
     if ((surface.spec > 0.0) || (surface.reflection > 0.0))
     {
          /* Calculate reflected ray's direction as a unit vector */
          VecDot(ray->dir,normal, K);
          VecComb(1.0/fabs(K), ray->dir, 2.0, normal, reflRay.dir);
          VecNorm(&(reflRay.dir));

          /* Assign intersection point as reflected ray's origin */
          reflRay.origin = point;
     }

     /* Initialize resulting color to ambient term */
     RGBAssignScale(surface.ambient, surface.color, color);

     /* Loop through all lights in scene to add their contribution */
     lightSource = myLightList;
     while (lightSource)
     {
          /* Create light ray for the current light source */
          /* and distance between intersection and current light */
          distanceT = createLightRay(lightSource, point, &lightRay);

          /* Get raw light contribution from current light source */
          lightColor = getLightColor(lightSource, object, &lightRay);

          /* Calculate diffusely reflected light contribution */
          VecDot(normal, lightRay.dir, dDiffuse);
          if ((dDiffuse > EPSILON) && (surface.diffuse > 0.0))
          {
```

```
                /* Light intensity distance attenuation */
                if (lightColor > 1.0)
                {
                     /* Distance attenuation model is: 1/d2 */
                     K = 1.0 / (distanceT * distanceT);

                     /* Scale the light color */
                     RGBScale(lightColor, K, lightColor);
                }

                /* Add diffuse component to result color */
                dDiffuse *= surface.diffuse;
                color += lightColor * surface.color * dDiffuse;
           }

           /* Calculate the specular contribution */
           if (surface.spec > 0.0)
           {
                /* Calculate specular coefficient */
                VecDot(reflRay.dir, lightRay.dir, spec);

                /* If the contribution is significant */
                if (spec > EPSILON)
                {
                     /* Exponent used to control the spot size */
                     spec = pow(spec, surface.rough)*surface.spec;

                     /* Add specular component to result color */
                     color += lightColor * spec;
                }
           }

           /* Go to next light source in light list */
           lightSource = lightSource->next;
      }

      /* If the current object is reflective, trace a reflected ray */
```

```
K = surface.reflection;
if (K > 0.0)
{
      /* Increase recursion level to avoid infinite loop */
      recurseLevel++;

      /* Call trace with reflected ray */
      trace(&reflRay, &newColor, recurseLevel);

      /* Add the contribution of the reflected ray */
      color += newColor * K;

      /* Decrease recursion level */
      recurseLevel--;
}

/* If current object is refractive */
K = surface.refraction;
if ((K > 0.0) && (surface.refrIndex > 1.0))
{
      /* Assign refraction indexes */
      n1 = 1.0;
      n2 = surface.refrIndex;

      /* Calculate refraction direction */
      refractionDirection(n1, n2, ray->dir, normal, refrRay.dir);

      /* Refraction color contribution */
      RGBCreate(0.0, 0.0, 0.0, newColor);

      /* Assign the point as the refracted ray's origin */
      refrRay.origin = point;

      /* Increase recursion level to avoid infinite loop */
      recurseLevel++;

      /* Call trace with refracted ray */
```

```
            trace(&refrRay, &newColor, recurseLevel);
            /* Decrease the recursion level */
            recurseLevel--;

            /* Add contribution of the refracted ray */
            color += newColor * K;
      }

      /* Finally, add contribution from fog, if any*/
      if(myHaze.density > 0.0)
      {
            haze = 1.0 - pow(1.0 - myHaze.density, distanceT);
            color = haze * myHaze.color + (1.0 - haze) * color;
      }
}
```

B.2.3 Ray/Primitive Intersection Functions

Four types of primitives are supported by the raytracer: plane, sphere, triangle, and polygon. Each primitive has its own intersection test routine. Triangles and polygons are planar objects; their intersection routines contain that of the plane (if a given ray doesn't intersect the triangle's embedding plane, it doesn't intersect the triangle!).

What follows are explanations of the different intersection routines:

- Sphere/ray intersection: Two main methods exist: an arithmetic one and a geometric one. In the arithmetic method, the parametric equations of the ray and the sphere are combined. The intersection condition is then turned into solving a polynomial equation of the second degree (i.e., finding the values of t for which $a\,t^2 + b\,t + c = 0$). In the geometric method, the ray's and the sphere's geometric properties are used to express the intersection condition. Both methods are fast, but the geometric method involves slightly fewer operations. The ray/sphere intersection routine was thus implemented following Eric Haines' ray/sphere geometric solution in [1].
- Plane/ray intersection: This routine is the simplest to implement. It involves only two dot products. The first one checks if the ray is parallel to the plane (if it is parallel, there is of course no intersection). The second one helps in checking if the intersection is behind the ray's origin (if it is behind the origin of the ray, there is no intersection). The implementation follows Eric Haines' ray/plane algorithm in [1].
- Triangle/ray intersection: This routine is slightly more complex than the previous ones. Several methods exist. I have implemented the routine based on [2]. In this

method, the routine first checks for an intersection with the triangle's embedding plane. If there is such an intersection, the intersection routine tries to resolve a parametric equation expressing the intersection condition. More precisely, take a triangle made of three vertices, *V0*, *V1*, and *V2*, and a point *P*. *P* will lie inside this triangle if, for the following equation:

$$\overrightarrow{V_0P} = \alpha \overrightarrow{V_0V_1} + \beta \overrightarrow{V_0V_2}$$

the following conditions are verified:

$$\alpha \geq 0, \beta \geq 0, \alpha + \beta \leq 1$$

The first equation is then turned into a set of three parametric equations (one for each coordinate). The trick here is to project the triangle onto one of the primary planes (i.e., *xy*, *xz*, or *yz*). To determine the plane of projection, you simply need to determine the dominant axis of the normal vector of the triangle, and take the plane that is perpendicular to that axis. By projecting the triangle onto its primary plane, the set of equations with *V0*, *V1*, *V2*, and *P* are reduced to only two simple equations with the two unknowns α and β. It is then trivial to resolve these two equations and thus be able to check for the three conditions stated above.

- Polygon/ray intersection: Since a polygon can be considered as a series of triangles (if there are *n* vertices in the polygon, the polygon can be seen as *n* – 2 triangles). The intersection routine is the same as that for a triangle except that it loops through the triangles making the polygon. Figure B.3 shows how a polygon can be split in a series of triangles by connecting its vertices (V0, V1, etc.):

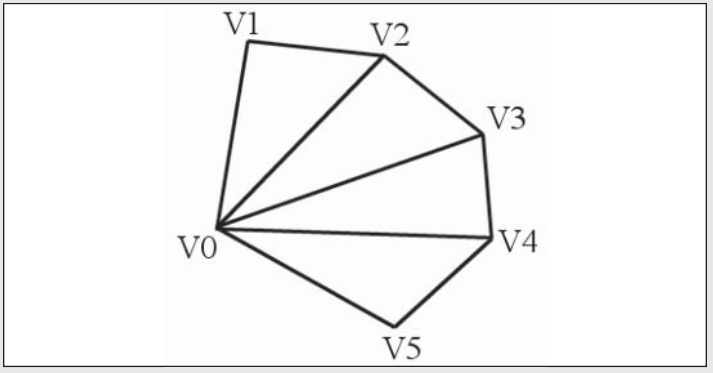


Figure B.3. A polygon as a series of triangles

B.2.4 File List and Auxiliary Functions

The following list gives all the files in the project along with the different auxiliary functions they contain.

- image.c: This file contains all the image manipulation routines. The openImage() function calls the openBMP(), openPPM(), or openTGA() function depending on the output image type. Likewise, writeImageLine() and closeImage() call the functions corresponding to the image file type.
- intersect.c: This file contains the two main intersection routines: intersectAll() and intersectShadow(). The first function is the main intersection routine of the raytracer;

it checks if a given ray intersects an object in the scene. It then stores the intersection information in the intersection structure passed as a parameter. It is called by the trace() function. The second function is the main intersection routine of the raytracer for light/shadow rays. It checks if a given ray intersects an object in the scene that is not the object given as a parameter; it doesn't store the intersection information, but simply returns 1 if there is at least an intersection, and 0 otherwise. It is called by the getLightColor() function.

- light.c: This file contains three light source related functions: createLightRay(), getLightColor(), and makeLight(). createLightRay() creates the light ray for the given light source; it returns the distance between the given intersection point and the given light. getLightColor() returns the color of the light if the ray is not blocked by an object; it returns the color black if the ray is blocked by an object in the scene. Finally, makeLight() simply creates a light and adds it to the light list of the scene.
- main.c: This file contains only the main() function described in the previous section.
- parser.c: This file contains the implementation of the parser as explained before.
- primitives.c: This file contains all primitive related functions. Two functions deal with linked list manipulation: addToList(), which adds a primitive to a given list (object list or light list), and freeList(), which removes all primitives from a given list to free allocated memory when the program exits. The other functions are used directly by the raytracer to manipulate the supported primitives. For each primitive, three functions are defined. The first one is the core intersection routine between that primitive and a ray (functions planeIntersect(), triangleIntersect(), polygonIntersect(), and sphereIntersect()). The second function returns the normal of the primitive at a given point on its surface (functions planeNormal(), triangleNormal(), polygonNormal(), and sphereNormal()). Finally, the third function allocates the memory for the primitive, initializes the different variables related to it, and adds it to the primitive list of the scene to render (functions makePlane(), makeTriangle(), makePolygon(), and makeSphere()).
- ray.c: This file contains only two functions pertaining to ray manipulation. initProjection() initializes the projection according to the camera of the scene and also initializes all primary ray origins based on this camera. initRay() initializes the given primary ray's direction corresponding to a given pixel on the image plane. Both functions are called by the raytrace() function only.
- shade.c: This file contains the shade() function explained in detail previously. It also contains the auxiliary function refractionDirection(), which calculates the refracted ray's direction based on refraction indexes. It was implemented following Andrew S. Glassner's optics for transmission in [1].

- surface.c: This file contains only the setSurfaceAttrib() function, which attaches surface parameters to a given primitive.
- trace.c: This file contains the raytrace() and trace() functions explained previously.
- vector.c: This file contains the VecNorm() function, which normalizes a given vector. All other vector manipulation routines have been implemented as macros and are part of the vector.h file.

B.3 The Raytracing Program

This last section gives a few examples of scenes rendered with the raytracing program implemented throughout this appendix. I also give some pointers to extend it.

B.3.1 Renderings

Following are a few scenes rendered with the raytracing program. These scenes can be found on the companion CD, in the Scenes directory of the raytracer's directory: Code\Appendix B - simpleRaytracer.

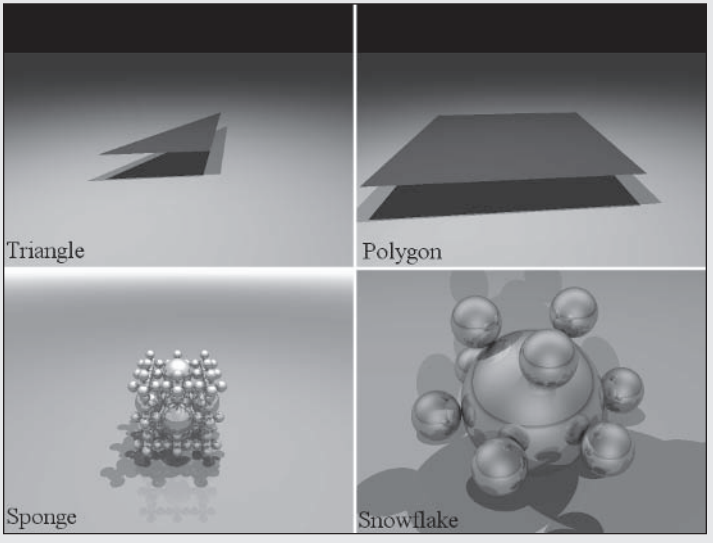


Figure B.4. Simple raytracer scenes with supported primitives

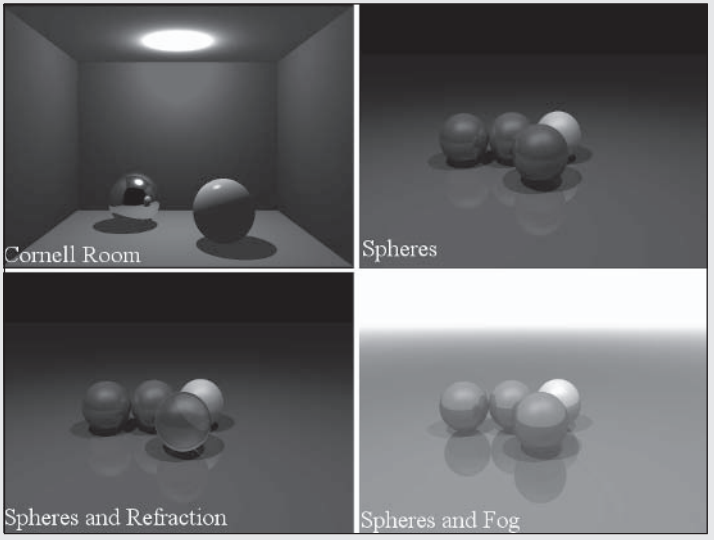


Figure B.5. Simple raytracer supported primitives and some features

B.3.2 Extending the Raytracer

The raytracing program provided with this book can serve as the basis for further extensions and for a more advanced renderer. The following list suggests different areas of improvement:

- Acceleration techniques: As you start playing with this program, you will quickly see that as scenes become larger and larger, the corresponding scenes render more slowly. This is because there is no acceleration technique implemented. For every single call to the trace() function or to the shade() function, the program will test for an intersection between the given ray and all primitives in the scene. Chapter 3 lists different acceleration techniques; one that could be implemented here could use bounding hierarchies volume (BHV). See [1] for more information.
- Texturing: At the moment, the shade() function implements a simple shading model with only solid colors for all surfaces. Several texturing techniques could be implemented including image mapping, normal perturbation (bump mapping), procedural texturing, etc. [3] is a good starting point for this vast subject.
- CSG and more primitives: Only basic primitives are supported (sphere, plane, triangle, and polygon). More primitives can be added and [4] gives numerous pointers to implement ray/primitive intersection routines. Another interesting primitive-related functionality to implement would be CSG (constructive solid geometry). This technique allows combining primitives using Boolean operations to derive other interesting primitives (intersection, union, etc.).
- Stochastic raytracing: By adding Monte Carlo algorithms to the raytracer, it is possible to implement numerous new features, including depth of field, smooth shadows, global illumination, fuzzy reflection, and refraction. [1] is once again a good starting point on the subject. [5] lists recent and modern algorithms for global illumination techniques.

Conclusion

This appendix is only an introduction to this fascinating and complex rendering algorithm. The program that was implemented is fairly simple, but it provides a good basis to further extend the rendering quality achievable. Please refer to Chapter 3 for various books and online references on the subject, in addition to those listed below.

References

[1] Glassner, Andrew S., *An Introduction to Raytracing*, Boston: Morgan Kaufmann, 1989.

[2] Badouel, D., "An Efficient Ray-Polygon Intersection," *Graphics Gems*, Boston: Academic Press, 1990, p. 390.

[3] Ebert, David S., et al., *Texturing and Modeling: A Procedural Approach*, Boston: Academic Press, 2003.

[4] Site dedicated to real-time rendering in general: http://www.realtimerendering.com

[5] Dutré, Philip, Philippe Bekaert, and Kavita Bala. *Advanced Global Illumination*, Natick, MA: AK Peters, 2003.

This page intentionally left blank.

Appendix C

The Lighting and Shading Frameworks

Introduction

In this appendix, I explain the different test bed programs developed to run the different types of shaders found in this book. The programs have been developed on and for different platforms, using the Direct3D and/or OpenGL libraries. More importantly, they were developed in order to run the different shaders coded in the HLSL, Cg, and GLSL shading languages. Throughout this book, these programs have been referred to as frameworks because their sole purpose is to set up the environment for the shaders, run the shaders, and allow the end user to interact with the shaders.

I first describe the DirectX/HLSL framework. More precisely, I explain in this first part how the DirectX test bed programs to run the HLSL shaders have been developed (since all programs are slightly different from one shader to another). In the second section, I explain the OpenGL/Cg framework. This program will serve as the basis for interested readers to adapt all shaders in the book. Indeed, because of the similarities between HLSL and Cg, the adaptation from DirectX/HLSL to OpenGL/Cg should be fairly easy. This second section explains a program that runs simple Cg shaders with different primitives. Finally, the third section of this appendix gives a similar description for the OpenGL/GLSL test bed program. The newer OpenGL shading language presents an interesting approach for developing applications with OpenGL shaders. This section shows how to adapt the HLSL shaders developed throughout this book to the OpenGL shading language.

C.1 DirectX/HLSL Framework

The DirectX/HLSL framework is used for much of the text that concentrates specifically on shaders. Some OpenGL users might find this less than useful at first glance, but the bulk of the interesting material is in the shaders themselves. The DirectX framework is very simple. In most cases, the applications have been kept as minimal as possible to allow the reader to concentrate on the shaders or other higher-level concepts. As such, the frameworks should not be seen as ideal starting points for games but rather simple skeletons on which one can build the higher-level material.

C.1.1 Requirements

Appendix A outlines the basic requirements and setup needed to compile basic DirectX applications. As long as your project links to the Direct3D libraries and the source files include the Direct3D header files, you should be able to compile the code without any effort.

Having said that, compilation does not necessarily equate to success. The applications included with this book have been tested with DirectX 9 class hardware, but users of older hardware might experience problems, especially with the more advanced pixel shaders. If this is the case, the best way to successfully run the applications is to switch the device to the reference device. This will allow you to see the output of the application, but the frame rate will probably be painfully slow. In some cases, readers with pre-DX9 hardware might be better off reading the material for the theory and the approach and then adapting the shaders to run on DX8 hardware at a lower level of quality (such as only using the vertex shaders).

C.1.2 Design

The design of the DirectX sample applications is extremely simple. Every sample is comprised of two classes; one handles the basic device creation and setup functions (CSampleApplication), and the other handles the application-specific code (CLightingApplication). The application-specific code in CLightingApplication is detailed in each chapter, so this section will only explain the core functionality found in CSampleApplication.

C.1.2.1 Introduction

The bulk of CSampleApplication focuses on Direct3D device creation and maintenance. This should already be familiar to most readers. If not, the DirectX SDK includes several tutorial samples that walk the user through each step of device creation and maintenance. In light of that, I will not cover that material here. However, some readers might find that they need to tweak some of the creation parameters to work with their hardware. The settings I use in the code are selected to work with a broad range of DX9 hardware, but users of older or newer cards might have to change the flags and other settings.

In addition to setup, the core class also handles some rudimentary interface functionality and some helper functions that handle loading and creating both shaders and models. That is the material I will focus on.

C.1.2.2 User Interface

There are two elements to the user interface: the diagnostic output to the user and some basic key handling for changing shaders and models. Experienced users might want to change the input side of the interface to accommodate their own needs.

On the output side, the application provides users with the instantaneous frame rate, as well as the name of the current shader. In the case of the frame rate, I have timed everything that happens before the call to Present, which eliminates the influence of the refresh rate of the monitor. This number is therefore artificially high, but it does give you an idea of the relative performance difference between various shaders. Readers are encouraged to adapt the code to fit their own timing needs. Better yet, there are tools available from Microsoft and other vendors that will give you more in-depth diagnostic information. Once the sample applications give you a rough indication of performance, you might want to use other tools to get better information.

For input, CSampleApplication includes the simple function HandleMessage for basic key handling. This is the same one you could use to add mouse handling if needed. I have included a basic set of keyboard commands, outlined in the following table.

Key	*Short Description*
V	Change the current vertex shader (if applicable)
P	Change the current pixel shader (if applicable)
M	Change the current mesh/model
W	Zoom in
S	Zoom out
A and D	Spin the mesh

The user viewpoint is hard-coded, but key commands could easily be added to change that. The purpose of the existing set of key commands is to give the reader an easy way to view the shader results while keeping the framework code to the barest minimum.

C.1.2.3 Data Structures and Instantiation

The sample application framework provides the basic facilities needed to instantiate both shaders and meshes. To do this, I have created three different structures that define vertex shaders, pixel shaders, and model data. Once a sample application instantiates arrays of these structures with the appropriate creation information, CSampleApplication will take care of the instantiation of the actual entities. The three structures are defined in the following tables.

VERTEX_SHADER_INFO	
char *pName	The human-readable shader name as supplied by the higher-level application. You can set this name to whatever you want.
char *pFileName	The name of the file that contains the shader code. These files are assumed to contain HLSL code.
char *pTargetName	The preferred compilation target for this shader. The majority of vertex shaders in this book are targeted to VS1.1, but you can easily experiment with different targets by changing this parameter.
LPDIRECT3DVERTEXSHADER9 pShader	This is the actual shader, assuming it's been successfully created. (Filled by the framework)

PIXEL_SHADER_INFO	
char *pName	The human-readable shader name as supplied by the higher-level application. You can set this name to whatever you want.
char *pFileName	The name of the file that contains the shader code. These files are assumed to contain HLSL code.
char *pTargetName	The preferred compilation target for this shader.
LPDIRECT3DPIXELSHADER9 pShader	This is the actual shader, assuming it's been successfully created. (Filled by the framework)

MODEL_INFO	
char *pName	The human-readable model name as supplied by the higher-level application. You can set this name to whatever you want.
char *pFileName	The name of the file that contains the mesh data. This can be NULL for algorithmically created meshes.
LPD3DXMESH pMesh	A pointer to the mesh object. (Filled by the framework)
LPDIRECT3DVERTEXBUFFER9 pVertexBuffer	A pointer to the vertex buffer. Because of the way I render with shaders, I don't rely on the D3DXMESH render helper methods and I use the vertex and index buffers directly. (Filled by the framework)
LPDIRECT3DINDEXBUFFER9 pIndexBuffer	The index buffer for the mesh. (Filled by the framework)
long NumFaces	The number of faces used in the mesh (needed for draw calls). (Filled by the framework)
long NumVertices	The number of vertices used in the mesh (needed for draw calls). (Filled by the framework)

Once CLightingApplication creates an array of these structures, CSampleApplication uses the structures in the functions CreateShaders and CreateVertices. The following code shows how CreateShaders uses the D3DX helper functions to help create vertex and pixel shaders.

```
void CSampleApplication::CreateShaders()
{
     LPD3DXBUFFER pShaderCode = NULL;

     for (long i = 0; i < NUM_VERTEX_SHADERS; i++)
     {
          if (m_VertexShaders[i].pFileName)
          {
              D3DXCompileShaderFromFile(m_VertexShaders[i].pFileName,
                NULL, NULL, "main", m_VertexShaders[i].pTargetName,
                NULL, &pShaderCode, NULL, NULL);

               m_pD3DDevice->CreateVertexShader(
                              (DWORD*)pShaderCode->GetBufferPointer(),
                              &(m_VertexShaders[i].pShader));

            pShaderCode->Release();
          }
     }

     for (long i = 0; i < NUM_PIXEL_SHADERS; i++)
     {
          if (m_PixelShaders[i].pFileName)
          {
               D3DXCompileShaderFromFile(m_PixelShaders[i].pFileName,
                                         NULL, NULL, "main",
                                         m_PixelShaders[i].pTargetName,
                                         NULL, &pShaderCode, NULL, NULL);

               m_pD3DDevice->CreatePixelShader(
                              (DWORD*)pShaderCode->GetBufferPointer(),
                              &(m_PixelShaders[i].pShader));

            pShaderCode->Release();
          }
     }
}
```

Similarly, CreateVertices uses the model information structure to load any models (from .X files) specified by the higher-level application. Once all of the model files are loaded, CSampleApplication adds a set of default shapes to the set of models. Therefore, it is possible to test the application without loading any meshes from .X files.

```
void CSampleApplication::CreateVertices()
{
     for (long i = 0; i < NUM_MODELS; i++)
     {
          if (m_Models[i].pFileName)
          {
               D3DXLoadMeshFromX(m_Models[i].pFileName,
                                 D3DXMESH_MANAGED,
                                 m_pD3DDevice, NULL, NULL,
                                 NULL, NULL,
                                 &m_Models[i].pModel);
          }
          else
               break;
     }
```

Once the function reaches a structure that does not include a filename, it begins creating shapes using D3DX and placing the resulting data in the subsequent MODEL_INFO structures.

```
     m_Models[i].pName = "Box";
     D3DXCreateBox(m_pD3DDevice, 10.0f, 10.0f, 10.0f,
                   &m_Models[i].pModel, NULL);

     [...create more shapes using similar D3DX
         functions...]
```

Once all the meshes are loaded and created, the application must massage the vertex data to make sure that the vertices are in the format I have chosen for all of the vertex shaders. This task is made easier by the CloneMesh function, which handles cloning the vertex data and translating it into new formats. Once the mesh is cloned, the function extracts the vertex and index buffer so that they can be used in draw calls.

```
     for (; i >= 0; i--)
     {
          LPD3DXMESH pTempMesh;

          m_Models[i].pModel->CloneMesh(D3DXMESH_MANAGED,
                     m_pVertexElements, m_pD3DDevice,
                     &pTempMesh);
          m_Models[i].pModel->Release();
          m_Models[i].pModel = pTempMesh;
          m_Models[i].pModel->GetVertexBuffer(
                     &m_Models[i].pModelVertices);
          m_Models[i].pModel->GetIndexBuffer
                     &m_Models[i].pModelIndices);

          D3DXComputeNormals(m_Models[i].pModel, NULL);

          m_Models[i].NumFaces = m_Models[i].pModel->
                     GetNumFaces();
          m_Models[i].NumVertices = m_Models[i].pModel->
                     GetNumVertices();
     }
}
```

Once these entities are created, CLightingApplication can use the resulting data in whatever way it sees fit. The use of

the resulting shader pointers and vertex buffers can be seen in all of the actual samples.

C.2 OpenGL/Cg Framework

In this section I explain how the test bed program for Cg shaders was developed. As hinted at before, for portability reasons, OpenGL is used in conjunction with Cg. You will find the full source code, project files, and precompiled binaries on the CD in the following directory: Code\Appendix C - OGLCgFramework.

C.2.1 Requirements

As you already saw in Appendix A, the following tools and libraries are required to build the OpenGL/Cg framework:

- OpenGL: OpenGL is used here as the primary 3D API for rendering. Alternatively, Mesa is an open-source library implementing the OpenGL APIs that can be used on most operating systems.
- Visual Studio .NET 2003: This IDE is used to compile the C++ programs on MS Windows platforms.
- GCC: For Linux platforms, GCC is used to compile C++ programs.
- GLUT: In order to develop cross-platform programs with OpenGL, GLUT 3.6 or above is needed. It can be found at [1] and [2].
- Cg: NVIDIA's Cg Toolkit is necessary for the development of vertex and fragment shaders. It can be found for both Linux and Windows at [3].

C.2.2 Design

Let's see now how the program was designed.

C.2.2.1 Introduction

The following list describes what the program does:

- The program can successively display various 3D primitives (sphere, cube, cone, and torus).
- The program can display the primitives with the standard OpenGL shading model.
- The program can display the primitives shaded by a specific shader.
- The program can display the primitives in wireframe or filled mode.
- The program allows using the keyboard to interact with the shader (modifying variables).
- The program can be easily adapted to accept any shader.

C.2.2.2 User Interface

The user interface is limited to the use of the keyboard to interact with the program. The mouse is not used for any interaction (no rotation).

The following table shows the default keyboard mapping with corresponding actions:

Key	*Description*
y/Y	Toggle between the different display modes (wireframe and filled mode)
q/Q	Clean up and exit the program
g/G	Toggle between the OpenGL and shader shading model
s	Scale down the primitive
S	Scale up the primitive
>	Cycle up through the different primitives: sphere, cube, cone, and torus
<	Cycle back through the different primitives: torus, cone, cube, and sphere

The application handles these interactions through the keyboard. It then hands control over to the shader-specific keyboard routine (handleKey(), see section C.2.2.4), thus allowing easily modifiable shader variables at run time and automatic display of the results.

C.2.2.3 Data Structures

The program contains only three data structures. The first one, *CGOGLShaderData*, holds a shader's metadata: the name of the file containing the Cg source code of the shader, a short description of the shader (displayed at run time on the console), the Cg program corresponding to this shader, and its Cg profile. The second structure is used to hold the camera parameters: position, target, field of view, up vector, ratio of the image plane, and the near and far clipping plane distances. Finally, the last structure holds the properties of a simple point light source: position and color.

Figure C.1 shows the UML class diagram that summarizes the different instance variables of these structures.

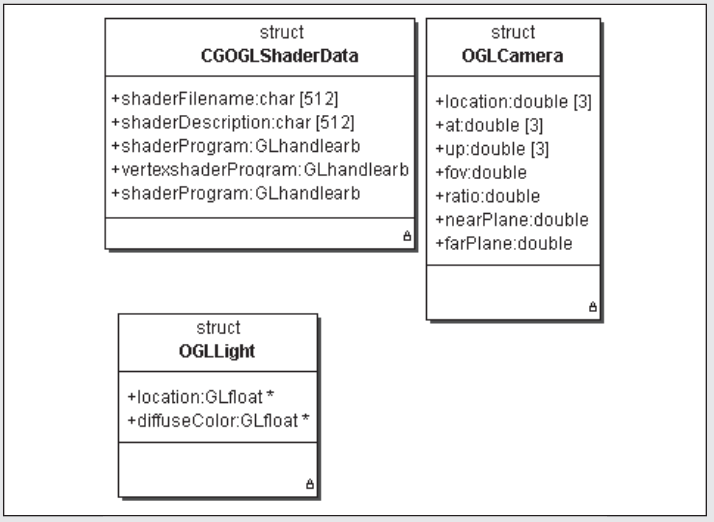


Figure C.1. OpenGL/Cg framework class diagram

C.2.2.4 Functions and Files

The functions are located in two main files: core.c and application.c (with the corresponding header files: core.h and application.h). The core.c file contains, as its name implies, the core functions of the test bed. These functions will be common to all such programs and won't change from one shader to the next. The core functions handle the following:

- Creation of the window
- Initialization of the GLUT callback functions
- Initialization of the OpenGL environment (default OpenGL lighting and shading model)
- Handling of keyboard interaction (as described in the previous section)
- Creation of the Cg context
- Creation and loading of the shader program from a source file
- Display of 3D primitives (sphere, cube, cone, and torus)
- Calling the shader program when the shader display modes are selected

The application.c file contains four functions that are common to all shaders. The first function, handleKey(), is called from the core program keyboard handling routine and is used to add interaction with the shader. It is therefore easy to modify shader variables and automatically see the results. Two functions, initShader() and initShaderData(), are called when the core application loads. The former simply initializes the shader metadata (name of the shader program source file, shader description, and the Cg profile used at run time). The latter binds local variables to shader program variables; this allows modification of the shader program behavior and automatically seeing the results on the screen. The last function located in the application.c file, setupDisplay(), can be used to set up any parameter for the shader program. This function is called from the core program just before calling the execution of the shader program (through the cgGLEnableProfile() function) in the main display routine.

The following table shows the different functions of the programs with the file in which they are located and a brief description of each.

Function Name	*File*	*Description*
checkCgError	core.c	Checks for a Cg error and exit by printing the error message if an error occured. Parameters: none Returns: nothing
cleanup	core.c	Cleans up allocated memory and Cg working environment. Parameters: none Returns: nothing
credits	core.c	Shows code credits Parameters: none Returns: nothing
display	core.c	Main rendering routine. It sets up the camera and calls the function to render the scene in the current lighting and display modes.

Function Name	File	Description
display (cont.)		Parameters: none Returns: nothing
help	core.c	Displays a short help message on the command line Parameters: none Returns: nothing
initGL	core.c	Prepares the OpenGL rendering state by calling various OpenGL functions (glEnable(), glHint(), etc.). Parameters: none Returns: nothing
keyboard	core.c	GLUT keyboard callback function Parameters: the key code and the mouse coordinates when the key was pressed Returns: nothing
main	core.c	The program's main entry point Parameters: the number of arguments to the program and the list of these arguments Returns: an integer value (that can be used by the calling batch file/shell script)
parseArgs	core.c	Parses the command-line arguments to the program and sets various global variables Parameters: the number of arguments to the program and the list of these arguments Returns: nothing
reshape	core.c	GLUT window resizing callback function Parameters: the width and height of the resized window Returns: nothing
usage	core.c	Shows the different command-line switches of the program Parameters: none Returns: nothing
handleKey	application.c	Handles keyboard input specific to the shader Parameters: the key code and the mouse coordinates when the key was pressed Returns: nothing
initShader	application.c	Initializes the shader metadata Parameters: none Returns: nothing
initShaderData	application.c	Binds the local variables to shader program variables Parameters: none Returns: nothing
setupDisplay	application.c	Passes the required parameters to the shader before using them for rendering Parameters: none Returns: nothing

C.2.2.5 Program Flow

The program entry point (the main function main()) works as follows:

1. It first initializes the different default variables of the program: shading model (OpenGL by default), default primitive to render (sphere), primitive size, window resolution (800 x 600), window position when created, and display mode (filled mode by default).
2. It then parses the command-line arguments by calling the parseArgs() function.
3. It calls GLUT's initialization routine (glutInit()), sets GLUT's display mode (double buffering and RGB with glutInitDisplayMode()), and initializes the window resolution and initial position (glutInitWindowSize() and glutInitWindowPosition()) before creating the rendering window (glutCreateWindow()).
4. It then initializes the OpenGL rendering environment by calling the initGL() function. This is where the OpenGL shading model is initialized, the single light is created, and the camera parameters are set.
5. The different GLUT callback functions are then assigned: the display function (glutDisplayFunc() with display()), the reshape function (glutReshapeFunc() with reshape()), and the keyboard function (glutKeyboardFunc() with keyboard()).
6. The Cg working environment is set by calling the appropriate functions: Cg context creation (cgCreateContext()), initialization of the shader program metadata (the shader-specific initShader() in application.c), creation of the shader program (cgCreateProgramFromFile()), loading of the shader program (cgGLLoadProgram()), and binding of shader variables (the shader-specific initShaderData() in application.c).
7. The program then enters GLUT's main loop (glutMainLoop()).
8. Before exiting, the cleanup() function is called. This is where the Cg environment is destroyed.

The main display function (display()) in core.c works as follows:

1. It sets up the 3D view (calls to gluPerspective() and gluLookAt()).
2. The shader program is activated by calling the cgGLBindProgram() function, then calling the shader-specific function setupDisplay() from application.c (where the data binding between the program and the shader program is performed), and finally calling the cgGLEnableProfile() function. These calls are simply bypassed if the OpenGL shading model is selected.
3. It calls the appropriate GLUT function for the selected primitive (glutSolidSphere(), glutSolidCube(), glutSolidCone(), or glutSolidTorus()).
4. If the shader display mode is selected, the cgGLDisableProfile() function is called to disable the shader program.

By modifying the four functions in the application.c file it is therefore easy to load any shader and bind local variables to shader parameters. The possibility to have customized keyboard interaction also makes it easy to modify the shader at run time and get immediate visual feedback.

C.2.3 Results

In this section, I briefly show the results obtained with two Cg shaders. These shaders are almost identical to those developed in HLSL. Only a minor modification has been performed: Shader register parameters have been changed to shader input variables for easier manipulations with Cg and OpenGL.

C.2.3.1 OpenGL Shading

Figure C.2 shows the different primitives rendered in filled mode and wireframe mode using OpenGL standard shading with a red point light source.

C.2.3.2 Simple Shader

In order to illustrate the use of the OpenGL/Cg framework, the ambient shader ambient.hsl was adapted to run within the simplest version of the test bed program. The resulting shader, ambient.cg, is almost identical to the original HLSL shader; it simply returns a plain color for the input vertex (the ambient term). In order to also illustrate interaction with the shader, the OGLCgFramework program allows dynamically changing the ambient color of the primitive displayed. To achieve this, the handleKey() function in the application.c

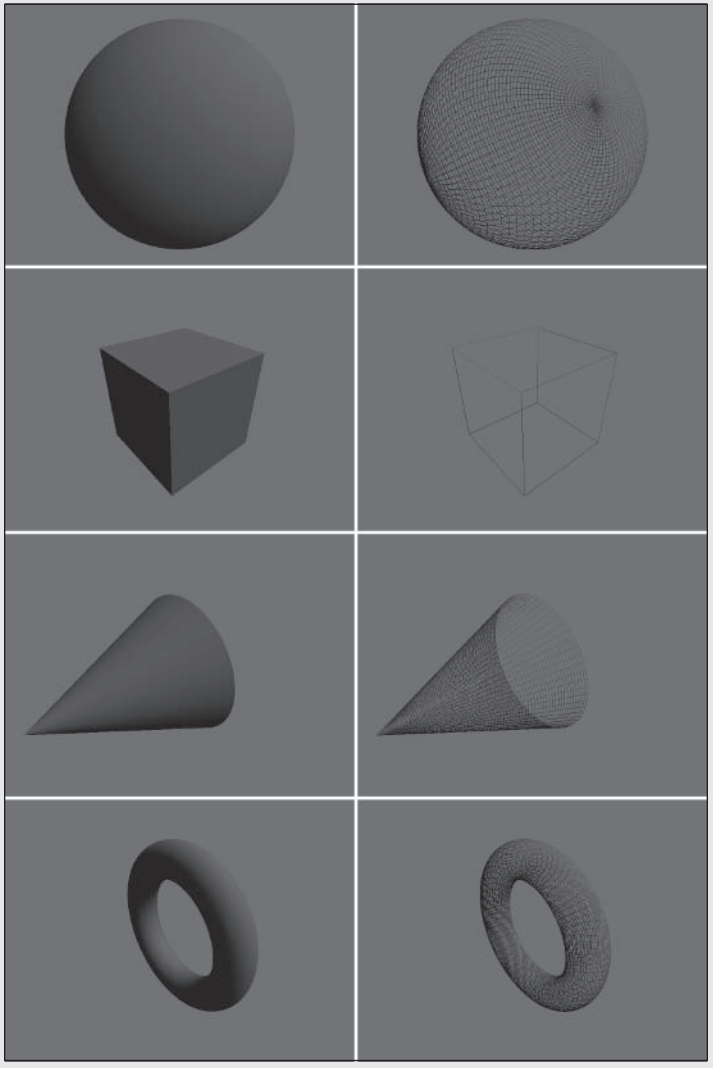


Figure C.2. Simple primitives shaded in OpenGL

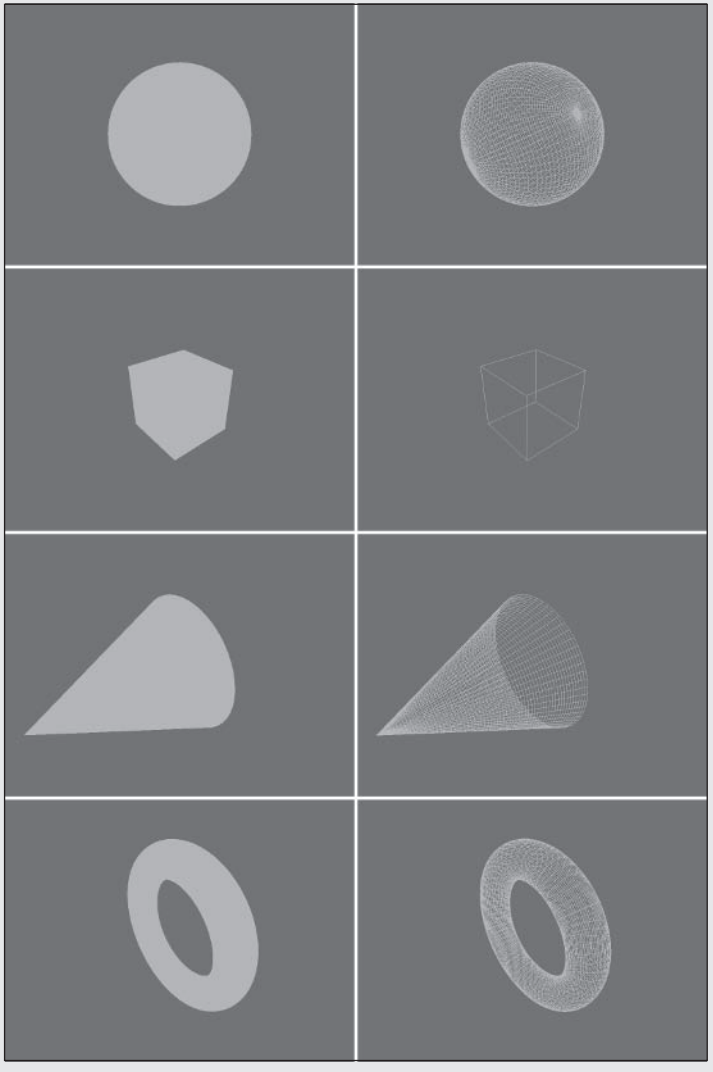


Figure C.3 Simple primitives with the ambient shader

file simply maps the "m" and "n" keys to yellow and green respectively.

Figure C.3 shows the different primitives displayed in filled mode and wireframe mode with a yellow ambient color.

C.2.3.3 Advanced Shader

Another more interesting shader was adapted to illustrate the use of the framework — the Phong shader with a point light source. In this case, the HLSL shader uses register variables to pass input parameters; this was changed in the Cg version by passing all parameters directly to the shader's main function.

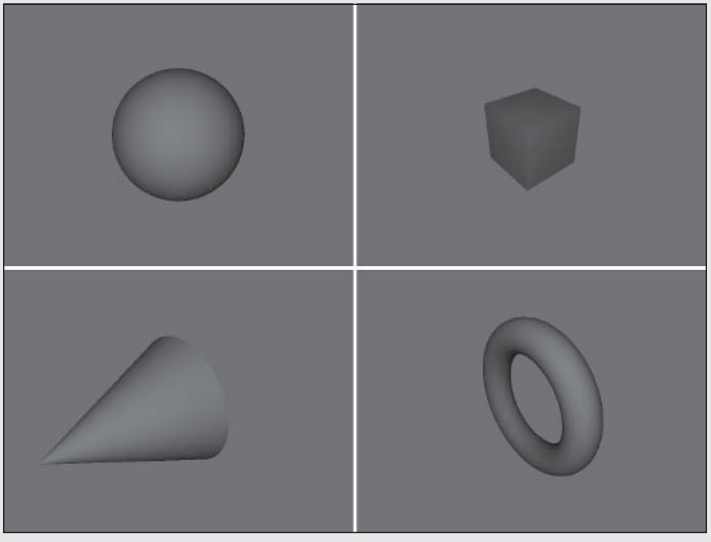


Figure C.4. Simple primitives displayed with the Phong/point light source Cg shader

In this case, the OGLCgPhongPoint program allows dynamically changing the diffuse color of the model displayed by using the same keys ("m" and "n" for yellow and white respectively). The program also makes it possible to adapt the intensity of the point light source of the shader by using the "l" (scale up) and "L" (scale down) keys. Here again, the handleKey() function in the application.c file was modified to provide this functionality.

Figure C.4 shows the different primitives displayed in filled mode with the Phong/point light source Cg shader.

C.3 OpenGL/GLSL Framework

In this section I explain how the test bed program for OpenGL shaders was developed. This framework will be interesting to OpenGL fans who want to adapt and easily test the shaders developed throughout this book. You will also find the full source code, project files, and precompiled binaries on the CD in the following directory: Code\Appendix C - OGLGLSLFramework\.

C.3.1 Requirements

As you already saw in Appendix A, the following tools and libraries are required to build the OpenGL/GLSL framework:

- OpenGL: OpenGL is used here as the primary 3D API for rendering. Mesa is an alternative open-source library implementing the OpenGL APIs.
- Visual Studio .NET 2003: This IDE is used to compile the framework on Windows platforms.
- GLUT: In order to develop cross-platform programs with OpenGL, GLUT 3.6 or above is needed. It can be found at [1] and [2].
- 3Dlabs' GL Shading Language SDK: This SDK is necessary for the development of vertex and fragment shaders on 3Dlabs hardware. It can be found for Windows at [4].
- ATI's GL Shading Language SDK: This SDK is necessary for the development of vertex and fragment shaders on ATI hardware. It can be found for Windows at [5].

C.3.2 Design

Let's see how the program was designed.

C.3.2.1 Introduction

This framework is very similar to the OpenGL/Cg one. The following list describes what it does:

- The program can successively display various 3D primitives (a sphere, a cube, a cone, and a torus).
- The program can display the primitives with the standard OpenGL shading model.
- The program can display the primitives shaded by a specific shader.

- The program can display the primitives in wireframe or filled mode.
- The program allows using the keyboard to interact with the shader (modifying variables).
- The program can be easily adapted to accept any GLSL shaders.

C.3.2.2 User Interface

The test bed program has the same default key mapping as the OpenGL/Cg framework.

The following table shows the default keyboard mapping with corresponding actions.

Key	*Description*
y/Y	Toggle between the different display modes (wireframe and filled mode)
q/Q	Clean up and exit the program
g/G	Toggle between OpenGL and shader shading models
s	Scale down the primitive
S	Scale up the primitive
>	Cycle forward through the different primitives: sphere, cube, cone, and torus
<	Cycle backward through the different primitives: torus, cone, cube, and sphere

The program also hands control over to the shader-specific keyboard routine (handleKey()), thus allowing for easy modification of shader variables at run time and automatically seeing the results.

C.3.2.3 Data Structures

The program contains only three data structures. The camera and point light structures are the same as before. The GLSLShaderData data structure contains the shader's metadata: the names of the files containing the GLSL source code (vertex and fragment shader), a short description of the shader (displayed at run time in the console), the GLSL program handle, and the GLSL handles for the vertex and fragment programs. Figure C.5 shows the UML class diagram that summarizes the different instance variables of these structures.

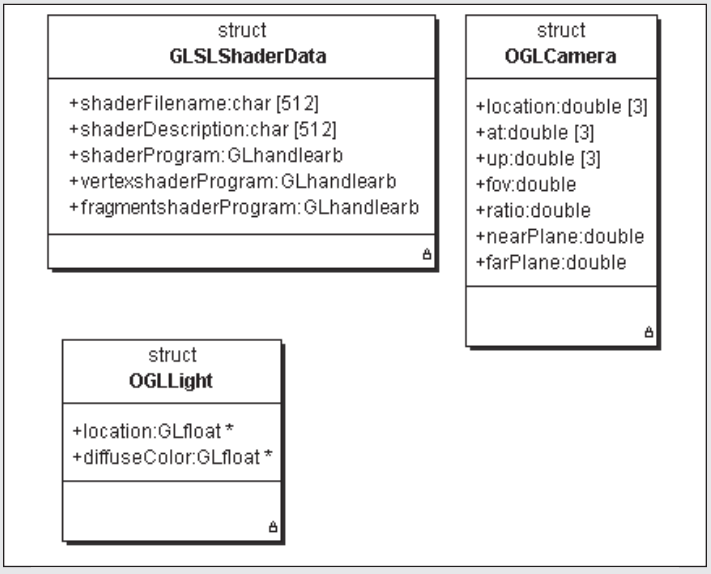


Figure C.5. UML class diagram of the GLSL framework

C.3.2.4 Functions and Files

Here again, the functions are located in two main files: core.c and application.c (with the corresponding header files core.h and application.h). The core.c file contains the core functions of the test bed program. These functions handle the following:

- Creation of the OpenGL window
- Initialization of the GLUT callback functions
- Initialization of the OpenGL environment (default OpenGL lighting and shading model)
- Initialization of the GLSL environment (different function calls depending on the SDK used)
- Handling of keyboard interaction (as described in the previous section)
- Loading and compilation of the shader programs from source files
- Linking and creation of the GLSL shader program
- Display of 3D primitives (sphere, cube, cone, and torus)
- Calling the shader program when the shader display mode is selected

The application.c file contains exactly the same routines as for the OpenGL/Cg framework, the only difference being that GLSL API calls replace CG APIs calls.

The following table shows the new functions developed specifically for this framework.

Function Name	*File*	*Description*
compileShader	core.c	This function has been adapted from the ATI GLSL SDK program example. It compiles the shader referenced by the shader handle given as the input parameter. It returns the compilation status. Parameters: a GLhandleARB representing the shader to compile Returns: an integer value representing the compilation status
linkProgram	core.c	This function has been adapted from the ATI GLSL SDK program example. It links the program referenced by the program handle given as the input parameter. It returns the link status. Parameters: a GLhandleARB representing the program to link Returns: an integer value representing the link status
loadShaderFile	core.c	This function loads the shader source code from a text file and assigns it the shader referenced by the shader handle given as the input parameter. Parameters: the name of the shader source file and the shader to which the source code needs to be assigned Returns: nothing
loadShaders	core.c	Loads and compiles the shaders and links the program by calling the functions above Parameters: none Returns: nothing

C.3.2.5 Program Flow

The program's flow is also very similar to that of the OpenGL/Cg framework. The main function, main(), works as follows:

1. It first initializes the different default variables of the program: shading model (OpenGL by default), default primitive to render (sphere), primitive size, window resolution (800 x 600), window position when created, and display mode (filled mode by default).
2. It then parses the command-line arguments by calling the parseArgs() function.
3. It calls GLUT's initialization routine (glutInit()), sets GLUT's display mode (double buffering and RGB with glutInitDisplayMode()), and initializes the window resolution and initial position (glutInitWindowSize() and glutInitWindowPosition()) before creating the rendering window (glutCreateWindow()).
4. If the 3Dlabs SDK is used, it initializes the GLSL function pointers. If the ATI SDK is used, it initializes the OpenGL environment and the different extensions necessary (call to SetupGL1_2(), SetupARBExtensions(), and SetupGLSLExtensions()).
5. It then initializes the OpenGL rendering environment by calling the initGL() function. This is where the OpenGL shading model is initialized, the single light is created, and the camera parameters are set.
6. The different GLUT callback functions are then assigned: the display function (glutDisplayFunc() with display()), the reshape function (glutReshapeFunc() with reshape()), and the keyboard function (glutKeyboardFunc() with keyboard()).
7. The GLSL working environment is set by calling the appropriate functions: initialization of the shader program metadata (the shader-specific initShader() in application.c), loading, compilation and linking of the vertex and fragment shaders (loadShaders()), and binding of shader variables (the shader-specific initShaderData() in application.c).
8. The program then enters GLUT's main loop (glutMainLoop()).
9. Before exiting, the cleanup() function is called. This is where the GLSL environment is destroyed.

The shader loading function (loadShaders()) in core.c works as follows:

1. It creates the program and shader objects by calling the GLSL function glCreateProgramObjectARB().
2. It loads the vertex and fragment shader source code from the specified source files (extension .vert and .frag) by calling the loadShaderFile() function in core.c. This function in turns calls the glShaderSourceARB() function to load the source code into the GLSL environment.

3. The compileShader() function is then called to compile the shaders. The GLSL function glCompileShaderARB() is used to do this.
4. The compiled shaders can then be attached to the program (call to glAttachObjectARB()).
5. The last step consists of linking the resulting program (call to the linkProgram() in core.c, which in turns calls the glLinkProgramARB() function).

Finally, the main display function (display()) in core.c works as follows:

1. It sets up the 3D view (calls to gluPerspective() and gluLookAt()).
2. The shader program is activated by calling the glUseProgramObjectARB() function, then calling the shader-specific function setupDisplay() from application.c (where the data binding between the program and the shader program is performed). If the OpenGL shading model is selected, the glUseProgramObjectARB() function is called with 0 as the parameter (to disable the shader).
3. It calls the appropriate GLUT function for the selected primitive (glutSolidSphere(), glutSolidCube(), glutSolidCone(), or glutSolidTorus()).

By changing the four functions in the application.c file, it is therefore easy to load any shader and bind local variables to shader parameters. The possibility of having customized keyboard interaction also makes it easy to modify the shader at run time and get immediate visual feedback.

C.3.3 Results

Let's see the results obtained with the GLSL test bed program.

C.3.3.1 OpenGL Shading

The OpenGL shading is exactly the same as with the OpenGL/Cg framework. Refer to the previous section for screen shots of the results obtained.

C.3.3.2 Simple Shader

To illustrate the use of the OpenGL/GLSL framework, a very simple shader was adapted from Cg/HLSL: the ambient shader. This shader doesn't do much except assign a flat ambient color to the given primitive. The handleKey() function is exactly the same as that of the OGLCgFramework program: The "m" key sets the ambient color to yellow, and the "n" key sets the ambient color to green. The initShaderData() and setupDisplay() functions have been changed slightly to set the input parameter to the shader program by calling the glGetUniformLocationARB() and glUniform3fvARB() functions.

Figure C.6 shows the different primitives displayed in filled mode with a green ambient color.

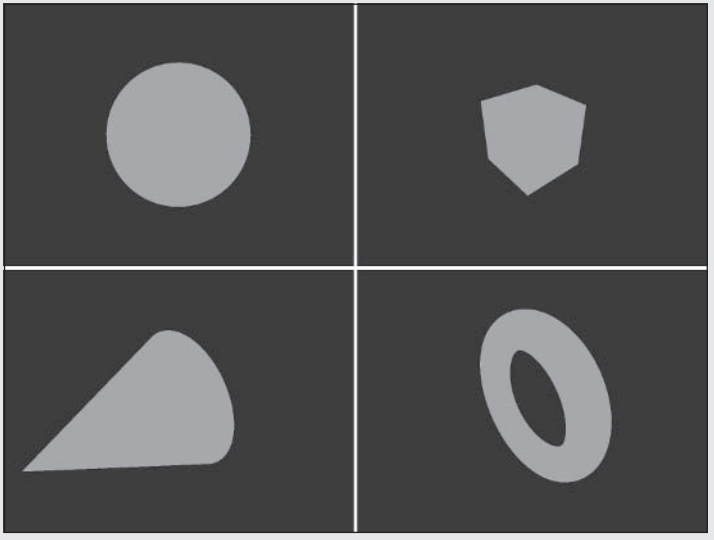


Figure C.6. Simple primitives rendered with the ambient GLSL shader

References

[1] GLUT main site, http://www.sgi.com/software/opengl/glut.html

[2] GLUT Win32 main site, http://www.pobox.com/~nate/glut.html

[3] Cg main site, http://developer.nvidia.com/view.asp?IO=cg_toolkit

[4] 3Dlabs OpenGL SL site, http://www.3dlabs.com/support/developer/ogl2/index.htm

[5] ATI GLSL SDK, http://www.ati.com/developer/sdk/RadeonSDK/Html/Samples/OpenGL/GLSLSimpleShader.html

Other Resources

ATI Developer's site, http://www.ati.com/developer/

Main DirectX web site, http://www.microsoft.com/directx

DirectX developer site, http://msdn.microsoft.com/directx

OpenGL community site, http://www.opengl.org

Index

P

R

Y

Z

Looking for more?

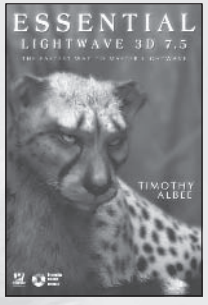
Check out Wordware's market-leading Graphics Library featuring the following new releases, backlist, and upcoming titles.

LightWave 3D 8 Texturing
1-55622-285-8 • $49.95
6 x 9 • 504 pp.

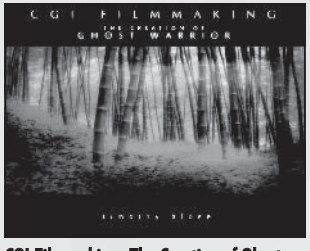
LightWave 3D 8 Lighting
1-55622-094-4 • $54.95
6 x 9 • 536 pp.

LightWave 3D 8: 1001 Tips & Tricks
1-55622-090-1 • $39.95
6 x 9 • 648 pp.

LightWave 3D 7.5 Lighting
1-55622-354-4 • $69.95
6 x 9 • 496 pp.

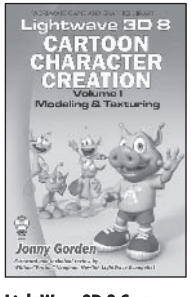


Essential LightWave 3D 7.5
1-55622-226-2 • $44.95
6 x 9 • 424 pp.

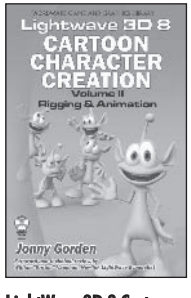


CGI Filmmaking: The Creation of Ghost Warrior
1-55622-227-0 • $49.95
9 x 7 • 344 pp.

Coming Soon



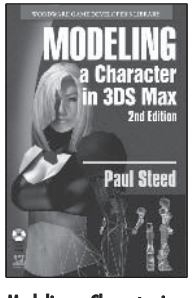
LightWave 3D 8 Cartoon Character Creation Vol. I
1-55622-253-X • $49.95
6 x 9 • 496 pp.



LightWave 3D 8 Cartoon Character Creation Vol. II
1-55622-254-8 • $49.95
6 x 9 • 440 pp.

3ds max Lighting
1-55622-401-X • $49.95
6 x 9 • 536 pp.

Essential LightWave 3D 8
1-55622-082-0 • $44.95
6 x 9 • 600 pp.



Modeling a Character in 3ds max
1-55622-088-X • $44.95
6 x 9 • 600 pp.

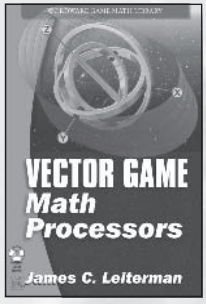
LightWave 3D 8 Character Animation
1-55622-099-5 • $49.95
6 x 9 • 400 pp.

Visit us online at www.wordware.com for more information.
Use the following coupon code for online specials: advlight2920

for more?

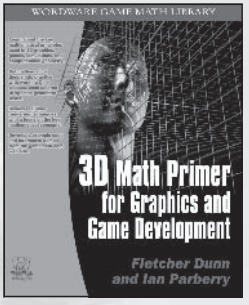
leading Game Developer's Library releases and backlist titles.

Java 1.4 Game Programming
1-55622-963-1 • $59.95
6 x 9 • 672 pp.

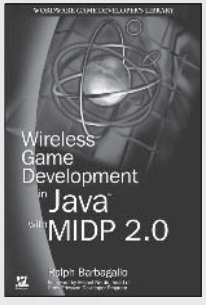


Vector Game Math Processors
1-55622-921-6 • $59.95
6 x 9 • 528 pp.

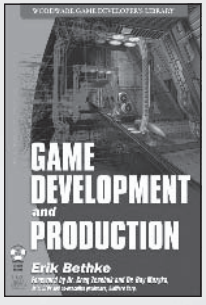
Game Design Foundations
1-55622-973-9 • $39.95
6 x 9 • 400 pp.



3D Math Primer for Graphics and Game Development
1-55622-911-9 • $49.95
7½ x 9¼ • 448 pp.

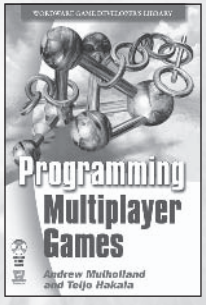


Wireless Game Development in Java with MIDP 2.0
1-55622-998-4 • $39.95
6 x 9 • 360 pp.



Game Development and Production
1-55622-951-8 • $49.95
6 x 9 • 432 pp.

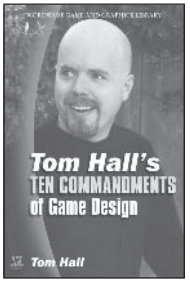
Official Butterfly.net Game Developer's Guide
1-55622-044-8 • $49.95
6 x 9 • 424 pp.



Programming Multiplayer Games
1-55622-076-6 • $59.95
6 x 9 • 576 pp.

Coming Soon

OpenGL Game Development
1-55622-989-5 • $59.95
6 x 9 • 500 pp.



Tom Hall's Ten Commandments of Game Design
1-55622-075-8 • $49.95
6 x 9 • 500 pp.

Use the following coupon code for online specials: advlight2920

This page intentionally left blank.

About the Companion CD

The companion CD contains the following directories:

- Code: This directory contains all the code in the book, with resources (models, textures, probes, etc.) needed to run the programs.
- Extras: This directory contains the installation program for Luminance Radiosity Studio by Stéphane Marty.
- Libraries: This directory contains the SDKs and libraries required to build the programs of the book. See Appendix A to install these and build the programs.

Please see the readme.txt file in each directory for more information.

For the latest versions of the programs, patches, extras, etc., check out http://www.advancedrenderingtechniques.com.

Enjoy!

CD/Source Code Usage License Agreement

Please read the following CD/Source Code usage license agreement before opening the CD and using the contents therein:

1. By opening the accompanying software package, you are indicating that you have read and agree to be bound by all terms and conditions of this CD/Source Code usage license agreement.
2. The compilation of code and utilities contained on the CD and in the book are copyrighted and protected by both U.S. copyright law and international copyright treaties, and is owned by Wordware Publishing, Inc. Individual source code, example programs, help files, freeware, shareware, utilities, and evaluation packages, including their copyrights, are owned by the respective authors.
3. No part of the enclosed CD or this book, including all source code, help files, shareware, freeware, utilities, example programs, or evaluation programs, may be made available on a public forum (such as a World Wide Web page, FTP site, bulletin board, or Internet news group) without the express written permission of Wordware Publishing, Inc. or the author of the respective source code, help files, shareware, freeware, utilities, example programs, or evaluation programs.
4. You may not decompile, reverse engineer, disassemble, create a derivative work, or otherwise use the enclosed programs, help files, freeware, shareware, utilities, or evaluation programs except as stated in this agreement.
5. The software, contained on the CD and/or as source code in this book, is sold without warranty of any kind. Wordware Publishing, Inc. and the authors specifically disclaim all other warranties, express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the disk, the program, source code, sample files, help files, freeware, shareware, utilities, and evaluation programs contained therein, and/or the techniques described in the book and implemented in the example programs. In no event shall Wordware Publishing, Inc., its dealers, its distributors, or the authors be liable or held responsible for any loss of profit or any other alleged or actual private or commercial damage, including but not limited to special, incidental, consequential, or other damages.
6. One (1) copy of the CD or any source code therein may be created for backup purposes. The CD and all accompanying source code, sample files, help files, freeware, shareware, utilities, and evaluation programs may be copied to your hard drive. With the exception of freeware and shareware programs, at no time can any part of the contents of this CD reside on more than one computer at one time. The contents of the CD can be copied to another computer, as long as the contents of the CD contained on the original computer are deleted.
7. You may not include any part of the CD contents, including all source code, example programs, shareware, freeware, help files, utilities, or evaluation programs in any compilation of source code, utilities, help files, example programs, freeware, shareware, or evaluation programs on any media, including but not limited to CD, disk, or Internet distribution, without the express written permission of Wordware Publishing, Inc. or the owner of the individual source code, utilities, help files, example programs, freeware, shareware, or evaluation programs.
8. You may use the source code, techniques, and example programs in your own commercial or private applications unless otherwise noted by additional usage agreements as found on the CD.

Warning:

By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement. Additionally, opening the CD package makes this book nonreturnable.