Foundations

of

Game Engine Development

Foundations

of

Game Engine Development

VOLUME 2

RENDERING

---

by Eric Lengyel

Foundations of Game Engine Development
Volume 2: Rendering

About the cover: The cover image is a scene from a game entitled *The 31st*, developed with the Tombstone Engine. Artwork by Javier Moya Pérez.

Contents

Preface

This book explores the mathematical foundations of real-time rendering methods used in modern game engines. This vast subject includes an enormous variety of techniques that must balance the goals of visual quality and performance. These goals are achieved through the application of solid engineering principles, and we emphasize that fact by including a large amount of technical detail. Because details require a lot of space, we could not hope to cover every major topic in the field of computer graphics in a single volume, so we focus primarily on the most important fundamentals. But even those topics have become rather complex, and this book is filled with some lengthy expositions as a result.

The six chapters in this book are numbered five through ten, and this numbering is a continuation of the four chapters appearing in the first volume. In this second volume, we assume that the reader is familiar with the subject matter covered in Chapters 1–3, which includes vectors, matrices, transforms, and basic 3D geometry. We do not rely on any knowledge of the advanced algebra covered in Chapter 4. The first volume is not required reading for someone who is already proficient in mathematics, but there are occasional references in this book to specific locations in Volume 1. Those can be identified by a first number of 1, 2, 3, or 4 in the $x.y$ notation used for numbering sections, equations, etc.

We begin the second volume with a general overview of graphics processing in Chapter 5. This includes an introduction to color science, a discussion of game world structure, and a light review of graphics hardware capabilities. Chapter 6 is much more mathematical, and it contains a thorough treatment of projections. In Chapter 7, we move on to the topic of shading, where we describe the details of basic rendering concepts as well as some advanced texture mapping techniques. That is followed by a rather long discussion of lighting and shadows in Chapter 8.

The focus changes somewhat in Chapter 9 where we address performance on a large scale by visiting the topics of visibility determination and occlusion culling. Finally, Chapter 10 presents several advanced rendering techniques in extreme detail. Much of this combines the author's own original research with existing methods to provide highly practical engineering advice.

This book does not teach the use of any specific graphics API or shading language, and it is not intended to be a primary source of information about how to use a graphics processor in general. Many excellent resources on these topics are available, and we prefer not to include redundant material here. However, it was necessary to choose certain conventions employed by graphics APIs and shading languages to be used throughout this book. The choices of axes in various coordinate systems generally agree with Direct3D and Vulkan. Shader code is written in a generic hybrid of HLSL and GLSL. The types and built-in functions found in our shaders use the names defined by HLSL, but uniform constants and texture functions use the GLSL syntax for its succinctness. Conversion to the reader's shading language of choice is expected to be a simple task.

There are many example code listings in this book, and most of them are much more complicated than the listings found in Volume 1. Programs that run on the CPU are written in C++, and programs that run on the GPU are written in our generic shading language. None of these examples are written in "pseudo-code". With the exception of minor tweaks that might be necessary to compile for a specific environment, all of the examples consist of production-ready code that works. The code listings can be downloaded from the website given below, and a license for its usage is included. To save space, a lot of the code appearing in the book is written in a highly compact form, sometimes to the detriment of readability. The code on the website is more loosely formatted. In the case of the marching cubes algorithm discussed at the end of Chapter 10, some data tables have been omitted from the code listings in the book, but they are included with the code on the website.

We assume that the reader has a solid understanding of basic trigonometry and the linear algebra topics covered by Chapters 1–3 of the first volume. We also assume a proficiency in calculus (including Newton's method) because its use is necessary at many places throughout this book. On the engineering side, we assume that the reader has a working knowledge of the C++ language, some experience using a GPU shading language, and an understanding of how standard floating-point numbers operate.

Important equations and key results appearing in the text are boxed with a blue outline. This is intended both to highlight the most valuable information and to make it easier to find when using the book as a reference.

Each chapter concludes with a set of exercises, and many of those exercises ask for a short proof of some kind. The exercises are designed to provide additional educational value, and while many of them have easy solutions, others are more involved. To ensure that getting stuck doesn't deprive any reader of a valuable learning experience, the answers to all of the exercises are provided on the website cited below.

This book is the second volume in a series that covers a wide range of topics related to game engine development. The official website for the *Foundations of Game Engine Development* series can be found at the following address:

foundationsofgameenginedev.com

This website contains information about all of the books in the series, including announcements, errata, code listings, and answers to exercises.

Mathematical Conventions

We continue to use the mathematical conventions and much of the same notation that was introduced in the first volume:

- A vector $\mathbf{v}$ representing a direction vector is written in a bold style, and a vector $\boldsymbol{p}$ representing a point is written in a bold script style. These are differentiated because, when extending to 4D homogeneous coordinates, a direction $\mathbf{v}$ acquires a w coordinate of zero, and a point $\boldsymbol{p}$ acquires a w coordinate of one.
- The notation $(\mathbf{p}\,|\,w)$ represents a homogeneous point with $\mathbf{p} = (x, y, z)$, the notation $[\mathbf{n}\,|\,d]$ represents a plane with normal direction $\mathbf{n}$ and distance to origin d, and the notation $\{\mathbf{v}\,|\,\mathbf{m}\}$ represents a line with direction $\mathbf{v}$ and moment $\mathbf{m}$.
- Indexes referring to rows and columns of a matrix are zero-based. For example, the notation M_{00} refers to the upper-left entry of the matrix $\mathbf{M}$.
- The notation $\mathbf{M}_{[j]}$ refers to the zero-based j-th column of the matrix $\mathbf{M}$. If $\mathbf{M}$ is a 4×4 transformation matrix having a fourth row of $[0\ \ 0\ \ 0\ \ 1]$, then $\mathbf{M}_{[0]}$, $\mathbf{M}_{[1]}$, and $\mathbf{M}_{[2]}$ are direction vectors, and $\mathbf{M}_{[3]}$ is a point.
- Ordinary vectors are treated as column matrices. Normals and planes, which are antivectors, are treated as row matrices.
- A matrix $\mathbf{M}$ transforms a vector $\mathbf{v}$ as $\mathbf{M}\mathbf{v}$. A normal is transformed as $\mathbf{n}\mathbf{M}^{-1}$.
- If $\mathbf{v}$ is a 4D vector, then the notation $\mathbf{v}_{xyz}$ refers to the 3D vector consisting of the x, y, and z components.

In this volume, we make use of two common shader functions, saturate and normalize, in mathematical expressions appearing outside shader code. They are abbreviated $\text{sat}(x)$ and $\text{nrm}(\mathbf{v})$ and defined as follows.

$$\text{sat}(x) = \min\left(\max(x,0),1\right)$$

$$\text{nrm}(\mathbf{v}) = \mathbf{v}/\|\mathbf{v}\|$$

Acknowledgements

David Hoffmann
Roland Holm
Erik Høyrup Jørgensen
Yuri Karabatov
Soufiane Khiat
Christopher Kingsley
Attila Kocsis
Kim Kulling
Yushuo Liu
Yunlong Ma
Diane McGraw
Steven Mello
Nicholas Milkovits
Javier Moya Pérez
Kamil Niechajewicz
Daan Nijs
Norbert Nopper
Gabor Pap

Filoteo Pasquini
Nikolaos Patsiouras
Raul Pena
Phillip Powers
Michael Quandt
Darren Ranalli
Randall Rauwendaal
Lane Roathe
Jorge Rodriguez
Nathan Rowe
Jonathan Rumion
Raphael Sabouraud
Soner Sen
Chris Sexton
Michael D. Shah
Torgrim Bøe Skårsmoen
Adam Skoglund

Sean Slavik
Daniel Smith
Aaron Spehr
Diogo Teixeira
Runar Thorstensen
Geoffrey Tucker
Kevin Ubay-Ubay
Jim Vaughn
Lars Viklund
Ken Voskuil
Shawn Walker-Salas
Albert Wang
Andreas Weis
Ian Whyte
Graham Wihlidal
Myron Wu
Robert Zasio

Graphics Processing

The rendering components of a game engine have the job of producing a two-dimensional image representing the current state of a three-dimensional world as seen from a given viewing position at a particular point in time. Each of these two-dimensional images is called a *frame*, and the speed at which they are produced is called the *frame rate*. Most applications need to render images at a frame rate of at least 30 frames per second, and many need no less than 60 frames per second. This allows very little time, no more than about 33 milliseconds and 16 milliseconds, respectively, for each frame to be drawn. The entire process can be extremely complex, and it usually involves an enormous amount of computation despite the fact that game engines are painstakingly designed to minimize their own workloads. When a frame is rendered, a sophisticated system involving large amounts of code for determining what parts of the world are visible under the current camera configuration typically runs on the CPU. The code that ultimately determines what color is assigned to each pixel on the display runs on the *graphics processing unit* (GPU) as a specialized set of relatively small programs called *shaders*. This chapter provides an overview of the basic parts of this whole process and the general capabilities of the graphics hardware.

5.1 Pixels

During each frame, the visual output of a game engine is basically a large array of colored pixels that form a detailed image. The term *pixel*, which is a shortened form of *picture element*, refers to the smallest individually addressable component of an image, and the dimensions of an image are usually characterized by the number of pixels composing its full width and height. The term *resolution* is often used to

mean the dimensions of an image, but it can also refer to the number of pixels per unit distance, such as pixels per centimeter or pixels per inch. There are many standard image dimensions that correspond to the full display size of computer monitors and television sets, and several of the most common of these, both current and historical, are listed in Table 5.1. With time, the display resolutions have naturally increased in overall size, and the standard *aspect ratio*, the ratio of the width to the height, has generally converged to 16:9.

Standard Name	Display Resolution	Mpixels	Aspect Ratio
VGA Video Graphics Array	640×480	0.307	4:3
XGA Extended Graphics Array	1024×768	0.786	4:3
HD / 720p High Definition	1280×720	0.922	16:9
SXGA Super Extended Graphics Array	1280×1024	1.31	5:4
FHD / 1080p Full High Definition	1920×1080	2.07	16:9
QHD Quad High Definition	2560×1440	3.69	16:9
4K UHD Ultra High Definition	3840×2160	8.29	16:9
8K UHD Ultra High Definition	7680×4320	33.2	16:9

Table 5.1. This is a list of some of the most common standard display resolutions available along with an illustration of their relative sizes over the time period in which hardware-accelerated 3D game engines have existed.

Each of the millions of pixels on a typical display has an independent color determined by a mixture of red, green, and blue light emitted by the screen in various proportions at the pixel's location. The brightness of the pixel corresponds to the overall intensity of the three colors of light as the ratio among them is maintained. This system for producing color imagery is called *RGB color*, and it is the standard way in which game engines compute the color information that is ultimately displayed to the user through the GPU.

The RGB color data for an image is stored in a block of memory called a *frame buffer*. Depending on the format, each pixel can occupy up to 128 bits of space inside the frame buffer, but images meant for display typically use 32 or 64 bits per pixel. The storage for each pixel is divided into four *channels* that are normally interpreted as 8-bit unsigned integers in the case of 32 bits per pixel and as 16-bit floating point values in the case of 64 bits per pixel. Three of the channels contain the intensities of the red, green, and blue color components. The fourth channel is called the *alpha channel*, and it can contain any extra per-pixel information that may be useful to the application. The alpha channel is not displayed, and it exists because it's much easier for computers to work with multiples of four than it is to work with multiples of three. The term *RGBA color* refers to a four-channel color that has red, green, blue, and alpha components.

The 32-bit RGBA pixel format having 8-bit unsigned integer channels is shown in Figure 5.1. Here, the value of each channel can be one of 256 values in the range 0 to 255, or in hexadecimal, `0x00` to `0xFF`. This number is an encoding of a fractional intensity value in the range 0.0 to 1.0, representing the minimum and maximum intensities possible. When the graphics hardware reads an 8-bit channel, it divides by 255 to obtain the corresponding intensity value.

Byte 0	Byte 1	Byte 2	Byte 3
Red	Green	Blue	Alpha

Figure 5.1. This is a typical byte layout for a 32-bit RGBA pixel format in which one byte is used to store values in the range 0–255 for red, green, blue, and alpha channels.

5.2 Color Science

As a first step before learning about the methods used to render the 3D images, it is important to understand how humans perceive color and why the RGB system works, so we begin with a short introduction to *color science*. Color science is a vast subject that could easily fill an entire book, but we limit our discussion to the material necessary to understand the theoretical basis for the RGB color standard and the proper ways to use it in practice.

The reason that RGB color is so prevalent as a way of producing color imagery has to do with the anatomy of human vision. Under brightly lit conditions, the perception of light near the center of the eye is dominated by three types of *cone cells* in the retina that are sensitive to different ranges of electromagnetic radiation within the visible spectrum. This is called *photopic vision*, and it differs from *scotopic vision* under dimly lit conditions in which *rod cells* with no color discerning abilities dominate. The three types of cone cells are classified as L cones, M cones, and S cones for long, medium, and short wavelengths. The relative sensitivities to light due to each of the three types of cones are shown in Figure 5.2 as functions of wavelength, where the radiant intensity of the light remains constant. The L cones make the greatest contribution to the perceived brightness of light because roughly 63% of the cone cells in the retina are L cones, with the M cones accounting for about 31% and the S cones making up the remaining 6%.

The total brightness perceived by human vision regardless of color is represented by the *luminosity function $V(\lambda)$*, also shown in Figure 5.2, and it roughly approximates the sum of the contributions from the three different types of cone cells. The luminosity function peaks at a wavelength of 555 nm in the green-yellow part of the visible spectrum, and this is therefore the approximate color that humans perceive as the brightest. Conversely, blues, violets, and deep reds are perceived as much darker colors because they lie on the tail ends of the luminosity function.

The light that we see, either emitted directly from a light source or reflected from various surfaces, is usually composed of many different wavelengths, and the specific composition is called the *spectral power distribution* of the light. This distribution can be expressed as a function $P(\lambda)$ giving the power corresponding to each wavelength λ. The total amount of power emitted by a light source is called its *radiant flux Φ_E*, and it is equivalent to the integral

$$\Phi_E = \int_\lambda P(\lambda)\,d\lambda, \tag{5.1}$$

where λ ranges over all wavelengths. Radiant flux is usually measured in watts, and the function $P(\lambda)$ typically has units of watts per nanometer (W/nm). The

subscript E stands for "energetic", and it indicates that the radiant flux Φ_E is based purely on the energy emitted from a light source. Because human perception of brightness depends on wavelength, radiant flux concentrated near one wavelength may not appear to have the same brightness as an equal amount of radiant flux concentrated near another wavelength. A different quantity of measurement called the *luminous flux* Φ_V takes this into account by weighting the spectral power distribution $P(\lambda)$ by the luminosity function $V(\lambda)$ inside the integral to produce

$$\Phi_V = 683 \int_\lambda V(\lambda) P(\lambda) \, d\lambda. \tag{5.2}$$

Luminous flux is measured in units called *lumens*, which have the symbol lm. The subscript V stands for "visual", and it indicates that the luminous flux Φ_V incorporates the wavelength-dependent sensitivity of human vision. The constant factor of 683 appearing in Equation (5.2) reflects the fact that, by definition, there are 683 lumens per watt where the luminosity function peaks (with a weight of one) at 555 nm. The value 683 was chosen so that quantities based on luminous flux measured in lumens would closely match historical units such as candlepower.

Regardless of how many different wavelengths of light are present, the three types of cone cells are capable of generating only three separate signals that can be sent to the brain for processing, and the intensities of those three signals are combined to form the final perceived color. The magnitude of each signal is determined by a weighted sum of the intensity of the light at each wavelength, where the weights are given by the sensitivity curves shown in Figure 5.2. In the continuous sense, the signal is the integral of the product of the light's spectral power distribution and the cone sensitivity function over the entire visible range of wavelengths.

Because there are infinitely many ways that the same three signal magnitudes could be produced by integrating different spectral power distributions, colors perceived by the brain do not have unique spectral compositions. It is possible for two (or many) different mixtures of wavelengths to appear as identical in color even though they have significantly different spectral power distributions, and such matching compositions are called *metamers*. All that matters is that the L, M, and S cones are stimulated with the same magnitudes, called the *tristimulus values*, and the perception of color does not change. Furthermore, when two compositions of light are mixed together, their separate L, M, and S tristimulus values add in an almost exact linear way to produce the tristimulus values of the mixture, which is know as *Grassmann's law* (named after the same Hermann Grassmann as the Grassmann algebra introduced in Chapter 4).
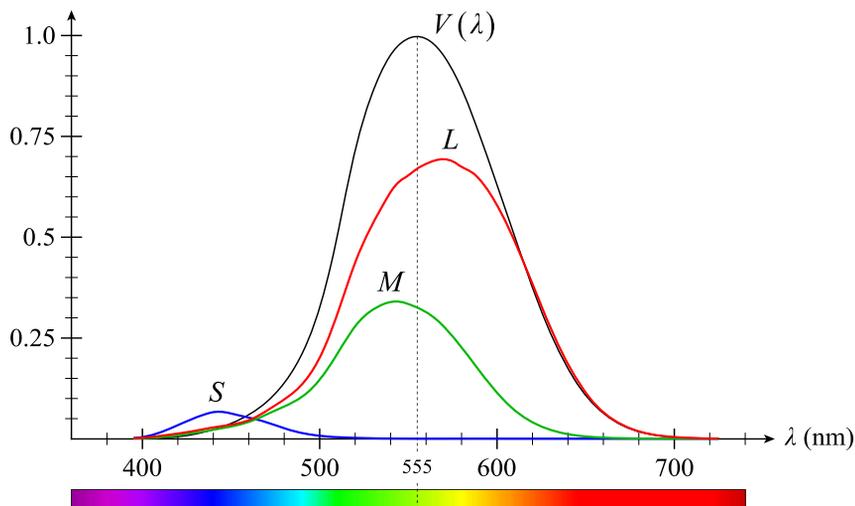
Figure 5.2. The red, green, and blue curves represent the relative sensitivities to light for an average observer at long, medium, and short wavelengths λ due to the stimulation of L, M, and S cone cells. The black curve is the luminosity function $V(\lambda)$ corresponding to the perceived brightness of light with the same radiant intensity at any particular wavelength, and it is roughly the sum of the other three curves.

5.2.1 The CIE RGB Color Space

The linearity of the additive nature of the tristimulus values means that they form a three-dimensional vector space. However, this vector space is a bit peculiar because its basis vectors do not correspond to tristimulus values that are physically possible. There are no wavelengths of light that stimulate only one of the three types of cone cells without also stimulating at least one of the other two. If we do want our basis vectors to have physical meaning, then we can choose a set of three discrete wavelengths, called *primaries*, that can be mixed together to produce a wide range of perceived colors. This concept led to color matching experiments in the 1920s that employed red, green, and blue lights as a basis for L, M, and S cone stimulation.

In the color matching experiments, the wavelengths of the green and blue lights were defined to be 546.1 nm and 435.8 nm, respectively, because they correspond to emission lines for mercury vapor lamps. These wavelengths could be easily reproduced, and they are close to the peak sensitivities of the M and S cones in the green and blue parts of the spectrum. For the red light, a wavelength of

700 nm was chosen because the L cones have a wide range of sensitivity and account for most of the perception of reddish colors even though they peak in effectiveness around the yellow part of the spectrum. Light with an exact wavelength of 700 nm was more difficult to produce at the time, but some amount of error was acceptable due to the fact that very little distinction among perceived colors occurs in that part of the spectrum.

The result of the experiments was the set of color matching functions $\bar{r}(\lambda)$, $\bar{g}(\lambda)$, and $\bar{b}(\lambda)$ shown in Figure 5.3 providing the amounts of the red, green, and blue primaries needed to match monochromatic light of a given wavelength λ. These functions were standardized in 1931 by the International Commission on Illumination (abbreviated CIE for its French name *Commission internationale de l'éclairage*), and they became the bedrock upon which almost all of color science is now built by defining the *CIE RGB color space*. This is not the same RGB color space that virtually all display devices use today, but it does constitute the basis from which the modern standard is derived, as discussed below.

Figure 5.3. These are the color matching functions for the CIE RGB color space. Each of the functions $\bar{r}(\lambda)$, $\bar{g}(\lambda)$, and $\bar{b}(\lambda)$ is multiplied by a given spectral power distribution and integrated over all visible wavelengths λ to determine the amounts of the R, G, and B primaries that combine to reproduce the perceived color.

Note that light of a single wavelength not equal to one of the primaries cannot be exactly reproduced by mixing the *R*, *G*, and *B* primaries together. This is reflected in the fact that the color matching functions all have ranges in which they take on negative values. This is obvious for the red function, but it also occurs very subtly for the green and blue functions. As an example, consider an attempt to match the color of light having a wavelength of 500 nm using a mixture of the green and blue primaries. When green light is added at its wavelength of 546.1 nm, it causes a much greater stimulation of the *L* cones than would normally occur at 500 nm, as can be seen at these two wavelengths in Figure 5.2. Basically, too much red has been added, and thus the function $\bar{r}(\lambda)$ must be negative at 500 nm to compensate. Since a contribution from a primary cannot be negative, we think of a negative value in one of the color matching functions to be an addition of the same amount to the monochromatic light that we are attempting to match, now creating a mixture of two wavelengths that is as close as we can get to the pure spectral color. This limitation defines the *gamut* of a color space, which is the set of all colors that can be produced by mixing together the primaries in any ratio.

Given an arbitrary spectral power distribution $P(\lambda)$, the color matching functions determine the corresponding amounts of the *R*, *G*, and *B* primaries through the integrals

$$R = \int_{\lambda} \bar{r}(\lambda) P(\lambda)\, d\lambda$$

$$G = \int_{\lambda} \bar{g}(\lambda) P(\lambda)\, d\lambda$$

$$B = \int_{\lambda} \bar{b}(\lambda) P(\lambda)\, d\lambda, \tag{5.3}$$

where the wavelength λ covers the full visible spectrum and is usually taken to be the range 380–780 nm. The color matching functions are based on experimental data, and their values are tabulated at regularly spaced wavelengths, so the integrals actually represent finite sums over a large number of data points.

5.2.2 The CIE XYZ Color Space

The CIE RGB color space has a couple qualities that, at the time of its standardization, were seen as somewhat undesirable if it was to be used as an international standard upon which all color measurements would be based. First, the possible negative values generated by Equation (5.3) were considered unwieldy. Second, a light's *luminance*, the perceived brightness without regard for color, was mixed into all three of the *R*, *G*, and *B* values. It was preferable to have all perceptible

colors expressed in terms of positive tristimulus values and to have luminance given directly by one of the tristimulus values. This led to the creation of the *CIE XYZ color space* that is defined by the exact linear transformation

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 2.768892 & 1.751748 & 1.130160 \\ 1 & 4.590700 & 0.060100 \\ 0 & 0.056508 & 5.594292 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{5.4}$$

with respect to the CIE RGB color space and its underlying experimental data.

When the transformation in Equation (5.4) is applied to the RGB color matching function $\bar{r}(\lambda)$, $\bar{g}(\lambda)$, and $\bar{b}(\lambda)$ shown in Figure 5.3, it produces the XYZ color matching functions $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$ shown in Figure 5.4, which each have positive values over the entire visible spectrum. Furthermore, the entries in the second row of the matrix in Equation (5.4) were carefully chosen so that the function $\bar{y}(\lambda)$ would be as close as possible to the luminosity function $V(\lambda)$ shown in Figure 5.2, and these are usually accepted as being exactly equal. Thus, the Y component of a color directly corresponds to its luminance, or apparent brightness, in the XYZ color space.
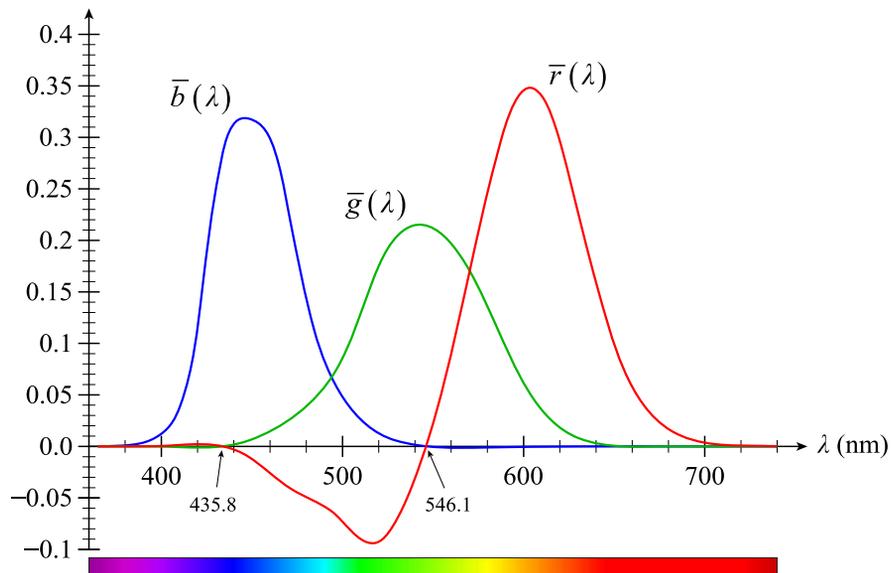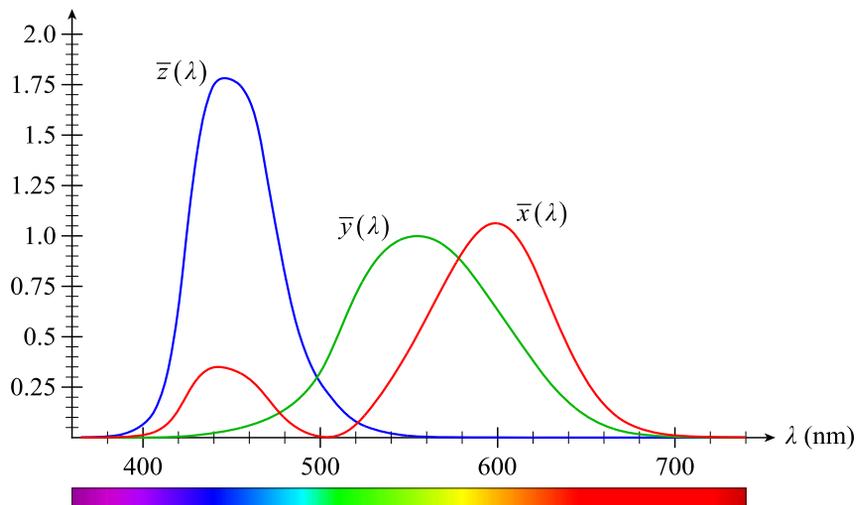

Figure 5.4. These are the color matching functions for the CIE XYZ color space. Each of the functions $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$ is multiplied by a given spectral power distribution and integrated over all visible wavelengths λ to determine the values of the abstract X, Y, and Z components of the perceived color.

The X and Z components are somewhat abstract quantities that correspond to *chromaticity*, which is a measure of perceived color without regard for brightness. However, these components scale with the luminance Y as a color becomes lighter or darker. Normalized chromaticity coordinates x, y, and z that are always in the range $[0,1]$ and do not change with luminance are calculated by scaling a color (X, Y, Z) so that it lies on the plane $X + Y + Z = 1$ using the formulas

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{Y}{X+Y+Z}, \quad \text{and} \quad z = \frac{Z}{X+Y+Z}. \tag{5.5}$$

Because these scaled values sum to one, only two of them are needed to fully describe the chromaticity of a color, and the third value is redundant. We can choose any two, but the standard practice is to take the x and y chromaticity coordinates and combine them with the original Y luminance value to form what is known as the *xyY color space*. This is the universal standard through which any color visible to an average human observer, whether it be monochromatic or a mixture of many wavelengths, can be given a precise numerical specification, and it serves as a basis for defining other color spaces. Given the x, y, and Y values for a particular color, we can calculate the X and Z values using the inverse formulas

$$X = \frac{x}{y} Y \quad \text{and} \quad Z = \frac{z}{y} Y = \frac{1-x-y}{y} Y. \tag{5.6}$$

These are useful for converting among color spaces that are defined by the chromaticities of their primaries.

A two-dimensional plot of the colors corresponding to the xy chromaticity coordinates is shown in Figure 5.5. This is called the *CIE xy chromaticity diagram*, and even though it is not possible to reproduce all of them accurately in print, every color that can be perceived by the average human eye is included. The curved line forming most of the outer boundary is the *spectral locus* consisting of monochromatic colors. The chromaticity coordinates of a point on the spectral locus is found by using the $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$ color matching functions shown in Figure 5.4 to determine the X, Y, and Z components for a single wavelength and then applying Equation (5.5) to obtain x and y. Any color not lying on the spectral locus is a mixture of multiple wavelengths. In particular, the shortest and longest wavelengths on the spectral locus are connected by the *line of purples* near the bottom of the diagram. This line contains all of the fully saturated shades of purple that cannot be produced by any single wavelength of light. Any pair of xy coordinates falling outside the colored region of the diagram is considered to be the chromaticity of an "imaginary" color because it has no physical manifestation.
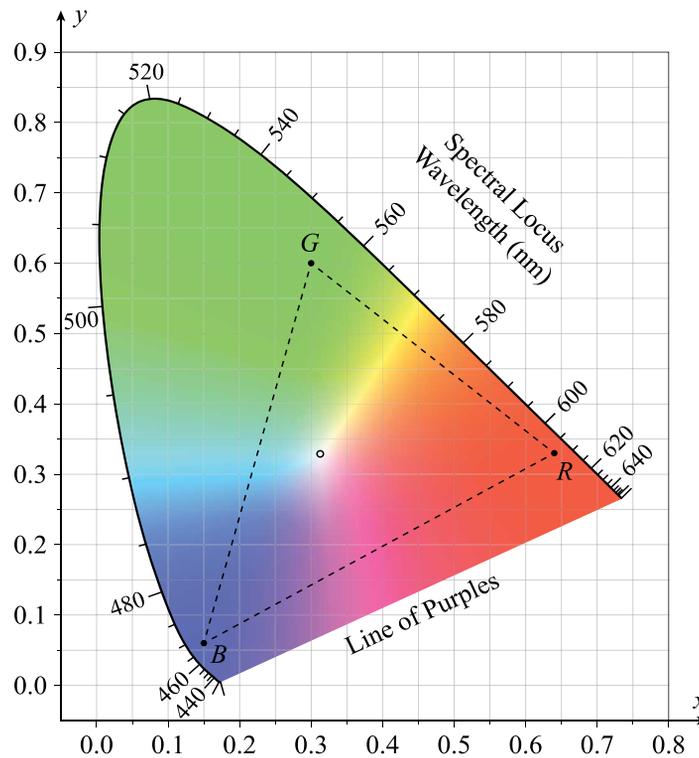
Figure 5.5. The CIE *xy* chromaticity diagram encompasses all colors visible to an average human observer. The curved line is the spectral locus, and it represents the monochromatic colors composed of a single wavelength of light. The dashed triangle is the sRGB color gamut enclosing all mixtures of the red, green, and blue primaries. The small circle inside the gamut is the D65 white point corresponding to typical daylight.

The $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$ color matching functions are normalized to have the same values when integrated for a flat light source having the same power at every visible wavelength. This means that for such an ideal light source, $X = Y = Z$, and thus the *xy* chromaticity coordinates of perfectly white light are $\left(\frac{1}{3}, \frac{1}{3}\right)$. Of course, a flat spectral power distribution doesn't actually occur in everyday settings, so the CIE has defined *standard illuminants* that have specific relative powers given for a table of visible wavelengths. The standard illuminant that pertains to computer graphics is called *illuminant D65*, and its spectral power distribution is shown in Figure 5.6. Illuminant D65 is designed to approximate average daylight in a wide range of geographic locations throughout the year. The number 65
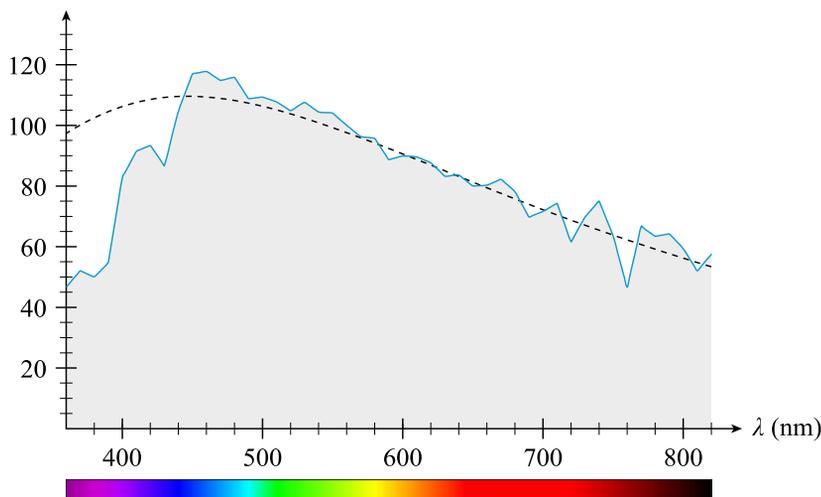
Figure 5.6. The solid blue line is the spectral power distribution of the standard illuminant D65, which is designed to approximate typical daylight. The dashed black line is the distribution given by Planck's black-body radiation law for a temperature of 6500 K.

indicates that the illuminant is correlated with ideal spectral power distribution predicted by Planck's black-body radiation law for a temperature of 6500 K, which is shown as the dashed line in the figure.

When we integrate the product of the $\bar{x}(\lambda)$, $\bar{y}(\lambda)$, and $\bar{z}(\lambda)$ color matching functions shown in Figure 5.4 and the spectral power distribution of illuminant D65 shown in Figure 5.6, the resulting chromaticity coordinates (x_W, y_W) are given by

$$x_W = 0.3127$$
$$y_W = 0.3290. \tag{5.7}$$

These coordinates define the *white point* for the illuminant, and this location is shown on the chromaticity diagram in Figure 5.5. The luminances of the primaries in an additive color space are typically defined by requiring that a mixture of the primaries in equal proportions produces the given chromaticity of the white point at a luminance of $Y_W = 1$.

5.2.3 The sRGB Color Space

In the late 1990s, chromaticities of red, green, and blue primaries were chosen to closely match the display devices of the time, and they define what is known as the

standard RGB color space, or as it's commonly called, *sRGB color*. (This is different from the CIE RGB color space, but because the sRGB primaries are defined in terms of chromaticities in the CIE XYZ color space, there is a concrete relationship between the two.) The sRGB color space is the color space in which most computer graphics applications do their work by default, and it can be assumed as the correct color space when no other color space has been specified.

The exact chromaticity coordinates (x_R, y_R), (x_G, y_G), and (x_B, y_B) of the sRGB primaries are defined as

$$
\begin{aligned}
(x_R, y_R) &= (0.64, 0.33) \\
(x_G, y_G) &= (0.30, 0.60) \\
(x_B, y_B) &= (0.15, 0.06).
\end{aligned}
\tag{5.8}
$$

These primaries are shown as the points R, G, and B in Figure 5.5, and the triangle connecting them represents the color gamut of the sRGB color space. Colors outside of this triangle cannot be produced on a device that works only with positive values for each of the primaries. A large region of colors is excluded from the gamut in the upper-left part of the diagram, but the spectral locus is stretched out in that region. Thus, larger distances there correspond to the same perceptible differences in color as do smaller distances in other regions, so the gamut's coverage isn't as bad as it might seem based on area comparisons.

The sRGB color space defines the color white as having the chromaticity coordinates given by Equation (5.7). This allows us to calculate luminances Y_R, Y_G, and Y_B of the red, green, and blue primaries by requiring that the colors (x_R, y_R, Y_R), (x_G, y_G, Y_G), and (x_B, y_B, Y_B) sum to the white point $(x_W, y_W, 1)$. We cannot add colors directly with their (x, y, Y) coordinates, but we can add their (X, Y, Z) coordinates, which we obtain by applying Equation (5.6). This yields the equality

$$
\begin{bmatrix}
x_W/y_W \\
1 \\
z_W/y_W
\end{bmatrix}
=
\begin{bmatrix}
x_R/y_R & x_G/y_G & x_B/y_B \\
1 & 1 & 1 \\
z_R/y_R & z_G/y_G & z_B/y_B
\end{bmatrix}
\begin{bmatrix}
Y_R \\
Y_G \\
Y_B
\end{bmatrix},
\tag{5.9}
$$

from which we can solve for the luminances by inverting the 3×3 matrix to obtain

$$
\begin{aligned}
Y_R &= 0.212639 \\
Y_G &= 0.715169 \\
Y_B &= 0.072192.
\end{aligned}
\tag{5.10}
$$

These values tell us how much of the primaries to mix together in order to produce what is considered white light in the sRGB color space.

In order to find a 3×3 matrix $\mathbf{M}_{\text{sRGB}}$ that transforms any color from XYZ space into sRGB space, we can enforce the requirement that each primary, having the chromaticity given by Equation (5.8) and the luminance given by Equation (5.10), maps to a color (R, G, B) in which the component corresponding to the primary is one and the other two components are zero. This can be written as the matrix equation

$$\mathbf{M}_{\text{sRGB}} \begin{bmatrix} \dfrac{x_R}{y_R} Y_R & \dfrac{x_G}{y_G} Y_G & \dfrac{x_B}{y_B} Y_B \\[2ex] Y_R & Y_G & Y_B \\[2ex] \dfrac{z_R}{y_R} Y_R & \dfrac{z_G}{y_G} Y_G & \dfrac{z_B}{y_B} Y_B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{5.11}$$

where the columns of the identity matrix on the right represent the red, green, and blue primaries at full brightness in sRGB space, and the columns of the matrix on the left are the corresponding coordinates in XYZ space. We solve for the conversion matrix $\mathbf{M}_{\text{sRGB}}$ by simply inverting the matrix of primary XYZ coordinates to obtain

$$\mathbf{M}_{\text{sRGB}} = \begin{bmatrix} 3.240970 & -1.537383 & -0.498611 \\ -0.969244 & 1.875968 & 0.041555 \\ 0.055630 & -0.203977 & 1.056972 \end{bmatrix}, \tag{5.12}$$

where each entry has been rounded to six decimal places. As expected, the matrix $\mathbf{M}_{\text{sRGB}}$ maps the D65 white point to the sRGB color $(1,1,1)$, as expressed by the equation

$$\mathbf{M}_{\text{sRGB}} \begin{bmatrix} x_W/y_W \\ 1 \\ z_W/y_W \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}. \tag{5.13}$$

The inverse of $\mathbf{M}_{\text{sRGB}}$ is the matrix of XYZ primaries shown in Equation (5.11), and it provides the conversion in the opposite direction from the sRGB color space back to the XYZ color space. Its entries are given by

$$\mathbf{M}_{\text{sRGB}}^{-1} = \begin{bmatrix} 0.412391 & 0.357584 & 0.180481 \\ 0.212639 & 0.715169 & 0.072192 \\ 0.019331 & 0.119195 & 0.950532 \end{bmatrix}, \tag{5.14}$$

where again, each entry has been rounded to six decimal places. The entries in the second row of $\mathbf{M}_{\text{sRGB}}^{-1}$ are the luminances Y_R, Y_G, and Y_B given by Equation (5.10), and they are the coefficients by which the components of a color (R, G, B) are multiplied to calculate that color's luminance Y, as expressed by

$$Y = 0.212639R + 0.715169G + 0.072192B. \qquad (5.15)$$

This is the formula used to convert a color image into a grayscale image containing only luminance information. The three different coefficients represent the relative apparent brightness of each primary based on the luminosity function, and they sum to one because the RGB color $(1, 1, 1)$, which is white at full intensity, must have a luminance of one.

The shader programs that render an image typically perform calculations involving colors by treating each color as a 3D vector whose x, y, and z components represent intensities of the standard red, green, and blue primaries. Two colors can be added together componentwise just like vectors, and this is a perfectly valid operation due to Grassmann's law. It is also valid to multiply a color by a scalar in the same way as a vector in order to increase or decrease its brightness. These two operations can be applied to full-intensity red, green, and blue colors acting as basis vectors with the values $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$ to produce the RGB color cube shown in Figure 5.7. In the figure, we can see the three faces where one of the components has a constant value of one, and the other two components vary from zero to one. The cube demonstrates how red and green combine to form yellow, green and blue combine to form cyan, blue and red combine to form magenta, and all three primary colors combine to form white. On the back side of the cube, colors on each face have a component with a constant value of zero, and they meet at $(0, 0, 0)$ to form black. The diagonal connecting $(0, 0, 0)$ to $(1, 1, 1)$ contains all of the gray levels between black and white.

There are many situations in which two colors need to be multiplied together. For example, the color of a light may be expressed as one RGB color, and the color that a surface reflects may be expressed as another RGB color. To determine what color of light would reach the viewer, and thus what color the surface appears to have, the light color and reflection color are simply multiplied together. Such a product between colors is calculated by componentwise multiplication, and this usually produces an acceptable approximation to physical reality. Multiplying colors in this way is not generally correct, but we cannot do better without additional information. The proper way to multiply two colors is to calculate the product of their spectra at enough discrete wavelengths to yield accurate results. Continuing the example of reflected light, the spectral power distribution of the emitted light would need to be multiplied by the reflection spectrum of the surface. That product
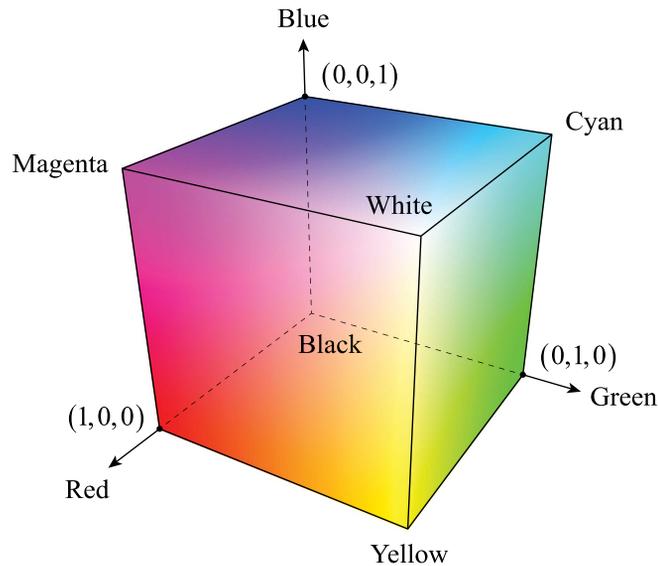
Figure 5.7. The RGB color cube demonstrates how linear combinations of red, green, and blue in varying proportions form all of the colors that can be displayed.

would then need to be multiplied by the red, green, and blue color matching curves for the sRGB color space and integrated to determine the intensities of each component that should be displayed. For reasons that include computational expense and storage requirements, game engines do not typically go to such lengths to produce a physically correct result and simply make do with RGB color in multiplicative contexts.

The definition of a simple data structure holding the components of an RGB color is shown in Listing 5.1. The structure also includes an alpha component, so it is called `ColorRGBA`, and it has floating-point members named `red`, `green`, `blue`, and `alpha` that can be accessed directly. A constructor taking four values can be used to initialize the color, and the fourth value can be omitted to assign a default value of one to the alpha component. As done with the mathematical data structures presented in Volume 1, a default constructor is explicitly included here so that it's possible to declare a color object without performing any initialization. Overloaded operators are provided for multiplying and dividing colors by scalar values and for adding, subtracting, and multiplying colors in a componentwise manner.

Listing 5.1. This is the definition of a simple data structure holding the three components of an RGB color and an alpha value. Overloaded operators are provided for scalar multiplication and division as well as componentwise color addition, subtraction, and multiplication.

```cpp
struct ColorRGBA
{
    float     red, green, blue, alpha;

    ColorRGBA() = default;

    ColorRGBA(float r, float g, float b, float a = 1.0F)
    {
        red = r; green = g; blue = b; alpha = a;
    }

    ColorRGBA& operator *=(float s)
    {
        red *= s; green *= s; blue *= s; alpha *= s;
        return (*this);
    }

    ColorRGBA& operator /=(float s)
    {
        s = 1.0F / s;
        red *= s; green *= s; blue *= s; alpha *= s;
        return (*this);
    }

    ColorRGBA& operator +=(const ColorRGBA& c)
    {
        red += c.red; green += c.green; blue += c.blue; alpha += c.alpha;
    }

    ColorRGBA& operator -=(const ColorRGBA& c)
    {
        red -= c.red; green -= c.green; blue -= c.blue; alpha -= c.alpha;
    }

    ColorRGBA& operator *=(const ColorRGBA& c)
    {
        red *= c.red; green *= c.green; blue *= c.blue; alpha *= c.alpha;
    }
};

inline ColorRGBA operator *(const ColorRGBA& c, float s)
{
    return (ColorRGBA(c.red * s, c.green * s, c.blue * s, c.alpha * s));
}

inline ColorRGBA operator /(const ColorRGBA& c, float s)
```

```
{
    s = 1.0F / s;
    return (ColorRGBA(c.red * s, c.green * s, c.blue * s, c.alpha * s));
}

inline ColorRGBA operator +(const ColorRGBA& a, const ColorRGBA& b)
{
    return (ColorRGBA(a.red + b.red, a.green + b.green,
                      a.blue + b.blue, a.alpha + b.alpha));
}

inline ColorRGBA operator -(const ColorRGBA& a, const ColorRGBA& b)
{
    return (ColorRGBA(a.red - b.red, a.green - b.green,
                      a.blue - b.blue, a.alpha - b.alpha));
}

inline ColorRGBA operator *(const ColorRGBA& a, const ColorRGBA& b)
{
    return (ColorRGBA(a.red * b.red, a.green * b.green,
                      a.blue * b.blue, a.alpha * b.alpha));
}
```

5.3 Gamma Correction

In the days of cathode ray tube (CRT) displays, the brightness of each color channel belonging to a pixel was determined by a nonlinear function of the input signal due to the way in which the electrical circuitry behaved. The displayed brightness V_{display} of a particular input value V_{signal} was given by the relationship

$$V_{\text{display}} = V_{\text{signal}}^{\gamma}, \tag{5.16}$$

where the exponent γ was typically in the range 2.0 to 2.5. This function is called a *gamma curve* after the Greek letter used for the exponent, and although not necessary from an engineering standpoint, newer display technologies intentionally maintain this relationship for the sake of consistency.

   The existence of the gamma curve means that the final colors calculated when a world is rendered should not be directly displayed because most intensities will appear too dark. This is demonstrated in Figure 5.8, where a linear ramp of grayscale intensities is shown as it would appear on a normal display with a gamma value of about 2.2. Due to Equation (5.16), the intensity value of 0.5 has an actual brightness of about 22% when displayed, much less than the 50% brightness that was intended. To compensate for this effect, *gamma correction* must be applied to

Linear values

0.0   0.25   0.5   0.75   1.0

Gamma corrected values
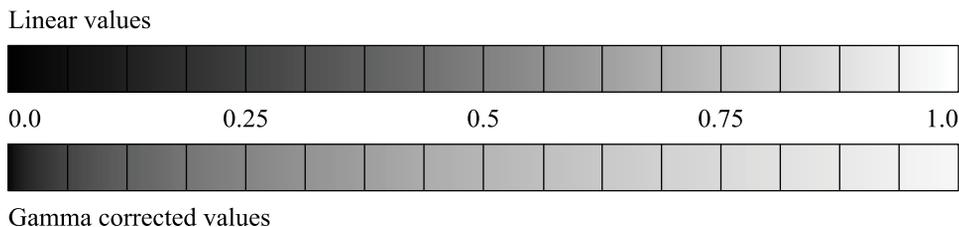
Figure 5.8. These two grayscale ramps illustrate the difference between linear intensity values and gamma corrected intensity values as they would be displayed on a monitor.

color values before they are stored in the frame buffer. An intensity value is gamma corrected by raising it to the power $1/\gamma$ to perform the inverse operation of Equation (5.16). The result is demonstrated by the second grayscale ramp in Figure 5.8, for which the correct brightness is displayed at all intensity values.

When a color is described as being encoded as sRGB, it usually implies that the intensity of each component has been gamma corrected and is stored as a non-linear value, and we will use the term *sRGB* for that exact meaning. A color that has not been gamma corrected is described simply as *linear*. (Both types of colors make use of the same set of sRGB primaries.) The sRGB standard defines conversion functions between linear components and sRGB components that are a little different from the relationship in Equation (5.16) in order to avoid an infinite derivative near zero. Each component c of a linear color is gamma corrected to convert it to a component of an sRGB color using the function f_{sRGB} defined by

$$f_{\text{sRGB}}(c) = \begin{cases} 12.92c, & \text{if } c \le 0.0031308; \\ 1.055c^{1/2.4} - 0.055, & \text{if } c > 0.0031308. \end{cases} \tag{5.17}$$

In the reverse direction, each component c of an sRGB color is converted to a component of a linear color using the function f_{linear} defined by

$$f_{\text{linear}}(c) = \begin{cases} \dfrac{c}{12.92}, & \text{if } c \le 0.04045; \\ \left(\dfrac{c + 0.055}{1.055}\right)^{2.4}, & \text{if } c > 0.04045. \end{cases} \tag{5.18}$$

These two functions are plotted in Figure 5.9, and they are exact inverses of each other. Even though an exponent of 2.4 is used by these conversion functions, the overall shape of f_{linear} closely approximates Equation (5.16) with the gamma value of 2.2 used by most displays, and the function $f(c) = c^{2.2}$ is shown in the figure for comparison. What this means is that the function f_{sRGB} is very close to the inverse of the gamma curve applied by the display, so the gamma correction applied by f_{sRGB} causes the displayed intensities to appear linear.

Gamma correction has an additional benefit when images are stored in a low-precision format such as one that uses eight bits per channel. The human eye is able to detect smaller differences in brightness at low intensities than it can at high intensities. Therefore, if intensities are stored in a linear manner, then too many discrete values are dedicated to bright intensities that cannot be distinguished from each other, and too few discrete values are dedicated to dim intensities where distinguishing adjacent values is easy. Applying gamma correction redistributes the 256 available values so that fewer of them correspond to bright intensities, and more of them correspond to dim intensities. This allows greater precision at the intensity levels where the eye is more discerning. It applies to images that are authored directly in the gamma-corrected space or images stored at a higher bit depth
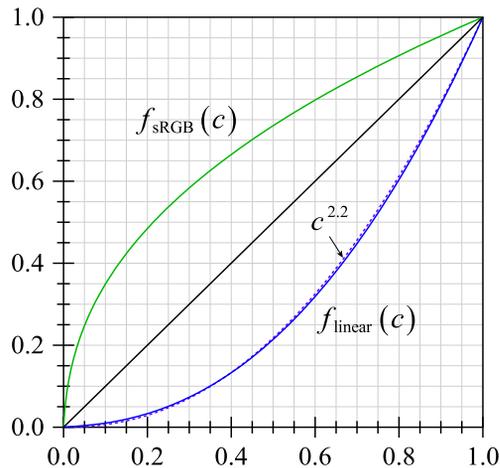
Figure 5.9. The function f_{sRGB}, shown as the green curve, converts linear colors to sRGB colors by performing gamma correction. The function f_{linear}, shown as the blue curve, is the inverse function that converts sRGB colors to linear colors by removing gamma correction. The dashed purple curve is the 2.2 gamma curve applied by the display, which is approximately the same as f_{linear}, causing gamma corrected colors to appear to be linear.

that have gamma correction applied before they are converted to the lower precision of eight bits per channel. For this reason, gamma correction is sometimes called *gamma compression* because some of the higher precision is retained by the redistribution of intensity values.

Most digital image formats, such as JPEG and PNG, store colors with gamma-corrected intensities in the sRGB color space by default. However, rendering calculations that involve lighting, reflections, and atmospheric effects all need to occur in a linear space. The final colors then need to be gamma corrected before they can be displayed. To make all of this easier, the graphics hardware is capable of automatically performing the conversions given by Equations (5.17) and (5.18) at the appropriate times as discussed at various points in this book.

5.4 World Structure

It is common for a complete game world to contain many thousands or even millions of objects. These objects are generically referred to as *nodes*, and they can often be categorized into a diverse group of specific types based on their functions within the virtual environment, as described by the following examples.

- A geometry node, usually made up of triangle meshes, represents a solid object that can participate in both rendering and collision detection.
- A light node represents one of several different kinds of light sources.
- A camera node represents the point of view from which the world is rendered.
- An effect node represents any type of visual effect that isn't necessarily solid geometry, including fire, particle systems, lightning bolt, laser beams, scorch marks, and light shafts.
- A trigger node represents a volume of space that could initiate a scripted sequence of events when some other object, like a character, passes through it.
- A shape node represents a simplified component of a rigid body controlled by the physics simulation.

Most nodes are simultaneously members of a number of independent data structures serving different purposes. As discussed below in this section, each node usually belongs to a transform hierarchy that establishes local coordinate systems. As discussed later in Chapter 9, geometry nodes may also belong to some kind of spatial organization structure used by the game engine to efficiently determine what subset of nodes contribute to the final rendered image. Volume 4 will discuss how nodes representing rigid bodies are often stored in additional data structures used by the physics simulation. All of these ways of organizing the world exist so

that it is possible for a game engine to finish the many tasks it needs to perform within the short time available during each frame.

5.4.1 Coordinate Spaces

Game engines make use of several different coordinate systems throughout the rendering process. In particular, each type of node has a local coordinate system called *object space* that has the same configuration for every node of that type. The origin and axis directions in object space are purposely chosen to be a natural fit for the geometry of the object so they are easy to work with. For example, the object-space origin o_{object} for every box node could be located at a specific corner of the box, and the coordinate axes would naturally be aligned parallel to its edges, as shown in Figure 5.10. In the case of a light node, the object-space origin o_{light} would ordinarily coincide with some central point of emission, and one of the coordinate axes would be aligned to the primary direction in which light is emitted, if any. Similarly, the object-space origin o_{camera} for a camera node would usually be located at its center of projection, and one of the coordinate axes would be aligned to the viewing direction, as done with the camera's z axis in Figure 5.10.
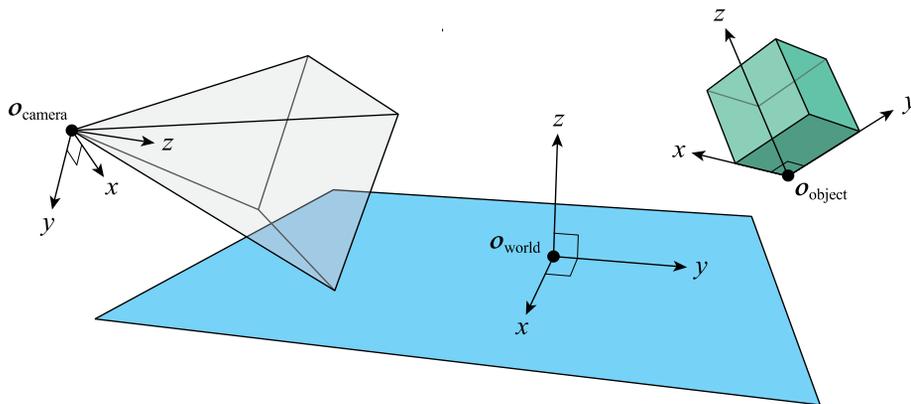
Figure 5.10. Every node has a local coordinate system called object space in which the origin and axes have a natural configuration. For the green box, the object-space origin o_{object} coincides with one of the box's corners, and the axes are parallel to the box's edges. For the camera on the left side of the figure, the object-space origin o_{camera} coincides with the center of projection, and the z axis points in the direction the camera is looking. Objects are positioned and oriented relative to a fixed global coordinate system called world space having the origin o_{world}.

When talking about lights, cameras, or other specific types of nodes, we frequently refer to the object space associated with those nodes using less generic terms such as *light space* or *camera space* instead of referring to them as the object space of the light or the object space of the camera. The shorter terms concisely identify the pertinent nodes and remove possible ambiguities in contexts involving multiple object-space coordinate systems.

Camera space is an especially important coordinate system because it represents the orientation of the viewer. As described in Section 5.4.3 below, vertices are ultimately transformed into coordinate systems that are aligned to camera space as the objects they compose are rendered. When it comes to the coordinate axes in camera space, everybody seems to agree that the x axis points to the right, that the y axis is perpendicular to the viewing direction, and that the z axis is parallel to the viewing direction. However, different rendering systems use different conventions for whether the y axis points upward or downward and whether the z axis points directly into the visible scene or directly away from it. The choice of axis configuration determines whether camera space has a right-handed or left-handed coordinate system. In this book, we choose to have the y axis point downward and to have the z axis point into the scene, as shown in Figure 5.10. This creates a right-handed coordinate system in which all three axes point in the directions of increasing viewport coordinates. The inherent consistency also avoids some negations that are easy to forget, thus lessening the difficulty of implementation.

There are countless situations in which two or more nodes must interact in some way. For example, whenever a camera can see an object that needs to be rendered, whenever a light illuminates an object, and whenever two physical objects collide, they are interacting. In order to perform meaningful calculations involving more than one node, we must be able to establish the spatial relationships among them. This is done by introducing a global coordinate system called *world space* and giving every node a 4×4 transformation matrix $\mathbf{M}$ that represents its position and orientation in that space. The matrix $\mathbf{M}$ transforms vectors and points from a node's object-space coordinate system into the world-space coordinate system. The first three columns of $\mathbf{M}$ correspond to the world-space directions in which the node's object-space coordinate axes point, and the fourth column of $\mathbf{M}$ corresponds to the world-space position of the node's object-space origin. World space is fixed, and as nodes move around the world, their transformation matrices are updated to reflect their new positions and orientations. When an interaction occurs between two nodes A and B, any geometric calculations involved must be carried out in a common coordinate system. This can be accomplished by transforming all vector-like quantities belonging to both nodes into world space using their transformation matrices $\mathbf{M}_A$ and $\mathbf{M}_B$ or by transforming them from the object

space of one node into the object space of the other node using one of the matrix product $\mathbf{M}_A^{-1}\mathbf{M}_B$ or $\mathbf{M}_B^{-1}\mathbf{M}_A$.

One common choice for world space is shown in Figure 5.10. Here, the x and y axes lie in the horizontal plane, and the z axis points upward, creating a right-handed coordinate system. This is the definition of world space used by many game engines and modeling applications, and it's the one we prefer to use in this book. Its configuration is a natural fit for environments in which the ground exists and objects such as characters and vehicles are able to move in any horizontal direction. (Having a z axis that points upward from a surface is also consistent with a coordinate system called *tangent space*, which used extensively in Chapter 7.) The x axis typically points to the east, and the y axis typically points to the north, just as they would on a 2D map of the world. Changes to the direction in which an object is facing, which would usually determine its direction of motion, generally occur in the x-y plane without involving the third dimension along the z axis, and this makes it easy to understand what transformations need to be made when an object makes a turn.

Another common choice for world space aligns the y axis with the up direction and places the x and z axes in the horizontal plane. Some game engines and modeling applications use this definition of world space by default. Its configuration may be preferable for environments in which most movement occurs in a plane perpendicular to the ground, and horizontal motion is restricted to a single set of forward and backward directions. Where it makes sense to talk about directions on a map, the x axis typically points to the east just as it does in a z-up configuration. To form a right-handed coordinate system with the y axis pointing upward, the z axis must now point to the south.

When there is a mismatch between the world-space conventions used by two different applications, it can cause problems when moving geometric data between them. This is particularly true when objects are created in a modeling application that uses one up direction, and those objects need to be imported into a game engine that uses the other up direction. Fortunately, it is easy to convert from the y-up convention to the z-up convention or the other way around. Any 4D homogeneous vector can be transformed from world space with the y axis pointing upward to world space with the z axis pointing upward using the 4×4 matrix

$$\mathbf{M}_{z\text{-up}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{5.19}$$

When applied to a homogeneous point $\mathbf{p}$, which is just a direction vector if $p_w = 0$, the transformed point $\mathbf{p}'$ can be written as

$$\mathbf{p}' = \mathbf{M}_{z\text{-up}}\mathbf{p} = (p_x, -p_z, p_y, p_w). \qquad (5.20)$$

The reverse transformation from world space with the z axis pointing upward to world space with the y axis pointing upward is given by

$$\mathbf{M}_{y\text{-up}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad (5.21)$$

which is, of course, the inverse of $\mathbf{M}_{z\text{-up}}$. Its effect on a homogeneous point $\mathbf{p}$ can be written as

$$\mathbf{p}' = \mathbf{M}_{y\text{-up}}\mathbf{p} = (p_x, p_z, -p_y, p_w). \qquad (5.22)$$

The process of converting a 4×4 matrix $\mathbf{H}$, such as an object-space to world-space transform, between y-up and z-up coordinate systems is slightly more involved. In this case, we must use Equation (2.8) to perform the transformation, which states that for any 4×4 matrix $\mathbf{M}$ that transforms vectors from one coordinate system to another, the matrix $\mathbf{H}$ is transformed into the matrix $\mathbf{MHM}^{-1}$. Since $\mathbf{M}_{z\text{-up}}$ and $\mathbf{M}_{y\text{-up}}$ have very simple forms, we can easily calculate this product and see how the entries of $\mathbf{H}$ are rearranged. If $\mathbf{H}$ is a matrix that transforms vectors into a world space with the y axis pointing upward, then the corresponding matrix $\mathbf{H}'$ that transforms vectors into a world space with the z axis pointing upward is given by

$$\mathbf{H}' = \mathbf{M}_{z\text{-up}}\mathbf{H}\mathbf{M}_{z\text{-up}}^{-1} = \begin{bmatrix} H_{00} & -H_{02} & H_{01} & H_{03} \\ -H_{20} & H_{22} & -H_{21} & -H_{23} \\ H_{10} & -H_{12} & H_{11} & H_{13} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (5.23)$$

Similarly, if $\mathbf{H}$ is a matrix that transforms vectors into a world space with the z axis pointing upward, then the corresponding matrix $\mathbf{H}'$ that transforms vectors into a world space with the y axis pointing upward is given by

$$\mathbf{H}' = \mathbf{M}_{y\text{-up}}\mathbf{H}\mathbf{M}_{y\text{-up}}^{-1} = \begin{bmatrix} H_{00} & H_{02} & -H_{01} & H_{03} \\ H_{20} & H_{22} & -H_{21} & H_{23} \\ -H_{10} & -H_{12} & H_{11} & -H_{13} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \qquad (5.24)$$

It's important to understand that the up directions in world space and camera space are completely independent, and the choice of up direction in one of the two coordinate systems does not determine the up direction in the other. In most games, the player is able to look up and down, and the associated rotation of camera space means that the camera's local up direction could point in any direction, perhaps even parallel to the ground, when transformed into world space. Up directions for each space should usually be chosen in isolation to be whatever makes the most sense without considering other coordinate systems. However, there are sometimes cases in which the camera's rotational transform is highly restricted, and using the same up direction (usually the y axis) in both world space and camera space is justified by the increased ease of remembering how the axes are oriented.

5.4.2 Transform Hierarchy

In a typical game engine, all of the nodes in a world are organized into a *transform hierarchy* that determines their spatial positions and orientations. This hierarchy forms a tree structure in which the root node represents world space. All other nodes are either direct or indirect descendants of the root node and are thus positioned relative to the coordinate system established by the root node. It is often the case that a large number of the nodes making up the environment are attached directly to the root node because they are independent objects that don't form part of a more complex unit. For example, if there was a set of immovable boulders resting on the terrain in an outdoor environment, then each boulder would be independently attached directly to the root node in the transform hierarchy. (Note that the boulders may be separately organized in other data structures like those described in Chapter 9.) The transformation matrices associated with these nodes provide their final positions and orientations in world space. Because so many nodes are immediate children of the root node, the transform hierarchy tends to have a large breadth but, for most branches of the tree, very little depth.

More complex models, especially those that have moving parts, are usually organized into a hierarchy that has a deeper tree structure. Whenever one node should naturally follow the motion of another node but still have some freedom of motion locally, it is attached to the other node in the transform hierarchy and becomes a *subnode*. This gives rise to tree structures like the skeleton shown in Figure 5.11. In this example, a top-level node named "Skeleton" represents the entire character model and would be attached to the root node. The transform of the model's top-level node represents the position and orientation of the whole model in world space. However, the transform of every other node in the model represents its position and orientation relative to its parent node in the hierarchy.
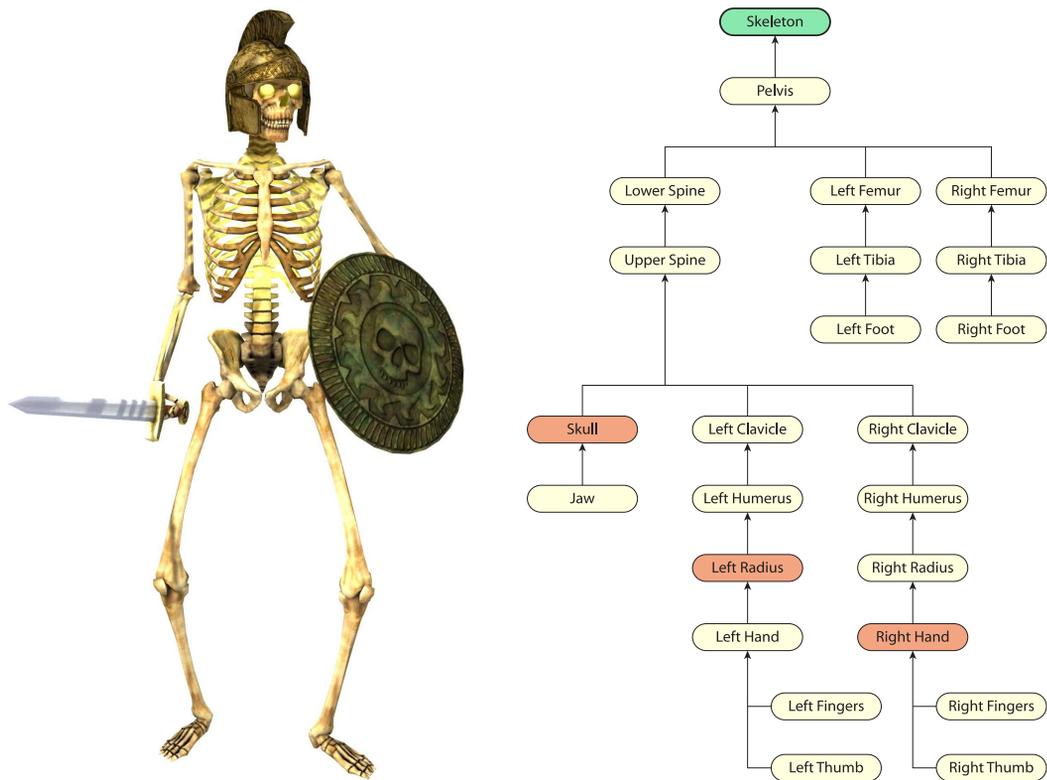
Figure 5.11. This is the node transform hierarchy for a skeleton model. Each node's object transform determines its position and orientation relative to the node it is attached to. In this example, separate models for various weapons, shields, and helmets can be attached to the "Right Hand", "Left Radius", and "Skull" nodes, respectively, so that they track the movements of those parts of the skeleton model.

To transform from object space to world space for a particular node in the model, we must multiply all of the local transformation matrices belonging to that node and all of its ancestors in the model. For example, the transform $\mathbf{W}_{\text{Skull}}$ from object space to world space for the "Skull" node in the figure is given by

$$\mathbf{W}_{\text{Skull}} = \mathbf{M}_{\text{Skeleton}} \mathbf{M}_{\text{Pelvis}} \mathbf{M}_{\text{Lower Spine}} \mathbf{M}_{\text{Upper Spine}} \mathbf{M}_{\text{Skull}}, \tag{5.25}$$

where each matrix $\mathbf{M}$ is the local transformation matrix for the node in the subscript. To avoid calculating long products like this for every node, the object-to-world transforms can be determined in a top-down order and stored for later

reference whenever the model moves. Assuming the "Skeleton" node is attached directly to the root node of the world, its object-to-world transform $\mathbf{W}_{\text{Skeleton}}$ is simply equal to its local transformation matrix $\mathbf{M}_{\text{Skeleton}}$. Then, the object-to-world transform $\mathbf{W}_{\text{Pelvis}}$ for the "Pelvis" node is given by

$$\mathbf{W}_{\text{Pelvis}} = \mathbf{W}_{\text{Skeleton}}\mathbf{M}_{\text{Pelvis}}, \tag{5.26}$$

the object-to-world transform $\mathbf{W}_{\text{Lower Spine}}$ for the "Lower Spine" node is given by

$$\mathbf{W}_{\text{Lower Spine}} = \mathbf{W}_{\text{Pelvis}}\mathbf{M}_{\text{Lower Spine}}, \tag{5.27}$$

and so on until we reach the leaf nodes of the tree.

Transform hierarchies make it easy to attach models to various nodes belonging to other models. This is done so that separate models can track the movements of the parts they're attached to. For a character like our skeleton, a weapon such as the sword in Figure 5.11 is attached to the right hand by making it a subnode of the "Right Hand" node and giving it a local transformation matrix that puts it in the appropriate relative position and orientation. The sword is intentionally kept as a separate model that is not part of the original skeleton model so it can easily be replaced by different weapon models. Similarly, various shields can be attached to the skeleton by making them subnodes of the "Left Radius" node, and various helmets can be attached by making them subnodes of the "Skull" node. This arrangement forms the basis for a system of interchangeable parts allowing each character model to use many different weapons and each weapon model to be used by many different characters.

5.4.3 Vertex Transformations

When a triangle mesh is rendered by the GPU, the 3D object-space position of each of its vertices must ultimately be transformed into a 2D position on the display device. This is accomplished through a series of transformations that move vertices from one coordinate system to another until they reach the coordinate system of the *viewport*, the rectangular area on the display device into which the image is being rendered. The node containing the triangle mesh knows only its own transformation $\mathbf{M}_{\text{object}}$ from object space to world space. Likewise, the camera through which the world is being viewed knows only its own transformation $\mathbf{M}_{\text{camera}}$ from camera space to world space. We can combine these two matrices to form a new transformation

$$\mathbf{M}_{\text{MV}} = \mathbf{M}_{\text{camera}}^{-1}\mathbf{M}_{\text{object}} \tag{5.28}$$

$\mathbf{M}_{object}$ — — → [**World Space**] — — → $\mathbf{M}_{camera}^{-1}$

$\mathbf{M}_{camera}^{-1}\mathbf{M}_{object}$

[**Object Space**] — — → Model-View Matrix — — → [**Camera Space**]

MVP Matrix → [**Clip Space**]

Projection Matrix

[**Clip Space**] → Perspective Divide

[**Device Space**]

Viewport Transformation
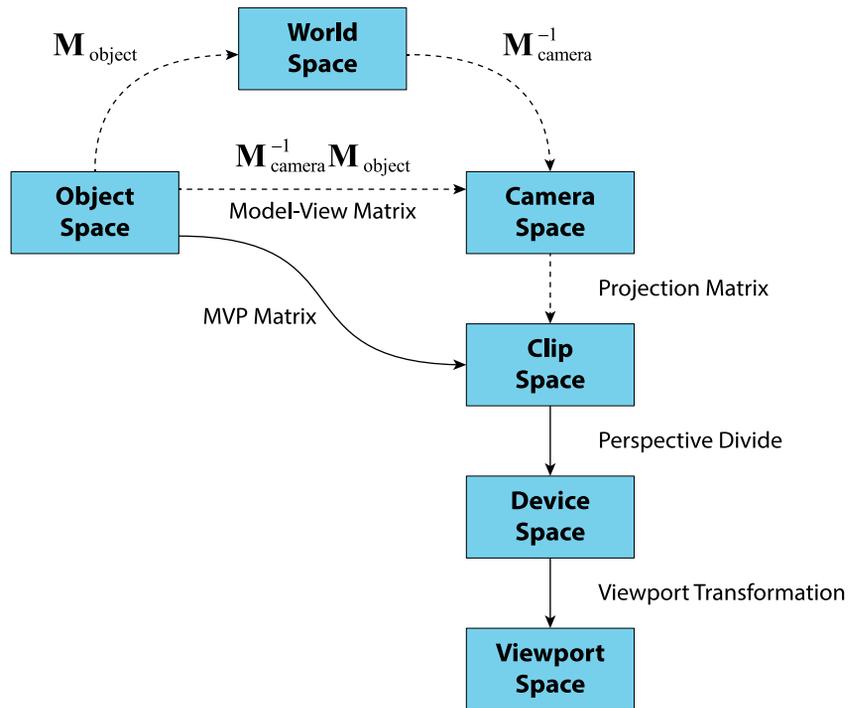
[**Viewport Space**]

Figure 5.12. Vertices originally specified with object-space coordinates undergo a number of transformations that ultimately produce x and y coordinates in a rectangular viewport on the display device and a z coordinate corresponding to depth within the view volume. The model-view-projection (MVP) matrix transforms directly from object space to clip space, bypassing world space and camera space. The perspective divide converts 4D homogeneous coordinates in clip space into 3D coordinates in device space. Finally, the viewport transformation converts normalized device coordinates in device space into viewport coordinates measured in pixels.

called the *model-view* matrix. The model-view matrix transforms vertices directly from object space into camera space, skipping over world space entirely, as illustrated in Figure 5.12.

After vertices have been transformed into camera space, they still require one additional transformation before the GPU can determine their final positions in the viewport. This last transformation is called the *projection*, and it is a topic extensive enough that we dedicate the entirety of Chapter 6 to it. The projection is performed by a 4×4 matrix $\mathbf{P}$ that transforms vertices from camera space into *clip space*, where the vertices then have 4D homogeneous coordinates representing both their

positions in the viewport and their depths along the camera's viewing direction. The model-view matrix is often multiplied by the projection matrix to form the *model-view-projection* matrix

$$\mathbf{M}_{\text{MVP}} = \mathbf{P}\mathbf{M}_{\text{MV}} = \mathbf{P}\mathbf{M}_{\text{camera}}^{-1}\mathbf{M}_{\text{object}},\qquad(5.29)$$

or just *MVP matrix* for short. As shown in Figure 5.12, the MVP matrix transforms vertices directly from object space to clip space, passing over both world space and camera space altogether. As described in the next section, the GPU runs a program for each vertex when a triangle mesh is rendered, and that program normally performs the task of transforming an object-space vertex position into clip space by multiplying it by the MVP matrix.

　　Clip space is so named because it's the coordinate space in which the GPU has enough information to determine whether each triangle extends outside the volume of space visible to the camera. This could mean that a triangle extends beyond the edges of the viewport, or that its depth extends outside the allowed range in front of the camera. Such triangles need to be *clipped* so that only the parts inside the visible volume of space remain. The projection matrix is designed so that a clip-space vertex position $(x_{\text{clip}}, y_{\text{clip}}, z_{\text{clip}}, w_{\text{clip}})$ is inside the visible volume of space when the inequalities

$$-w_{\text{clip}} \leq x_{\text{clip}} \leq w_{\text{clip}}$$
$$-w_{\text{clip}} \leq y_{\text{clip}} \leq w_{\text{clip}}$$
$$0 \leq z_{\text{clip}} \leq w_{\text{clip}}\qquad(5.30)$$

are all satisfied. (Some older graphics libraries use a clip space in which z_{clip} must lie in the range $[-w_{\text{clip}}, w_{\text{clip}}]$ like the x and y coordinates, but there are advantages to using the range $[0, w_{\text{clip}}]$ employed by all newer graphics libraries.) These inequalities correspond to a set of six clipping planes that bound the view volume. Whenever a triangle has a homogeneous vertex position $\mathbf{p}$ satisfying all of the inequalities in Equation (5.30) and another homogeneous vertex position $\mathbf{q}$ that does not satisfy all of them, a new vertex location has to be calculated on the edge connecting $\mathbf{p}$ and $\mathbf{q}$ where it crosses the bounding plane between them. For example, suppose that $q_x > q_w$, which means that the vertex $\mathbf{q}$ lies beyond the right side of the viewport. We can calculate a parameter t along the line segment from $\mathbf{p}$ to $\mathbf{q}$ telling us where the new vertex should be located by solving the equation

$$p_x + t(q_x - p_x) = p_w + t(q_w - p_w).\qquad(5.31)$$

The value of t is given by

$$t = \frac{p_x - p_w}{(p_x - p_w) - (q_x - q_w)}, \tag{5.32}$$

and the location of the new vertex is thus $\mathbf{p} + t(\mathbf{q} - \mathbf{p})$, where the interpolation is carried out for all four homogeneous coordinates.

Once clipping has been performed by the GPU, vertices are transformed from four-dimensional clip space into three-dimensional *normalized device coordinates*, or *device space* for short, by dividing each vertex position by its w coordinate. This operation is called the *perspective divide* because it is ultimately what's responsible for making objects far away from the camera appear to be smaller in the viewport than objects closer to the camera, as discussed in Chapter 6. In device space, the coordinates $(x_{\text{device}}, y_{\text{device}}, z_{\text{device}})$ of a vertex position inside the visible volume of space satisfy the inequalities

$$-1 \le x_{\text{device}} \le 1$$
$$-1 \le y_{\text{device}} \le 1$$
$$0 \le z_{\text{device}} \le 1 \tag{5.33}$$

regardless of the dimensions of the viewport, so the view volume is normalized to a $2 \times 2 \times 1$ box.

Vertices are finally transformed into *viewport space* (also known as *window coordinates* or *screen coordinates*) on the GPU by remapping the device-space x and y coordinates to the ranges $[0, w]$ and $[0, h]$, where w and h are the width and height of the viewport, in pixels. The device-space z coordinate is remapped to the range $[d_{\min}, d_{\max}]$ representing the minimum and maximum depths of the camera's view volume. The viewport transformation, shown as the final step in Figure 5.12, is thus given by the equations

$$x_{\text{viewport}} = \frac{w}{2}(x_{\text{device}} + 1)$$

$$y_{\text{viewport}} = \frac{h}{2}(y_{\text{device}} + 1)$$

$$z_{\text{viewport}} = (d_{\max} - d_{\min})z_{\text{device}} + d_{\min}. \tag{5.34}$$

(It is ordinarily the case that $d_{\min} = 0$ and $d_{\max} = 1$, in which case z_{viewport} has the same value as z_{device}.) Once a triangle has been clipped and its vertices have been transformed into viewport space, the GPU identifies the pixels belonging to the interior of the triangle and runs a program for each one of them to determine the colors they are filled with. This process is described in the next section.

5.5 The Graphics Pipeline

When the GPU is instructed to draw an object, a complex sequence of operations is launched on the graphics hardware. Vertex and triangle data defining the object's geometry flows through various functional units where it is transformed into the final colors that appear on the screen. These stages of transformation are collectively called the *graphics pipeline* or *rendering pipeline*, and all of the major components are laid out in Figure 5.13.

The stages of the graphics pipeline can be classified into two general types, *fixed-function* stages and *programmable* stages. The fixed-function stages correspond to specialized hardware units that each perform a specific task. These stages are all configurable to some degree, but they generally carry out transformations in the same way every time they are utilized. The programmable stages, on the other hand, are capable of running arbitrary programs called *shaders* to carry out their designated transformations. The term "shader" historically referred to a small program that exclusively performed lighting and shading calculations as a surface was rendered, but the scope and usefulness of that type of program has grown well beyond its original purpose. Now, a "shader" can mean any program that runs on the GPU regardless of what kind of calculations it carries out.

There are several different kinds of shaders that each serve a particular function in the graphics pipeline, and most of them operate on vertex and triangle geometry in some way before projection and clipping occur. The earliest programmable GPUs supported only vertex shaders and pixel shaders, and those two are the only programmable stages of the graphics pipeline that are required to be active. The optional geometry shader, hull shader, and domain shader stages were added to later GPUs, and they are active when some kind of geometry amplification is being performed. The output ultimately produced by the active geometry processing stages is passed to the fixed-function rasterization stage, where triangles are broken up into the individual pixels that they cover in the viewport. After this point, operations are performed at the per-pixel level by the pixel shader and the frame buffer operations. This section reviews what happens in each of these components of the graphics pipeline to the extent necessary to follow the rest of the book.

5.5.1 Geometry Processing

At a fundamental level, the graphics hardware is able to render three types of geometric *primitives*: points, lines, and triangles. Naturally, a point is defined by one vertex, a line is defined by two vertices, and a triangle is defined by three vertices. The vast majority of objects rendered by a game engine are composed of triangles,
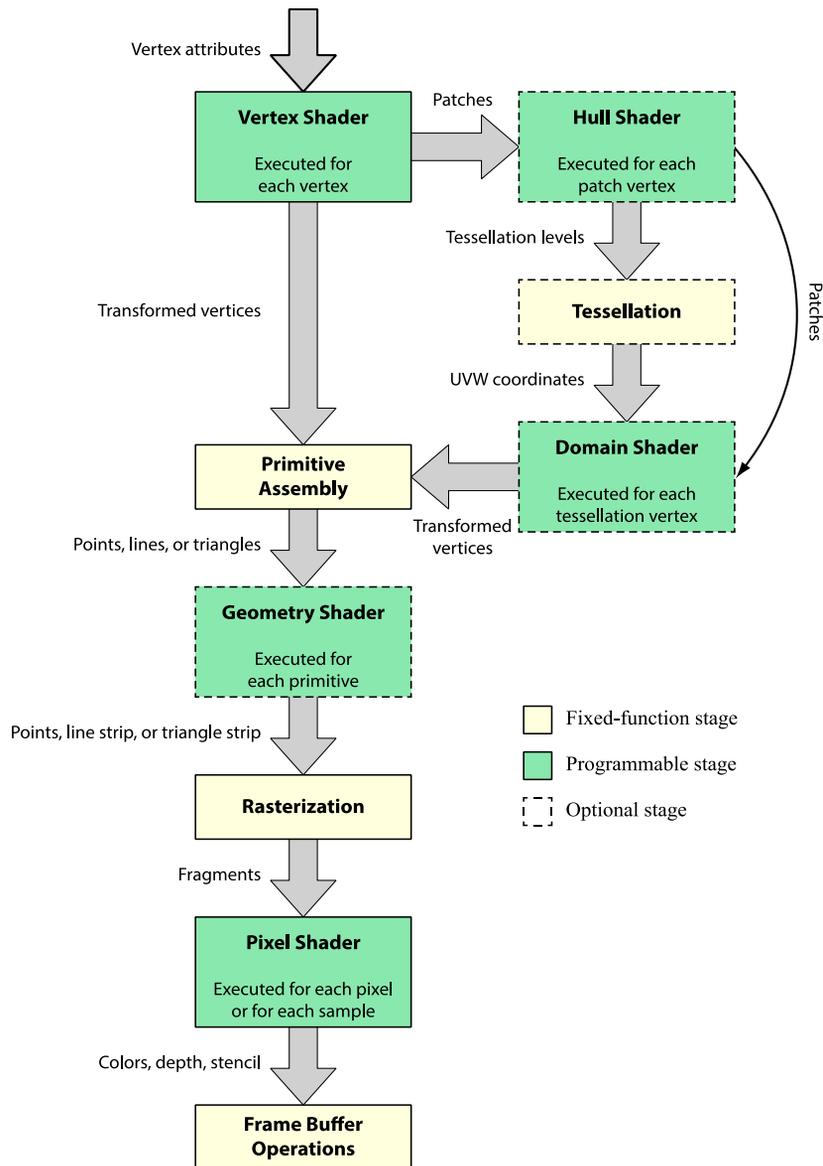
Figure 5.13. The graphics pipeline consists of several programmable (green) and fixed-function (yellow) stages. The pipeline begins where vertex attributes are consumed by the vertex shader and ends where output colors, and possibly depth and stencil data, are written to the frame buffer. All of the stages preceding rasterization pertain to geometry processing, and the two stages following rasterization operate on individual pixels.

but points and lines appear in special cases. For example, a particle system might initially be submitted for drawing as a list of points, but each of those points would then typically be replaced by a pair of triangles in the geometry shader stage. (And those triangles would often be billboards, described in Section 10.2.) With the exception of some discussion of points and lines in this section, we will concentrate exclusively on triangles for all of our rendering needs in this book.

Every object the GPU renders begins its journey through the graphics pipeline as an array of vertices that are fed into the vertex shader stage. The initial geometric structure defined by each set of vertices is specified by configuring the graphics hardware to render one of the topologies listed in Table 5.2. Among these topologies are lists of independent points, lines, and triangles, which are the geometric primitives that we have already mentioned. The topologies also include line strips, triangle strips, and triangle fans that each represent a connected sequence of primitives that share vertices between each pair of lines or triangles. The final topology, a list of patches, is the one specified when tessellation is active.

As discussed in Section 3.1, each vertex in a triangle mesh is usually shared among several triangles. It would be possible to render a mesh containing m triangles by independently storing all three vertices for each triangle in the vertex array and telling the hardware to draw a triangle list with $3m$ vertices. However, this would not only require a lot of vertex data to be duplicated, but it would cause many identical vertices to be processed by the vertex shader to obtain identical results. To avoid this wastefulness, GPUs allow a layer of indirection called an *index array* to be utilized during a drawing operation. The index array is a list of integer values that reference vertices by their positions within the vertex array. Instead of rendering each triangle with three consecutive elements in the vertex array, we render a triangle with three consecutive elements in the index array, and those indices refer to three arbitrary elements in the vertex array. Furthermore, when the vertex at a specific index is processed, the graphics hardware is able to save the outputs of the vertex shader in dedicated on-chip storage called the *post-transform cache*. This allows the outputs to be reused without running the vertex shader again if the same index appears in another triangle soon afterward. The `Triangle` data structure shown in Listing 5.2 provides a convenient way to store the three vertex indices used by each triangle.

The vertex array containing an object's geometry is composed of one or more *vertex attributes*, and each of those attributes can have between one and four numerical components. At a minimum, there is one attribute that holds the position of each vertex, typically as a 3D vector of object-space x, y, and z coordinates. Most of the time, the vertex array contains several more attributes holding additional per-vertex information. It is very common for a vertex array to contain 3D

Mode	Example	Description
Point List		A list of n independent points. Point k is given by vertex k.
Line List		A list of $n/2$ independent lines. Line k is composed of the vertices $\{2k, 2k+1\}$.
Line Strip		A strip of $n-1$ connected lines. Line k is composed of the vertices $\{k, k+1\}$.
Triangle List		A list of $n/3$ independent triangles. Triangle k is composed of the vertices $\{3k, 3k+1, 3k+2\}$.
Triangle Strip		A strip of $n-2$ connected triangles. Triangle k is composed of the vertices $\{k, k+1, k+2\}$ when k is even and the vertices $\{k, k+2, k+1\}$ when k is odd, wound in those orders.
Triangle Fan		A fan of $n-2$ connected triangles. Triangle k is composed of the vertices $\{0, k+1, k+2\}$.
Patch List		A list of independent patches. Each patch is composed of a set of control points used in the tessellation stages.
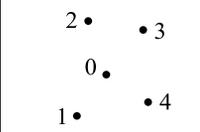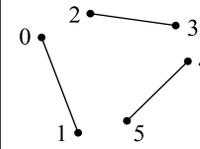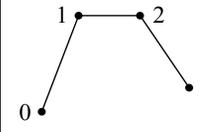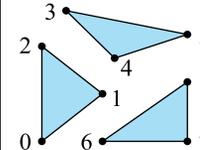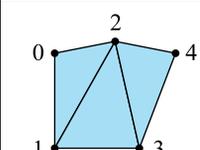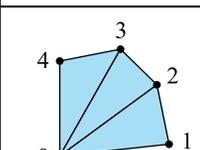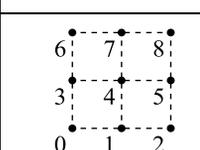
Table 5.2. When a set of n vertices is passed to the graphics hardware for rendering, one of these topologies specifies how they are initially assembled into primitive geometries that the GPU is able to draw.

Listing 5.2. The `Triangle` data structure holds the three indices that reference the vertices making up a single triangle, and they are stored as 16-bit unsigned integers.

```
struct Triangle
{
    uint16    vertexIndex[3];
};
```

normal vectors, 3D tangent vectors, and one or more sets of 2D texture coordinates to facilitate the shading calculations discussed in Chapter 7. Other attributes may specify per-vertex colors, data needed for mesh deformation, or any kind of custom information needed by the shaders. As shown in Figure 5.14, all of the attributes are usually interleaved in memory so that the data for each individual vertex is stored as a contiguous structure. Organizing the data in this way is good for cache performance while the GPU fetches the attributes for each vertex. In this example, every vertex has four attributes comprising eleven 32-bit floating-point values. The offset from the beginning of one vertex's data to the beginning of the next vertex's data is called the *stride* of the vertex array, and it is 44 bytes in this case.

GPUs are able to fetch values stored in many different formats and automatically convert them to floating-point numbers before they are used in a shader. When the full range or precision of a 32-bit floating-point value is not necessary, attributes can be stored in a format that occupies less space. This is often done for normal vectors and tangent vectors because they ordinarily have unit length and thus don't require a large range. It would be perfectly reasonable to store their components in a 16-bit signed normalized integer format. Colors often require even less precision and are typically stored as four 8-bit unsigned normalized integer values with gamma correction.

Vertex positions must be sent to the rasterization stage of the graphics pipeline in clip-space coordinates. This means that the last active shader stage before that point is responsible for transforming the object-space vertex positions into clip space using the MVP matrix. The shader code for this simple 4D matrix-vector multiplication is shown in Listing 5.3. In this example, we have specified the MVP matrix as an array of four 4D vectors representing its rows to avoid the ambiguity surrounding the storage order of matrix entries. If the vertex shader is the only enabled geometry processing stage, which is often the case, then this transformation would take place there. Otherwise, it could take place in either the geometry shader or the domain shader.

The final geometry processing shader can also pass vertex attributes straight through to the rasterization stage without modification. This is often done with texture coordinates. Other vertex attributes may undergo different transformations

such as a change from object-space to world-space coordinates before they are output by the shader. Additional information such as camera and light positions are commonly output in a specific coordinate space as well. All of these attributes, specified for each vertex, are later interpolated inside each triangle to provide smoothly varying values to the pixel shader.
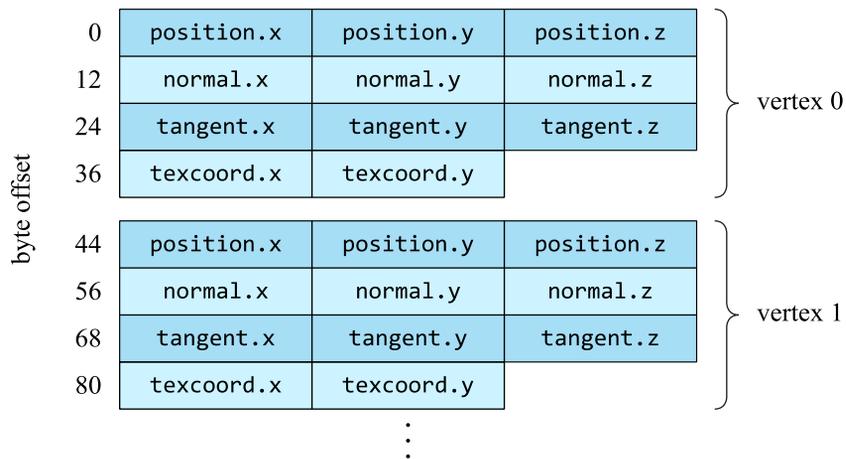
Figure 5.14. Vertex attributes are typically interleaved as shown. In this example, there are four attributes comprising a 3D position, a 3D normal vector, a 3D tangent vector, and one set of 2D texture coordinates. Each of the components are stored as a 32-bit floating-point value. The stride from one vertex to the next is the total number of bytes occupied by the attribute data for each vertex, which is 44 bytes in this case.

Listing 5.3. This vertex shader code transforms the 3D object-space vertex coordinates given by `position` into 4D clip-space coordinates using the MVP matrix. The positions are treated as points with an implicit w coordinate of one. Each entry of the `mvp` array holds one row of the MVP matrix.

```
uniform float4    mvp[4];

float4 TransformVertex(float3 position)
{
    return (float4(dot(position, mvp[0].xyz) + mvp[0].w,
                   dot(position, mvp[1].xyz) + mvp[1].w,
                   dot(position, mvp[2].xyz) + mvp[2].w,
                   dot(position, mvp[3].xyz) + mvp[3].w));
}
```

5.5.2 Pixel Processing

The fixed-function rasterization stage performs several operations on the primitives defined by the transformed vertices that have been output by the geometry processing stages. Once these operations have been completed, the result is a set of *fragments* that correspond to pixel-sized pieces of the original primitives. The data associated with each fragment includes its viewport-space coordinates and the information necessary to calculate interpolated values for each vertex attribute supplied with the primitive it came from. The pixel shader is executed once for each fragment to calculate a final color that will be written to the frame buffer (or blended with the existing contents of the frame buffer). Because of the close relationship between fragments and pixels, the term *fragment shader* is used by some graphics libraries in place of the term pixel shader.

Clipping

The first thing that happens in the rasterization stage is clipping. Because calculating new vertex positions and attribute values can be expensive, GPUs do not typically test the coordinates x_{clip} and y_{clip} against the exact boundary of the viewport by comparing them to $\pm w_{\text{clip}}$ as in Equation (5.30). Instead, primitives are allowed to extend outside the viewport by a somewhat large distance, and clipping occurs only if either x_{clip} or y_{clip} lies outside the range given by $\pm b w_{\text{clip}}$ for some constant factor b. This extra space surrounding the viewport is called the *guard band*, and it is illustrated in Figure 5.15. Primitives lying completely beyond any one of the four edges of the viewport can be quickly discarded by the GPU. Otherwise, clipping in the x or y direction may be necessary, but it's uncommon due to the large size of the guard band. However, the GPU must still ensure that no fragments are generated outside the viewport, and this is accomplished through the use of what's called the *scissor test*. The GPU has at least one internal scissor rectangle that makes sure rendering stays inside the current render target, and the application is able to configure an externally exposed scissor rectangle to serve any purpose. Fragments are generated only inside the intersection of these scissor rectangles. Establishing a scissor rectangle can thus be used as an optimization to reduce the number of pixels processed inside the viewport without modifying any geometry, and an important case is described in Section 8.2.1.

Face culling

After any necessary clipping has been performed, the perspective divide and viewport transformation are applied to the vertices of each surviving primitive. For triangles, the GPU then optionally performs *face culling*. This is an operation used to
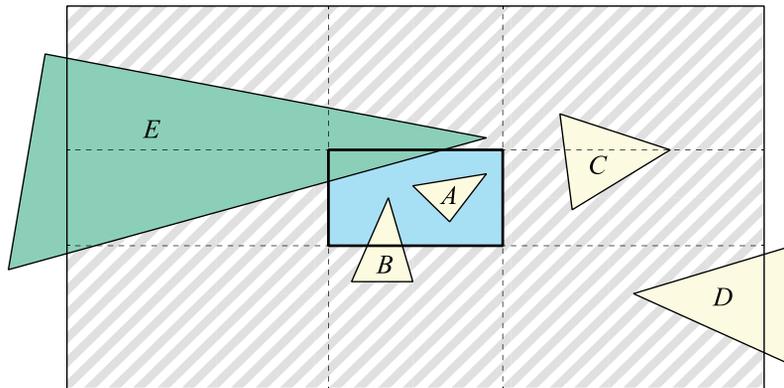
Figure 5.15. The viewport, shown as the blue rectangle in the center, is surrounded by a guard band, shown as the larger hatched region. Triangle A is completely inside the viewport, so it is neither clipped nor scissored. Triangle B extents outside the viewport and is affected by scissoring, but it is not clipped because it lies entirely inside the guard band. Triangles C and D are both discarded because all of their vertices lie beyond the viewport's right edge. Finally, triangle E is clipped because it extends outside the guard band and cannot be discarded based on the positions of its vertices.

quickly eliminate any triangles that are facing away from the camera on the back side of an object where they can't be visible. The determination of whether a triangle is front facing or back facing is based on the signed area of the triangle in viewport space. For a triangle having 2D vertex coordinates (x_0, y_0), (x_1, y_1), and (x_2, y_2) after the viewport transformation, the signed area A is given by

$$A = \frac{1}{2}[(x_2 - x_0)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_0)]. \tag{5.35}$$

This formula produces a positive value when the three vertices are wound in a counterclockwise direction in a viewport space for which the x axis points to the right and the y axis points downward. GPUs allow an application to choose which winding direction (clockwise or counterclockwise) is considered front facing, and either front-facing triangles, back-facing triangles, or neither may be culled during the rasterization stage.

Rasterization

Triangles that remain after the face culling operation has been performed are finally ready to be divided into fragments, and this is where the rasterization stage gets its

name. The mathematically precise edges of each triangle are laid onto the pixel grid, as shown in Figure 5.16, and the triangle's interior is converted to a raster image, or *rasterized*. A pixel is considered to be inside a triangle only if the point at its center is contained within the triangle's three edges. Tie breaking rules are applied to pixels lying exactly on an edge so that no pixel is filled twice by adjacent triangles sharing two vertex positions.

The black dots in Figure 5.16 indicate which pixels are rendered for the specific triangle shown. However, these are not the only pixels for which the pixel shader is actually executed. The GPU generates fragments for all four pixels in every 2×2 block, called a *quad*, having at least one pixel inside the triangle. This means that the pixel shader is also run for every pixel containing a red dot in the figure, but the outputs of those pixel shader invocations are discarded and do not affect the contents of the frame buffer. The reason fragments are generated in 2×2 quads has to do with texture filtering. To select an appropriate mipmap level when a texture map is sampled, the GPU needs to know what the derivatives of the texture coordinates are with respect to the x and y directions. The most effective method of calculating them is to take the differences between the values assigned to the texture coordinates by the pixel shader in adjacent pixels. With the exception of cases involving noncoherent control flow, valid differences are guaranteed to be available when pixel shaders are always executed for 2×2 quads.
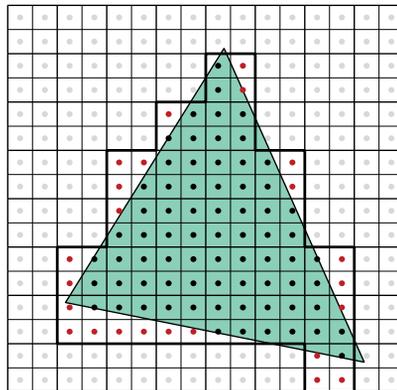
Figure 5.16. A triangle is rasterized by filling the pixels having center points inside the triangle's three edges, indicated by black dots. To allow the calculation of derivatives, the pixel shader is executed for all four pixels belonging to each 2×2 quad containing at least one black dot, illustrated by the black outline containing the full set of such quads. The red dots indicate pixels for which the outputs of the pixel shader are not used.

Because the fragments generated for each triangle always come in 2×2 quads, the pixels belonging to any quad straddling the boundary between adjacent triangles are processed *twice* by the pixel shader. For each pixel, the output is written to the frame buffer only for one of the two triangles, and the output for the other invocation of the pixel shader is thrown out. The graphics hardware can sometimes mitigate the extra expense of running pixel shaders for these "helper" invocations, such as by suppressing unneeded texture fetches, but the cost associated with 2×2 quad rendering is generally something that we just have to live with. For simple pixel shaders, it doesn't make a huge difference, but for more complex pixel shaders, there can be a significant impact on performance, especially when rendering a lot of small triangles. It is therefore considered good practice to eliminate edges or to minimize their lengths through the choice of triangulation where it's practical to do so. (See Exercise 10.)

Rendering a *full-screen pass* is a common case in which a large edge between two triangles can be removed to save a little processing time. A full-screen pass refers to an operation that is intended to affect every pixel in the viewport, and it's often used to apply some kind of computation to the entire visible scene. For example, the ambient occlusion, atmospheric shadowing, and motion blur techniques described in Chapter 10 all make use of full-screen passes. The most straightforward way to cover the whole viewport would be to place vertices at the four corners and render two triangles that form a screen-sized rectangle. However, this would mean that a small amount of time is wasted rendering pixels in the 2×2 quads along the diagonal twice. A better method is to render a single triangle of larger size that contains the entire viewport as illustrated in Figure 5.17. The vertex coordinates in the figure are expressed directly in normalized device coordinates and can be passed through the vertex shader without any transformation by an MVP matrix. Even though two of the vertices lie far outside the viewport, they are still well within a typical guard band, so no clipping occurs. The triangle is simply scissored by the hardware, and every pixel in the viewport is processed exactly one time.

Interpolation

When a pixel shader runs, it has access to the values of a triangle's vertex attributes after they have been interpolated at the pixel's center. As explained in detail in Section 6.2, the attributes must normally be interpolated in a "perspective-correct" manner before they can be used in the pixel shader, and a common way to implement the pertinent calculations makes use of the pixel's *barycentric coordinates* in viewport space. The barycentric coordinates of a point q are a set of three weights by which the vertex positions p_0, p_1, and p_2 can be multiplied so that they sum to
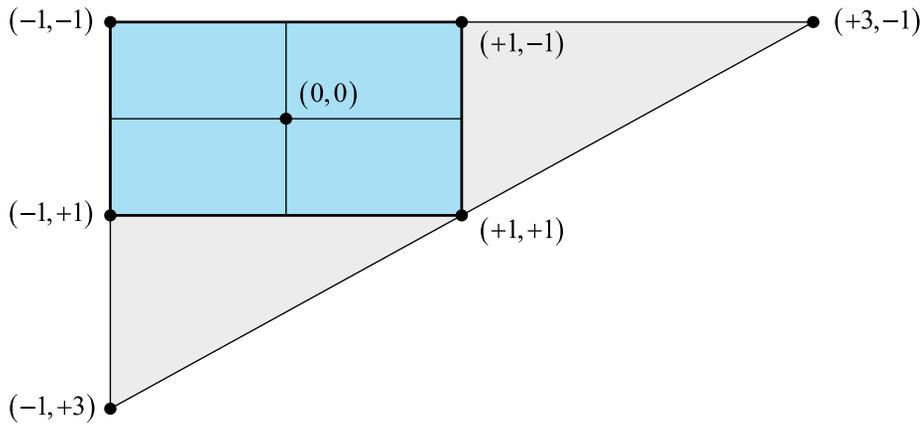
Figure 5.17. A full-screen pass can be rendered using a single triangle by placing two of the vertices outside the viewport. The coordinates shown here are expressed directly in normalized device coordinates, where the viewport itself occupies the space between -1 and $+1$.

the point q. By requiring that the weights themselves sum to one, we need only two independent values, which we call u and v. The point q is then expressed as

$$q = (1-u-v)\, p_0 + u p_1 + v p_2. \tag{5.36}$$

As shown in Figure 5.18, the value of u corresponds to the perpendicular distance between the point q and the edge connecting the vertices p_0 and p_2. This distance is normalized so that u is always in the range $[0,1]$ as long as the point q is inside the triangle. If q lies on the edge connecting p_0 and p_2, then $u = 0$, and if q is coincident with p_1, then $u = 1$. Similarly, the value of v corresponds to the perpendicular distance between the point q and the edge connecting the vertices p_0 and p_1, where a value of one is reached when $q = p_2$.

Barycentric coordinates summing to one can be calculated as a ratio of areas. For example, the value of u in Figure 5.18 is equal to the area of the triangle formed by q, p_0, and p_2 divided by the area of the whole triangle. The areas A_u and A_v of the subtriangles corresponding to u and v can be expressed as

$$A_u = \frac{1}{2}\big[(x_2 - x_0)(q_y - y_0) - (y_2 - y_0)(q_x - x_0)\big]$$

$$A_v = \frac{1}{2}\big[(q_x - x_0)(y_1 - y_0) - (q_y - y_0)(x_1 - x_0)\big]. \tag{5.37}$$

where $\boldsymbol{p}_0 = (x_0, y_0)$, $\boldsymbol{p}_1 = (x_1, y_1)$, and $\boldsymbol{p}_2 = (x_2, y_2)$. When we divide these values by the total area A, given by Equation (5.35), and rewrite in terms of the components of the point $\boldsymbol{q}$, we find that the barycentric coordinates u and v are given by

$$u = \frac{A_u}{A} = \frac{y_0 - y_2}{2A} q_x + \frac{x_2 - x_0}{2A} q_y + \frac{x_0 y_2 - x_2 y_0}{2A}$$

$$v = \frac{A_v}{A} = \frac{y_1 - y_0}{2A} q_x + \frac{x_0 - x_1}{2A} q_y + \frac{x_1 y_0 - x_0 y_1}{2A}. \tag{5.38}$$

These formulas are often called a triangle's *plane equations* because they are equivalent to taking dot products between the point $\boldsymbol{q}$ and planes representing two of the edges, where all z coordinates are zero. The three fractions appearing in each formula, which define the planes, are constants that can be precomputed one time when a triangle is rasterized. Once the planes are known, the values of u and v corresponding to any point $\boldsymbol{q}$ at a pixel center inside the triangle can each be calculated by the GPU with nothing more than a pair of multiply-add operations.
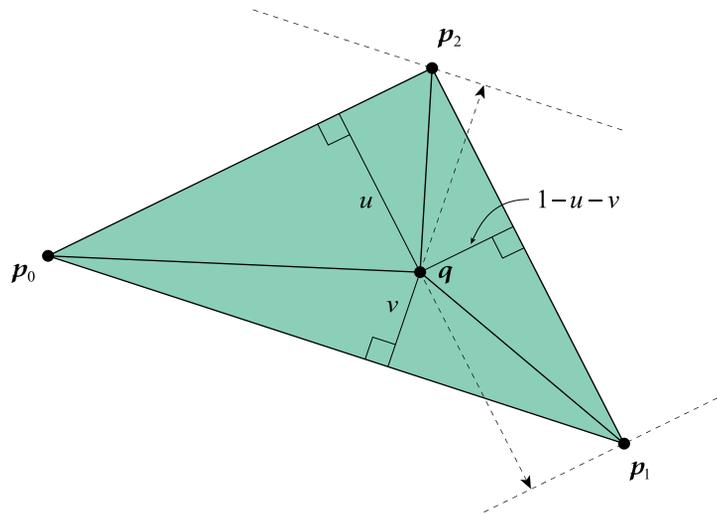
Figure 5.18. The barycentric coordinates of the point $\boldsymbol{q}$ allow it to be expressed as the weighted sum of the vertices $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, and $\boldsymbol{p}_2$. The values of u, v, and $1-u-v$ correspond to perpendicular distances between the triangle's edges and the point $\boldsymbol{q}$. These distances are normalized so that the distance from any one vertex to the edge connecting the other two vertices is always one.

Multisampling

In the rasterization process described earlier, every pixel is considered to be either fully inside a triangle or fully outside. This leads to jagged edges because it doesn't account for the extremely common occurrence that pixels are partially covered by multiple triangles. A simple way to overcome this problem is to use *supersampling*, a technique in which a scene is rendered at a resolution higher than that of the final image to be displayed. After rendering is complete, the larger image is scaled down to the final size, and every pixel in the displayed image is given the average color of several pixels that were originally rendered. It is typical for the larger image to have twice the resolution in both the *x* and *y* directions so that every pixel in the displayed image is an average of a 2×2 block. This method is very effective, but it is also very expensive. Not only are the memory requirements much greater, but the processing requirements are much higher because the number of pixels that need to be shaded has quadrupled.

A method called *multisample antialiasing (MSAA)* strikes a compromise between rendering speed and image quality, and it is widely available on modern graphics hardware. MSAA still uses the larger amount of storage that is required by supersampling, but it does not normally execute the pixel shader any more times than it would have originally. Instead, the pixel shader is executed one time for each pixel at the final display resolution, and the outputs are distributed to multiple *samples* covering that pixel in the higher-resolution storage buffer. Each sample corresponds to a specific location inside the pixel as shown in Figure 5.19. The rasterization rules are modified so that any pixel having at least one sample point inside a triangle, as opposed to only the exact center, are considered part of the triangle. The pixel shader is still executed as if the pixel center is covered, even if that is not true, and vertex attributes are normally interpolated at the pixel center. (However, GPUs provide a *centroid* interpolation mode that changes this in cases when the pixel center is outside the triangle.) The outputs of the pixel shader are
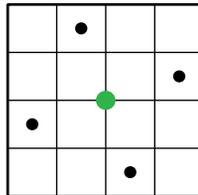
Figure 5.19. Typical sample locations for $4\times$ rotated grid multisampling are shown as black dots inside the single pixel depicted by the outer box. The vertex attributes are interpolated at the pixel center specified by the green dot.

stored in the higher-resolution frame buffer only for the samples that lie inside the triangle. This allows the samples corresponding to a single pixel to hold colors generated by multiple triangles that covered the pixel in different ways during the rendering process. When the high-resolution image is scaled down for display, these colors are averaged together, resulting in a smooth boundary where foreground triangles are drawn on top of background triangles.

5.5.3 Frame Buffer Operations

There are a number of fixed-function operations that are applied to individual pixels before and after the pixel shader runs, and the sequence in which they occur is illustrated in Figure 5.20. The scissor test was described in the previous section, and it is applied first to prevent any further processing for pixels that do not lie inside the scissor rectangle. The depth test, stencil test, and depth bounds test are also capable of removing pixels from the pipeline, and they are described below. The blend operation is applied last to any pixels that make it past all of the preceding tests, and it combines the output of the pixel shader with the contents of the frame buffer.

Depth test

The *depth test* is the mechanism through which the GPU performs hidden surface removal at a per-pixel granularity (or in the case of multisampling, at a per-sample granularity). Higher-level methods that can remove entire occluded objects are discussed in Chapter 9. The concept of the depth test is simple. In addition to a buffer that holds the color of each pixel, storage is allocated for a *depth buffer*, also called a *Z buffer*, that holds the depth of each pixel with respect to the direction the camera
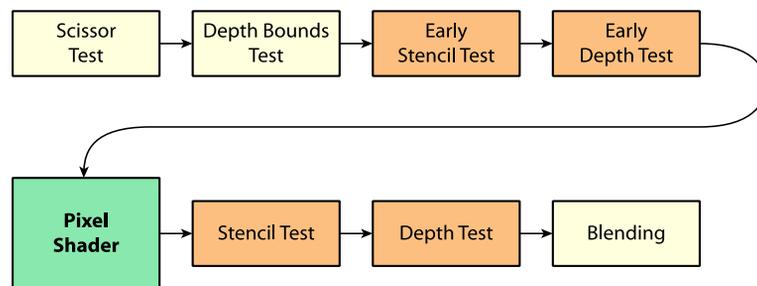
Figure 5.20. Various per-pixel operations are applied before and after the pixel shader runs. Under most circumstances, the depth test and stencil test can be applied *early*, before the pixel shader runs.

is pointing. When a triangle is rendered, the depth of each fragment in the triangle is compared to the depth already stored in the depth buffer at the fragment's location. If the new depth is less than the previous depth, meaning that it's closer to the camera, then the depth test *passes*. In this case, the pixel is rendered, and the depth buffer is updated with the new depth value. Otherwise, the depth test *fails*, meaning that the pixel must lie behind a closer surface that was already rendered. In this case, the pixel is thrown out, and no change is made to either the color buffer or the depth buffer.

As long as the pixel shader doesn't modify the depth assigned to a fragment during rasterization, the depth test can be performed *early*, before the pixel shader runs. If the early depth test fails for all of the non-helper pixels in a 2×2 quad, then the pixel shader can be skipped for the entire quad, saving a significant amount of computation. Most modern hardware takes early rejection even further by using a hierarchical depth buffer. Extra memory is allocated to hold the current maximum depth over all pixels in each tile covering 4×4, 8×8, or larger blocks of pixels. When a triangle is rasterized, the GPU calculates the minimum depth of the triangle for each tile that the triangle intersects. If that minimum depth is still greater than the maximum depth already rendered for the tile, then all of the triangle's fragments within the tile are guaranteed to fail the depth test. The GPU can then safely cull the entire tile at once, avoiding further per-pixel operations.

In order to make most effective use of a hierarchical depth buffer, it is common for a game engine to render a *depth prepass* to establish the contents of the depth buffer by itself without performing any other rendering calculations. Most hardware is able to render exclusively to the depth buffer with extremely high performance. A depth prepass is rendered with the intention of eliminating overdraw in a subsequent pass when complex shaders are executed to determine the colors that are ultimately stored in the frame buffer. Ideally, with the early depth test enabled for the entire scene, a pixel shader is run only one time for each pixel, and it's never the case that pixels belonging to a foreground triangle overwrite pixels previously rendered for a background triangle.

Depth is a quantity that must be interpolated inside a triangle using perspective-correct methods, and the exact calculation is given in Section 6.2. On older graphics hardware, depths were stored as 24-bit integers, but they are now almost always stored as 32-bit floating-point values. For reasons having to do with the nonlinear nature of perspective-correct depths and the precision distribution of floating-point numbers, the traditional method of storing smaller values for closer depths is now often abandoned in favor of a reversed method in which larger values are actually stored for closer depths. The details are discussed in Section 6.3.3.

Because depth is stored with finite precision, triangles intended to be rendered in the same plane generally do not produce the same depth values for their fragments unless they have exactly the same vertex positions. When two triangles lying in coincident planes are rendered, which happens when a "decal" is applied to a surface as described in Section 10.1, the result is an artifact called *Z fighting*. When the second triangle is rendered, some of its fragments will pass the depth test and others will not, and the pattern changes randomly as the camera moves. To eliminate this problem, the hardware includes *polygon offset* functionality that modifies the depth of each fragment. This allows the depths for one triangle to be artificially increased or decreased without changing the vertex positions so that it always appears in front of the other triangle. As discussed in Section 8.3.4, the use of polygon offset is also an important part of shadow mapping.

Stencil test

The *stencil test* is a mechanism through which a mask can be generated and later utilized to control whether rendering occurs at each pixel (or in the case of multisampling, at each sample). An additional buffer, the *stencil buffer*, stores an 8-bit unsigned integer for every pixel, and it is compared to a reference value specified by the application to determine whether the stencil test passes or fails. The stencil test interacts with the depth test, and different operations can be performed on the stencil buffer depending on the outcomes of both the stencil test and the depth test. A nontrivial example of a stencil-based technique that takes advantage of this functionality is the stencil shadow algorithm described in Section 8.4, which uses the stencil test is used to mask off pixels that cannot receive light.

Like the depth test, the stencil test can usually be performed before the pixel shader runs. If the stencil test fails, then the pixel shader is skipped, and no change is made to the color buffer, but the stencil buffer may still be modified. In fact, the stencil test can be configured to perform different operations in three different cases. One operation is performed when the stencil test fails, regardless of the outcome of the depth test. The other two operations are performed when the stencil test passes, and which one depends on whether the depth test passes or fails. The specific operations that can be performed by the vast majority of GPUs are listed in Table 5.3. Some newer GPUs support additional operations and are also able to output a per-pixel reference value from the pixel shader (which would require that the stencil test take place after the pixel shader runs).

The stencil test can be independently configured for pixels belonging to front-facing and back-facing triangles. That is, a triangle mesh can be rendered without face culling enabled, and different stencil operations can be performed depending on whether a fragment was generated from a front-facing or back-facing triangle.

Stencil Operation	Description
Keep	No change is made to the stencil value.
Zero	The stencil value is set to zero.
Replace	The stencil value is set to the reference value.
Invert	The stencil value is inverted bitwise.
Increment and saturate	The stencil value is incremented by one unless its value is 0xFF, in which case no change is made.
Decrement and saturate	The stencil value is decremented by one unless its value is 0x00, in which case no change is made.
Increment and wrap	The stencil value is incremented by one, and a value of 0xFF becomes the value 0x00.
Decrement and wrap	The stencil value is decremented by one, and a value of 0x00 becomes the value 0xFF.

Table 5.3. Most GPUs support these eight stencil operations. Separate operations can be specified based on whether pixels belong to front-facing or back-facing triangles and based on whether they pass both the stencil test and depth test, they pass the stencil test but fail the depth test, or they fail the stencil test.

The stencil shadow algorithm requires that both front and back faces of a shadow volume are rendered, and the ability to specify separate operations allows an entire shadow volume to be rendered with a single command. Without this added configurability, rendering front faces and back faces with different stencil operations would require two commands with an intervening change to the hardware state.

Depth bounds test

The *depth bounds test* is a special function that can be used to eliminate fragments before the pixel shader runs. It works by discarding fragments for which the depth value *already stored* in the depth buffer falls outside an application-specified range of depths in viewport space. The depth of the fragment itself does not matter, so the depth bounds test can always be performed early. If it fails, then the pixel shader and depth test are skipped, and no stencil operations are performed.

The depth bounds test is, in some ways, the analog of the scissor test along the z axis. The difference is that the depth bounds test operates on the depth of geometry that was previously rendered and not on the depth of any fragments currently

being rendered. Due to this property, it only has any practical utility after the contents of the depth buffer have been established. The depth bounds test was originally designed as an optimization for the stencil shadow algorithm, discussed in Section 8.4, which does indeed require a fully rendered depth buffer. A different application, in which the depth bounds test is used to cull pixels not affected by a light source, is described in Section 8.2.2.

Blending

After a fragment has passed all of the above tests and the pixel shader has been executed, the results are finally written to the frame buffer. The color output by the pixel shader is called the *source* color S, and the existing color stored in the frame buffer at the pixel's location is called the *destination* color D. The source color can simply overwrite the destination color, or both colors can be combined through an operation called *blending*. The blending operations supported by most GPUs are shown in Table 5.4. The factors X and Y appearing in the table are weighting factors that can be zero, one, or a variety of values derived from four possible inputs. The source and destination colors themselves constitute two of these inputs. The third possible input is a secondary or "dual" source color T that can be output from the pixel shader for the sole purpose of participating in the blending operation. The fourth possible input is an application-specified constant color C. The ways in which these four colors can be used as weighting factors are listed in Table 5.5.

Blend Operation	Formula	Description
Add	$XS + YD$	The weighted source color is added to the weighted destination color.
Subtract	$XS - YD$	The weighted destination color is subtracted from the weighted source color.
Reverse subtract	$YD - XS$	The weighted source color is subtracted from the weighted destination color.
Minimum	$\min(S, D)$	The minimum of the source color and destination color is selected.
Maximum	$\max(S, D)$	The maximum of the source color and destination color is selected.

Table 5.4. Most GPUs support these five operations for blending the source color S and the destination color D. The formulas are applied componentwise, and the weighting factors X and Y can be any of the values listed in Table 5.5.

Blend Factor	RGBA Value
Zero	$(0,0,0,0)$
One	$(1,1,1,1)$
Source color 0	(S_r, S_g, S_b, S_a)
Source color 1	(T_r, T_g, T_b, T_a)
Destination color	(D_r, D_g, D_b, D_a)
Constant color	(C_r, C_g, C_b, C_a)
Inverse source color 0	$(1-S_r, 1-S_g, 1-S_b, 1-S_a)$
Inverse source color 1	$(1-T_r, 1-T_g, 1-T_b, 1-T_a)$
Inverse destination color	$(1-D_r, 1-D_g, 1-D_b, 1-D_a)$
Inverse constant color	$(1-C_r, 1-C_g, 1-C_b, 1-C_a)$
Source alpha 0	(S_a, S_a, S_a, S_a)
Source alpha 1	(T_a, T_a, T_a, T_a)
Destination alpha	(D_a, D_a, D_a, D_a)
Constant alpha	(C_a, C_a, C_a, C_a)
Inverse source alpha 0	$(1-S_a, 1-S_a, 1-S_a, 1-S_a)$
Inverse source alpha 1	$(1-T_a, 1-T_a, 1-T_a, 1-T_a)$
Inverse destination alpha	$(1-D_a, 1-D_a, 1-D_a, 1-D_a)$
Inverse constant alpha	$(1-C_a, 1-C_a, 1-C_a, 1-C_a)$

Table 5.5. Each of the weighting factors used in the blending operation can be zero, one, or a value derived from the primary source color *S*, the secondary or "dual" source color *T*, the destination color *D*, or a constant color *C*.

One blend operation and pair of blend factors *X* and *Y* can be specified for all three color channels (red, green, and blue), and a separate blend operation and pair of blend factors *X* and *Y* can be specified for the alpha channel. The formulas used to perform blending are applied componentwise.

To simply replace the destination color with the source color, the blend operation is set to add, the factor *X* is set to one, and the factor *Y* is set to zero. It is common for the alpha channel of the source color to contain a measure of opaqueness in which zero is completely invisible, one is completely opaque, and values

in between represent varying levels of transparency. In this case, the source color is blended with the destination color using the formula

$$S_a S + (1 - S_a) D, \tag{5.39}$$

which corresponds to a factor X set to the source alpha and a factor Y set to the inverse source alpha.

If the red, green, and blue components of the destination color D are stored in the frame buffer as gamma-corrected values, then they cannot be used directly in the blending operation because a weighted average of nonlinear intensities does not yield a correct result. In this case, the graphics hardware has to first linearize the destination color by applying Equation (5.18). The blending operation is then performed correctly in linear space, and Equation (5.17) is applied to the result to transform it back into gamma space. The same conversions are not applied to the alpha channel because it always contains linear values.

Most GPUs are capable of writing colors to *multiple render targets (MRT)* in the frame buffer using two or more colors output by the pixel shader. When this functionality is enabled, blending operations can usually be specified separately for each render target, but it may not be possible to use the secondary source color T at the same time. (One use of multiple render targets is described in the context of motion blur in Section 10.7.1.)

Exercises for Chapter 5

1. Calculate the memory storage requirements for 32-bit and 64-bit pixel formats for each of the display resolutions listed in Table 5.1.

2. Calculate 3×3 matrices that transform linear colors (without gamma correction) between the sRGB and CIE RGB color spaces.

3. Calculate the xy chromaticities of the red, green, and blue primaries in the CIE RGB color space to three significant figures.

4. The ultra-high-definition television standard defines a wide color gamut based on monochromatic red, green, and blue primaries that we call uRGB in this exercise. The white point used by uRGB is the same as that used in sRGB, derived from illuminant D65, but the xy chromaticities of the red, green, and blue primaries are given by $(0.708, 0.292)$, $(0.170, 0.797)$, and $(0.131, 0.046)$, respectively. Calculate 3×3 matrices that transform linear colors (without gamma correction) from sRGB to uRGB and from uRGB to sRGB.

5. Determine the coefficients of a cubic polynomial $g(x)$ approximating the function $f_{\text{linear}}(c)$ given by Equation (5.18). The polynomial should satisfy the conditions $g(0) = 0$, $g'(0) = 0$, $g(1) = f_{\text{linear}}(1)$, and $g'(1) = f'_{\text{linear}}(1)$.

6. Suppose we have a world space coordinate system W in which the x axis points north, the y axis points east, and the z axis points down. Find 3×3 matrices that convert vectors between W and the two right-handed world-space coordinate systems in which x points east (in both systems) and either the y axis or z axis points up.

7. Suppose the skeleton model in Figure 5.11 can fire laser beams from its eyes in the direction of the "Skull" node's local x axis. Describe how to calculate world-space direction in which the laser beams would fire.

8. Let **p** and **q** be the 4D homogeneous positions of two vertices after they have been transformed into clip space, and suppose that $p_z \geq 0$ and $q_z < 0$. Find a formula that produces the x and y coordinates of the point on the edge connecting **p** and **q** has a z coordinate of zero.

9. Let $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, and $\boldsymbol{p}_2$ be the two-dimensional vertex positions for a triangle in viewport space. Show that Equation (5.35) is equivalent to the wedge product

$$A = \frac{1}{2}[(\boldsymbol{p}_2 - \boldsymbol{p}_0) \wedge (\boldsymbol{p}_1 - \boldsymbol{p}_0)].$$

10. Consider all possible triangulations of a regular octagon whose vertices lie on a circle of radius one. Determine which triangulation minimizes the sum of the lengths of the five interior edges that are required.

Chapter 6

Projections

Geometry inside a 3D world seen through a virtual camera is mathematically transformed into geometry that can be displayed in a 2D viewport using a process called *projection*. While the basic concepts of a projection are rather simple, various forces of practical engineering cause the best implementation choice to be found at the end of a more complicated path. After an introduction to the view frustum, this chapter spends a lot of space describing how projection matrices work, and it provides a complete discussion of the intricacies that lead us to the most effective methods of projection. We finish with the description of an advanced technique that manipulates a projection matrix for special rendering purposes.

6.1 The View Frustum

Recall that the camera space we introduced in Chapter 5 is configured so that the camera lies at the origin, the x axis points to the right, the y axis points downward, and the z axis points in the direction that the camera is looking. The region of space containing every point in the world that is visible to the camera is a six-sided volume called the *view frustum*. As shown in Figure 6.1, the view frustum is aligned to the coordinate axes in camera space, and it has the shape of a rectangular pyramid with its apex at the camera's position. The pyramid is truncated by a *near plane* perpendicular to the viewing direction (the positive z axis) at a distance n from the camera, and this truncation is the reason that we use the term "frustum". The base of the pyramid, which is at the back of the volume from the camera's perspective, lies in the *far plane* at a distance f from the camera. Points with z coordinates less than the near plane distance n or greater than the far plane distance f are considered to be invisible.
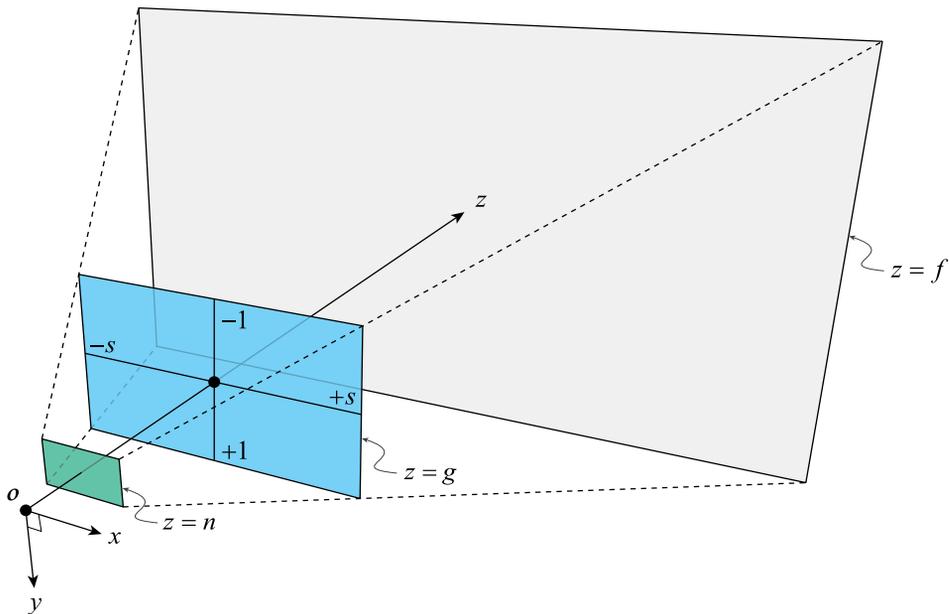
Figure 6.1. This is the shape of the view frustum for a display having an aspect ratio of $s = 16/9$. The near plane (green), projection plane (blue), and far plane (gray) are each perpendicular to the viewing direction (the positive z axis), and they lie at the distances n, g, and f from the camera, respectively.

The rectangular shape of the view frustum is determined by the *aspect ratio* of the viewport. The aspect ratio s is defined as the width of the viewport divided by its height. There is a third plane perpendicular to the viewing direction at a distance g from the camera called the *projection plane*. On the projection plane, the viewport corresponds to the rectangle extending from $y = -1$ to $y = +1$ in the vertical direction and from $x = -s$ to $x = +s$ in the horizontal direction. These dimensions and the projection plane distance g determine the angles of the four lateral planes of the pyramid, completing the set of six planes that bound the view frustum.

The angle between the left and right frustum planes is called the horizontal *field of view* angle φ_x, and the angle between the top and bottom frustum planes is called the vertical field of view angle φ_y. These angles are often given the designations FOVx and FOVy. Either of these angles can be set to a desired value, and the distance g to the projection plane can then be calculated so that the projection plane intersects the side planes of the view frustum at $x = \pm s$ and $y = \pm 1$. As shown in Figure 6.2, the field of view angles satisfy the trigonometric relationships

$$\tan \frac{\varphi_x}{2} = \frac{s}{g} \quad \text{and} \quad \tan \frac{\varphi_y}{2} = \frac{1}{g}. \tag{6.1}$$

One of these equations can be used to calculate g for a given angle φ_x or φ_y. The other field of view angle is then given by one of the inverse equations

$$\varphi_x = 2 \tan^{-1} \frac{s}{g} \quad \text{and} \quad \varphi_y = 2 \tan^{-1} \frac{1}{g}. \tag{6.2}$$

The distance g to the projection plane is sometimes called the *focal length* of the camera. A short focal length corresponds to a wide field of view, and a long focal length corresponds to a narrow field of view. By increasing or decreasing the distance g, the camera can be made to zoom in and zoom out, respectively.

It is common for a game engine to lock the vertical field of view angle φ_y to a constant value, such as 60 degrees, which also determines a constant value for the distance g. The horizontal field of view angle φ_x is then determined by the aspect ratio s, which may be allowed to vary depending on the dimensions of the display. In the case that a fixed value for φ_x is also required, which is equivalent to assuming
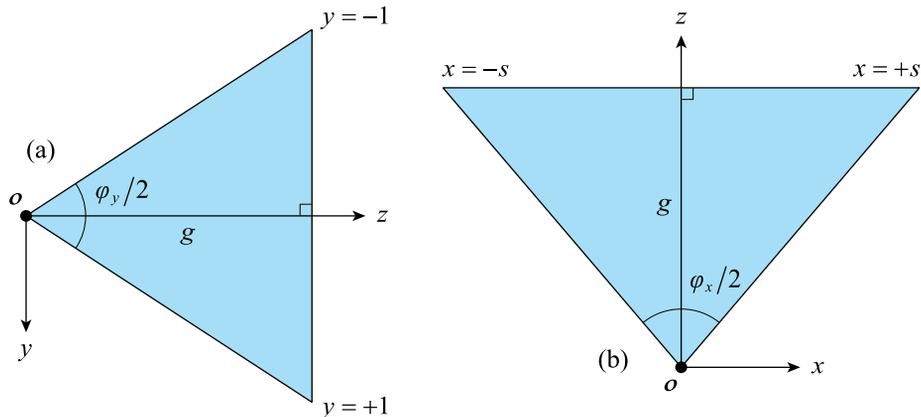
Figure 6.2. The distance g to the projection plane is related to the horizontal and vertical field of view angles φ_x and φ_y as shown. (a) The angle φ_y between the top and bottom planes of the view frustum corresponds to y coordinates in the range $[-1,+1]$ on the projection plane. (b) The angle φ_x between the left and right planes of the view frustum corresponds to x coordinates in the range $[-s,+s]$ on the projection plane, where s is the aspect ratio of the viewport.

a specific aspect ratio such as $s = 16/9$, the viewport cannot possibly fill the entire display if its actual aspect ratio is different. If the display's aspect ratio is less than s, then black bars are typically shown above and below the viewport to enforce an exact rectangular shape for the rendered image. If the display's aspect ratio is greater than s, then black bars are shown to the left and right of the viewport.

The camera-space normal directions of the near, far, left, and right planes of the view frustum are shown in Figure 6.3. The near plane has the normal direction $(0,0,1)$ aligned with the view direction along the z axis, and the far plane has the opposite normal direction $(0,0,-1)$. Each normal direction of the four lateral planes is a 90-degree rotation of the midpoint of an edge of the viewport rectangle on the projection plane. For example, the midpoint of the left edge lies at the point $(-s, 0, g)$, and it is rotated 90 degrees clockwise to obtain the normal direction $(g, 0, s)$ for the left plane. The midpoint $(s, 0, g)$ of the right edge is rotated 90 degrees counterclockwise to obtain the normal direction $(-g, 0, s)$ for the right plane.

The 4D camera-space coordinates of all six planes bounding the view frustum are listed in Table 6.1, and they have been scaled so that their normals have unit length. Each plane faces inward so that points inside the view frustum always lie at a positive distance from all six planes, where the distance is given by the dot product between the plane and the point. Any point outside the view frustum must be on the negative side of at least one of the planes. The property can be used to
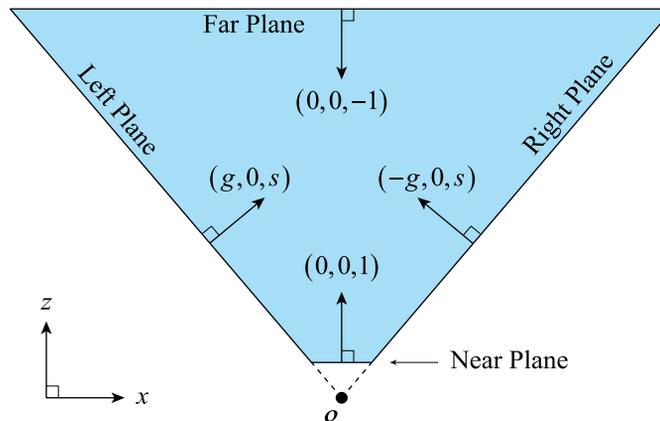
Figure 6.3. These are the inward-pointing normal directions for the near, far, left, and right planes of the view frustum in camera space. Not shown, the normal directions for the top and bottom planes are $(0, g, 1)$ and $(0, -g, 1)$, respectively.

Frustum Plane	Camera-Space (x, y, z, w)
Near	$(0,0,1,-n)$
Far	$(0,0,-1,f)$
Left	$\dfrac{1}{\sqrt{g^2+s^2}}(g,0,s,0)$
Right	$\dfrac{1}{\sqrt{g^2+s^2}}(-g,0,s,0)$
Top	$\dfrac{1}{\sqrt{g^2+1}}(0,g,1,0)$
Bottom	$\dfrac{1}{\sqrt{g^2+1}}(0,-g,1,0)$

Table 6.1. These are the 4D camera-space coordinates for the six planes bounding the view frustum, where n is the near plane distance, f is the far plane distance, g is the projection plane distance, and s is the aspect ratio of the viewport.

perform object culling in world space, as discussed in Chapter 9. Once a camera-space view frustum plane $\mathbf{f}_{\text{camera}}$ has been calculated, the world-space plane $\mathbf{f}_{\text{world}}$ is given by

$$\mathbf{f}_{\text{world}} = \mathbf{f}_{\text{camera}}\mathbf{M}_{\text{camera}}^{-1}, \tag{6.3}$$

where $\mathbf{M}_{\text{camera}}$ is the transform from camera space to world space. (Equation (3.45) uses the adjugate to transform a plane, but in this case, we can assume the determinant is one, so the inverse and adjugate are the same.) Since the axes are orthogonal in camera space, the upper-left 3×3 portion of $\mathbf{M}_{\text{camera}}^{-1}$ is simply the transpose of the upper-left 3×3 portion of $\mathbf{M}_{\text{camera}}$, so the normal vectors for the world-space frustum planes end up being linear combinations of the columns of $\mathbf{M}_{\text{camera}}$. For example, the world-space normal $\mathbf{n}$ of the left frustum plane is given by

$$\mathbf{n} = g\mathbf{M}_{\text{camera}[0]} + s\mathbf{M}_{\text{camera}[2]}, \tag{6.4}$$

where we continue to use the notation $\mathbf{M}_{[i]}$ to refer to the zero-based column i of the matrix $\mathbf{M}$. To form a complete plane $[\mathbf{n} \mid d]$, we only need to calculate

$$d = -\mathbf{n} \cdot \mathbf{M}_{\text{camera}[3]}, \tag{6.5}$$

where the last column of $\mathbf{M}_{\text{camera}}$ gives the world-space position of the camera.

At the projection distance g in front of the camera, a plane perpendicular to the view direction cuts through the lateral planes of the view frustum at a rectangle bounded by the camera-space coordinates $x = \pm s$ and $y = \pm 1$. At an arbitrary distance u from the camera, these ranges are simply scaled by a factor of u/g. There are instances in which we need to know where the four points q_0, q_1, q_2, and q_3 at the corners of this rectangle are located in world space. Again using the matrix $\mathbf{M}_{\text{camera}}$ that transforms camera space into world space, we identify the camera position $c = \mathbf{M}_{\text{camera}[3]}$ and the three axis directions $\mathbf{x} = \mathbf{M}_{\text{camera}[0]}$, $\mathbf{y} = \mathbf{M}_{\text{camera}[1]}$, and $\mathbf{z} = \mathbf{M}_{\text{camera}[2]}$ for notational convenience. Then, for a view frustum having a projection distance g and an aspect ratio s, the points q_i are given by

$$q_0 = c + \frac{us}{g}\mathbf{x} + \frac{u}{g}\mathbf{y} + u\mathbf{z}, \qquad q_1 = c + \frac{us}{g}\mathbf{x} - \frac{u}{g}\mathbf{y} + u\mathbf{z},$$

$$q_2 = c - \frac{us}{g}\mathbf{x} - \frac{u}{g}\mathbf{y} + u\mathbf{z}, \qquad q_3 = c - \frac{us}{g}\mathbf{x} + \frac{u}{g}\mathbf{y} + u\mathbf{z}. \tag{6.6}$$

When $u = n$ and $u = f$, these points correspond to the rectangles on the boundary of the view frustum at the near plane and far plane.

6.2 Perspective-Correct Interpolation

The main purpose of a projection matrix is to transform 3D points from camera space into their corresponding positions on the projection plane so that a view of a 3D world can be displayed as a 2D image. Finding the projection of a point p involves the rather simple task of determining where a line segment connecting the origin (the camera position) to p intersects the projection plane. The x and y coordinates of the projected point q are given by

$$q_x = \frac{g}{p_z}\,p_x \quad \text{and} \quad q_y = \frac{g}{p_z}\,p_y. \tag{6.7}$$

The division by p_z is called the *perspective divide*, and it is responsible for objects farther away, which have larger z coordinates, appearing to be smaller on the screen than objects that are closer to the camera.

In order to incorporate the perspective divide into the linear transformation performed by a matrix, we work in homogeneous coordinates. By using a 4×4 matrix that moves p_z into the w coordinate, the perspective divide happens when the transformed 4D homogeneous vector q is projected back into 3D space. This is illustrated by the equation

$$q = \begin{bmatrix} g & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & g & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} gp_x \\ gp_y \\ gp_z \\ p_z \end{bmatrix}, \tag{6.8}$$

which produces the x and y coordinates given by Equation (6.7) after dividing by the w coordinate. Note that this transform also produces a z coordinate of g, which is what it must be for a point lying in the projection plane.

The matrix in Equation (6.8) does indeed perform the correct projection of any 3D point p with $p_z \neq 0$ onto the plane containing the 2D viewport, but it is not the form of the matrix that is actually used in computer graphics. The reason for this has to do with interpolation of values inside a triangle after its vertices have been projected, and an explanation of the correct way to perform interpolation with perspective division applied will allow us to derive the standard form of the projection matrix used in practice.

When a triangle is drawn by the GPU, the three vertices are projected into the viewport to determine what set of pixels the triangle covers. Values calculated by the vertex shader, such as depth, colors, and texture coordinates, are initially known only at the projected vertex locations, and the hardware must interpolate them inside the triangle in order to obtain the proper values at any particular pixel. Due to the perspective divide, however, the correct interpolation is not linear, and the GPU must use a special type of interpolation called *perspective-correct* interpolation to render triangles without producing unwanted nonlinear distortion.

Consider two camera-space vertex positions p_1 and p_2 and their projections q_1 and q_2 onto the plane at a distance g from the camera, as shown in Figure 6.4. When we take equally-spaced steps on the projection plane between the points q_1 and q_2, it is clear that these do not correspond to equally-spaced steps between the original vertex positions p_1 and p_2. More generally, a linearly interpolated point

$$q(t) = q_1(1-t) + q_2 t, \tag{6.9}$$

is not the projection of $p_1(1-t) + p_2 t$ for the same value of t except at the endpoints where $t = 0$ or $t = 1$. If we wanted to interpolate per-vertex values directly, we would have to unproject $q(t)$ back onto the plane containing the triangle and calculate a new interpolation parameter. This would be expensive, so we instead look for quantities that can be linearly interpolated on the projection plane.

Let $[\mathbf{n} \,|\, d]$ be the plane containing a triangle to be rendered, and assume that $d \neq 0$ because otherwise, the plane would be viewed edge-on and thus not be visible to the camera. A point $p = (x, y, z)$ inside the triangle must satisfy the equation
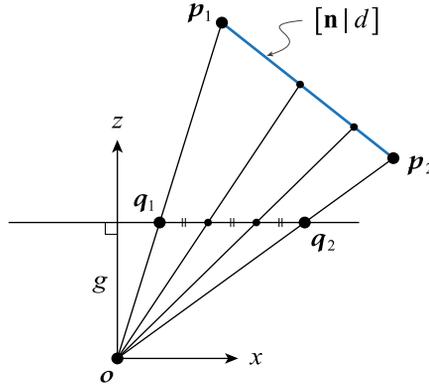
Figure 6.4. Two camera-space vertex positions p_1 and p_2 are projected onto the plane at a distance g from the camera as the points q_1 and q_2. Positions interpolated at equal-sized steps between q_1 and q_2 do not correspond to positions interpolated at equal-sized steps between the original vertices p_1 and p_2.

$$[\mathbf{n}\,|\,d]\cdot p = n_x x + n_y y + n_z z + d = 0. \tag{6.10}$$

The projection q of the point p is given by

$$q = \frac{g}{z}\,p, \tag{6.11}$$

and the reverse projection of q is thus

$$p = \frac{z}{g}\,q. \tag{6.12}$$

When we substitute this expression for p in Equation (6.10), we have

$$\left(\frac{n_x}{g}q_x + \frac{n_y}{g}q_y + n_z\right)z + d = 0. \tag{6.13}$$

Solving this equation for z would produce an expression in which the coordinates of q appear in the denominator, and this is not something that can be linearly interpolated. However, if we instead solve for the reciprocal of z, then we get

$$\frac{1}{z} = -\frac{n_x q_x}{gd} - \frac{n_y q_y}{gd} - \frac{n_z}{d} = -\frac{\mathbf{n}\cdot q}{gd}, \tag{6.14}$$

and the right side of this equation is a linear function of the coordinates of q. The reciprocal of z is a quantity that can be linearly interpolated, and for a point $q(t)$ given by Equation (6.9), we can write

$$\frac{1}{z(t)} = -\frac{\mathbf{n} \cdot q(t)}{gd}$$

$$= -\frac{\mathbf{n} \cdot q_1(1-t) + \mathbf{n} \cdot q_2 t}{gd}$$

$$= \frac{1}{z_1}(1-t) + \frac{1}{z_2}t, \tag{6.15}$$

where z_i is the z coordinate of p_i, and $z(t)$ represents the depth interpolated with perspective correctness.

Equation (6.15) is the fundamental principle upon which depth buffering is based. At the same time that the vertices of a triangle are projected into the viewport, the camera-space depths of those vertices are inverted to produce reciprocal z values that can be linearly interpolated across the face of the triangle. It is this interpolated reciprocal that actually gets used in the depth test, and not the value of z itself. This allows the graphics hardware to avoid a costly reciprocal operation at each pixel during the rasterization process. It's only when the depth test passes and the pixel shader is executed that the reciprocal operation is performed to recover the actual depth value, which turns out to be necessary for calculating interpolated values of vertex attributes.

Let a_1 and a_2 be the components of a particular vertex attribute associated with the positions $p_1 = (x_1, y_1, z_1)$ and $p_2 = (x_2, y_2, z_2)$. For example, these values could each represent the red component of a vertex color or the one of the texture coordinates specified at each vertex. The linearly interpolated value $a(t)$ between the two vertices must correspond to the same fraction of the total difference $a_2 - a_1$ as does the interpolated depth $z(t)$ with $z_2 - z_1$. That is, the value of $a(t)$ must satisfy the equation

$$\frac{a(t) - a_1}{a_2 - a_1} = \frac{z(t) - z_1}{z_2 - z_1}. \tag{6.16}$$

Substituting the reciprocal of the right side of Equation (6.15) for $z(t)$ gives us

$$\frac{a(t) - a_1}{a_2 - a_1} = \frac{z_1 t}{z_2(1-t) + z_1 t} \tag{6.17}$$

after some algebraic simplification. When we solve for $a(t)$, we have

$$a(t) = \frac{a_1 z_2 (1-t) + a_2 z_1 t}{z_2 (1-t) + z_1 t}. \tag{6.18}$$

Multiplying the numerator and denominator by $1/z_1 z_2$ produces

$$a(t) = \frac{\dfrac{a_1}{z_1}(1-t) + \dfrac{a_2}{z_2}t}{\dfrac{1}{z_1}(1-t) + \dfrac{1}{z_2}t} = \frac{\dfrac{a_1}{z_1}(1-t) + \dfrac{a_2}{z_2}t}{\dfrac{1}{z(t)}}, \tag{6.19}$$

and multiplying both sides by $1/z(t)$ finally yields

$$\frac{a(t)}{z(t)} = \frac{a_1}{z_1}(1-t) + \frac{a_2}{z_2}t. \tag{6.20}$$

This tells us that the quotient of any vertex attribute and the depth at the same vertex can be linearly interpolated on the projection plane.

When the projection matrix is applied to a vertex, the camera-space depth z is moved to the w coordinate of the result, just as in Equation (6.8). As such, it is common to express the interpolation equations in terms of the projected w coordinates. Furthermore, all three vertices are involved in the interpolation calculations, and two parameters u and v are required to form a set of barycentric coordinates. When generalized to a projected triangle, Equations (6.15) and (6.20) become

$$\boxed{\frac{1}{w(u,v)} = \frac{1}{w_1}(1-u-v) + \frac{1}{w_2}u + \frac{1}{w_3}v} \tag{6.21}$$

and

$$\boxed{\frac{a(u,v)}{w(u,v)} = \frac{a_1}{w_1}(1-u-v) + \frac{a_2}{w_2}u + \frac{a_3}{w_3}v.} \tag{6.22}$$

For each of the three vertices of a triangle, after they have been transformed by the projection matrix, the GPU calculates $1/w_i$. One method that could then be used to perform attribute interpolation is to multiply every component a_i of each vertex attribute by the result to obtain a_i/w_i. These quantities, which all include the reciprocal of the depth stored in the coordinate w_i, are linearly interpolated at each pixel covered by a triangle using Equations (6.21) and (6.22). In the pixel shader, the reciprocal of the interpolated value of $1/w(u,v)$ is calculated to obtain

the perspective-correct depth $w(u, v)$ at the pixel location, and each interpolated value of $a(u, v)/w(u, v)$ is multiplied by that depth to obtain the perspective-correct attribute value $a(u, v)$ for the same pixel.

Because there are often many values a_i that would need to be multiplied by $1/w_i$, most modern GPUs take a different approach in which the barycentric coordinates u and v themselves are interpolated in a perspective-correct manner. Once their values have been determined at a point inside a triangle, they can be used to calculate the attribute values $a(u_{pc}, v_{pc})$ with ordinary linear interpolation, where the subscript "pc" indicates that perspective correction has been applied. In this case, no per-vertex multiplications by $1/w_i$ need to be performed. We show in detail how this works for the u parameter, and the same is true for the v parameter.

For any values u_1 and u_2 assigned to two vertices having w coordinates w_1 and w_2, the perspective-correct interpolated value $u_{pc}(t)$ is given by

$$u_{pc}(t) = w(t)\left[\frac{u_1}{w_1}(1-t) + \frac{u_2}{w_2}t\right], \tag{6.23}$$

which follows directly from Equation (6.20) after replacing z coordinates with w coordinates. Since $u_{pc}(t)$ is a parameter that is required to interpolate between known values at two vertices, we must assign $u_1 = 0$ and $u_2 = 1$. Doing so produces the simpler expression

$$u_{pc}(t) = \frac{w(t)}{w_2}t. \tag{6.24}$$

Now, continuing to use w coordinates instead of z coordinates, if we reorganize Equation (6.15) as

$$1 - t = \frac{w_1}{w(t)} - \frac{w_1}{w_2}t \tag{6.25}$$

and plug this expression for $1 - t$ into Equation (6.20), then we have

$$a(t) = a_1\left(1 - \frac{w(t)}{w_2}t\right) + a_2\frac{w(t)}{w_2}t \tag{6.26}$$

after multiplying by $w(t)$. This contains two instances of $u_{pc}(t)$ given by Equation (6.24), so it can be rewritten as

$$a(t) = a_1(1 - u_{pc}(t)) + a_2 u_{pc}(t), \tag{6.27}$$

which is just ordinary linear interpolation.

To make use of this method, the GPU calculates barycentric coordinates u and v in viewport space for each pixel inside a triangle, as described in Section 5.5.2. The reciprocal of the w coordinate at the pixel is then given by

$$\frac{1}{w(u,v)} = \frac{1}{w_1}(1-u-v) + \frac{1}{w_2}u + \frac{1}{w_3}v. \tag{6.28}$$

This value is inverted and used to calculate the perspective-correct barycentric coordinates with the formulas

$$u_{\mathrm{pc}} = \frac{w(u,v)}{w_2}u \ \text{ and } \ v_{\mathrm{pc}} = \frac{w(u,v)}{w_3}v. \tag{6.29}$$

These values are all that are needed to then interpolate any number of per-vertex attributes with the equation

$$a(u_{\mathrm{pc}}, v_{\mathrm{pc}}) = a_1(1-u_{\mathrm{pc}} - v_{\mathrm{pc}}) + a_2 u_{\mathrm{pc}} + a_3 v_{\mathrm{pc}}. \tag{6.30}$$

The perspective correction is performed entirely with Equations (6.28) and (6.29) at a fixed cost regardless of how many attributes are needed by a pixel shader.

6.3 Projection Matrices

The purpose of the projection matrix is to transform geometry inside the view frustum from camera space to clip space using homogeneous coordinates. In the case of a perspective projection, the projection matrix also sets up the perspective divide by moving the camera-space depth into the w coordinate. In the three-dimensional slice of clip space where $w = 1$, the transformed view frustum is shaped like a box called the *canonical view volume*, as shown in Figure 6.5. The canonical view volume is a representation of the view frustum after the perspective divide occurs, and it is independent of the field of view angles, the aspect ratio of the viewport, and the distances to the near and far planes. The x and y coordinates of any point inside the view frustum both fall into the range $[-1, +1]$ in the canonical view volume, and the z coordinates between the near and far planes are mapped to the range $[0,1]$. These are called *normalized device coordinates*, and they are later scaled to the dimensions of the frame buffer by the viewport transform.

Throughout this chapter, we will be talking about coordinates in four different phases of transformation between camera space and viewport space. To make it clear which space any particular point or coordinate values belong to, we use the

subscript labels shown in Figure 6.6. The subscript "camera" means that the coordinates describe a position in 3D camera space, the subscript "clip" is used for 4D homogeneous coordinates in clip space produced by the projection matrix, the subscript "device" indicates 3D normalized device coordinates after the division by w_{clip}, and the subscript "viewport" refers to the final 3D coordinates after the viewport transform has been applied.
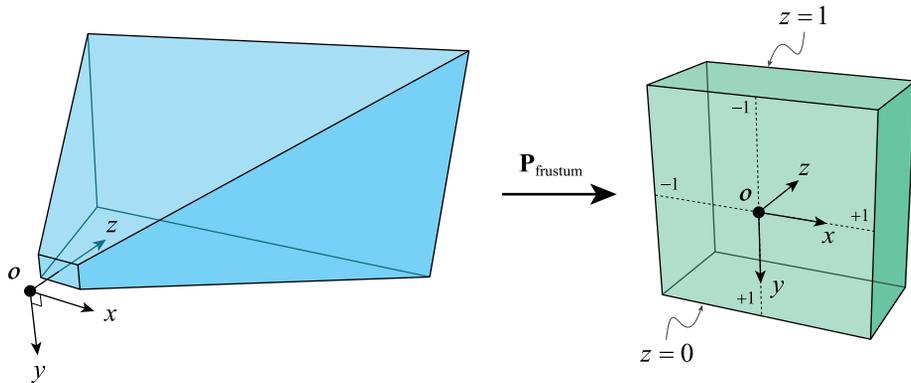
Figure 6.5. The projection matrix $\mathbf{P}_{\text{frustum}}$ transforms the view frustum into the canonical view volume, where points inside the view frustum have normalized device coordinates in the range $[-1, +1]$ along the x and y axes and in the range $[0, 1]$ along the z axis after the perspective divide occurs.
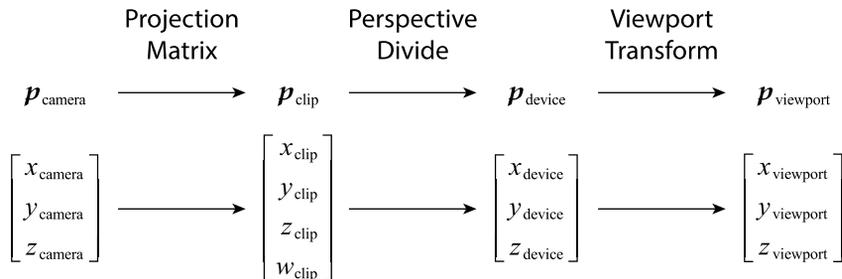
Figure 6.6. The projection matrix transforms from 3D "camera" coordinates to 4D homogeneous "clip" coordinates. The perspective divide then converts clip coordinates into 3D normalized "device" coordinates. The viewport transform finally scales and offsets device coordinates so they become "viewport" coordinates.

6.3.1 Perspective Projection Matrices

The exact form of the perspective projection matrix, which we call $\mathbf{P}_{\text{frustum}}$, can be determined by enforcing the range requirements of the canonical view volume after the perspective divide has been applied through the division by the w coordinate. We have already seen in Equation (6.7) that the x and y coordinates of a camera-space point $\boldsymbol{p}_{\text{camera}} = (x_{\text{camera}}, y_{\text{camera}}, z_{\text{camera}})$ are projected into the viewport by multiplying them by g/z_{camera}. If $\boldsymbol{p}_{\text{camera}}$ is inside the view frustum, then the projected y coordinate falls in the range $[-1, +1]$, but the projected x coordinate falls into the range $[-s, +s]$, so we also need to divide the x coordinate by s to fit it into the canonical view volume. This means that the projection matrix has the form

$$\mathbf{P}_{\text{frustum}} = \begin{bmatrix} g/s & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & A & B \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{6.31}$$

where the entries A and B used to calculate the projected z coordinate have yet to be ascertained. The homogeneous clip-space point $\boldsymbol{p}_{\text{clip}} = (x_{\text{clip}}, y_{\text{clip}}, z_{\text{clip}}, w_{\text{clip}})$ produced by this matrix is given by

$$\boldsymbol{p}_{\text{clip}} = \mathbf{P}_{\text{frustum}} \boldsymbol{p}_{\text{camera}} = \begin{bmatrix} (g/s) x_{\text{camera}} \\ g y_{\text{camera}} \\ A z_{\text{camera}} + B \\ z_{\text{camera}} \end{bmatrix}. \tag{6.32}$$

The third row of $\mathbf{P}_{\text{frustum}}$ determines the z coordinate in clip space. It ordinarily does not depend on x_{camera} or y_{camera}, so the first two entries in the row are zero, leaving two unknowns A and B in the last two entries. After division by w_{clip}, which is always equal to z_{camera}, the projected z coordinates of points between the near and far planes of the view frustum must be in the range $[0, 1]$. It is this projected value of $z_{\text{clip}}/w_{\text{clip}}$ that is ultimately used in the depth test as it is linearly interpolated over a triangle. Because the reciprocal of depth must be interpolated, as discussed in Section 6.2, it is remarkably convenient that

$$\frac{z_{\text{clip}}}{w_{\text{clip}}} = A + \frac{B}{z_{\text{camera}}}, \tag{6.33}$$

and the camera-space depth z_{camera} appears as a reciprocal, just as required. All we need to do is calculate the values of A and B that produce the depth $z_{\text{clip}}/w_{\text{clip}} = 0$

on the near plane where $z_{camera} = n$ and the depth $z_{clip}/w_{clip} = 1$ on the far plane where $z_{camera} = f$, which are expressed by the equations

$$A + \frac{B}{n} = 0 \quad \text{and} \quad A + \frac{B}{f} = 1. \tag{6.34}$$

Solving these for A and B gives us

$$A = \frac{f}{f-n} \quad \text{and} \quad B = -\frac{nf}{f-n}, \tag{6.35}$$

and the complete perspective projection matrix is thus given by

$$\mathbf{P}_{frustum} = \begin{bmatrix} g/s & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & \dfrac{f}{f-n} & -\dfrac{nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{6.36}$$

The fact that B is a negative value means that depths that were larger in camera space are still larger in clip space after the perspective divide. Taking the reciprocal reverses the order of depths, but the subtraction then reverses the order again. A function that constructs the projection matrix $\mathbf{P}_{frustum}$ and returns a Matrix4D data structure is shown in Listing 6.1.

Listing 6.1. This code creates a 4×4 matrix that represents the perspective projection $\mathbf{P}_{frustum}$ for a view frustum having a vertical field of view specified by fovy and returns it in a Matrix4D data structure. The aspect ratio of the viewport is given by s, and the distances to the near and far planes are given by n and f.

```
Matrix4D MakeFrustumProjection(float fovy, float s, float n, float f)
{
    float g = 1.0F / tan(fovy * 0.5F);
    float k = f / (f - n);

    return (Matrix4D(g / s, 0.0F, 0.0F, 0.0F,
                     0.0F,  g,    0.0F, 0.0F,
                     0.0F, 0.0F,  k,    -n * k,
                     0.0F, 0.0F,  1.0F, 0.0F));
}
```

In our version of clip space, the x, y, and z axes point in the same directions as they do in camera space. This convention is followed by some rendering systems, but others invert the y axis so that it points upward instead of downward, creating a left-handed clip space. When this is the case, the second row of the projection matrix needs to be negated to produce the correct clip coordinates, which amounts to simply replacing g with $-g$ in Equation (6.36). Some rendering systems also require that z coordinates fall in the range $[-1, +1]$ after the perspective divide instead of the range $[0, 1]$. The values of A and B are a little different in that case, as shown by Exercise 8.

Unlike the matrix shown in Equation (6.8), the perspective projection matrix given by Equation (6.36) is invertible, and it is has the form

$$\mathbf{P}_{\text{frustum}}^{-1} = \begin{bmatrix} s/g & 0 & 0 & 0 \\ 0 & 1/g & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & -\dfrac{f-n}{nf} & \dfrac{1}{n} \end{bmatrix}. \tag{6.37}$$

The lower-right 2×2 portion of this matrix can be used to transform a value z_{device}, taken from an existing depth buffer, back into camera space. The vector $(z_{\text{device}}, 1)$ is transformed into the camera-space coordinates $(z_{\text{camera}}, w_{\text{camera}})$, where we must temporarily use homogeneous coordinates, and we then divide by w_{camera} to obtain

$$\frac{z_{\text{camera}}}{w_{\text{camera}}} = \frac{nf}{z_{\text{device}} n + (1 - z_{\text{device}}) f}. \tag{6.38}$$

Figure 6.7 illustrates how the standard projection matrix $\mathbf{P}_{\text{frustum}}$ given by Equation (6.36) transforms depths from camera space to normalized device coordinates, both inside and outside the view frustum. Camera-space depths falling between the near plane where $z_{\text{camera}} = n$ and far plane where $z_{\text{camera}} = f$ are mapped precisely into the canonical view volume where $0 \leq z_{\text{device}} \leq 1$. All camera-space depths lying beyond the far plane, no matter how large, are squeezed into the small space between $z_{\text{device}} = 1$ and $z_{\text{device}} = f/(f-n)$, and this range becomes smaller as the far plane is moved farther away from the camera. Conversely, all of the depths between the camera and the near plane, in the range $0 \leq z_{\text{camera}} < n$, are stretched into the entire range of negative values of z_{device}, extending all the way to $-\infty$. The remaining set of possible depths, consisting of all values lying behind the camera with $z_{\text{camera}} < 0$, is wrapped around to the opposite side of the view frustum and ends up occupying the range of positive values where $z_{\text{device}} > f/(f-n)$.
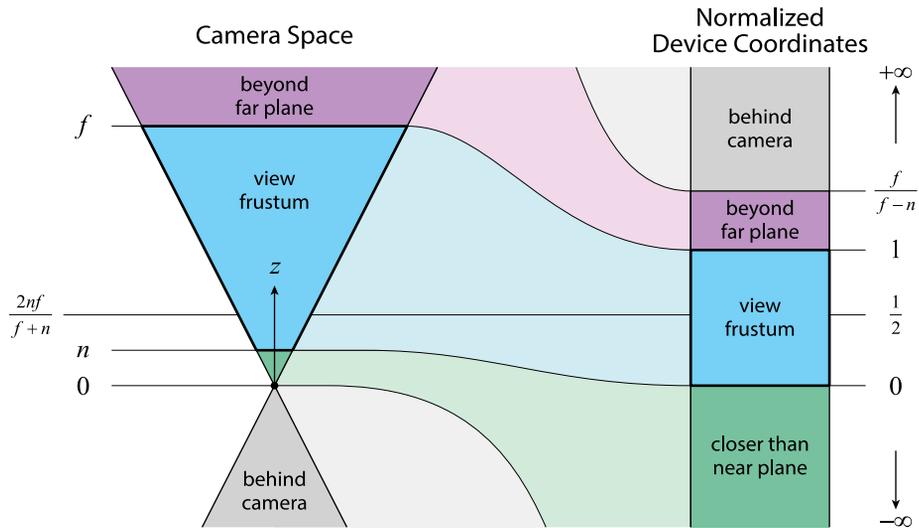
Figure 6.7. This diagram illustrates how depths in camera space are transformed into normalized device coordinates by the standard perspective projection matrix $\mathbf{P}_{\text{frustum}}$.

6.3.2 Infinite Projection Matrices

It is often impractical, or at least inconvenient, for the view frustum to have a far plane beyond which no objects can be rendered. Some game engines need to draw a very large world in which the camera can see many kilometers in any direction, and things like the sky, sun, moon, and star fields need to be rendered much farther away to ensure that they don't intersect any world geometry. Fortunately, it is possible to push the far plane out to a literally infinite distance from the camera, and doing so has become a common practice. Game engines may still choose to cull objects beyond some maximum viewing distance, but the projection matrix will no longer cause triangles to be clipped against a far plane.

The infinite perspective projection matrix $\mathbf{P}_{\text{infinite}}$ is derived by taking the limit as the distance f to the far plane tends to infinity in the matrix $\mathbf{P}_{\text{frustum}}$ given by Equation (6.36) to arrive at

$$\mathbf{P}_{\text{infinite}} = \lim_{f \to \infty} \mathbf{P}_{\text{frustum}} = \begin{bmatrix} g/s & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & 1 & -n \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{6.39}$$

A camera-space depth z_{camera} is transformed by this matrix into a normalized device depth

$$z_{device} = 1 - \frac{n}{z_{camera}}, \tag{6.40}$$

which is in the range $[0,1]$ as long as $z_{camera} \geq n$, and it approaches one as z_{camera} becomes arbitrarily large. For the infinite projection matrix, the space shown beyond the far plane in Figure 6.7 shrinks to nothing.

Objects can be rendered at infinity by using homogeneous vertex positions that have w coordinates equal to zero. These positions can be specified in world space or in object space, and they behave as direction vectors as they are transformed into camera space, which means that the camera's position has no effect on them. When we apply the infinite projection matrix given by Equation (6.39) to a camera-space direction vector, we get

$$\begin{bmatrix} g/s & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & 1 & -n \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \\ 0 \end{bmatrix} = \begin{bmatrix} (g/s)\,x_{camera} \\ gy_{camera} \\ z_{camera} \\ z_{camera} \end{bmatrix}. \tag{6.41}$$

The values of z_{clip} and w_{clip} in the result are equal, so the value z_{device} after the perspective divide is exactly one. This corresponds to a point lying on the far plane at an infinite distance from the camera.

The realities of floating-point round-off error often make the matrix $\mathbf{P}_{infinite}$ unusable in practice for rendering objects at infinity unless we make a small modification. When the projection matrix is combined with the model-view matrix, the exactness of the entries of $\mathbf{P}_{infinite}$ can be lost, and it's possible for the calculated value of z_{device} to be slightly larger than one, which causes random clipping to occur. The solution to this problem is to modify the projection matrix so that it produces depths in the range $[0, 1-\varepsilon]$ for some tiny constant value ε. This can be achieved by multiplying the third row of $\mathbf{P}_{infinite}$ by $1-\varepsilon$ to obtain the new matrix

$$\mathbf{P}_{infinite}^{*} = \begin{bmatrix} g/s & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & 1-\varepsilon & -n(1-\varepsilon) \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{6.42}$$

The value of ε needs to be large enough that it is still significant compared to the floating-point value of one from which it is subtracted. For 32-bit numbers, choos-

ing a value around $\varepsilon = 2^{-20}$, which is about 10^{-6}, is usually safe and effective. A function that constructs the modified infinite projection matrix $\mathbf{P}^*_{\text{infinite}}$ and returns a `Matrix4D` data structure is shown in Listing 6.2.

Listing 6.2. This code creates a 4×4 matrix that represents the infinite perspective projection $\mathbf{P}^*_{\text{infinite}}$ for a view frustum having a vertical field of view specified by `fovy` and returns it in a `Matrix4D` data structure. The aspect ratio of the viewport is given by `s`, the distance to the near plane is given by `n`, and the value of ε in Equation (6.42) is given by `e`. If $\varepsilon = 0$, then the returned matrix is $\mathbf{P}_{\text{infinite}}$ given by Equation (6.39).

```
Matrix4D MakeInfiniteProjection(float fovy, float s, float n, float e)
{
    float g = 1.0F / tan(fovy * 0.5F);
    e = 1.0F - e;

    return (Matrix4D(g / s, 0.0F, 0.0F, 0.0F,
                     0.0F,  g,    0.0F, 0.0F,
                     0.0F, 0.0F,  e,   -n * e,
                     0.0F, 0.0F, 1.0F, 0.0F));
}
```

6.3.3 Projected Depth Precision

The projection matrix and perspective divide map camera-space depths inside the view frustum to the range $[0,1]$, but the mapping is nonlinear, and this leads to a highly uneven distribution of discrete representable values in the depth buffer. Consider the value $z_{\text{device}} = \frac{1}{2}$ that is halfway through the full range of depths in normalized device coordinates. We can calculate the original value of z_{camera} for which the projection matrix $\mathbf{P}_{\text{frustum}}$ produces this particular depth by solving the equation

$$\frac{f}{f-n} - \frac{nf}{(f-n)z_{\text{camera}}} = \frac{1}{2}. \tag{6.43}$$

The answer is

$$z_{\text{camera}} = \frac{2nf}{f+n}, \tag{6.44}$$

and this is shown above in Figure 6.7 in order to illustrate the disproportionality between the view frustum in camera space and the canonical view volume in clip space. Since f is expected to be much larger than n, this value of z_{camera} is very close

to $2n$, with equality being reached as f tends to infinity. What the above calculation means is that entirely half of the final range of depths is dedicated to the tiny difference between n and at most $2n$ in camera space, and the other half must be assigned to the vast expanse beyond that.

Ironically, the design of floating-point numbers causes drastically greater precision to be available in the range $[0, \frac{1}{2})$, which corresponds to a very small distance, compared to the relatively scant precision available in the range $[\frac{1}{2}, 1)$, which corresponds to a very large distance. In 32-bit floating-point, there are 2^{23} (roughly 8.4 million) possible mantissa values for each particular exponent value. Of the 126 possible exponent values used for numbers smaller than one, 125 of them correspond to numbers less than $\frac{1}{2}$, and the remaining solitary exponent value corresponds to all of the numbers greater than or equal to $\frac{1}{2}$. As shown in Figure 6.8(a), the result is a rather ridiculous clustering of representable depths close to the near plane. In this diagram, the values of z_{device} produced by the projection matrix $\mathbf{P}_{\text{frustum}}$ are graphed as a function of z_{camera}. The horizontal gridlines represent individual values that can be written into the depth buffer, where the number of distinct mantissas per exponent has been limited to ten for illustrative purposes. The vertical lines are drawn at each value of z_{camera} where a horizontal line intersects the curve. The rapidly growing distance between vertical lines as camera-space depth increases demonstrates the equally rapid loss of precision as geometry lies farther away from the camera.

The situation just described is the opposite of what we would like to have. However, the problem is easily remedied by redesigning the projection matrix so that the clip-space depths between the near and far planes still get mapped to the range $[0, 1]$, but in the reverse order. Enforcing this requirement produces the new projection matrix

$$\mathbf{R}_{\text{frustum}} = \begin{bmatrix} g/s & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & \dfrac{n}{n-f} & -\dfrac{nf}{n-f} \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{6.45}$$

where the values n and f have simply been swapped in each of the places they previously appeared in the matrix $\mathbf{P}_{\text{frustum}}$ given by Equation (6.36). When the matrix $\mathbf{R}_{\text{frustum}}$ is applied, the graph of z_{device} is turned upside down, as shown in Figure 6.8(b), and all of the extra precision available for small values of z_{device} is now allocated to depths far away from the camera where they are needed the most.
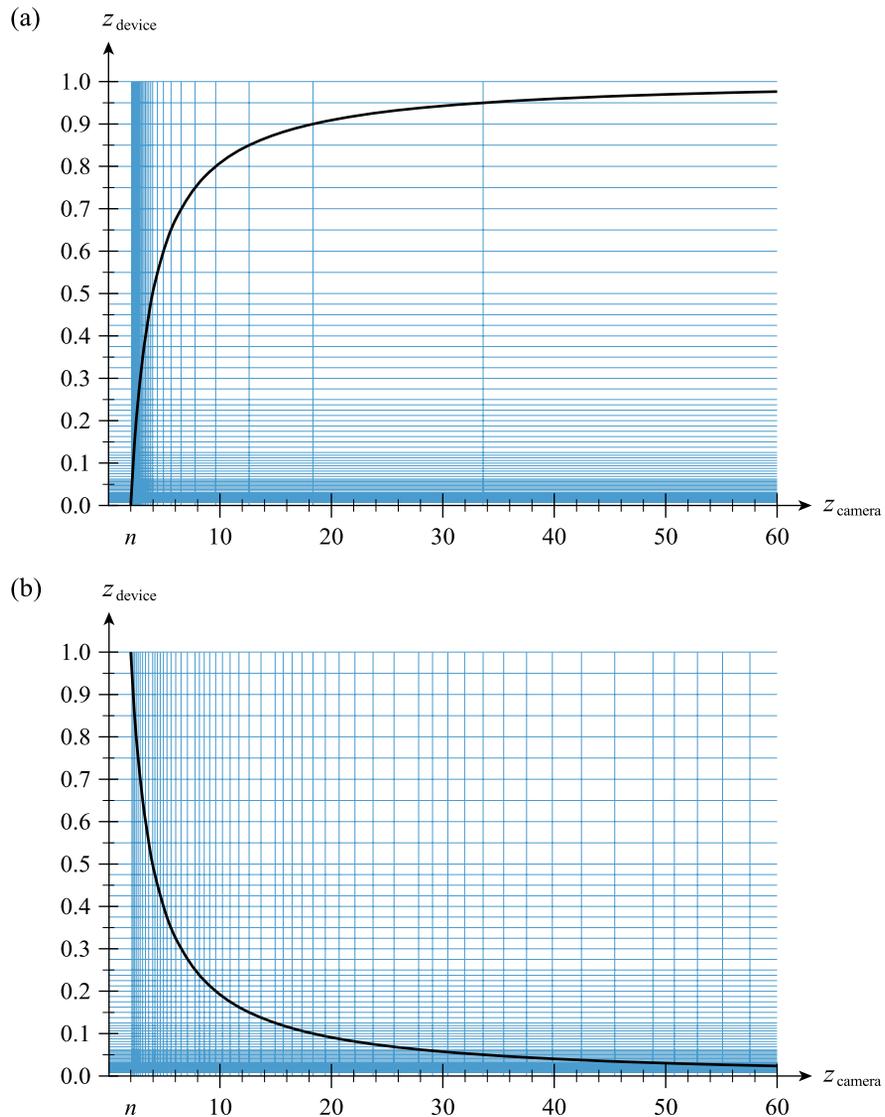
Figure 6.8. The values of z_{device} produced by the projection matrix and the perspective divide are graphed as functions of z_{camera}. (a) The matrix $\mathbf{P}_{\text{frustum}}$ given by Equation (6.36) causes depths to increase with distance from the camera, but almost all of the available precision is wasted close to the near plane, leaving very little farther out. (b) The matrix $\mathbf{R}_{\text{frustum}}$ given by Equation (6.45) causes depths to decrease with distance from the camera, and the available precision is much more evenly distributed over the whole view frustum.

In order to take advantage of the reversing projection, a game engine must make one further adjustment, and that is to reverse the direction of the depth test because objects closer to the camera now write greater values to the depth buffer. A function that constructs the reverse projection matrix $\mathbf{R}_{\text{frustum}}$ and returns a `Matrix4D` data structure is shown in Listing 6.3.

As before, we can push the far plane to infinity for the reversing projection matrix. The result is the matrix $\mathbf{R}_{\text{infinite}}$ given by

$$\mathbf{R}_{\text{infinite}} = \lim_{f \to \infty} \mathbf{R}_{\text{frustum}} = \begin{bmatrix} g/s & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & 0 & n \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{6.46}$$

which is even simpler than $\mathbf{P}_{\text{infinite}}$ as measured by the number of nonzero entries. To prevent floating-point round-off error from assigning slightly negative values of z_{device} to vertex positions at infinity, we can again make a small modification that causes this projection matrix to map depths into the range $[\varepsilon, 1]$. With this change, we obtain

$$\mathbf{R}_{\text{infinite}}^* = \begin{bmatrix} g/s & 0 & 0 & 0 \\ 0 & g & 0 & 0 \\ 0 & 0 & \varepsilon & n(1-\varepsilon) \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{6.47}$$

which is the reversed-depth analog of the matrix $\mathbf{P}_{\text{infinite}}^*$ given by Equation (6.42). A function that constructs the modified reverse infinite projection matrix $\mathbf{R}_{\text{infinite}}^*$ and returns a `Matrix4D` data structure is shown in Listing 6.4.

When a 32-bit floating-point depth buffer is being used in a rendering system that defines the range of z_{device} values in the canonical view volume to be $[0,1]$, there are no disadvantages to using the projection matrix $\mathbf{R}_{\text{infinite}}^*$ as long as having a far plane clipping is not a necessary feature. This matrix can usually serve as the default perspective projection matrix because it provides excellent depth buffer precision, avoids clipping artifacts for objects rendered at infinity, and still enables clipping at the near plane. We only need to keep in mind that the depth test ordinarily passes when z_{device} is *greater* than the value stored in the depth buffer at each pixel.

Listing 6.3. This code creates a 4×4 matrix that represents the reversing perspective projection $\mathbf{R}_{\text{frustum}}$ for a view frustum having a vertical field of view specified by `fovy` and returns it in a `Matrix4D` data structure. The aspect ratio of the viewport is given by `s`, and the distances to the near and far planes are given by `n` and `f`.

```
Matrix4D MakeRevFrustumProjection(float fovy, float s, float n, float f)
{
    float g = 1.0F / tan(fovy * 0.5F);
    float k = n / (n - f);

    return (Matrix4D(g / s, 0.0F, 0.0F, 0.0F,
                     0.0F,  g,    0.0F, 0.0F,
                     0.0F,  0.0F, k,    -f * k,
                     0.0F,  0.0F, 1.0F, 0.0F));
}
```

Listing 6.4. This code creates a 4×4 matrix that represents the reversing infinite perspective projection $\mathbf{R}_{\text{infinite}}^{*}$ for a view frustum having a vertical field of view specified by `fovy` and returns it in a `Matrix4D` data structure. The aspect ratio of the viewport is given by `s`, the distance to the near plane is given by `n`, and the value of ε in Equation (6.47) is given by `e`. If $\varepsilon = 0$, then the returned matrix is $\mathbf{R}_{\text{infinite}}$ given by Equation (6.46).

```
Matrix4D MakeRevInfiniteProjection(float fovy, float s, float n, float e)
{
    float g = 1.0F / tan(fovy * 0.5F);

    return (Matrix4D(g / s, 0.0F, 0.0F,  0.0F,
                     0.0F,  g,    0.0F,  0.0F,
                     0.0F,  0.0F, e,     n * (1.0F - e),
                     0.0F,  0.0F, 1.0F,  0.0F));
}
```

6.3.4 Orthographic Projections

Up to this point, we have been discussing perspective projections in which the viewable volume of space is defined by horizontal and vertical angles opening at a single point, the camera position. In a perspective projection, vertices belonging to visible objects are connected to that one camera position to determine where they fall on the projection plane. There is another kind of projection called an *orthographic* projection in which vertices are projected not to a single point but to the entire plane containing the camera position. An orthographic projection is also known as a *parallel* projection because, as shown in Figure 6.9, the line segments connecting vertices to the projection plane are all parallel to each other.
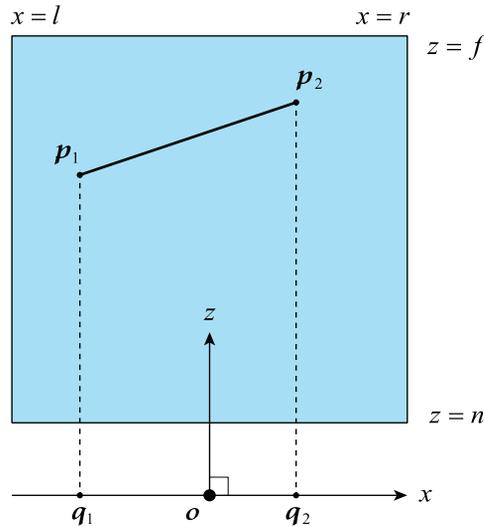
Figure 6.9. In an orthographic projection, vertices p_1 and p_2 in the view volume are projected in the direction parallel to the camera-space z axis onto the points q_1 and q_2 in the plane containing the camera.

The need for an orthographic projection often arises when some point of view can be considered infinitely far away, like the sun. When a shadow map is rendered for such a light source, an orthographic projection is used because the light rays can be considered parallel. (See Section 8.3.) Orthographic projections are also used in tools like level editors to show blueprint views of world geometry.

Compared to perspective projections, the derivation of the matrix used to perform an orthographic projection is very straightforward. There is no perspective divide to consider, and thus no perspective-correct interpolation or depth buffer precision issues to worry about. An orthographic projection simply scales and offsets camera-space coordinates so that one volume of space shaped like a box is resized to another volume of space shaped like a box, the canonical view volume. We describe the boundary of the visible region of camera space by specifying minimum and maximum values along each of the three axes. In the x direction, coordinates range from a left plane at $x = l$ to a right plane at $x = r$. In the y direction, coordinates range from a top plane at $y = t$ to a bottom plane at $y = b$. And in the z direction, coordinates range from a near plane at $z = n$ to a far plane at $z = f$. The orthographic projection matrix $\mathbf{P}_{\text{ortho}}$ is determined by requiring that these ranges are transformed into the range $[-1, +1]$ in the x and y directions and into the range $[0, 1]$ in the z direction, yielding

$$\mathbf{P}_{\text{ortho}} = \begin{bmatrix} \dfrac{2}{r-l} & 0 & 0 & -\dfrac{r+l}{r-l} \\ 0 & \dfrac{2}{b-t} & 0 & -\dfrac{b+t}{b-t} \\ 0 & 0 & \dfrac{1}{f-n} & -\dfrac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{6.48}$$

A function that constructs the projection matrix $\mathbf{P}_{\text{ortho}}$ and returns a `Matrix4D` data structure is shown in Listing 6.5.

Note that there are no requirements on the boundary planes of the view volume in relation to the camera position. In the z direction, the camera can be in front of the near plane, beyond the far plane, or even in between the two planes inside the view volume. The same is true in the x and y directions with respect to the side planes. However, it is common to place the camera at a point on the near plane that is halfway between the left, right, top, and bottom planes. In this case, the orthographic projection matrix simplifies to

$$\mathbf{P}_{\text{ortho}} = \begin{bmatrix} 2/w & 0 & 0 & 0 \\ 0 & 2/h & 0 & 0 \\ 0 & 0 & 1/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6.49}$$

where w, h, and d are the width, height, and depth of the view volume.

Listing 6.5. This code creates a 4×4 matrix that represents the orthographic projection $\mathbf{P}_{\text{ortho}}$ and returns it in a `Matrix4D` data structure. The left, right, top, and bottom sides of the view volume are given by `l`, `r`, `t`, and `b`. The distances to the near and far planes are given by `n` and `f`.

```
Matrix4D MakeOrthoProjection(float l, float r, float t, float b, float n, float f)
{
   float w_inv = 1.0F / (r - l), h_inv = 1.0F / (b - t), d_inv = 1.0F / (f - n);

   return (Matrix4D(2.0F * w_inv,    0.0F,        0.0F, -(r + l) * w_inv,
                    0.0F,     2.0F * h_inv, 0.0F, -(b + t) * h_inv,
                    0.0F,         0.0F,     d_inv,    -n * d_inv,
                    0.0F,         0.0F,      0.0F,       1.0F));
}
```

6.3.5 Frustum Plane Extraction

In clip space, the fixed planes bounding the canonical view volume have a rather simple form, as shown in Figure 6.10. This due to the fact that their normal vectors are all parallel to the coordinate axes and their distances to the origin are always either zero or one. As a consequence, for a given projection matrix $\mathbf{P}$, it is particularly easy to determine what the camera-space planes bounding the view volume must be.

Recall that planes are 4D antivectors and that they transform between coordinate systems in the opposite sense of vectors. The projection matrix $\mathbf{P}$ transforms vectors, regarded as matrices having a single column, from camera space to clip space. The same matrix $\mathbf{P}$, not its inverse, also transforms antivectors, regarded as matrices having a single row, the other way from clip space to camera space. Thus, the plane $\mathbf{f}_{\text{clip}}$ in clip space is transformed into the plane $\mathbf{f}_{\text{camera}}$ in camera space with the formula

$$\mathbf{f}_{\text{camera}} = \mathbf{f}_{\text{clip}}\mathbf{P}. \tag{6.50}$$

The boundary planes of the canonical view volume each have at most two nonzero components, and each one is either $+1$ or -1. This means that the camera-space planes can be extracted from the matrix $\mathbf{P}$ by calculating the sum or difference of
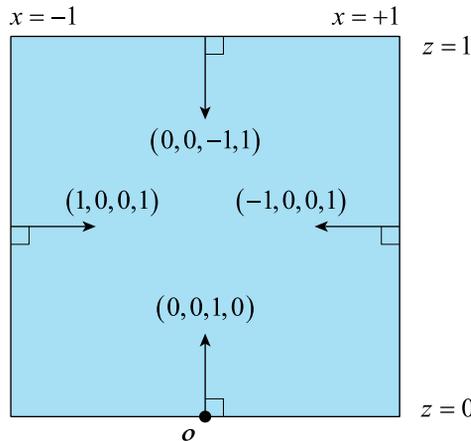
Figure 6.10. This diagram shows four of the six planes bounding the canonical view volume in clip space. Each plane's normal vector is aligned to one of the coordinate axes, and each plane's w coordinate is either zero or one. The top and bottom planes not shown have coordinates $(0,1,0,1)$ and $(0,-1,0,1)$.

two unscaled rows of $\mathbf{P}$ in most cases, with the only exception being the near plane, which is simply equal to the third row of $\mathbf{P}$ by itself.

The six clip-space planes bounding the canonical view volume are listed in Table 6.2 along with the formulas for calculating their camera-space counterparts with the rows of a projection matrix $\mathbf{P}$. These formulas are valid for *any* projection matrix, perspective or orthographic, with a finite or infinite far plane, or with the oblique near plane described in Section 6.4 below. (However, an infinite far plane is extracted from the projection matrix as its dual point at the origin in camera space.) In the case of a reversing projection matrix $\mathbf{R}$ that maps the near plane to one and the far plane to zero, the formulas for near and far plane extraction are swapped, as explicitly shown by separate entries in Table 6.2. Keep in mind that the planes calculated with these formulas will not generally be normalized and may need to be rescaled depending on their intended use.

Frustum Plane	Clip-Space (x, y, z, w)	Camera-Space Formula
Left	$(1,0,0,1)$	$\mathbf{P}_3 + \mathbf{P}_0$
Right	$(-1,0,0,1)$	$\mathbf{P}_3 - \mathbf{P}_0$
Top	$(0,1,0,1)$	$\mathbf{P}_3 + \mathbf{P}_1$
Bottom	$(0,-1,0,1)$	$\mathbf{P}_3 - \mathbf{P}_1$
Conventional Projection Matrix		
Near	$(0,0,1,0)$	$\mathbf{P}_2$
Far	$(0,0,-1,1)$	$\mathbf{P}_3 - \mathbf{P}_2$
Reversing Projection Matrix		
Near	$(0,0,-1,1)$	$\mathbf{R}_3 - \mathbf{R}_2$
Far	$(0,0,1,0)$	$\mathbf{R}_2$

Table 6.2. Clip-space plane coordinates (x, y, z, w) are listed for the six planes bounding the canonical view volume. The formulas for extracting the equivalent camera-space planes are given in terms of the rows of the projection matrix $\mathbf{P}$, where the notation $\mathbf{P}_i$ represents the zero-based i-th row of $\mathbf{P}$. For a projection matrix $\mathbf{R}$ that reverses the direction of depth, the formulas for the near and far planes are swapped.

6.4 Oblique Clipping Planes

In some advanced techniques utilized by various game engines, the final scene is a composite of images rendered from multiple camera views. For example, this happens when the objects visible to the primary camera include things like a mirror, a reflective water surface, or a magical portal through which you can see a distant part of the world. When those objects are rendered, they use secondary images that must have already been generated for completely different camera positions. There is often a boundary plane that separates the secondary camera views from the primary camera view. In the case of a reflection through the world-space boundary plane $\mathbf{k}$, the transform $\mathbf{M}_{camera}$ from camera space to world space used by the primary camera is turned into the transform

$$\mathbf{M}_{reflection} = \mathbf{H}_{reflect}(\mathbf{k})\mathbf{M}_{camera} \tag{6.51}$$

to be used by the camera that first renders the reflected image, where $\mathbf{H}_{reflect}(\mathbf{k})$ is the matrix given by Equation (3.34) that performs a reflection through a plane.

When rendering the secondary images, a problem arises when geometry penetrates the boundary plane. This is exemplified in Figure 6.11(a), where a reflection is rendered for a water surface, and partially submerged stone pillars are drawn into the secondary image with a camera transform that flips everything upside down. The reflection should contain only those parts of the pillars that are above the water surface, but the submerged parts are also visible in the top image of the figure, which is clearly incorrect. One solution to this problem is to enable a user clipping plane to prevent any underwater geometry from being rendered in the reflection image, and this produces the correct result shown in Figure 6.11(b). While effective, a user clipping plane inconveniently requires additional state management and, for each unique material, a separate vertex shader that outputs a clip distance, possibly with a small performance cost. Furthermore, the additional clipping plane is almost always redundant with the near plane because it slices through the view frustum at a distance much farther from the camera. Imagine an upward tilted view from the underwater camera used to render a reflection. Only the geometry above the water surface, beyond the boundary plane from the camera's perspective, should be rendered. Anything that would be clipped by the near plane would certainly also be clipped to a greater extent by the boundary plane.

A user clipping plane can be avoided by employing a trick that modifies the projection matrix in such a way that the conventional near plane is replaced by an arbitrary plane that faces away from the camera. This technique exploits the existing hardware clipping functionality at no additional performance cost and achieves the same results shown in Figure 6.11(b). The only downside is that the far plane

Figure 6.11. Partially submerged stone pillars are rendered with a reflective water surface. (a) No clipping is performed at the boundary plane, and underwater parts are incorrectly shown in the reflection image. (b) The projection matrix is modified so that the near plane is replaced by the boundary plane, and ordinary frustum clipping ensures that only geometry above the water surface are shown in the reflection image.

is adversely affected, even for infinite projection matrices, and this impacts depth precision, but we will be able to minimize the damage.

Let $\mathbf{k}$ be the boundary plane with camera-space coordinates (k_x, k_y, k_z, k_w). This would normally have been transformed into the object space for a camera used to render a secondary image. We assume $k_w < 0$ so that the camera lies on the negative side of the plane. The plane's normal vector points inward with respect to the view frustum as shown in Figure 6.12. For a conventional projection matrix $\mathbf{P}$, one that does not reverse the direction of depths in clip space, the near plane corresponds directly to $\mathbf{P}_2$ according to Table 6.2. (Projection matrices that do reverse depths are discussed below.) We easily accomplish our goal of causing the oblique plane $\mathbf{k}$ to act as the near plane by simply replacing the third row of the projection matrix with $\mathbf{k}$. In the process, however, we also modify the far plane $\mathbf{f} = \mathbf{P}_3 - \mathbf{P}_2$ in an unintuitive way due to its dependence on both the third and fourth rows, and the change is rather destructive.

When the near plane is not perpendicular to the camera-space z axis, the near plane and far plane are no longer parallel to each other, and they intersect at a line contained in the x-y plane. (See Exercise 11.) As a result, replacing the third row of a projection matrix by the arbitrary plane $\mathbf{k}$ causes the far plane $\mathbf{f}$ to be reoriented as shown in Figure 6.12. Clipping no longer occurs at the original far plane, but at this new oddly positioned far plane. Additionally, depth in normalized device coordinates is now dependent on all three of the x, y, and z coordinates in camera
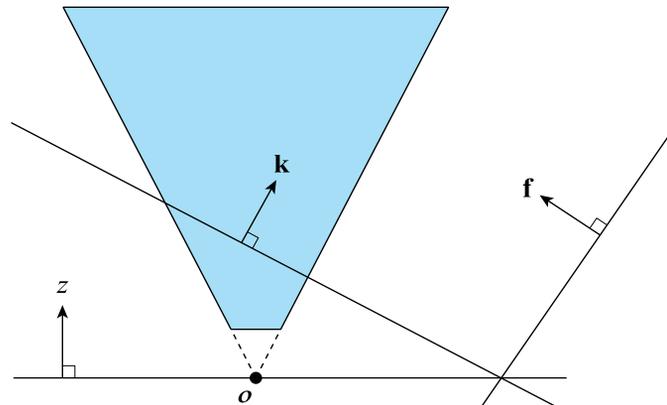
Figure 6.12. The camera-space boundary plane $\mathbf{k}$ slices through the view frustum at an oblique angle. It must be the case that $k_w < 0$ so that the origin o is on its negative side. The far plane $\mathbf{f}$ must intersect the boundary plane $\mathbf{k}$ at a line contained in the x-y plane, and this causes it to have an unusual orientation.

space, and it increases from zero at the plane $\mathbf{k}$ to one at the plane $\mathbf{f}$. Fortunately, we can exert some control over the exact orientation of $\mathbf{f}$ by multiplying $\mathbf{k}$ by a positive scale factor m. Doing so does not change the geometric meaning of the near clipping plane, and it allows us to optimize our usage of the full range of depths under the constraint that $\mathbf{f}$ does not end up chopping off any part of the original view frustum. We assume that the original projection matrix $\mathbf{P}$ has the form

$$\mathbf{P} = \begin{bmatrix} P_{00} & 0 & P_{02} & 0 \\ 0 & P_{11} & P_{12} & 0 \\ 0 & 0 & P_{22} & P_{23} \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{6.52}$$

where the presence of P_{02} and P_{12} makes the following derivation inclusive of off-center perspective projections. The inverse of $\mathbf{P}$ is given by

$$\mathbf{P}^{-1} = \begin{bmatrix} 1/P_{00} & 0 & 0 & -P_{02}/P_{00} \\ 0 & 1/P_{11} & 0 & -P_{12}/P_{11} \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1/P_{23} & -P_{22}/P_{23} \end{bmatrix}. \tag{6.53}$$

In the modified projection matrix, the third row is replaced by the scaled clipping plane $m\mathbf{k}$. Since the far plane is given by the difference between the fourth row and third row, and we cannot change the fourth row, we must have

$$\mathbf{f} = \mathbf{P}_3 - m\mathbf{k}. \tag{6.54}$$

What we would like to do is calculate a value of m that causes the far plane to include the original view frustum, but nothing more. This can be accomplished by forcing $\mathbf{f}$ to contain the camera-space vertex v_{camera} on the boundary of the original view frustum that is most distant from the plane $\mathbf{k}$. This vertex is one of the four corners of the view frustum on the original far plane, and which one depends on the signs of the x and y components of $\mathbf{k}$. In clip space, points on the far plane always have $z_{\text{clip}} = 1$, so we can express the most distant vertex in clip space as

$$v_{\text{clip}} = \left(\text{sgn}\left(\frac{k_x}{P_{00}} \right), \text{sgn}\left(\frac{k_y}{P_{11}} \right), 1, 1 \right). \tag{6.55}$$

Here, k_x and k_y have been transformed into clip space with the matrix $\mathbf{P}^{-1}$. In the case of a projection that does not invert the x or y axes, P_{00} and P_{11} are both positive,

so the signs of the camera-space values of k_x and k_y could simply be used to calculate $\mathbf{v}_{\text{clip}}$, and this is what we'll use as we continue. The camera-space vertex $\mathbf{v}_{\text{camera}}$, in homogeneous coordinates, is then given by

$$\mathbf{v}_{\text{camera}} = \mathbf{P}^{-1}\mathbf{v}_{\text{clip}} = \begin{bmatrix} (\text{sgn}(k_x) - P_{02})/P_{00} \\ (\text{sgn}(k_y) - P_{12})/P_{11} \\ 1 \\ (1 - P_{22})/P_{23} \end{bmatrix}. \tag{6.56}$$

If $\mathbf{P}$ represents an infinite projection matrix, then the resulting w coordinate is zero, and $\mathbf{v}_{\text{camera}}$ can be interpreted as the direction to a point at infinity. The requirement that the far plane $\mathbf{f}$ contain the $\mathbf{v}_{\text{camera}}$ means that the dot product between them must be zero, from which we obtain

$$\mathbf{f} \cdot \mathbf{v}_{\text{camera}} = \mathbf{P}_3 \cdot \mathbf{v}_{\text{camera}} - m\mathbf{k} \cdot \mathbf{v}_{\text{camera}} = 0. \tag{6.57}$$

Using $\mathbf{P}_3 = \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$, this is easily solved for m to arrive at

$$m = \frac{1}{\mathbf{k} \cdot \mathbf{v}_{\text{camera}}}. \tag{6.58}$$

Replacing the third row of $\mathbf{P}$ by the scaled plane $m\mathbf{k}$ produces the view volume shown in Figure 6.13. The near plane has been moved to coincide with the new boundary plane $\mathbf{k}$, and the far plane has been oriented so that it contains the most distant vertex $\mathbf{v}_{\text{camera}}$ of the original view frustum, minimizing the angle between $\mathbf{k}$ and $\mathbf{f}$. The region shaded blue corresponds to the volume of camera space that gets reshaped into the canonical view volume by the modified projection matrix. Nothing in this region is clipped, and it often extends to infinity in some directions regardless of whether the original view frustum has a finite far plane. The vector field shown in the figure represents the gradient of z_{device} after the perspective divide. This field demonstrates how increasing depths follow circular arcs that are perpendicular to both the near plane $\mathbf{k}$, where $z_{\text{device}} = 0$, and the far plane, where $z_{\text{device}} = 1$.

In the case that the original projection matrix is one that reverses the depths in clip space, some minor changes to the above derivation are necessary. First, points on the far plane have z coordinates of zero in clip space, so the value of $\mathbf{v}_{\text{clip}}$ in Equation (6.55) is modified accordingly to

$$\mathbf{v}_{\text{clip}} = (\text{sgn}(k_x), \text{sgn}(k_y), 0, 1). \tag{6.59}$$
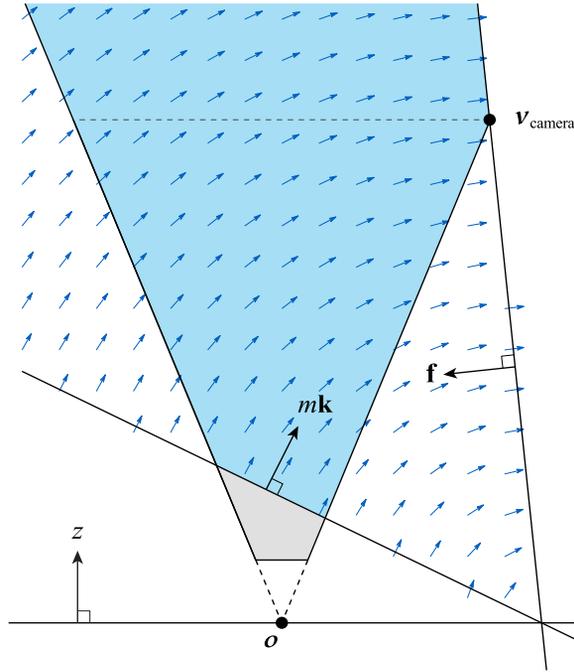
Figure 6.13. The boundary plane **k** can be scaled by a factor m specifically calculated so that the far plane **f** is rotated into the optimal orientation that contains the most distant point v_{camera} on the far plane of the original view frustum. The vector field represents the gradient of z_{device}, and it follows circular arcs around the line where **k** and **f** intersect in the x-y plane.

Calling the reversing projection matrix **R**, the value of v_{camera} is now given by

$$v_{\text{camera}} = \mathbf{R}^{-1} v_{\text{clip}} = \begin{bmatrix} \left(\operatorname{sgn}(k_x) - R_{02} \right)/R_{00} \\ \left(\operatorname{sgn}(k_y) - R_{12} \right)/R_{11} \\ 1 \\ R_{22}/R_{23} \end{bmatrix}, \tag{6.60}$$

which differs from the value in Equation (6.56) only in the w coordinate. As shown in Table 6.2, the near plane for a reversing projection matrix is equal to $\mathbf{R}_3 - \mathbf{R}_2$, and this is the quantity must be replaced by $m\mathbf{k}$, so we have the relationship

$$\mathbf{R}_2 = \mathbf{R}_3 - m\mathbf{k} = \left(-mk_x, -mk_y, 1 - mk_z, -mk_w \right). \tag{6.61}$$

Since $\mathbf{R}_2$ is equal to the far plane, the value of m is calculated in exactly the same way as in Equations (6.57) and (6.58).

Functions that modify a conventional projection matrix $\mathbf{P}$ and a reversing projection matrix $\mathbf{R}$ to have an oblique near clipping plane are shown in Listing 6.6. The w coordinate of v_{camera} is calculated with a slightly different formula in each case. The only other way in which the two functions are not identical is that the third row of $\mathbf{P}$ is replaced by $m\mathbf{k}$, but the third row of $\mathbf{R}$ is replaced by $\mathbf{R}_3 - m\mathbf{k}$.

Listing 6.6. These two functions modify the conventional projection matrix P and the reversing projection matrix R in place so either type has the oblique near clipping plane k and the best far plane orientation possible.

```
void ModifyProjectionNearPlane(Matrix4D& P, const Plane& k)
{
    Vector4D vcamera((sgn(k.x) – P(0,2)) / P(0,0),
                     (sgn(k.y) – P(1,2)) / P(1,1),
                     1.0F, (1.0F – P(2,2)) / P(2,3));

    float m = 1.0F / Dot(k, vcamera);
    P(2,0) = m * k.x;
    P(2,1) = m * k.y;
    P(2,2) = m * k.z;
    P(2,3) = m * k.w;
}

void ModifyRevProjectionNearPlane(Matrix4D& R, const Plane& k)
{
    Vector4D vcamera((sgn(k.x) – R(0,2)) / R(0,0),
                     (sgn(k.y) – R(1,2)) / R(1,1),
                     1.0F, R(2,2) / R(2,3));

    float m = –1.0F / Dot(k, vcamera);
    R(2,0) = m * k.x;
    R(2,1) = m * k.y;
    R(2,2) = m * k.z + 1.0F;
    R(2,3) = m * k.w;
}
```

Exercises for Chapter 6

1. Let $\mathbf{M}_{\text{camera}}$ be the transform from object space to world space for a camera, and consider a view frustum with projection plane distance g, aspect ratio s, and distances n and f to the near and far planes. Find formulas for the six bounding planes of the view frustum in terms of the columns of $\mathbf{M}_{\text{camera}}$.

2. Let $\mathbf{l}$, $\mathbf{r}$, $\mathbf{t}$, and $\mathbf{b}$ be the world-space left, right, top, and bottom planes for a view frustum. Find formulas using Plücker coordinates $\{\mathbf{v} \,|\, \mathbf{m}\}$ that express the four lines at which these planes intersect in world space such that the direction $\mathbf{v}$ of each line points away from the camera position. Express the same four lines using the antiwedge product to form 4D bivectors.

3. Determine the camera-space coordinates for the eight vertices of a view frustum having aspect ratio s, projection plane distance g, near plane distance n, and far plane distance f.

4. Suppose that the view frustum having aspect ratio s and projection plane distance g is circumscribed by a right circular cone with its apex at the camera position. (a) Given the cone's height h (along the z axis), find a formula for the cone's radius r at its base. (b) Determine the sine of the angle α between the z axis and the lateral surface of the cone.

5. Suppose that a camera is looking in the horizontal direction, parallel to the ground, and that the view frustum has a projection plane distance g and a near plane distance n. Calculate the minimum height above the ground that the camera position must have to avoid the ground being clipped by the near plane inside the view frustum.

6. Calculate the inverses of the projection matrices $\mathbf{P}_{\text{infinite}}$, $\mathbf{P}^*_{\text{infinite}}$, $\mathbf{R}_{\text{frustum}}$, $\mathbf{R}_{\text{infinite}}$, and $\mathbf{R}^*_{\text{infinite}}$.

7. Derive an off-center projection matrix similar to $\mathbf{P}_{\text{frustum}}$ but for which the viewport extends from $x = l$ to $x = r$ and from $y = t$ to $y = b$ on the projection plane at a distance g from the camera. This rectangle should correspond to the range $[-1, +1]$ in both the x and y directions after the perspective divide.

8. Derive a projection matrix similar to $\mathbf{P}_{\text{frustum}}$ but for which the canonical view volume extends from -1 to $+1$ in the z direction. That is, for a near plane distance n and far plane distance f, the camera-space range $[n, f]$ is transformed into the clip-space range $[-1, +1]$ after the perspective divide.

9. Using the result from the previous exercise, calculate a projection matrix similar to $\mathbf{P}_{\text{infinite}}$ but for which the camera-space range of z coordinates $[n, \infty]$ is transformed into the clip-space range $[-1, +1]$ after the perspective divide.

10. Suppose that the projection matrix $\mathbf{P}_{\text{frustum}}$ has been modified by replacing its third row with the scaled plane $m\mathbf{k}$, where m is given by Equation (6.58). Show that setting $\mathbf{k} = (0, 0, 1, -n)$ recovers the original matrix $\mathbf{P}_{\text{frustum}}$ because it is then the case that $m = f/(f - n)$.

11. Show that if the third row of a perspective projection matrix is not perpendicular to the camera-space z axis, then the near plane and far plane must intersect at a line $\{\mathbf{v} \,|\, \mathbf{m}\}$ that lies in the x-y plane.

12. For a projection matrix that has an oblique near clipping plane $\mathbf{k}$, consider the scalar field $z_{\text{device}}(\mathbf{p})$ in camera space defined as the z coordinate produced by projecting the point $\mathbf{p}$ and performing the perspective divide. Let $\{\mathbf{v} \,|\, \mathbf{m}\}$ be the line where the near and far planes intersect, and let $\mathbf{q}$ be the point on this line that is closest to $\mathbf{p}$. Prove that the gradient of $z_{\text{device}}(\mathbf{p})$, as shown for a particular configuration in Figure 6.13, is perpendicular to both $\mathbf{v}$ and $\mathbf{p} - \mathbf{q}$.

13. Let $\mathbf{P}$ be a perspective projection matrix for which the camera-space range of z coordinates $[n, f]$ between the near and far planes is transformed into the clip-space range $[-1, +1]$ after the perspective divide. For an arbitrary clipping plane $\mathbf{k}$, determine what the third row of $\mathbf{P}$ should be replaced by so that the near plane is repositioned to coincide with $\mathbf{k}$ and the far plane optimally includes the original view frustum.

14. Suppose that the third and fourth rows of a projection matrix have the generic form given by

$$\begin{bmatrix} A & B & C & D \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

If we wanted to modify this matrix so that the depths it produced were offset by a constant distance δ_{device} after the perspective divide, we could replace C by $C + \delta_{\text{device}}$. However, this would correspond to varying offsets in camera space depending on the location of a point $\mathbf{p}$. Derive a formula for δ_{device} that would produce a depth offset equivalent to the change in depth produced by transforming the point $(p_x, p_y, p_z - \delta_{\text{camera}})$ with the original projection matrix for a given camera-space offset of δ_{camera} at the specific coordinates of $\mathbf{p}$. (We subtract δ_{camera} so that positive values corresponds to offsets toward the camera for points in the view frustum.)

Shading

Once a camera transform and projection matrix have been established, we can calculate the positions of a model's vertices in the viewport. We know exactly where the triangles making up the geometric shape of the model are going to be rendered and what pixels they will cover on the screen. *Shading* is the process through which we decide what color is ultimately written to each of those pixels. For every pixel inside each triangle, a pixel shader is run to calculate the apparent brightness and color at the corresponding point on the surface of the model. We calculate these quantities by considering all of the light reaching that point and applying properties of the surface's material to determine how much of the light is reflected toward the camera.

Shading calculations are performed primarily by a pixel shader, but it almost always receives interpolated information that was generated earlier in the graphics pipeline, usually by the vertex shader. A typical pixel shader can be divided into three components representing different parts of the path that light takes before it is detected by the camera. First, the pixel shader determines the color and brightness of the incoming light when reaches the surface. This accounts for the possibility that the light was blocked by some intervening geometry, in which case the point being shaded is in shadow. The pixel shader next performs its reflection calculations to figure out how much light leaves the surface in the direction pointing toward the camera. Finally, the pixel shader may apply fog calculations to account for interactions with the medium filling the space between the surface and the camera. In this chapter, we discuss only the component that handles surface reflection, and we save the other two components for Chapter 8. For now, we assume that the brightness of the light reaching the surface is already known, and we keep in mind that fog can later be applied to the reflection.

7.1 Rendering Fundamentals

In Section 5.2, we discussed what it means for light to have a particular color, and we used the luminance Y to describe its brightness. In that setting, however, the luminance was normalized to the range $[0,1]$ of displayable intensity levels, and it didn't really have any physical meaning. When we render a scene, we are mainly concerned with how much light is bouncing around, how it interacts with different surfaces, and how bright it is when it reaches the camera. Before we can implement a rendering system, we first need to quantify exactly what we mean when we say things like "how much light" and "how bright". Then we need to understand how difficult it is to account for all of the light interacting with the scene and devise practical methods that produce acceptable approximations to physical reality.

7.1.1 Luminance

Suppose we are rendering a point p on a surface that is visible in the view frustum from a camera at the position c. A vector $\mathbf{v}$, called the *view vector*, is defined as

$$\mathbf{v} = \text{nrm}\left(c - p\right). \tag{7.1}$$

This vector points toward the camera from the surface, and it appears throughout the mathematics used in shading. The point p represents the tiny area on the surface that projects onto one pixel in the viewport. To assign a brightness to the pixel, we need to perform some kind of calculation producing a result that tells us how much light leaves the surface at p and travels specifically in the direction $\mathbf{v}$ toward the camera.

We measure the perceived brightness at the point p by first considering that some amount of luminous flux Φ_{V} is exiting the surface at every point in a small area dA surrounding the point p. This is illustrated in Figure 7.1 by the rays leaving the surface in many different directions from various locations. To determine how much of that visual power comes from a specific area, like the footprint of a single pixel, we must be able to express the power emitted per unit area as a function of position. This quantity is called the *luminous exitance M_{V}*, which is defined as

$$M_{\text{V}}\left(p\right) = \frac{d\Phi_{\text{V}}}{dA}. \tag{7.2}$$

The luminous exitance includes all of the light that leaves the surface in every possible direction, but we are only interested in the light traveling in the small
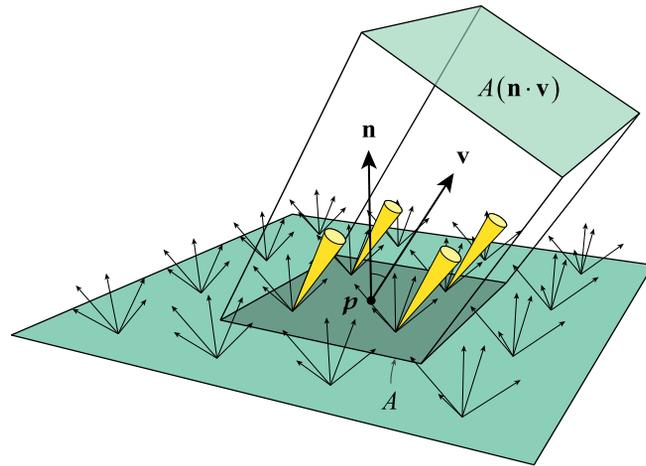
Figure 7.1. The luminous flux exiting a surface having normal vector **n** is represented by many rays surrounding the point $\boldsymbol{p}$. The darker region corresponds to the area A covered by one pixel, and the yellow cones indicate the bundle of directions that reach the pixel on the view frustum's projection plane. The luminance in the direction **v** is given by the amount of power, in lumens, emitted per steradian within the yellow cones and per square meter through the projected area $A(\mathbf{n} \cdot \mathbf{v})$.

bundle of directions within a solid angle $d\omega$ that leads to the area on the view frustum's projection plane covered by the pixel being rendered. This is illustrated by the yellow cones in Figure 7.1 that are aligned to the view vector **v**. Light traveling in other directions is not directly observed coming from the point $\boldsymbol{p}$, but it may bounce off other surfaces and eventually be reflected toward the camera from some other point in space. The power per unit area contained in the luminous exitance is distributed over a solid angle representing all of the directions in which light could leave the surface. This angle is typically the 2π steradians in a hemisphere aligned to the normal vector. The *luminance L_V* is a function of position $\boldsymbol{p}$ and direction **v** giving the amount of power emitted per unit solid angle per unit area, and it is defined as

$$L_V(\boldsymbol{p}, \mathbf{v}) = \frac{d^2 \Phi_V}{d\omega \, dA \cos \theta}. \tag{7.3}$$

The luminance basically tells us how much power is being emitted in a single direction from a single point. The angle θ appearing in Equation (7.3) is the angle

between the normal vector $\mathbf{n}$ and the view vector $\mathbf{v}$. Assuming that $\mathbf{n}$ and $\mathbf{v}$ are normalized, we can also write the definition of luminance as

$$L_{\mathrm{V}}(\boldsymbol{p}, \mathbf{v}) = \frac{d^2\Phi_{\mathrm{V}}}{d\omega\, dA\,(\mathbf{n}\cdot\mathbf{v})}. \tag{7.4}$$

Multiplying the area by $\mathbf{n}\cdot\mathbf{v}$ has the effect of projecting it onto a plane perpendicular to the direction $\mathbf{v}$, as shown in Figure 7.1. This accounts for the fact that the light emitted from an area A on a surface is concentrated into a smaller area from the perspective of the camera as the angle between $\mathbf{n}$ and $\mathbf{v}$ increases.

Luminance is a fundamental measurement in computer graphics, and it is the quantity that we will always calculate in our pixel shaders. Luminance has units of lumens per steradian per square meter ($\mathrm{lm}\cdot\mathrm{sr}^{-1}\cdot\mathrm{m}^{-2}$). We will discuss another unit called the *candela* in Section 8.1, and it is equivalent to lumens per steradian. It is common to see luminance expressed in terms of this unit as candelas per square meter (cd/m^2). The units of luminance are also called *nits* (nt), which are often used when describing the brightness of pixels on a display. In the sRGB standard, the luminance Y of a color is normalized so that a value of $Y = 1$ corresponds to an physical luminance L_{V} of 80 nt. This means that calculated values of L_{V} should be divided by 80 before being incorporated into a displayed color.

We can think of luminance as either leaving a surface or arriving at a surface, and this allows us to quantify the amount of luminous flux that is transferred from one point to another along the direction between them. We are particularly interested in how much power reaches the area of a pixel on the projection plane from the corresponding area on the surface we are rendering for that pixel. Suppose that we have calculated the outgoing luminance L_{surface} at a point $\boldsymbol{p}$ in the direction $\mathbf{v}$ toward the camera, and let $\boldsymbol{q}$ be the point where the line connecting $\boldsymbol{p}$ to the camera position $\boldsymbol{c}$ intersects the projection plane. As shown in Figure 7.2(a), the solid angle $\omega_{\mathrm{surface}}$ through which light emitted from the surface strikes the pixel is the angle subtended by the area A_{pixel} of the pixel. This angle represents the only set of directions in which light leaving the surface affects the brightness of the pixel. We must collect the light emitted in these directions over the area A_{surface}, shown separately in Figure 7.2(b), consisting of all points on the surface that project onto the pixel when connected by lines to the camera position $\boldsymbol{c}$. Assuming that the luminance L_{surface} is constant over a small solid angle $\omega_{\mathrm{surface}}$ and small area A_{surface}, the luminous flux Φ_{V} transmitted toward the pixel is given by

$$\Phi_{\mathrm{V}} = L_{\mathrm{surface}}\,\omega_{\mathrm{surface}}\,A_{\mathrm{surface}}^{\perp}, \tag{7.5}$$
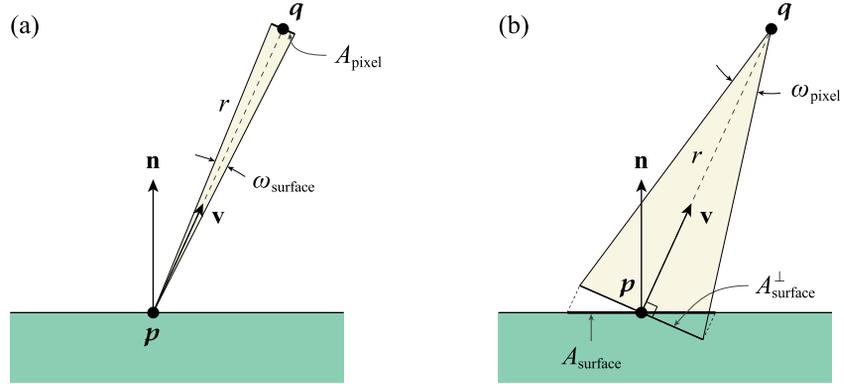
Figure 7.2. (a) Light leaves a surface at the point p in the directions that strike a pixel of area A_{pixel} at the point q, and this area subtends the solid angle ω_{surface}. (b) Light arrives at the pixel in the directions coming from the area A_{surface} covered by the pixel. The projection $A_{\text{surface}}^{\perp}$ of that area perpendicular to the view direction $\mathbf{v}$ subtends the solid angle ω_{pixel}.

where $A_{\text{surface}}^{\perp} = A_{\text{surface}} \left(\mathbf{n} \cdot \mathbf{v} \right)$ is the projection of the area onto a plane perpendicular to $\mathbf{v}$. On the receiving end at the point q, we can perform a similar calculation to determine the incoming luminance L_{pixel}. As shown in the figure, let ω_{pixel} be the solid angle subtended by the area $A_{\text{surface}}^{\perp}$. Then the luminous flux received at the pixel is given by

$$\Phi_{\text{V}} = L_{\text{pixel}} \omega_{\text{pixel}} A_{\text{pixel}}, \tag{7.6}$$

which we know is equal to the luminous flux Φ_{V} calculated in Equation (7.5) because it was limited to the exact amount of light directed toward the pixel. In the limit as the angles and areas become infinitesimal quantities $d\omega$ and dA, we can state the geometric relationships

$$dA_{\text{pixel}} = d\omega_{\text{surface}} r^{2} \quad \text{and} \quad dA_{\text{surface}}^{\perp} = d\omega_{\text{pixel}} r^{2}, \tag{7.7}$$

where r is the distance from p to q. For both of these equations to be true simultaneously, we must have

$$d\omega_{\text{surface}} \, dA_{\text{surface}}^{\perp} = d\omega_{\text{pixel}} \, dA_{\text{pixel}}. \tag{7.8}$$

We conclude that the luminance L_{surface} appearing in Equation (7.5) and the luminance L_{pixel} appearing in Equation (7.6) are equal. This means that we can express the power received by a pixel as

$$\Phi_{\text{V}} = L_{\text{surface}} \omega_{\text{pixel}} A_{\text{pixel}}, \tag{7.9}$$

which is independent of the distance to the surface and the area on the surface covered by the pixel. The values of ω_{pixel} and A_{pixel} are constant once the shape of the view frustum and resolution of the viewport have been established. If we have calculated L_{surface}, then we know exactly how much power is received at the pixel, and that tells us how bright it should be.

The result that we just derived demonstrates an intrinsic property possessed by every ray of light. The luminance carried by a ray through empty space is a constant quantity everywhere along the ray. (If the space is not empty, then the luminance can be absorbed or scattered, and this is the topic of Section 8.5.) When some amount of light leaves a point p with luminance L and travels to some other point q, the luminance the light has when it arrives at q is still the same value of L that it started with, regardless of how much distance was covered. This is what makes luminance a fundamental measurement of what amount of light is traveling between two points.

7.1.2 The Rendering Equation

The luminance $L_{\mathrm{out}}\left(p, \mathbf{v}\right)$ leaving a point p in a direction $\mathbf{v}$ can be a mixture of light coming from two sources. Some amount of light could be emitted by the surface itself, and this is easy to model as a function $L_{\mathrm{emission}}\left(p, \mathbf{n}, \mathbf{v}\right)$, where $\mathbf{n}$ is the surface normal. Everything else included in the luminance is due to the total sum of the portion of light $L_{\mathrm{in}}\left(p, \mathbf{l}\right)$ striking the surface from every direction $\mathbf{l}$ that ends up being reflected toward the camera. This is expressed by the equation

$$L_{\mathrm{out}}\left(p, \mathbf{v}\right) = L_{\mathrm{emission}}\left(p, \mathbf{n}, \mathbf{v}\right) + \int_{\Omega} L_{\mathrm{in}}\left(p, \mathbf{l}\right) f_{\mathrm{reflect}}\left(p, \mathbf{n}, \mathbf{v}, \mathbf{l}\right) \mathrm{sat}\left(\mathbf{n} \cdot \mathbf{l}\right) d\omega, \quad (7.10)$$

which is known as the *rendering equation*. The function $f_{\mathrm{reflect}}\left(p, \mathbf{n}, \mathbf{v}, \mathbf{l}\right)$ is a general placeholder for the calculations that describe the appearance of a surface by considering its *material* properties. This function can be very complicated, but it ultimately specifies how much light reaching the point p from the direction $\mathbf{l}$ is reflected in the direction $\mathbf{v}$, taking into account the normal vector $\mathbf{n}$. The meaning of the factor $\mathrm{sat}\left(\mathbf{n} \cdot \mathbf{l}\right)$ is discussed below in Section 7.2.

Equation (7.10) is an ideal description of the physical behavior of light in a scene, and it cannot be solved. Consider the reality that the luminance $L_{\mathrm{in}}\left(p, \mathbf{l}\right)$ arriving from any direction $\mathbf{l}$ must be equal to the value of $L_{\mathrm{out}}\left(p + u\mathbf{l}, -\mathbf{l}\right)$ leaving the point $p + u\mathbf{l}$ in the opposite direction on some other surface at a distance u away. That value of L_{out} is itself given by Equation (7.10) based on all of the incoming light at the point $p + u\mathbf{l}$, which may even include light coming back from the point p! This recursive process becomes absurdly complex as light continues

bouncing around. Instead of trying to account for all of these interactions directly, all practical rendering systems employ simplified illumination models to generate an approximation of a scene's true lighting within the time constraints imposed by the application.

Game engines typically split the task of rendering a scene into two major components. One component accounts for *direct* illumination from a set of discrete light sources in the scene. It handles only light that is able follow a direct path from its source to the point p being shaded, and it does not consider interactions with other surfaces that could cause more light to arrive at the same point through indirect paths. The other component accounts for *ambient* illumination, and this encompasses most of the complexities in the rendering equation because it accounts for all of the indirect lighting. The shaded color C_{shaded} generated at a point p by the combination of these two components can be expressed as

$$C_{\text{shaded}} = f_{\text{ambient}}\left(C_{\text{ambient}}, \mathbf{v}\right) + \sum_{k=1}^{n} f_{\text{direct}}\left(C_{\text{illum}}^{k}, \mathbf{n}, \mathbf{v}, \mathbf{l}_{k}\right). \qquad (7.11)$$

The output color C_{shaded} is an RGB color that incorporates a luminance value. To reduce clutter, the point p has been left out of this equation, but f_{ambient} and f_{direct} are still functions of position.

The function f_{ambient} in Equation (7.11) represents the contribution from ambient illumination, and more generally, from *any* source that is not attributed to direct illumination. For example, emission from a glowing material is included in the ambient term. Because it is composed of a mixture of light that may have undergone many random bounces, the ambient illumination present throughout the environment tends to vary smoothly as both a function of position and direction. The most basic implementation of ambient illumination makes the assumption that the luminance due to all indirect lighting has averaged out to a constant color C_{ambient} that is present everywhere and shines with equal brightness in every direction. In this case, the effect of C_{ambient} is simply added to the final color produced by other parts of the shading calculation. In more sophisticated implementations, the amount of ambient light changes with position and direction. The luminance due to indirect lighting is often precalculated for one or more directions at some set of sample positions distributed throughout the world. Those precalculated values are then interpolated during the rendering process to reconstruct a smoothly varying ambient function. This does not capture the changes in ambient light on a small scale, however, so ambient occlusion techniques such as the one presented in Section 10.5 are often used to handle the fine details.

The sum in Equation (7.11) represents the contribution from direct illumination by n separate light sources. The function f_{direct} calculates how much luminance coming from the direction $\mathbf{l}_k$ pointing toward the k-th light source is reflected in the direction $\mathbf{v}$ toward the camera. Since all of the light comes from one direction, we can assume that the incoming luminance $L_{\text{in}}^k (\boldsymbol{p}, \mathbf{l}_k)$ has already been integrated over the hemisphere Ω to produce the color

$$C_{\text{illum}}^k = \int_{\Omega} L_{\text{in}}^k (\boldsymbol{p}, \mathbf{l}_k) \, d\omega. \tag{7.12}$$

(Here, k is used as a superscript index and not as an exponent.) This color has units of lumens per square meter, and it represents the *illuminance* E_V at the point $\boldsymbol{p}$ due to the light source. Illuminance is a measure of how brightly a surface is lit, and it is discussed in detail in Section 8.1. In this chapter, we assume that a light's illuminance has already been calculated at any point $\boldsymbol{p}$ where it strikes a surface.

Using the illuminance C_{illum}^k, the function f_{direct} in Equation (7.11) usually has the form

$$f_{\text{direct}} \left(C_{\text{illum}}^k, \mathbf{n}, \mathbf{v}, \mathbf{l}_k \right) = C_{\text{illum}}^k f_{\text{BRDF}} (\mathbf{n}, \mathbf{v}, \mathbf{l}_k) \operatorname{sat} (\mathbf{n} \cdot \mathbf{l}_k), \tag{7.13}$$

where the function f_{BRDF} is called the *bidirectional reflectance distribution function (BRDF)*. The BRDF is a property of the surface material, and it turns an incoming illuminance from the direction $\mathbf{l}_k$ into an outgoing luminance in the direction $\mathbf{v}$. This means that a BRDF has units of inverse steradians (sr^{-1}). In general, the BRDF describes how incoming light is redistributed to every possible outgoing direction. Through the application of texture mapping, discussed below in Section 7.4, the BRDF is a function of position. It has the word "bidirectional" in its name because any physically plausible BRDF would have to be symmetric under exchange of the directions $\mathbf{v}$ and $\mathbf{l}_k$. That is, if a fraction f of the light arriving at the surface from the direction $\mathbf{l}_k$ gets reflected in the direction $\mathbf{v}$, then it must also be true that the same fraction f of light arriving from the direction $\mathbf{v}$ gets reflected in the direction $\mathbf{l}_k$. This property is known as *Helmholtz reciprocity*.

7.2 Diffuse Reflection

Many surfaces that may appear to be smooth actually have a rough geometric structure on a microscopic scale. This tends to cause any light incident on the surface to be reflected in random directions from different points. On a larger scale, the incoming light is redistributed over the entire hemisphere of outgoing directions, and this is called *diffuse reflection*. The simplest way to model diffuse reflection is to

assume that the outgoing distribution is *uniform*, meaning that light is reflected equally in all possible directions. The BRDF describing such a perfectly matte material is extremely easy to implement because it is just a constant value for every view vector **v**.

Suppose that light arrives at a surface from the direction **l** through a beam of cross-sectional area A, as shown in Figure 7.3. Given the illuminance C_{illum} of the incoming light, the power carried by the beam is equal to $C_{\text{illum}}A$. If the light direction makes an angle θ with the normal vector **n**, then the area on the surface struck by the beam is given by $A/\cos\theta$. The power contained in the beam is thus spread out over a larger area, so the actual illuminance E_V on the surface is equal to

$$E_V = \frac{C_{\text{illum}}A}{A/\cos\theta} = C_{\text{illum}}\cos\theta. \tag{7.14}$$

We never need to calculate the angle θ directly because its cosine is given by the simple dot product $\mathbf{n}\cdot\mathbf{l}$, as long as these two vectors have unit length. In a pixel shader, we calculate the illuminance as

$$\boxed{E_V = C_{\text{illum}}\,\text{sat}\left(\mathbf{n}\cdot\mathbf{l}\right),} \tag{7.15}$$

where the saturation prevents a negative illuminance from being calculated for surfaces that face away from the light source. The dependence on the cosine of the
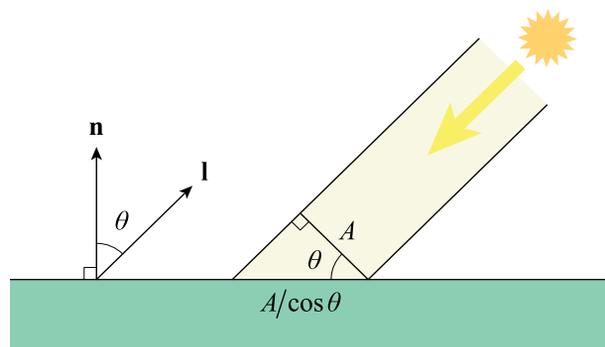
Figure 7.3. A surface is illuminated by a beam of light having cross-sectional area A coming from the direction **l**. The area receiving the light on the surface is larger by a factor of $1/\cos\theta$, where θ is the angle between **l** and the surface normal **n**. The illuminance, equal to the power received per unit area, decreases by a factor of $\cos\theta$ because the light is spread out over a larger area.

angle θ was introduced by Johann Heinrich Lambert (1728–1777), and Equation (7.14) is called *Lambert's cosine law* in his honor. Our slightly modified version of the cosine law in Equation (7.15) is the source of the factor sat $(\mathbf{n} \cdot \mathbf{l})$ appearing in Equations (7.10) and (7.13).

The light leaving the surface also obeys Lambert's cosine law, but in reciprocal form. Suppose that light is leaving an area A on the surface in the direction $\mathbf{v}$ toward the camera. Then the area of the exiting beam is given by $A(\mathbf{n} \cdot \mathbf{v})$. However this area perpendicular to the direction of travel is exactly what appears in the definition of luminance in Equation (7.4). Since luminance is the quantity that we need measure at the camera, we never actually use the dot product $\mathbf{n} \cdot \mathbf{v}$ in our shading calculations to account for the cosine law. The illuminated area appears to have the same brightness from every viewing angle.

A surface that redistributes light uniformly under diffuse reflection is called a *Lambertian* surface. Because it has been spread out over a range of directions, the outgoing light is never as bright as the incoming light that originated from only a single direction. Furthermore, some of the incoming light is not reflected at all because it is absorbed by the surface. The fraction of light that does get reflected is called the *albedo* of the surface's material, which we denote by ρ. This property of a material depends on the wavelength of light, and we specify different values of albedo for red, green, and blue primaries in order to assign a color C_{diffuse} to a material. We can keep this color separate from the albedo ρ by requiring C_{diffuse} to have a luminance Y of one. Then the wavelength-dependent albedo is given by the product ρC_{diffuse}.

For a material to be physically plausible, the amount of light reflected by a surface cannot exceed the amount of light incident upon it. Assuming for the moment that a material exhibits the maximum possible albedo $\rho = 1$ with a perfectly white color C_{diffuse}, the total luminous exitance M_V leaving a point on the surface must be equal to the total illuminance E_V arriving at that point. Let L represent the constant luminance reflected over all directions by the surface. We can determine what the value of L must be by integrating it over the solid angle Ω corresponding to the hemisphere of directions centered on the surface normal. This gives us the equation

$$M_V = \int_\Omega L \cos \theta \, d\omega = E_V, \tag{7.16}$$

where θ is the angle between the surface normal and the direction of the differential solid angle ω. This equation expands to the double integral

$$\int_0^{2\pi} \int_0^{\pi/2} L \cos \theta \sin \theta \, d\theta \, d\varphi = E_V, \tag{7.17}$$

from which we obtain

$$L = \frac{E_{\mathrm{V}}}{\pi}. \tag{7.18}$$

This result tells us that the maximum outgoing luminance in every direction is equal to the incoming illuminance over π steradians. The BRDF for a Lambertian surface must therefore be given by the constant function

$$f_{\mathrm{BRDF}}\left(\mathbf{n}, \mathbf{v}, \mathbf{l}\right) = \frac{\rho}{\pi} C_{\mathrm{diffuse}}. \tag{7.19}$$

When we plug this into Equation (7.13), we have

$$\boxed{f_{\mathrm{direct}}\left(C_{\mathrm{illum}}, \mathbf{n}, \mathbf{l}, \mathbf{v}\right) = \frac{\rho}{\pi} C_{\mathrm{diffuse}} C_{\mathrm{illum}} \ \mathrm{sat}\left(\mathbf{n} \cdot \mathbf{l}\right)} \tag{7.20}$$

as the shading formula for diffuse reflection. It is common for the various factors in the constant supplied by Equation (7.19) to be rolled into a single color value that is simply called the diffuse color of a material. In that case, the diffuse reflection is given by the product of the incoming illuminance, the diffuse color, and $\mathrm{sat}\left(\mathbf{n} \cdot \mathbf{l}\right)$.

Equation (7.18) tells us that to take light arriving from a single direction and distribute it uniformly over a hemisphere, we divide by π. This equation also holds for the reverse process. If we were to collect light arriving in equal amounts from every direction in a hemisphere and reflect it in a single direction, then we would multiply by π. In the case of diffusely reflected ambient light, both of these transformations take place. We multiply by π to gather the ambient light from every direction, but then we immediately divide by π to redistribute the ambient light right back to all of the same directions it came from. The ambient term for a Lambertian surface is thus given by

$$f_{\mathrm{ambient}}\left(C_{\mathrm{ambient}}, \mathbf{v}\right) = \rho C_{\mathrm{diffuse}} C_{\mathrm{ambient}}, \tag{7.21}$$

where C_{ambient} is the luminance of the ambient light. If the diffuse color incorporates the factor ρ/π, then we need to multiply by π to calculate the ambient reflection. To avoid having to do this, the factor of π is sometimes included in the ambient light color C_{ambient}.

The pixel shader code shown in Listing 7.1 implements diffuse shading for a single direct light source and flat ambient illumination using Equations (7.20) and

(7.21). An example showing a model shaded by this code is provided in Figure 7.4. The object-space normal vector **n** and light direction **l** should be output by the vertex shader and interpolated over a triangle for use in the pixel shader. Before we calculate the dot product $\mathbf{n} \cdot \mathbf{l}$ in the pixel shader, we must normalize both **n** and **l**. The normal vector **n** would ordinarily be provided as a vertex attribute that is already normalized to unit length, but it can become shorter when it is interpolated, so it needs to be normalized again in the pixel shader. The light direction **l** is often calculated in the vertex shader as the difference between the light position and the vertex position. It's important that this direction is *not* normalized in the vertex shader so that the correct difference in positions is interpolated over a triangle. The light position must also be specified in object space. For an object having the transform $\mathbf{M}_{\text{object}}$ and a light having the transform $\mathbf{M}_{\text{light}}$, the object-space position of the light is given by $\mathbf{M}_{\text{object}}^{-1}\mathbf{M}_{\text{light}[3]}$.

Listing 7.1. This pixel shader function calculates the ambient and direct shading terms for diffuse reflection from a Lambertian surface. The surface normal n and light direction l are the interpolated values output by the vertex shader, and they must be normalized before being passed to this function. The uniform constant diffuseColor holds the color $(\rho/\pi)C_{\text{diffuse}}$, the constant ambientColor holds the color πC_{ambient}, and the constant lightColor holds the color C_{illum}.

```
uniform float3  diffuseColor;    // (rho / pi) * C_diffuse
uniform float3  ambientColor;    // pi * C_ambient
uniform float3  lightColor;      // C_illum

float3 CalculateDiffuseReflection(float3 n, float3 l)
{
    float3 directColor = lightColor * saturate(dot(n, l));
    return ((ambientColor + directColor) * diffuseColor);
}
```

7.3 Specular Reflection

Of course, perfectly matte surfaces are not what we experience in the real world. Virtually all materials tend to reflect light more strongly near the direction **r** that represents the reflection of the light direction **l** through the surface normal **n**. This phenomenon is called *specular reflection*, and the highlight that it creates on a surface is called a *specularity*. If a material exhibited only specular reflection, then it would be a perfect mirror, and light coming from the direction **l** would be visible only if the reflected direction **r** lined up with the view vector **v**. The properties of

Figure 7.4. On the left, a barrel is illuminated by ambient light only. On the right, the barrel is illuminated by ambient light and one direct light source, the direction to which increases by 30 degrees to the right in each image. Lambert's cosine law causes the shading to darken as the angle between the surface normal and the direction to the light increases.

most materials fall somewhere in between a perfectly matte appearance and a perfectly mirror-like appearance, and we model such materials by shading a surface with both diffuse and specular contributions.

Theoretical models for rendering physically accurate specular reflection exist, but they can be very complex. The model we introduce here is a nonphysical approximation that has been in widespread use since the early days of computer graphics. The basic idea is that we first calculate a light's reflection direction $\mathbf{r}$ as

$$\mathbf{r} = 2\left(\mathbf{n} \cdot \mathbf{l}\right)\mathbf{n} - \mathbf{l}, \tag{7.22}$$

which follows from Equation (2.25), and then use the dot product $\mathbf{v} \cdot \mathbf{r}$ to estimate how close the reflection comes to the view vector $\mathbf{v}$. Since the specular highlight should be brightest when $\mathbf{v} \cdot \mathbf{r}$ is maximized, this dot product provides a convenient way to calculate some value representing the specular reflection. In a model called *Phong reflection*, the color of the specular reflection is given by

$$S = C_{\text{specular}} C_{\text{illum}} \left[\text{sat}\left(\mathbf{v} \cdot \mathbf{r}\right)\right]^{\alpha}, \tag{7.23}$$

where C_{specular} is the color of the specular highlights produced by the material. The exponent α, called the *specular power*, is an adjustable parameter that controls how sharp the specularity is. Higher values of α cause sharper highlights to be rendered because they cause the brightness to decrease more quickly as the directions $\mathbf{v}$ and $\mathbf{r}$ diverge. As with diffuse reflection, the saturation prevents nonzero specular highlights from occurring when $\mathbf{v} \cdot \mathbf{r}$ is negative.

A modification to the Phong model of specular reflection replaces the dot product in Equation (7.23) with $\mathbf{n} \cdot \mathbf{h}$, where the vector $\mathbf{h}$ is defined by

$$\mathbf{h} = \mathrm{nrm}\,(\mathbf{l} + \mathbf{v}).\qquad(7.24)$$

The vector $\mathbf{h}$ is called the *halfway vector* because, as illustrated in Figure 7.5, it lies halfway between the light direction $\mathbf{l}$ and the view vector $\mathbf{v}$. Using the halfway vector, the color of the specular reflection is now given by

$$S = C_{\mathrm{specular}} C_{\mathrm{illum}} \left[\mathrm{sat}\,(\mathbf{n} \cdot \mathbf{h}) \right]^{\alpha}.\qquad(7.25)$$

This produces results similar to Equation (7.23) because whenever the vectors $\mathbf{v}$ and $\mathbf{r}$ point in the same direction, the vectors $\mathbf{n}$ and $\mathbf{h}$ are also aligned to each other. However, the angle between $\mathbf{n}$ and $\mathbf{h}$ does not increase as quickly as the angle between $\mathbf{v}$ and $\mathbf{r}$ as the reflection direction diverges from the view vector, so a larger specular power α must be used to achieve roughly the same appearance. If the vectors $\mathbf{n}$, $\mathbf{v}$, and $\mathbf{l}$ all lie in the same plane, then the angle between $\mathbf{n}$ and $\mathbf{h}$ is exactly half the angle between $\mathbf{v}$ and $\mathbf{r}$. The shading model in which specular reflection is calculated with Equation (7.25) is called *Blinn-Phong reflection*. It is generally preferred over the original Phong model because the halfway vector $\mathbf{h}$ appears frequently in more advanced shading techniques. When we add the specular contribution to the diffuse reflection given by Equation (7.20), we have

$$f_{\mathrm{direct}}\,(C_{\mathrm{illum}}, \mathbf{n}, \mathbf{l}, \mathbf{v}) = \left\{ \frac{\rho}{\pi} C_{\mathrm{diffuse}} \, \mathrm{sat}\,(\mathbf{n} \cdot \mathbf{l}) + C_{\mathrm{specular}} \left[\mathrm{sat}\,(\mathbf{n} \cdot \mathbf{h}) \right]^{\alpha} \right\} C_{\mathrm{illum}}.\qquad(7.26)$$

This is the shading formula for the color of the combined diffuse and specular reflection attributed to a single light source.

The pixel shader code shown in Listing 7.2 calculates the specular shading for a single direct light source using the Blinn-Phong model in Equation (7.25). The result should be added to the diffuse shading calculated in Listing 7.1. The function in Listing 7.2 takes the halfway vector $\mathbf{h}$ as a parameter, and it should be calculated in the pixel shader with Equation (7.24) after the interpolated vectors $\mathbf{l}$ and $\mathbf{v}$ have been normalized. The function also takes the value of $\mathbf{n} \cdot \mathbf{l}$ as a parameter so that the specular reflection can be disabled in the case that the light source is behind the surface. If shadows are being rendered, then this parameter is not necessary because the shadow calculations will cause C_{illum} to be zero for surfaces facing away from the light source. An example showing a model shaded by this code is provided in Figure 7.6.
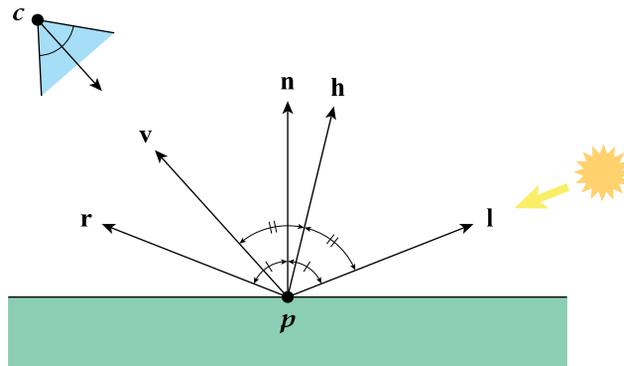
Figure 7.5. A point p is shaded on a surface with normal vector **n**. The view vector **v** points toward the camera at position c, and the light direction **l** points toward the light source. The vector **r** is the reflection of the light direction across the normal vector. The halfway vector **h** is the normalized sum of **l** and **v** lying halfway between the light direction and the view vector. When **v** = **r**, it is also the case that **n** = **h**.

Listing 7.2. This pixel shader function calculates the specular reflection produced by the Blinn-Phong reflection model with a specular power `alpha`. The interpolated surface normal n and halfway vector h must be normalized before being passed to this function. The uniform constant `specular-Color` holds the color C_{specular}, and the constant `lightColor` holds the color C_{illum}.

```
uniform float3   specularColor;   // C_specular
uniform float3   lightColor;      // C_illum

float3 CalculateSpecularReflection(float3 n, float3 h, float alpha, float nl)
{
    float highlight = pow(saturate(dot(n, h)), alpha) * float(nl > 0.0);
    return (lightColor * specularColor * highlight);
}
```


Figure 7.6. On the left, a sci-fi weapon is rendered with diffuse shading only. On the right, specular shading is added with specular powers of 10, 50, and 200 from left to right.

7.4 Texture Mapping

So far, the material properties that we have described, such as the diffuse reflection color C_{diffuse} and the specular reflection color C_{specular}, have been constants over the entire surface of an object. Visual detail is added to a surface, effectively allowing the material properties to vary as functions of position, through the application of *texture maps*. Modern GPUs support several different types of texture maps, and they can contain data stored in a wide variety of formats. Analogous to how one element of a displayed image is called a pixel, one element of a texture map is called a *texel* (a shortened form of *texture element*). Texture maps are usually applied to a triangle mesh by assigning predetermined *texture coordinates* to each vertex, but it is also possible to calculate texture coordinates in a shader for special purposes.

7.4.1 Texture Coordinates

The basic types of texture maps are shown in Figure 7.7. A texture map can be one-, two-, or three-dimensional, and up to three texture coordinates named u, v, and w are used to access it. As shown in the figure, we place the origin along each axis at the left, top, and front of the image. 1D and 2D texture maps can also be organized into arrays with multiple layers, and the layer that we access is specified by an extra coordinate. One additional type of texture map not shown in the figure is a cube texture map consisting of six square 2D faces, and this type is described in detail below.

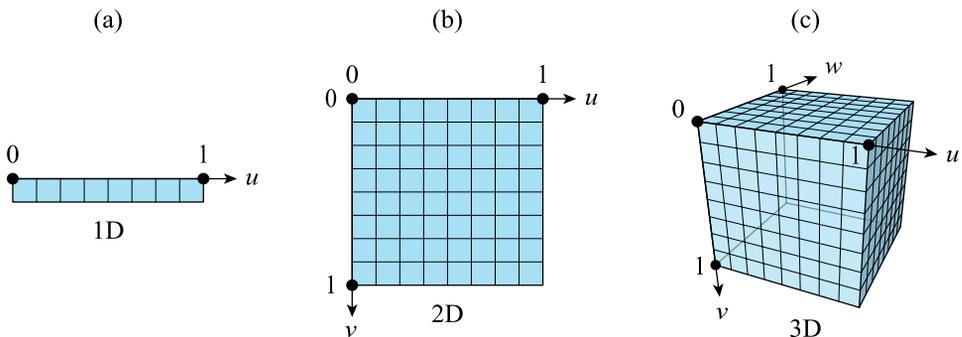**Figure 7.7.** (a) A 1D texture map is accessed with a single u coordinate. (b) A 2D texture map is accessed with a pair of texture coordinates (u, v), where the origin is at the upper-left corner of the image. (c) A 3D texture map is accessed with a triplet of texture coordinates (u, v, w). The range $[0,1]$ typically corresponds to the full size of the texture map.

Texture coordinates are usually normalized so that the range $[0,1]$ along any axis corresponds to the full size of the texture map, as illustrated in Figure 7.7. When a texture map is sampled with coordinates outside this range, the behavior is controlled by the wrapping modes that have been specified. Most of the time, we want the texture image to repeat. In this case, the integer part of each texture coordinate is discarded, and the fractional part in the range $[0,1)$ determines what texels are fetched.

The frame buffer itself can be treated as a 2D texture map, and additional data buffers matching the size of the frame buffer are often used in advanced rendering techniques. Methods that render special information to an offscreen frame buffer and later read it as a texture map appear several times in Chapter 10. In these cases, the texture coordinates used to fetch samples from the texture maps are not normalized to the range $[0,1]$, but instead cover the full width and height of the frame buffer, in pixels. It is convenient to access these kinds of texture maps with unnormalized texture coordinates because the coordinates of the pixel being rendered can be used directly as the texture coordinates in a pixel shader without any scaling.

7.4.2 Conventional Texture Mapping

The most common conventional use of texture mapping is the application of an ordinary 2D image to a triangle mesh. Although a texture map usually contains colors, it could also be used to hold whatever kind of high-resolution data that we might want to assign to a surface. For example, a texture map is often used to apply different diffuse or specular colors at different locations on a model, but a texture map could also be used to apply a different specular *power* to control the shininess of a model at different locations. Single-component values like the specular power are sometimes stored in the alpha channel of a texture map that contains color data in its red, green, and blue channels. This practice is also exemplified by the ambient occlusion values discussed later in Section 7.8.3. It may also be the case that the color channels do not contain colors at all. For instance, a special kind of texture map called a normal map contains the x, y, and z components of 3D vectors in its red, green, and blue channels. Normal maps and additional kinds of texture maps utilized in more advanced rendering methods are introduced later in this chapter.

The application of an ordinary texture map containing diffuse colors is demonstrated in Figure 7.8(a). A wireframe outline of the triangles to which the texture map is applied is shown in Figure 7.8(b). To aid in the visualization of how the texture image is mapped onto those triangles, another wireframe outline is shown in Figure 7.8(c), but the 3D vertex positions (x, y, z) have been replaced by the 2D texture coordinates (u, v) assigned to each vertex. This makes it possible to see
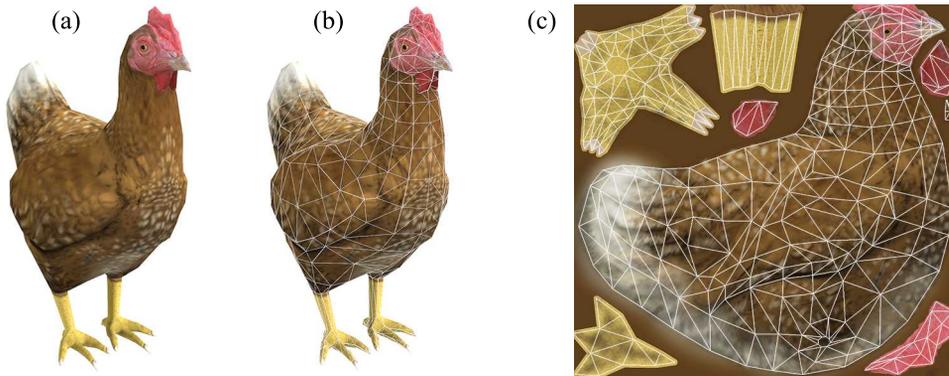
Figure 7.8. (a) A model of a chicken is rendered with a diffuse texture map. (b) The triangles composing the model are shown in a wireframe overlay. (c) The same triangles are overlaid on the texture image by placing each vertex at its (u, v) texture coordinates. This demonstrates how parts of the texture image are mapped to the interior of each triangle.

what part of the texture image fills the interior of each triangle. The colors in the texture image would normally be stored as gamma-corrected sRGB colors. The graphics hardware is capable of transforming these colors into linear colors automatically when a texture map is sampled. Shading calculations are then performed in linear space, and the results can be automatically converted back to sRGB color when written to the frame buffer.

7.4.3 Cube Texture Mapping

A *cube texture map* is a special type of texture map that consists of six 2D layers corresponding to the six faces of a cube. A cube texture map often contains an omnidirectional snapshot of an environment or some other kind of data that varies with direction relative to some central position. By design, a cube texture map is intended to be sampled with a 3D vector (x, y, z) instead of ordinary 2D texture coordinates (u, v). This vector would not typically be stored with a model's vertices, but it would instead be calculated inside a pixel shader. This is the case for the technique called *environment mapping* described below, which is the original motivation for the functionality provided by cube texture maps.

When a cube texture map is sampled using a vector (x, y, z), the GPU first selects the 2D face that it accesses by determining which component of the vector has the largest absolute value. For example, if $|x| > |y|$ and $|x| > |z|$, then the face sampled is either the $+x$ face or the $-x$ face depending on the sign of x. In the event

that two or more coordinates have the same absolute value, the z coordinate always has the highest precedence, and the y coordinate is selected over the x coordinate. After the face has been determined, normalized 2D coordinates (u, v) in the range $[0, 1]$ are calculated for that face using the formulas shown in Table 7.1. This allows each face to be sampled as an ordinary 2D texture map. The particular formulas listed in the table were chosen so that all the faces would have the same u and v axis directions when the cube is unwrapped and flattened as shown in Figure 7.9. This is how the cube looks when viewed from outside, and its configuration was originally designed for an external coordinate system in which the y axis is the up direction.

Face	u	v	$\mathbf{M}_{face}$
$+x$	$\dfrac{1-z/x}{2}$	$\dfrac{1-y/x}{2}$	$\begin{bmatrix} 0 & 0 & +1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix}$
$-x$	$\dfrac{1-z/x}{2}$	$\dfrac{1+y/x}{2}$	$\begin{bmatrix} 0 & 0 & -1 \\ 0 & -1 & 0 \\ +1 & 0 & 0 \end{bmatrix}$
$+y$	$\dfrac{1+x/y}{2}$	$\dfrac{1+z/y}{2}$	$\begin{bmatrix} +1 & 0 & 0 \\ 0 & 0 & +1 \\ 0 & +1 & 0 \end{bmatrix}$
$-y$	$\dfrac{1-x/y}{2}$	$\dfrac{1+z/y}{2}$	$\begin{bmatrix} +1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix}$
$+z$	$\dfrac{1+x/z}{2}$	$\dfrac{1-y/z}{2}$	$\begin{bmatrix} +1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & +1 \end{bmatrix}$
$-z$	$\dfrac{1+x/z}{2}$	$\dfrac{1+y/z}{2}$	$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$

Table 7.1. For each face of a cube texture map, formulas for the values of u and v provide the normalized texture coordinates at which the face is sampled. The matrix $\mathbf{M}_{face}$ gives the transform from the face's coordinate system to the camera's coordinate system when the face is rendered from the center of the cube.
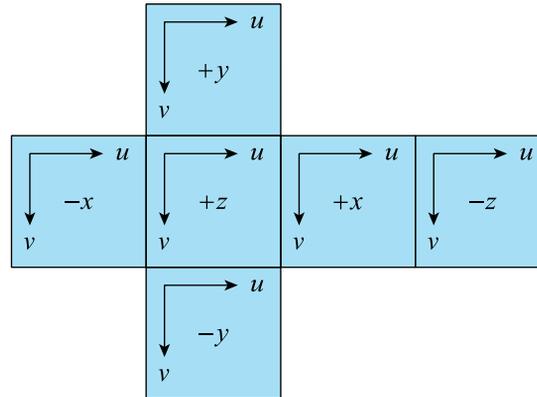
Figure 7.9. The u and v coordinate axes for each face of a cube texture map are shown on a flattened cube as it would appear from the outside. The x, y, and z axes of the cube point out of the page.

The images stored in a cube texture map are often rendered by the game engine itself. This could be done as a step taken when compiling a world, or it could happen dynamically as the environment changes. To render the image for each cube face with the correct orientation, we need to set up the camera transform properly, and this can be a little tricky. For each particular face, the x axis in camera space needs to be aligned with the direction that u coordinate points, and the y axis needs to be aligned with the v coordinate. The camera is positioned at the center of the cube, and its z axis has to point out of the face that we are rendering. As an example, consider the $+x$ face shown in Figure 7.9. Its u axis points in the $-z$ direction, and its v axis points in the $-y$ direction. These must be aligned with the camera-space $+x$ and $+y$ directions. Also, from a camera position *inside* the cube, the $+x$ axis pointing out of the face must be aligned with the camera-space $+z$ direction. We can express the mapping from the coordinate system of the face to the coordinate system of the camera with the matrix

$$\mathbf{M}_{\text{face}} = \begin{bmatrix} 0 & 0 & +1 \\ 0 & -1 & 0 \\ -1 & 0 & 0 \end{bmatrix}. \tag{7.27}$$

The first two columns of this matrix are the directions in which the u and v axes point, and the third column is the direction pointing out of the face. Matrices derived in a similar manner are listed in Table 7.1 for all six cube faces.

Let the matrix $\mathbf{M}_{cube}$ be the transformation from object space to world space for a node representing the location where a cube texture map is generated. The camera transformation $\mathbf{M}_{camera}$ for a single face of the cube is given by the product

$$\mathbf{M}_{camera} = \mathbf{M}_{cube}\mathbf{M}_{face}. \qquad (7.28)$$

This is the camera transformation that should be used when rendering a cube face image, where $\mathbf{M}_{face}$ is the matrix listed in Table 7.1 for the specific face. An important observation is that all of the cube face coordinate systems are left handed, and this means that each matrix $\mathbf{M}_{face}$ has a determinant of -1. As a result, the matrix $\mathbf{M}_{camera}$ given by Equation (7.28) contains a reflection, and we must compensate by inverting the rendering state that specifies the front face winding direction. If front faces are normally wound counterclockwise, then they will appear to be wound clockwise when rendering a cube face.

Environment mapping is a technique in which a cube texture map, called an *environment map* in this case, is used to produce the appearance of an approximate mirror-like reflection on shiny surfaces. We must first render the contents of an environment map to capture the details of the surrounding geometry at some location. An example environment map for an indoor area is shown in Figure 7.10(a). Once the environment map is available, we can sample it in a pixel shader by simply providing a vector $\mathbf{r}$ that corresponds to the reflection of the view vector $\mathbf{v}$ across the surface normal $\mathbf{n}$. This reflection vector is calculated with the formula

$$\mathbf{r} = 2(\mathbf{n} \cdot \mathbf{v})\mathbf{n} - \mathbf{v}, \qquad (7.29)$$

which can be used directly as the (x, y, z) texture coordinates with which the environment map is sampled. There is no need to normalize this vector because the divisions appearing in Table 7.1 automatically take care of scaling relative to the largest component.

The vector $\mathbf{r}$ given by Equation (7.29) has object-space coordinates, but we have to sample the environment map in its own local coordinate system. If the matrix $\mathbf{M}_{object}$ transforms from object space to world space, and the matrix $\mathbf{M}_{cube}$ transforms from the coordinate system of the cube texture map to world space, then the product

$$\mathbf{M}_{env} = \mathbf{M}_{cube}^{-1}\mathbf{M}_{object} \qquad (7.30)$$

can be used to transform vectors from object space directly into coordinate space used by the environment map. For the reflection vector $\mathbf{r}$ calculated with Equation (7.29), we would sample the environment map with the vector $\mathbf{M}_{env}\mathbf{r}$. This process is implemented in Listing 7.3, and it produces results like the reflections shown on the shiny surfaces in Figure 7.10(b).

(a)

(b)
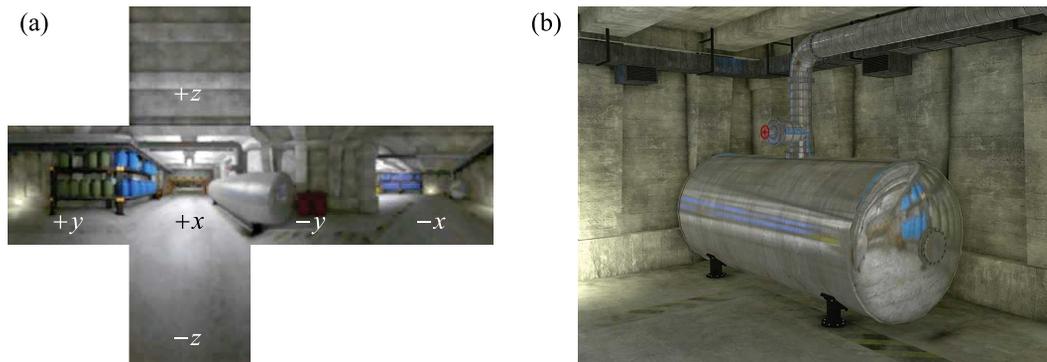
Figure 7.10. (a) An environment map has been rendered for an indoor location. This is how the six faces appear from inside the cube. The axis shown for each face points into the page. (b) The environment map has been applied to a tank and some pipes to give them a highly reflective appearance.

Listing 7.3. This pixel shader function calculates the reflection of the view vector v across the surface normal n, which are assumed to be normalized. It transforms the result into the local coordinate system of an environment map using the upper-left 3×3 portion of the matrix $\mathbf{M}_{env}$ given by Equation (7.30), whose rows are stored in the uniform constant Menv.

```
uniform TextureCube    environmentMap;
uniform float3         Menv[3];

float4 SampleEnvironmentMap(float3 n, float3 v)
{
    float3 r = n * (2.0 * dot(n, v)) - v;
    float3 texcoord = float3(dot(Menv[0], r), dot(Menv[1], r), dot(Menv[2], r));
    return (texture(environmentMap, texcoord));
}
```

7.5 Tangent Space

Up to this point, our shading calculations have been carried out in object space. More advanced techniques, some of which have become standard fixtures in game engines, are able to make use of texture maps that store finely detailed geometric information instead of colors. The most common example is the normal mapping technique in which a texture map contains vector data, and this is introduced in the next section. The numerical values stored in this kind of texture map are expressed in the coordinate system of the texture map itself so that the geometric details are

independent of any particular model. This allows a geometric texture map to be applied to any triangle mesh without having to account for the object-space coordinate system used by its vertices.

In the coordinate system of a texture map, the x and y axes are aligned to the horizontal and vertical directions in the 2D image, and the z axis points upward out of the image plane, as shown in Figure 7.11(a). If the origin of the texture map is located in the upper-left corner, then this constitutes a left-handed coordinate system. It is also possible to flip the texture upside down and put the origin in the lower-left corner to create a right-handed coordinate system. Either choice works fine because we will need to account for the handedness inherent in the mapping of the texture map to a surface anyway.

In order to perform shading calculations that use geometric information stored in a texture map, we need a way to transform between the coordinate system of the texture map and object space. This is done by identifying the directions in object space that correspond to the coordinate axes of the texture map. These object-space directions are not constant, but vary from triangle to triangle. For a single triangle, we can think of the texture map as lying in the triangle's plane with its x and y axes oriented in the directions that are aligned to the (u, v) texture coordinates assigned to the triangle's vertices. The z axis of the texture map points directly out of the plane, so it is aligned with the triangle's normal vector in object space. The x and y axes of the texture map point along directions that are tangent to the surface in object space, and at least one of these vectors needs to be calculated ahead of time.
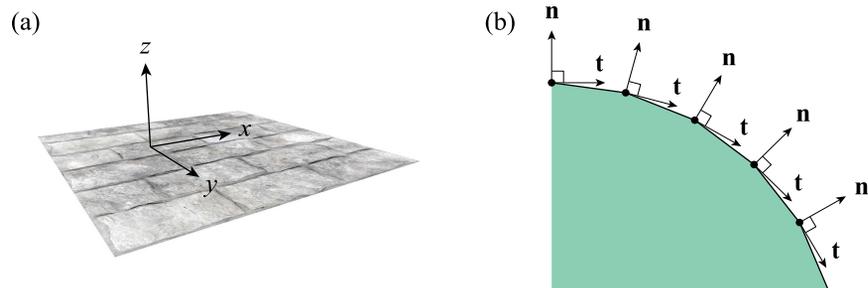
Figure 7.11. (a) In the coordinate system of a texture map, the x and y axes are aligned to the texel image, and the z axis points upward out of the image plane. (b) Each vertex in a triangle mesh has a normal vector **n** and a perpendicular tangent vector **t**, and both vectors form a smooth field over the entire model. The direction that the tangent vector points within the tangent plane is determined by the orientation of the texture map at each vertex.

As with normal vectors, we calculate an average unit-length tangent vector $\mathbf{t}$ for each vertex in a triangle mesh. This lets us create a smooth tangent field on the surface of a model, as shown in Figure 7.11(b). Although it may not be strictly true for the specific texture mapping applied to a model, we assume that the two tangent directions are perpendicular to each other, so a second tangent direction $\mathbf{b}$ called the *bitangent vector* can be calculated with a cross product. The three vectors $\mathbf{t}$, $\mathbf{b}$, and $\mathbf{n}$ form the basis of the *tangent frame* at each vertex, and the coordinate space in which the x, y, and z axes are aligned to these directions is called *tangent space*. We can transform vectors from tangent space to object space using the 3×3 matrix $\mathbf{M}_{\text{tangent}}$ given by

$$\mathbf{M}_{\text{tangent}} = \begin{bmatrix} \uparrow & \uparrow & \uparrow \\ \mathbf{t} & \mathbf{b} & \mathbf{n} \\ \downarrow & \downarrow & \downarrow \end{bmatrix}, \tag{7.31}$$

which has the vectors $\mathbf{t}$, $\mathbf{b}$, and $\mathbf{n}$ as its columns. Since this matrix is orthogonal, the reverse transformation from object space to tangent space is the transpose

$$\mathbf{M}_{\text{tangent}}^{\mathrm{T}} = \begin{bmatrix} \leftarrow & \mathbf{t}^{\mathrm{T}} & \rightarrow \\ \leftarrow & \mathbf{b}^{\mathrm{T}} & \rightarrow \\ \leftarrow & \mathbf{n}^{\mathrm{T}} & \rightarrow \end{bmatrix}, \tag{7.32}$$

where the vectors $\mathbf{t}$, $\mathbf{b}$, and $\mathbf{n}$ form the rows. The name *TBN matrix* is often used to refer to either one of these matrices.

Applying a little linear algebra to the vertex positions and their associated texture coordinates lets us calculate the tangent field for a triangle mesh. Let $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, and $\boldsymbol{p}_2$ be the three vertices of a triangle, wound in counterclockwise order, and let (u_i, v_i) represent the texture coordinates associated with the vertex $\boldsymbol{p}_i$. The values of u and v correspond to distances along the axes $\mathbf{t}$ and $\mathbf{b}$ that are aligned to the x and y directions of the texture map. This means that we can express the difference between two points with known texture coordinates as

$$\boldsymbol{p}_i - \boldsymbol{p}_j = (u_i - u_j)\mathbf{t} + (v_i - v_j)\mathbf{b}. \tag{7.33}$$

To determine what the vectors $\mathbf{t}$ and $\mathbf{b}$ are, we can form a system of equations using differences between the vertices on two of the triangle's edges. After making the definitions

$$\begin{aligned} \mathbf{e}_1 &= \boldsymbol{p}_1 - \boldsymbol{p}_0, \quad (x_1, y_1) = (u_1 - u_0, v_1 - v_0), \\ \mathbf{e}_2 &= \boldsymbol{p}_2 - \boldsymbol{p}_0, \quad (x_2, y_2) = (u_2 - u_0, v_2 - v_0), \end{aligned} \tag{7.34}$$

we can write this system very compactly as

$$\mathbf{e}_1 = x_1 \mathbf{t} + y_1 \mathbf{b}$$
$$\mathbf{e}_2 = x_2 \mathbf{t} + y_2 \mathbf{b}. \tag{7.35}$$

An equivalent matrix equation is

$$\begin{bmatrix} \uparrow & \uparrow \\ \mathbf{e}_1 & \mathbf{e}_2 \\ \downarrow & \downarrow \end{bmatrix} = \begin{bmatrix} \uparrow & \uparrow \\ \mathbf{t} & \mathbf{b} \\ \downarrow & \downarrow \end{bmatrix} \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix}, \tag{7.36}$$

where $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{t}$ and $\mathbf{b}$ are all column vectors. This equation is readily solved by inverting the 2×2 matrix of coefficients on the right side to produce

$$\begin{bmatrix} \uparrow & \uparrow \\ \mathbf{t} & \mathbf{b} \\ \downarrow & \downarrow \end{bmatrix} = \frac{1}{x_1 y_2 - x_2 y_1} \begin{bmatrix} \uparrow & \uparrow \\ \mathbf{e}_1 & \mathbf{e}_2 \\ \downarrow & \downarrow \end{bmatrix} \begin{bmatrix} y_2 & -x_2 \\ -y_1 & x_1 \end{bmatrix}. \tag{7.37}$$

To calculate an average tangent vector and bitangent vector at each vertex, we maintain sums of the vectors produced for each triangle and later normalize them. When values of $\mathbf{t}$ and $\mathbf{b}$ are calculated with Equation (7.37), they are added to the sums for the three vertices referenced by the triangle. The results are usually not exactly perpendicular, but unless the texture mapping is skewed to a significant degree, they should be *close* to perpendicular. We can nudge them the rest of the way by applying Gram-Schmidt orthonormalization. First, assuming the vertex normal vector $\mathbf{n}$ has unit length, we make sure the vertex tangent vector $\mathbf{t}$ is perpendicular to $\mathbf{n}$ by replacing it with

$$\mathbf{t}_\perp = \mathrm{nrm}\left(\mathbf{t} - (\mathbf{t} \cdot \mathbf{n}) \mathbf{n} \right) \tag{7.38}$$

using the rejection operation described in Section 1.6. (We use the subscript $\perp$ simply to mean that the vector has been orthonormalized.) The vertex bitangent vector $\mathbf{b}$ is then made perpendicular to both $\mathbf{t}$ and $\mathbf{n}$ by calculating

$$\mathbf{b}_\perp = \mathrm{nrm}\left(\mathbf{b} - (\mathbf{b} \cdot \mathbf{n}) \mathbf{n} - (\mathbf{b} \cdot \mathbf{t}_\perp) \mathbf{t}_\perp \right). \tag{7.39}$$

The vectors $\mathbf{t}_\perp$, $\mathbf{b}_\perp$, and $\mathbf{n}$ now form a set of unit-length orthogonal axes for the tangent frame at a vertex. Code that implements this entire process is provided in Listing 7.4.

Since the vectors are orthogonal, it is not necessary to store all three of the vectors $\mathbf{t}_\perp$, $\mathbf{b}_\perp$, and $\mathbf{n}$ for each vertex. Just the normal vector and the tangent vector will always suffice, but we do need one additional bit of information. The tangent frame can form either a right-handed or left-handed coordinate system, and which one is given by the sign of $\det(\mathbf{M}_{\text{tangent}})$. Calling the sign of this determinant σ, we can reconstitute the bitangent with the cross product

$$\mathbf{b}_\perp = \sigma\,(\mathbf{n}\times\mathbf{t}_\perp),\qquad(7.40)$$

and then only the normal and tangent need to be supplied as vertex attributes. An example showing the normal field and tangent field for a character model is provided in Figure 7.12. One possible way to communicate the value of σ to the vertex shader is by extending the tangent to a four-component vertex attribute and storing σ in the w coordinate. This is the method used in Listing 7.4, but a more clever approach might encode σ in the least significant bit of one of the x, y, or z coordinates of the tangent to avoid increasing the size of the vertex data.

Listing 7.4. This function calculates the per-vertex tangent vectors for the triangle mesh having `triangleCount` triangles with indices specified by `triangleArray` and `vertexCount` vertices with positions specified by `vertexArray`. The per-vertex normal vectors and texture coordinates are given by `normalArray` and `texcoordArray`. Tangents are written to `tangentArray`, which must be large enough to hold `vertexCount` elements. The determinant of the matrix $\mathbf{M}_{\text{tangent}}$ at each vertex is stored in the w coordinate of each tangent vector.

```
void CalculateTangents(int32 triangleCount, const Triangle *triangleArray,
            int32 vertexCount, const Point3D *vertexArray, const Vector3D *normalArray,
            const Point2D *texcoordArray, Vector4D *tangentArray)
{
    // Allocate temporary storage for tangents and bitangents and initialize to zeros.
    Vector3D *tangent = new Vector3D[vertexCount * 2];
    Vector3D *bitangent = tangent + vertexCount;
    for (int32 i = 0; i < vertexCount; i++)
    {
        tangent[i].Set(0.0F, 0.0F, 0.0F);
        bitangent[i].Set(0.0F, 0.0F, 0.0F);
    }

    // Calculate tangent and bitangent for each triangle and add to all three vertices.
    for (int32 k = 0; k < triangleCount; k++)
    {
        int32 i0 = triangle[k].index[0];
        int32 i1 = triangle[k].index[1];
        int32 i2 = triangle[k].index[2];
```

```
        const Point3D& p0 = position[i0];
        const Point3D& p1 = position[i1];
        const Point3D& p2 = position[i2];
        const Point2D& w0 = texcoord[i0];
        const Point2D& w1 = texcoord[i1];
        const Point2D& w2 = texcoord[i2];

        Vector3D e1 = p1 - p0, e2 = p2 - p0;
        float x1 = w1.x - w0.x, x2 = w2.x - w0.x;
        float y1 = w1.y - w0.y, y2 = w2.y - w0.y;

        float r = 1.0F / (x1 * y2 - x2 * y1);
        Vector3D t = (e1 * y2 - e2 * y1) * r;
        Vector3D b = (e2 * x1 - e1 * x2) * r;

        tangent[i0] += t;
        tangent[i1] += t;
        tangent[i2] += t;
        bitangent[i0] += b;
        bitangent[i1] += b;
        bitangent[i2] += b;
    }

    // Orthonormalize each tangent and calculate the handedness.
    for (int32 i = 0; i < vertexCount; i++)
    {
        const Vector3D& t = tangent[i];
        const Vector3D& b = bitangent[i];
        const Vector3D& n = normalArray[i];
        tangentArray[i].xyz() = Normalize(Reject(t, n));
        tangentArray[i].w = (Dot(Cross(t, b), n) > 0.0F) ? 1.0F : -1.0F;
    }

    delete[] tangent;
}
```

It is common for there to be discontinuities in a model's texture mapping, and this is in fact unavoidable for anything topologically equivalent to a sphere because a continuous nonvanishing tangent field is impossible. In these cases, vertices are duplicated along the triangle edges where the discontinuity occurs. The additional vertices have the same positions, but they could have different texture coordinates. Because they are indexed separately, their tangent vectors are not averaged, and this can lead to a visible boundary where an abrupt change in shading is visible. To avoid this, duplicates need to be identified so that their tangents can be averaged and set equal to each other, but only if the tangent frames have the same handedness and the tangents are pointing in similar directions.
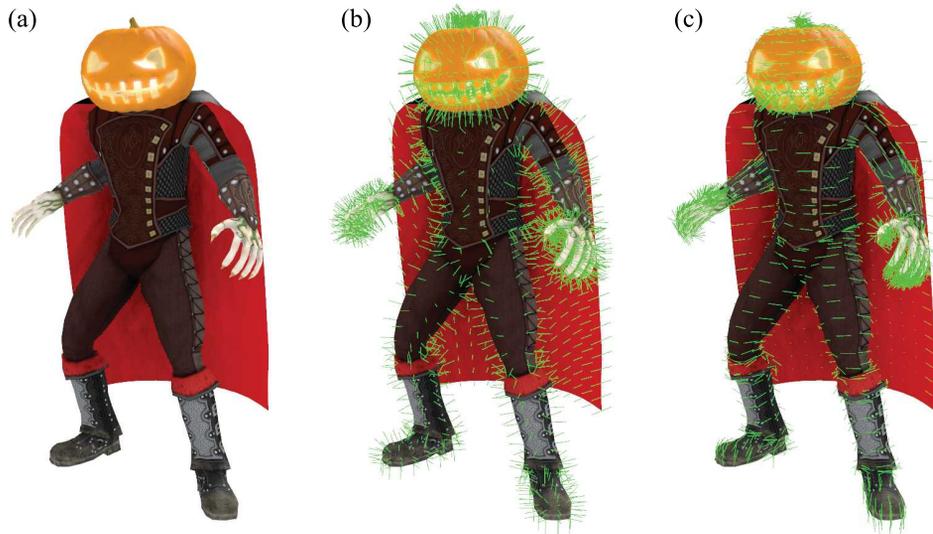
Figure 7.12. (a) A character model uses texture mapping techniques that require a tangent frame. (b) The normal vector corresponding to each vertex is shown as a green line starting at the vertex's position. (c) The tangent vector is shown for each vertex, and it is aligned to the *x* direction of the texture map at the vertex position.

7.6 Bump Mapping

Bump mapping is a technique that gives the surface of a model the appearance of having much greater geometric detail than is actually present in the triangle mesh. It works by using a special type of texture map call a *normal map* to assign a different normal vector to each point on a surface. When shading calculations account for these finely detailed normal vectors in addition to the smoothly interpolated geometric normal, it produces the illusion that the surface varies in height even though the triangles we render are still perfectly flat. This happens because the brightness of the reflected light changes at a high frequency, causing the surface to appear as if it were bumpy, and this is where the term bump mapping comes from. Because it uses a normal map, bump mapping is often called *normal mapping*, and the two terms can be used interchangeably. Figure 7.13 shows an example in which a wall is rendered as a flat surface having a constant normal vector and is rendered again with the varying normal vectors produced by bump mapping. The difference is already pretty remarkable, but we will be able to extend the concept even further with the addition of parallax and horizon mapping in the next two sections.
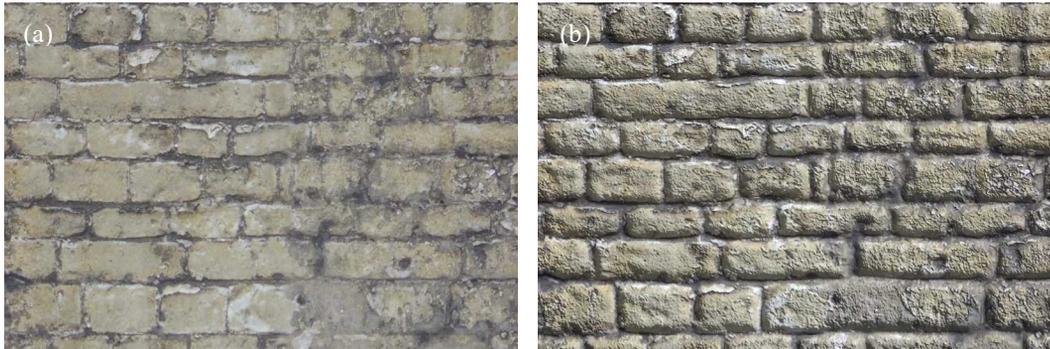
Figure 7.13. (a) A flat wall is rendered without bump mapping, so it has a constant normal vector across its surface. (b) The same wall is rendered with bump mapping, and the normal vector is modified by the vectors stored in a normal map. In both images, the direction to the light points toward the upper-right corner.

7.6.1 Normal Map Construction

Each texel in a normal map contains a unit-length normal vector whose coordinates are expressed in tangent space. The normal vector $(0, 0, 1)$ corresponds to a smooth surface because it is parallel to the direction pointing directly along the interpolated normal vector **n** in object space. Any other tangent-space normal vector represents a deviation from the smooth surface due to the presence of geometric detail. Though it is possible to generate a normal map based on some mathematical description of a surface, most normal maps are created by calculating slopes in a height field. A single-channel *height map* is typically supplied, and the values it contains correspond to the heights of the detailed geometry above a flat surface. For example, the height map that was used to create the normal map for the wall in Figure 7.13 is shown as a grayscale image in Figure 7.14(a).

To calculate the normal vector for a single texel in the normal map, we first apply central differencing to calculate slopes in the x and y directions. The values in the height map are usually interpreted as numbers in the range $[0,1]$, so they must be scaled by some constant factor to stretch them out into the full range of heights that are intended to be covered. Using a scale factor of s means that the maximum height is s times the width of a single texel. Let the function $h(i, j)$ represent the value, in the range $[0,1]$, stored in the height map at the coordinates (i, j). The two slopes d_x and d_y are then given by

$$d_x = \frac{\Delta z}{\Delta x} = \frac{s}{2}[h(i+1, j) - h(i-1, j)]$$

$$d_y = \frac{\Delta z}{\Delta y} = \frac{s}{2}[h(i, j+1) - h(i, j-1)]. \tag{7.41}$$

At the edges of the height map, care must be taken to either clamp the coordinates or wrap them around to the opposite side of the image, depending on the intended wrapping mode of the resulting normal map.

Once the slopes have been determined, we can express directions $\mathbf{u}_x$ and $\mathbf{u}_y$ that are tangent to the height field along the x and y axes as

$$\mathbf{u}_x = (1, 0, d_x) \quad \text{and} \quad \mathbf{u}_y = (0, 1, d_y). \tag{7.42}$$

Since these are independent directions in the height map's tangent plane, the normal vector $\mathbf{m}$ can be calculated with the cross product

$$\boxed{\mathbf{m} = \mathrm{nrm}(\mathbf{u}_x \times \mathbf{u}_y) = \frac{(-d_x, -d_y, 1)}{\sqrt{d_x^2 + d_y^2 + 1}}.} \tag{7.43}$$
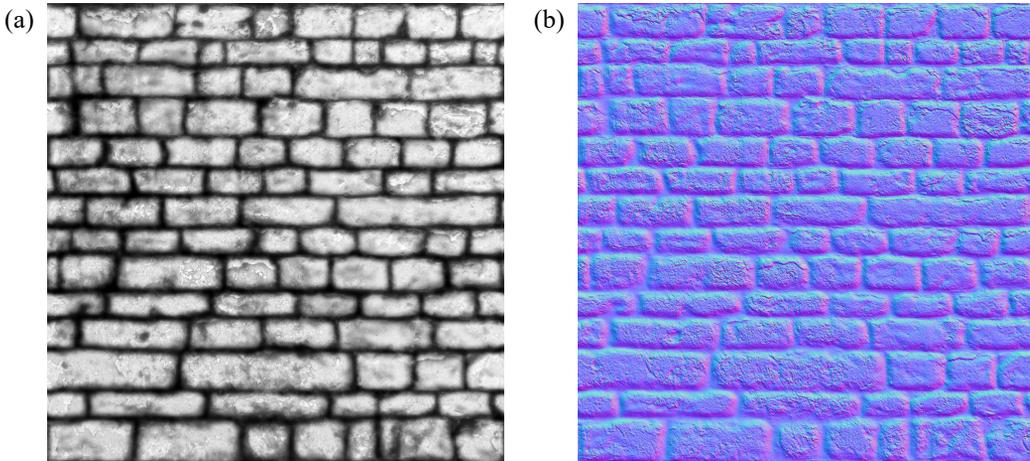
Figure 7.14. (a) A single-channel image contains a height map for a stone wall. Brighter values correspond to greater heights. (b) The corresponding normal map contains the vectors calculated by Equation (7.43) with a scale of $s = 24$. The components have been remapped to the range $[0,1]$.

This is the value that gets stored in the normal map. We use the letter **m** to avoid confusion with the object-space normal vector **n** defined at each vertex. The code shown in Listing 7.5 uses Equation (7.43) to construct a normal map from a height map that has already been scaled by the factor of s appearing in Equation (7.41). At the edges of the height map, this code calculates differences by wrapping around to the opposite side.

In the early days of GPU bump mapping, the three components of the normal vector had to be stored in a texture map whose color channels could hold values in the range $[0,1]$. Since the x and y components of **m** can be any values in the range $[-1,+1]$, they had to be remapped to the range $[0,1]$ by calculating $r = \frac{1}{2}x + \frac{1}{2}$ for the red channel and $g = \frac{1}{2}y + \frac{1}{2}$ for the green channel. Even though the z component

Listing 7.5. This function constructs a normal map corresponding to the scaled height map having power-of-two dimensions width × height specified by heightMap. Normal vectors are written to the buffer supplied by normalMap, which must be large enough to hold width × height values.

```
void ConstructNormalMap(const float *heightMap, Vector3D *normalMap,
                        int32 width, int32 height)
{
    for (int32 y = 0; y < size.y; y++)
    {
        int32 ym1 = (y - 1) & (height - 1), yp1 = (y + 1) & (height - 1);

        const float *centerRow = heightMap + y * width;
        const float *upperRow = heightMap + ym1 * width;
        const float *lowerRow = heightMap + yp1 * width;

        for (int32 x = 0; x < size.x; x++)
        {
            int32 xm1 = (x - 1) & (width - 1), xp1 = (x + 1) & (width - 1);

            // Calculate slopes.
            float dx = (centerRow[xp1] - centerRow[xm1]) * 0.5F;
            float dy = (lowerRow[x] - upperRow[x]) * 0.5F;

            // Normalize and clamp.
            float nz = 1.0F / sqrt(nx * nx + ny * ny + 1.0F);
            float nx = fmin(fmax(-dx * nz, -1.0F), 1.0F);
            float ny = fmin(fmax(-dy * nz, -1.0F), 1.0F);
            normalMap[x].Set(nx, ny, nz);
        }

        normalMap += width;
    }
}
```

of **m** is always positive (because normal vectors always point out of the surface), the same mapping was applied to it as well for the blue channel. When a normal vector encoded in this way is fetched from a texture map, the GPU performs the reverse mapping back to the range $[-1,+1]$ by multiplying by two and subtracting one. Because the normal vectors are shortened a little by linear interpolation when they are fetched from a texture map, they should be renormalized in the pixel shader. The normal map shown in Figure 7.14(b) uses this encoding scheme. The fact that the blue channel always contains values in the range $\left[\frac{1}{2},1\right]$ gives it the characteristic purple tint that normal maps are known for having.

   With the widespread availability of a much larger set of texture formats, we have better options. A texture map can have signed channels that store values in the range $[-1,+1]$, so we no longer need to remap to the range $[0,1]$. To save space, we can take advantage of two-channel formats by storing only the x and y components of a normal vector **m**. After these are fetched from a texture map in the pixel shader, we can reconstitute the z component by calculating

$$m_z = \sqrt{1 - m_x^2 - m_y^2} \tag{7.44}$$

under the assumption that **m** has unit length and m_z is positive. This method for storing normal vectors works well with compressed texture formats that support two uncorrelated channels. Equation (7.44) is implemented in Listing 7.6.

Listing 7.6. This pixel shader function fetches the x and y components of a normal vector from the 2D texture map specified by `normalMap` at the texture coordinates given by `texcoord`. The z component of the normal vector is reconstituted with Equation (7.44).

```
uniform Texture2D      normalMap;

float3 FetchNormalVector(float2 texcoord)
{
    float2 m = texture(normalMap, texcoord).xy;
    return (float3(m, sqrt(1.0 - m.x * m.x - m.y * m.y)));
}
```

7.6.2 Rendering with Normal Maps

To shade a surface that has a normal map applied to it, we perform the same calculations that we would perform without a normal map involving the view direction **v**, light direction **l**, and halfway vector **h**. The difference is that we no longer take any dot products with the interpolated normal vector **n** because it is replaced

by a normal vector **m** fetched from a normal map. Before we can take dot products with **m**, though, we have to do something about the fact that it is not expressed in the same coordinate system as **v** and **l**. We need to either transform **v** and **l** into tangent space to match **m** or transform **m** into object space to match **v** and **l**.

To transform any vector **u** from object space to tangent space, we multiply it by the matrix $\mathbf{M}_{\text{tangent}}^{\text{T}}$ given by Equation (7.32). The rows of this matrix are the per-vertex tangent **t**, bitangent **b**, and normal **n**, so the components of **u** simply become $\mathbf{t} \cdot \mathbf{u}$, $\mathbf{b} \cdot \mathbf{u}$, and $\mathbf{n} \cdot \mathbf{u}$ in tangent space. This is applied to the object-space view direction **v** and light direction **l** by the vertex shader function shown in Listing 7.7 after it calculates the bitangent vector with Equation (7.40). The resulting tangent-space view direction $\mathbf{v}_{\text{tangent}}$ and light direction $\mathbf{l}_{\text{tangent}}$ should then be output by the vertex shader so their interpolated values can be used in the pixel shader.

Shading can also be performed in object space by fetching the vector **m** from a normal map and then multiplying it by the matrix $\mathbf{M}_{\text{tangent}}$ in the pixel shader. The columns of $\mathbf{M}_{\text{tangent}}$ are the vectors **t**, **b**, and **n**, so the transformed normal vector $\mathbf{m}_{\text{object}}$ is given by

$$\mathbf{m}_{\text{object}} = m_x \mathbf{t} + m_y \mathbf{b} + m_z \mathbf{n}. \tag{7.45}$$

To perform this operation in the pixel shader, we need to interpolate the per-vertex normal and tangent vectors in addition to the view direction **v** and the light direction **l**, which now remain in object space. The handedness σ of the tangent frame

Listing 7.7. This vertex shader function calculates the tangent-space view direction $\mathbf{v}_{\text{tangent}}$ and light direction $\mathbf{l}_{\text{tangent}}$ for a vertex having object-space attributes `position`, `normal`, and `tangent` and returns them in `vtan` and `ltan`. The *w* coordinate of the tangent contains the handedness σ of the tangent frame used in the calculation of the bitangent vector.

```
uniform float3 cameraPosition;      // Object-space camera position.
uniform float3 lightPosition;       // Object-space light position.

void CalculateTangentSpaceVL(float3 position, float3 normal, float4 tangent,
                             out float3 vtan, out float3 ltan)
{
    float3 bitangent = cross(normal, tangent.xyz) * tangent.w;
    float3 v = cameraPosition – position;
    float3 l = lightPosition – position;
    vtan = float3(dot(tangent, v), dot(bitangent, v), dot(normal, v));
    ltan = float3(dot(tangent, l), dot(bitangent, l), dot(normal, l));
}
```

also needs to be passed from the vertex shader to the pixel shader. Since handedness is always constant over a triangle, it can be flat interpolated to save some computation. After interpolation, the normal and tangent vectors may not have unit length, and it's possible that they are no longer perpendicular. To correct for this, we have to orthonormalize them in the pixel shader before we calculate the bitangent vector with Equation (7.40). The pixel shader function shown in Listing 7.8 carries out these steps to construct the tangent frame. It then applies Equation (7.45) to transform a normal vector **m** fetched from a normal map into object space.

Listing 7.8. This pixel shader function fetches a normal vector from the 2D texture map specified by `normalMap` at the texture coordinates given by `texcoord` using the function in Listing 7.6. It then transforms it into object space by multiplying it by the matrix $\mathbf{M}_{\text{tangent}}$. The interpolated object-space normal, tangent, and handedness values are given by `normal`, `tangent`, and `sigma`.

```
uniform Texture2D        normalMap;

float3 FetchObjectNormalVector(float2 texcoord, float3 normal, float3 tangent,
                               float sigma)
{
    float3 m = FetchNormalVector(texcoord);
    float3 n = normalize(normal);
    float3 t = normalize(tangent - n * dot(tangent, n));
    float3 b = cross(normal, tangent) * sigma;
    return (t * m.x + b * m.y + n * m.z);
}
```

7.6.3 Blending Normal Maps

There are times when we might want to combine two or more normal maps in the same material. For example, a second normal map could add finer details to a base normal map in a high-quality version of the material. It's also possible that texture coordinate animation is causing two normal maps to move in different directions, as commonly used to produce interacting ripples on a water surface. We might also want to smoothly transition between two different normal maps. In general, we would like to have a function $f_{\text{blend}}\left(\mathbf{m}_1, \mathbf{m}_2, a, b\right)$ that calculates the weighted sum of two normal vectors $\mathbf{m}_1$ and $\mathbf{m}_2$ with the weights a and b. The sum must behave as if the original height maps from which $\mathbf{m}_1$ and $\mathbf{m}_2$ were derived had been added together with the same weights a and b, allowing a new normal vector to be calculated with Equation (7.43). We cannot simply calculate $a\mathbf{m}_1 + b\mathbf{m}_2$ because it does not satisfy this requirement. In particular, if one height map contains all zeros, then

it should have no effect on the sum, but blending the normal vectors directly would cause the results to be skewed toward the vector $(0,0,1)$.

Fortunately, we can easily recover the slopes d_x and d_y to which a normal vector corresponds. All we have to do is scale a normal vector by the reciprocal of its z coordinate to match the unnormalized vector in the numerator of Equation (7.43), effectively undoing the previous normalization step. These slopes are nothing more than scaled differences of heights, so they are values that we *can* blend directly. This leads us to the blending function

$$f_{\text{blend}}(\mathbf{m}_1, \mathbf{m}_2, a, b) = \text{nrm}\left(a\frac{x_1}{z_1} + b\frac{x_2}{z_2}, a\frac{y_1}{z_1} + b\frac{y_2}{z_2}, 1 \right), \qquad (7.46)$$

where $\mathbf{m}_1 = (x_1, y_1, z_1)$ and $\mathbf{m}_2 = (x_2, y_2, z_2)$.

By setting $a = 1 - t$ and $b = t$, we can smoothly transition from one normal map to another as the parameter t goes from zero to one. To combine two normal maps in such a way that they have an additive effect without diminishing the apparent size of the bumps encoded in either one, we apply Equation (7.46) with the weights $a = b = 1$. Since the result is normalized, we can multiply all three components by $z_1 z_2$, in which case the additive blending function is given by

$$f_{\text{add}}(\mathbf{m}_1, \mathbf{m}_2) = \text{nrm}(z_2 x_1 + z_1 x_2, z_2 y_1 + z_1 y_2, z_2 z_1). \qquad (7.47)$$

The weights a and b do not need to be positive. Using a negative weight for one normal map causes it to be subtracted from the other so that its bumps appear to be inverted.

7.7 Parallax Mapping

Plain bump mapping has its limitations. While it often looks good when a surface is viewed from a nearly perpendicular direction, the illusion of bumpiness quickly breaks down as the angle between the surface and the viewing direction gets smaller. This is due to the fact that the same texels are still rendered at the same locations on the surface from all viewing directions, betraying the flatness of the underlying geometry. If the surface features encoded in the normal map actually had real height, then some parts of the color texture would be hidden from view by the bumps, and other parts would be more exposed, depending on the perspective. A technique called *parallax mapping* shifts the texels around a little bit to greatly improve the illusion that a surface has varying height.

Parallax mapping works by first considering the height h and normal vector $\mathbf{n}$ mapped to each point on a surface. As shown in Figure 7.15, these values can be used to establish a plane $[\mathbf{n}\,|\,d]$ that is tangent to the bumpy surface at that point. This plane serves as a local approximation to the surface that can be used to calculate a texture coordinate offset that accounts for the viewing direction and produces the appearance of parallax. Since we need only the offset, we can assume that the original texture coordinates are $(0,0)$ for simplicity. The value of d is then determined by requiring that the point $(0,0,h)$ lies on the plane, from which we obtain

$$d = -n_z h. \tag{7.48}$$

For a particular tangent-space view direction $\mathbf{v}$, the point where the ray $\boldsymbol{o}+t\mathbf{v}$ intersects the plane $[\mathbf{n}\,|\,d]$ is approximately the point $\boldsymbol{p}$ that would be visible from the direction $\mathbf{v}$ if the surface actually had the height h at the point sampled on the flat geometry. The parameter t is calculated by solving the equation

$$[\mathbf{n}\,|\,d]\cdot(\boldsymbol{o}+t\mathbf{v}) = 0, \tag{7.49}$$

and the point $\boldsymbol{p}$ is thus given by

$$\boldsymbol{p} = \boldsymbol{o} + \frac{n_z h}{\mathbf{n}\cdot\mathbf{v}}\,\mathbf{v}. \tag{7.50}$$

The x and y coordinates of $\boldsymbol{p}$ provide the offset that should be added to the original texture coordinates. All texture maps used by the pixel shader, including the normal map, are then resampled at these new coordinates that include the parallax shift.

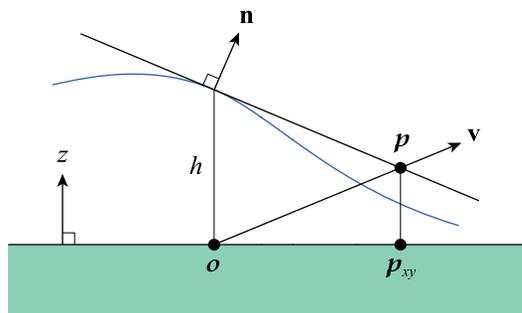**Figure 7.15.** The height h and normal vector $\mathbf{n}$ sampled at the point $\boldsymbol{o}$ are used to construct a plane that approximates the bumpy surface shown by the blue line. For a tangent-space view vector $\mathbf{v}$, the parallax offset is given by the x and y coordinates of the point $\boldsymbol{p}$ where the ray $\boldsymbol{o}+t\mathbf{v}$ intersects the plane.

In practice, the offset given by Equation (7.50) is problematic because the dot product $\mathbf{n} \cdot \mathbf{v}$ can be close to zero, or it can even be negative. This means that the offset can produce an arbitrarily large parallax shift toward or away from the viewer when $\mathbf{n}$ and $\mathbf{v}$ are nearly perpendicular. The usual solution to this problem, even though it has little geometric significance, is to gradually reduce the offset as $\mathbf{n} \cdot \mathbf{v}$ becomes small by simply multiplying by $\mathbf{n} \cdot \mathbf{v}$! This effectively drops the division from Equation (7.50) and gives us the new offset formula

$$\boldsymbol{p}_{xy} = n_z h \mathbf{v}_{xy}. \tag{7.51}$$

This offset generally produces good results, like those shown in Figure 7.16, and it is very cheap to calculate. Because we are no longer dividing out their magnitudes, however, we must ensure that both $\mathbf{n}$ and $\mathbf{v}$ have unit length before applying this formula.

The value of $n_z h$ in Equation (7.51) is precomputed for every texel in the normal map and stored in a separate *parallax map* having a single channel. To minimize storage requirements, an 8-bit signed format can be used in which texel values fall in the range $[-1, +1]$. Unsigned values h from the original height map are remapped to this range by calculating $2h - 1$ before multiplying by n_z and storing the results in the parallax map. A signed format is chosen so that texture coordinates are shifted both toward and away from the viewer when the full range of heights is well utilized. An original height of $h = 1/2$ corresponds to no offset, larger heights cause a surface to appear raised, and smaller values cause a surface to appear depressed. In the pixel shader, the sampled values of $n_z h$ are multiplied by $1/2$ to account for the doubling when they were converted to the signed format. They must also be multiplied by the same scale used when the normal map was generated so that the heights used in parallax mapping are the same as those that were used to calculate the per-texel normal vectors.

For bump maps containing steep changes in height, offsets given by Equation (7.51) can still be too large because the tangent plane becomes a poorer approximation as the size of the offset increases. Shifted color samples taken too far away from areas having a steep slope often produce visible artifacts. This problem can be eliminated in most cases by using k iterations and multiplying the offset by $1/k$ each time. A new value of $n_z h$ is fetched from the parallax map for each iteration, allowing each incremental parallax shift to be based on a different approximating plane.

The pixel shader code shown in Listing 7.9 implements parallax mapping with Equation (7.51), and it uses four iterations to mitigate the appearance of artifacts
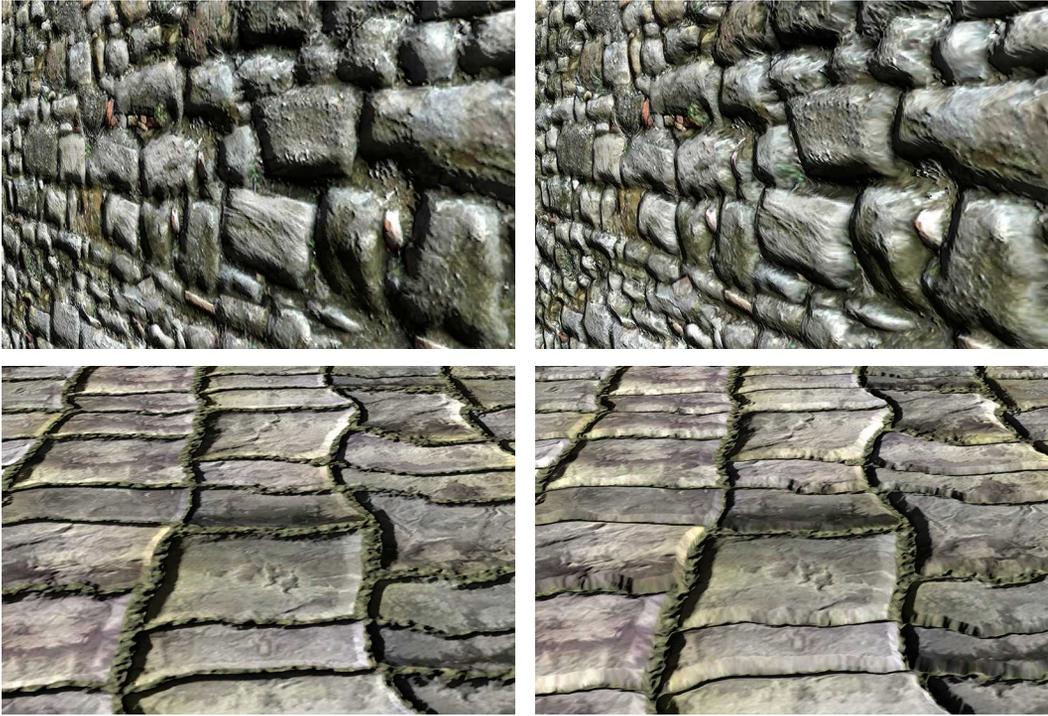
Figure 7.16. Two flat surfaces are rendered with only normal mapping in the left column. A parallax shift has been applied to the same surfaces in the right column. Texture coordinate offsets are calculated using Equation (7.51) with four iterations.

produced by steep slopes. The two-component scale value $\mathbf{u}$ passed to the function is given by

$$\mathbf{u} = \left(\frac{s}{2kr_x}, \frac{s}{2kr_y} \right), \tag{7.52}$$

and it accounts for several things that can be incorporated into a precalculated product. First, the it includes the height scale s that was originally used to construct the normal map. Second, since the heights are measured in units of texels, the parallax offsets must be normalized to the actual dimensions (r_x, r_y) of the height map, so the scale is multiplied by the reciprocals of those dimensions. (This is the only reason why there are two components.) Third, the scale includes a factor of $1/2$ to account for the multiplication by two when the heights were converted from unsigned to signed values. Fourth, the scale includes a factor of $1/k$, where k is the

number of iterations, so that each iteration contributes its proper share of the final result. Finally, the scale may include an extra factor not shown in Equation (7.52) that exaggerates the parallax effect.

Listing 7.9. This pixel shader function applies parallax mapping to the texture coordinates given by `texcoord` using four iterations and returns the final result. The texture specified by `parallaxMap` holds the signed values $n_z h$ belonging to the parallax map. The `vdir` parameter contains the tangent-space view direction, which must be normalized to unit length. The value of `scale` is given by Equation (7.52), where $k = 4$ in this code.

```
uniform Texture2D    parallaxMap;

float2 ApplyParallaxOffset(float2 texcoord, float3 vdir, float2 scale)
{
    float2 pdir = vdir.xy * scale;
    for (int i = 0; i < 4; i++)
    {
        // Fetch n.z * h from the parallax map.
        float parallax = texture(parallaxMap, texcoord).x;
        texcoord += pdir * parallax;
    }

    return (texcoord);
}
```

7.8 Horizon Mapping

While bump mapping with a parallax shift goes a long way toward making a flat surface look like it contains geometric detail, there is still one thing left that would complete the illusion and really make our materials look great. The bumps need to cast shadows. As shown in Figure 7.17, adding per-pixel shadows can make a dramatic difference even after bump and parallax mapping have been applied, especially when the light direction makes a large angle with the surface normal. These shadows are rendered with a technique call *horizon mapping*, which gets its name from the type of information that is stored in an additional texture map. Given an arbitrary light direction in tangent space, a horizon map tells us whether the light source is high enough above the horizon to exceed the height of any nearby bumps and thus illuminate the point being shaded. This allows us to cast high-quality shadows for the geometric detail stored in the original height map. The horizon mapping method is also fully dynamic, so the shadows reorient themselves correctly as the light source moves around.

Figure 7.17. Three examples of horizon mapping are shown. In the left column, only bump mapping with a parallax shift has been applied, and the lack of shadows in the crevices produces an unrealistic appearance. In the right column, horizon mapping has also been applied, and the shadows contribute considerably more visual information about the shape of the surface, giving the bumps the appearance of much greater volume. The light source is positioned to the upper right in all three cases.

For each texel, a horizon map contains eight values corresponding to eight different tangent directions spaced at 45-degree increments, and they are stored in the red, green, blue, and alpha channels in two layers of an array texture. Each value is the sine of the largest angle between the horizontal plane and the highest point in a 45-degree field centered on the direction to which the value corresponds. Even though the sines of horizon angles are precomputed and stored only for eight specific light directions, linear interpolation for arbitrary directions works very well in practice, and it reduces both storage space and computational expense. Furthermore, horizon maps can usually be constructed at lower resolutions compared to normal maps based on the same height map without a significant loss of rendering quality, especially if shadows with soft edges are desired. It is common for horizon maps to be half the width and height of a normal map, requiring only one quarter of the storage space of the full-resolution texture.

7.8.1 Horizon Map Construction

A horizon map is constructed from the same scaled height map that was used to construct a normal map and parallax map. For a texel at the coordinates (x, y), we read the base height h_0 from the height field and then search a neighborhood of some fixed radius r for larger height values. Whenever a greater height h is found at an offset of (i, j) texels, we calculate

$$\tan^2 \alpha = \frac{(h - h_0)^2}{i^2 + j^2},$$

(7.53)

where the angle α is the elevation angle between the horizontal plane at the height h_0 and the direction toward the higher point, as illustrated in Figure 7.18. Since the tangent is a monotonically increasing function over the range of angles that are possible in this setting, larger values given by Equation (7.53) correspond to larger elevation angles. The maximum values of the squared tangent are recorded for an array of horizontal directions holding several times the eight entries that we will ultimately need. After the whole neighborhood has been searched, each entry of this array is converted to a sine value with the trigonometric identity

$$\sin \alpha = \sqrt{\frac{\tan^2 \alpha}{\tan^2 \alpha + 1}}.$$

(7.54)

To obtain the eight values that we store in the horizon map, we simply average the entries in the array within a 45-degree span centered on the direction to which the value in the horizon map corresponds.
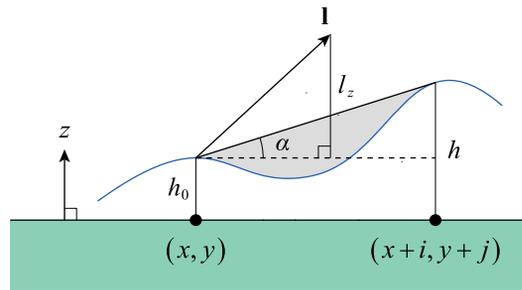
Figure 7.18. Each of the eight values stored in a horizon map for the texel location (x, y) is the sine of the maximum angle α made between the horizontal plane at the height h_0 and the direction to a nearby point having the height h at an offset of (i, j) texels. When rendering, the z component of the normalized tangent-space light direction $\mathbf{l}$ can be directly compared to $\sin \alpha$ to determine whether light can reach the point being shaded.

The code in Listing 7.10 constructs the horizon map for a given height map by searching the neighborhood within a 16-texel radius of each central location (x, y) and keeping track of the largest value of $\tan^2 \alpha$ for an array of 32 equally spaced horizontal directions. If the neighborhood extends beyond the bounds of the height map, then it wraps around to the opposite side. When we encounter a height at the neighborhood location $(x + i, y + j)$ that is greater than the height at the central location, the squared tangent can be applied to multiple array entries based on the angular size of the higher texel in the horizontal plane relative to the center. For simplicity, we assume that each texel has a radius of $\sqrt{2}/2$, in which case half the angular size of the higher texel is given by

$$\delta = \tan^{-1}\left(\frac{\sqrt{2}}{2 \, \|(i, j)\|} \right). \tag{7.55}$$

The squared tangent of α for this texel is applied to all array entries falling within an angular distance δ of the angle to the texel itself, which is equal to $\tan^{-1}(j/i)$.

The final values written to the horizon map are calculated by taking the average of the five nearest array entries for each multiple of 45 degrees. The red, green, blue, and alpha channels in the first layer of the horizon map contain the final values corresponding to directions making angles $0°$, $45°$, $90°$, and $135°$ with the positive x axis, respectively. The second layer contains the final values corresponding to directions making angles $180°$, $225°$, $270°$, and $315°$. These eight channels of two example horizon maps are shown separately in Figure 7.19 alongside the corresponding diffuse texture maps and normal maps.

As an added bonus, the same information that we calculate to create a horizon map can also be used to approximate the amount of ambient light that is occluded at each texel. This is discussed in more detail below in Section 7.8.3, but the code that generates an ambient light factor is included in Listing 7.10.

Listing 7.10. This function constructs a horizon map corresponding to the scaled height map having power-of-two dimensions width × height specified by heightMap and writes two layers of color data to the buffer supplied by horizonMap, which must be large enough to hold 2 × width × height values. The search neighborhood has a radius of 16 texels, and maximum squared tangent values are recorded for an array 32 horizontal direction angles. After the neighborhood has been searched, the array entries are converted to sine values using Equation (7.54), and groups of five are averaged to produce the final values written to the horizon map. An ambient occlusion map is also generated, and the results are written to the buffer specified by ambientMap, which must be large enough to hold width × height values. The value of ambientPower is used to exaggerate the occlusion.

```cpp
void ConstructHorizonMap(const float *heightMap, ColorRGBA *horizonMap,
                         float *ambientMap, float ambientPower, int32 width, int32 height)
{
    constexpr int kAngleCount = 32;      // Must be at least 16 and a power of 2.
    constexpr float kAngleIndex = float(kAngleCount) / two_pi;
    constexpr int kHorizonRadius = 16;

    for (int32 y = 0; y < height; y++)
    {
        const float *centerRow = heightMap + y * width;
        for (int32 x = 0; x < width; x++)
        {
            // Get central height. Initialize max squared tangent array to all zeros.
            float h0 = centerRow[x];
            float maxTan2[kAngleCount] = {};

            // Search neighborhood for larger heights.
            for (int32 j = -kHorizonRadius + 1; j < kHorizonRadius; j++)
            {
                const float *row = heightMap + ((y + j) & (height - 1)) * width;
                for (int32 i = -kHorizonRadius + 1; i < kHorizonRadius; i++)
                {
                    int32 r2 = i * i + j * j;
                    if ((r2 < kHorizonRadius * kHorizonRadius) && (r2 != 0))
                    {
                        float dh = row[(x + i) & (width - 1)] - h0;
                        if (dh > 0.0F)
                        {
                            // Larger height found. Apply to array entries.
                            float direction = atan2(float(j), float(i));
                            float delta = atan(0.7071F / sqrt(float(r2)));
```

```
                        int32 minIndex = int32(floor((direction - delta) * kAngleIndex));
                        int32 maxIndex = int32(ceil((direction + delta) * kAngleIndex));

                        // Calculate squared tangent with Equation (7.53).
                        float t = dh * dh / float(r2);
                        for (int32 n = minIndex; n <= maxIndex; n++)
                        {
                            int32 m = n & (kAngleCount - 1);
                            maxTan2[m] = fmax(maxTan2[m], t);
                        }
                    }
                }
            }
        }

        // Generate eight channels of horizon map.
        ColorRGBA *layerData = horizonMap;
        for (int32 layer = 0; layer < 2; layer++)
        {
            ColorRGBA color(0.0F, 0.0F, 0.0F, 0.0F);
            int32 firstIndex = kAngleCount / 16 + layer * (kAngleCount / 2);
            int32 lastIndex = firstIndex + kAngleCount / 8;

            for (int32 index = firstIndex; index <= lastIndex; index++)
            {
                float tr = maxTan2[(index - kAngleCount / 8) & (kAngleCount - 1)];
                float tg = maxTan2[index];
                float tb = maxTan2[index + kAngleCount / 8];
                float ta = maxTan2[(index + kAngleCount / 4) & (kAngleCount - 1)];

                color.red += sqrt(tr / (tr + 1.0F));
                color.green += sqrt(tg / (tg + 1.0F));
                color.blue += sqrt(tb / (tb + 1.0F));
                color.alpha += sqrt(ta / (ta + 1.0F));
            }

            layerData[x] = color / float(kAngleCount / 8 + 1);
            layerData += width * height;
        }

        // Generate ambient light factor.
        float sum = 0.0F;
        for (int32 k = 0; k < kAngleCount; k++) sum += 1.0F / sqrt(maxTan2[k] + 1.0F);
        ambientMap[x] = pow(sum * float(kAngleCount), ambientPower);
    }

    horizonMap += width;
    ambientMap += width;
    }
}
```

Diffuse Map   Normal Map   Horizon Map

R0 → G0 ↗ B0 ↑ A0 ↖

R1 ← G1 ↙ B1 ↓ A1 ↘

R0 → G0 ↗ B0 ↑ A0 ↖

R1 ← G1 ↙ B1 ↓ A1 ↘
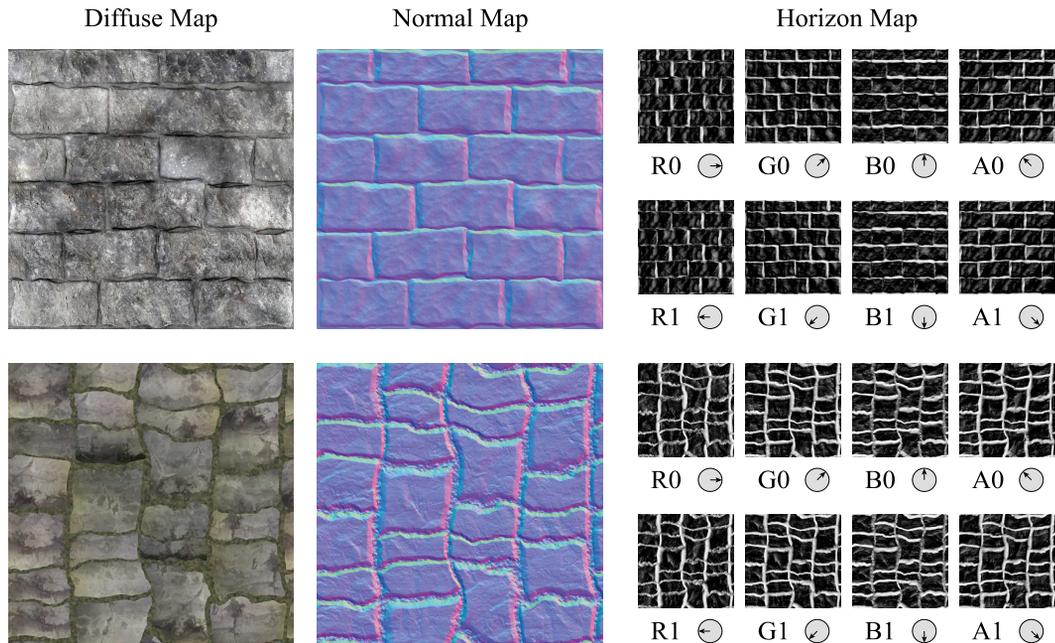
Figure 7.19. In each of these two examples, the eight channels of a horizon map are shown alongside the corresponding diffuse texture map and normal map. The horizon data corresponding to horizontal directions making angles of 0°, 45°, 90°, and 135° with the positive x axis are stored in the red, green, blue, and alpha channels of layer 0, and the horizon data for angles 180°, 225°, 270°, and 315° are stored in layer 1.

7.8.2 Rendering with Horizon Maps

In a pixel shader, the horizon mapping technique determines whether each pixel is illuminated by a light source or covered by a shadow. As described below, we would like a smooth transition between lit and shadowed pixels, so the horizon mapping calculation produces a value between zero and one specifying the fraction of incoming light that reaches each pixel. The ordinary reflected light color is then multiplied by this fraction to obtain the final lighting contribution.

The amount of computation needed to perform horizon mapping is surprisingly small considering the results. Most of the work involves determining the value of $\sin \alpha$ shown in Figure 7.18 for an arbitrary tangent-space light direction $\mathbf{l}$. For the sake of simplicity and speed, we linearly interpolate the values of $\sin \alpha$ stored in the horizon map for the two directions nearest the projected light direction (l_x, l_y).

It would be expensive to decide purely through computation which two channels of the horizon map should participate in the interpolation and with what weights. However, it is possible and quite convenient to store the interpolation weights for all eight channels in a special cube texture map that is accessed directly with the vector $\mathbf{l}$. We can encode weights for eight directions in a four-channel texture by taking advantage of the fact that if a weight is nonzero for one direction, then it must be zero for the opposite direction. This allows us to use positive weights when referring to any of the four channels in the first layer of the horizon map and negative weights when referring to any of the four channels in the second layer. In the weight cube map, we use a signed format with eight bits per channel to store values in the range $[-1, +1]$. For a four-component weight value $\mathbf{w}$ sampled with the light direction $\mathbf{l}$, we can compute the interpolated value of $\sin \alpha$ using the formula

$$\sin \alpha = \max(\mathbf{w}, 0) \cdot \mathbf{h}_0 + \max(-\mathbf{w}, 0) \cdot \mathbf{h}_1, \qquad (7.56)$$

where $\mathbf{h}_0$ and $\mathbf{h}_1$ are the four-channel sine values fetched from the two layers of the horizon map, and the max function is applied componentwise. The two dot products essentially mean that Equation (7.56) is a weighted sum of all eight horizon map channels, but at most two weights end up being nonzero.

The cube texture map containing the channel weights has the appearance shown in Figure 7.20 after remapping values to the range $[0, 1]$. A resolution of only 16×16 texels per face is sufficient in practice, and it requires a mere six kilobytes of storage. This texture can be generated by the code in Listing 7.11, which returns floating-point colors that must still be converted to a format having signed 8-bit components.

Once the interpolated value of $\sin \alpha$ has been calculated with Equation (7.56) using the information fetched from the horizon map, we can compare it to the sine of the angle that the vector $\mathbf{l}$ makes with the horizontal plane, which is simply l_z when $\mathbf{l}$ has been normalized to unit length. If $l_z \geq \sin \alpha$, then the light source is far enough above the horizon to illuminate the pixel being shaded. Otherwise, the pixel is in shadow. If we were to calculate a lighting factor F as

$$F = \begin{cases} 0, & \text{if } l_z < \sin \alpha; \\ 1, & \text{if } l_z \geq \sin \alpha, \end{cases} \qquad (7.57)$$

then the shadow would be correct, but its boundary would be hard and jagged as shown in Figure 7.21(a). We would instead like this lighting factor to smoothly transition from one to zero at the shadow's edge in order to produce the soft appearance shown in Figure 7.21(b). This can be accomplished by calculating

$$F = \eta\left(l_z - \sin\alpha\right) + 1 \tag{7.58}$$

and clamping it to the range $[0,1]$. The constant η is a positive number that determines how gradual the transition from light to shadow is. The value of F is always one when $l_z \geq \sin\alpha$, and it is always zero when $l_z \leq \sin\alpha - 1/\eta$. The transition takes place over a range of size $1/\eta$, so larger values of η produce harder shadows. The effect of choosing $\eta = 8$ is demonstrated in Figure 7.21(b).
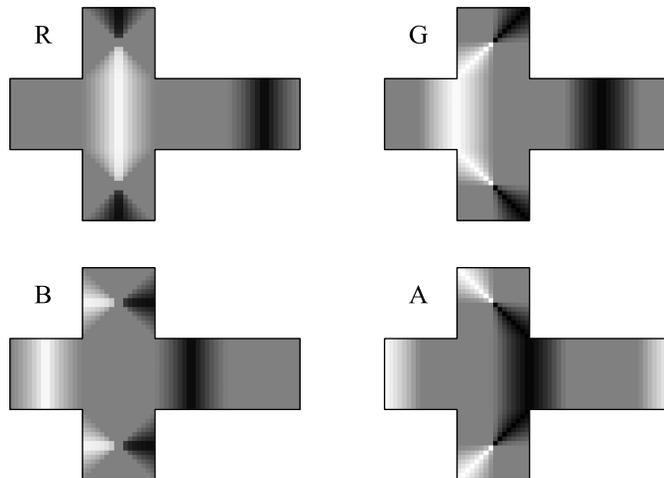
Figure 7.20. When sampled with coordinates given by the tangent-space light direction, this small cube texture map returns the channel weights used to linearly interpolate the sine values stored in the horizon map. For each of the separately shown red, green, blue, and alpha channels, black corresponds to a value of -1, and white corresponds to a value of $+1$.

The code in Listing 7.12 implements the horizon mapping technique. The texture coordinates passed to this function are the same as those used to fetch samples from the diffuse texture map and normal map after any parallax shift has been applied. The code calculates the linearly interpolated value of $\sin\alpha$ using Equation (7.56) and then applies Equation (7.58). For a specific tangent-space light direction that has already been normalized to unit length, the return value is the lighting factor by which the final shading contribution due to the light source should be multiplied.

Listing 7.11. This function generates the texel values for the cube texture map shown in Figure 7.20 as four-component floating-point colors. Each face is 16×16 texels, so the buffer supplied by `texel` must be large enough to hold 1536 values. The coordinates given by x and y fall in the centers of the 256 texels belonging to each face.

```
void GenerateHorizonCube(ColorRGBA *texel)
{
   for (int face = 0; face < 6; face++)
   {
      for (float y = -0.9375F; y < 1.0F; y += 0.125F)
      {
         for (float x = -0.9375F; x < 1.0F; x += 0.125F)
         {
            Vector2D  v;

            float r = 1.0F / sqrt(1.0F + x * x + y * y);
            switch (face)
            {
               case 0: v.Set(r, -y * r);        break;
               case 1: v.Set(-r, -y * r);       break;
               case 2: v.Set(x * r, r);         break;
               case 3: v.Set(x * r, -r);        break;
               case 4: v.Set(x * r, -y * r);    break;
               case 5: v.Set(-x * r, -y * r);   break;
            }

            float t = atan2(v.y, v.x) / pi_over_4;

            float red = 0.0F;
            float green = 0.0F;
            float blue = 0.0F;
            float alpha = 0.0F;

                 if (t < -3.0F) {red = t + 3.0F;   green = -4.0F - t;}
            else if (t < -2.0F) {green = t + 2.0F; blue = -3.0F - t;}
            else if (t < -1.0F) {blue = t + 1.0F;  alpha = -2.0F - t;}
            else if (t <  0.0F) {alpha = t;        red = t + 1.0F;}
            else if (t <  1.0F) {red = 1.0F - t;   green = t;}
            else if (t <  2.0F) {green = 2.0F - t; blue = t - 1.0F;}
            else if (t <  3.0F) {blue = 3.0F - t;  alpha = t - 2.0F;}
            else                {alpha = 4.0F - t; red = 3.0F - t;}

            texel->Set(red, green, blue, alpha);
            texel++;
         }
      }
   }
}
```
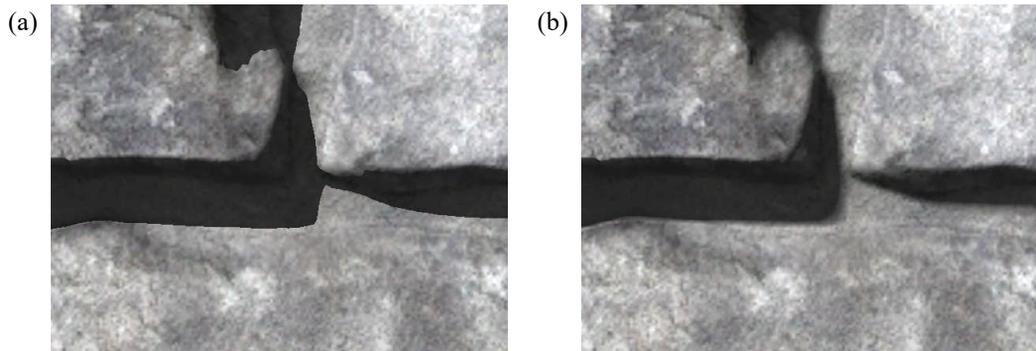
Figure 7.21. This close-up comparison shows the difference between a hard shadow and a soft shadow for a stone wall. (a) The illumination factor F is exactly one or zero, depending on whether $l_z \geq \sin \alpha$. (b) The illumination factor F is given by Equation (7.58) with $\eta = 8$.

Listing 7.12. This pixel shader code implements the horizon mapping technique. The texture specified by `horizonMap` is a 2D array texture map having two layers that contain the eight channels of the horizon map. The texture specified by `weightCube` is the special cube texture map that contains the channel weights for every light direction **l**. The `texcoord` parameter contains the same 2D texture coordinates used to sample the diffuse texture map, normal map, etc., after any parallax shift has been applied. The `ldir` parameter contains the tangent-space light direction, which must be normalized to unit length.

```
uniform Texture2DArray   horizonMap;
uniform TextureCube      weightCube;

float ApplyHorizonMap(float2 texcoord, float3 ldir)
{
    const float kShadowHardness = 8.0;

    // Read horizon channel factors from cube map.
    float4 weights = texture(weightCube, ldir);

    // Extract positive and negative weights for horizon map layers 0 and 1.
    float4 w0 = saturate(weights);
    float4 w1 = saturate(-weights);

    // Sample the horizon map and multiply by the weights for each layer.
    float s0 = dot(texture(horizonMap, float3(texcoord, 0.0)), w0);
    float s1 = dot(texture(horizonMap, float3(texcoord, 1.0)), w1);

    // Return lighting factor calculated with Equation (7.58).
    return (saturate((ldir.z - (s0 + s1)) * kShadowHardness + 1.0));
}
```

7.8.3 Ambient Occlusion Mapping

The information generated during the construction of a horizon map can also tell us how much ambient light is blocked at every texel. This allows us to create an *ambient occlusion map* containing a single channel of data that represents what fraction of incoming directions can receive ambient light. Despite having the word "occlusion" in its name, the values in the ambient occlusion map correspond to the amount of ambient light that is *not* blocked. Using this information to reduce the ambient luminance reaching a surface where it has features such as narrow pits and crevices makes bump mapping look much more realistic under low lighting levels. An example in which a stone surface is rendered with ambient light alone is shown Figure 7.22(a), and the surface appears very flat, even with a parallax map applied. The addition of an ambient occlusion map in Figure 7.22(b) produces much more variation in the brightness of the shading, and it brings out a lot of the geometric detail that was stored in the original height map.

Consider the elevation angle α to the nearby horizon, as previously shown in Figure 7.18. If the horizon existed at the same angle for all directions in the tangent plane, then the fraction F of ambient light reaching the texel would be given by

$$F = \int_0^{\pi/2-\alpha} \cos\theta \, d\theta, \tag{7.59}$$

where θ is the angle made with the z axis, and the cosine accounts for the Lambertian effect. The integral evaluates to $\cos\alpha$, and it does not need to be normalized

Figure 7.22. (a) A flat surface is rendered with a diffuse texture map and parallax map for a stone floor using ambient light only. (b) The same surface is rendered with an additional ambient occlusion map, and the geometric detail stored in the original height map is much more apparent.

because it has the value one when $\alpha = 0$. Using the value of $\tan^2 \alpha$ that is available when constructing a horizon map, we can calculate $\cos \alpha$ with the trigonometric identity

$$\cos \alpha = \frac{1}{\sqrt{\tan^2 \alpha + 1}}. \tag{7.60}$$

By simply averaging the values of $F = \cos \alpha$ corresponding to every direction in which $\tan^2 \alpha$ is calculated, we produce a good approximation to the overall fraction A of ambient light reaching the surface. In order to exaggerate the occlusion for artistic effect, we can calculate A^p for some power p and store the result in the ambient occlusion map. A value of $p = 2$ was used to generate the ambient occlusion map applied in Figure 7.22(b).

The calculation of A^p is implemented near the end of Listing 7.10, and it is stored as a floating-point value in the range $[0,1]$. It is convenient to take this value, convert it to an integer in the range $[0, 255]$, and store it in the alpha channel of the diffuse texture map. This texture map already has to be sampled during evaluation of the ambient shading term, and applying the ambient occlusion map is as simple as multiplying the whole ambient term by the factor stored in the alpha channel.

Exercises for Chapter 7

1. Let $\mathbf{n}$, $\mathbf{v}$, and $\mathbf{l}$ be the unit-length surface normal, view vector, and light direction. Define the vector $\mathbf{r}$ to be the reflection of the light direction across the normal, and define the vector $\mathbf{s}$ to be the reflection of the view direction across the normal. Prove that $\mathbf{r} \cdot \mathbf{v} = \mathbf{s} \cdot \mathbf{l}$.

2. Let $\mathbf{n}$, $\mathbf{v}$, $\mathbf{l}$, and $\mathbf{h}$ be the unit-length surface normal, view vector, light direction, and halfway vector. Show that

$$(\mathbf{n} \cdot \mathbf{h})^\alpha = \left[\frac{(\mathbf{n} \cdot \mathbf{l} + \mathbf{n} \cdot \mathbf{v})^2}{2(\mathbf{l} \cdot \mathbf{v} + 1)} \right]^{\alpha/2}.$$

3. Let $\boldsymbol{p}_0$, $\boldsymbol{p}_1$, and $\boldsymbol{p}_2$ be the vertices of a triangle, and let (u_i, v_i) be the texture coordinates associated with the vertex $\boldsymbol{p}_i$. Define the differences $\mathbf{e}_1$, $\mathbf{e}_2$, (x_1, y_1), and (x_2, y_2) as shown in Equation (7.34), and assume that the triangle's normal vector is given by $\mathbf{n} = \mathrm{nrm}(\mathbf{e}_1 \times \mathbf{e}_2)$. Determine a formula for the tangent vector $\mathbf{s}$ whose projected lengths onto the edges $\mathbf{e}_1$ and $\mathbf{e}_2$ are x_1 and x_2, respectively. That is, find a tangent vector $\mathbf{s}$ satisfying $\mathbf{e}_1 \cdot \mathbf{s} = x_1$ and $\mathbf{e}_2 \cdot \mathbf{s} = x_2$. Show

that $\mathbf{s} \cdot \mathbf{t} = 1$ and $\mathbf{s} \cdot \mathbf{b} = 0$, where $\mathbf{t}$ and $\mathbf{b}$ are the tangent and bitangent vectors given by Equation (7.37).

4. Define $\mathbf{A} \vdash \mathbf{B} = \mathbf{A} \vee \overline{\mathbf{B}}$ to be the *right interior product* between any blades $\mathbf{A}$ and $\mathbf{B}$. Since a surface normal is really a bivector, the cross product $\mathbf{n} \times \mathbf{t}$ between a normal vector $\mathbf{n}$ and a tangent vector $\mathbf{t}$ does not translate into a wedge product in Grassmann algebra in the same way that the cross product between two vectors does. Show that calculating a bitangent vector $\mathbf{b}$ with the formula $\mathbf{b} = \mathbf{n} \times \mathbf{t}$ is actually performing the operation $\mathbf{b} = \mathbf{n} \vdash \mathbf{t}$, where $\mathbf{n}$ is now recognized as a bivector and $\mathbf{t}$ remains an ordinary vector.

5. Determine a formula for a function $f_{\text{scale}}(\mathbf{m}, s)$ that scales a normal vector $\mathbf{m}$ fetched from a normal map in such a way that the result is what would be obtained by scaling the values in the height field from which $\mathbf{m}$ was derived by the factor s.

Chapter 8

Lighting and Shadows

The previous chapter discussed how light interacts with a surface having known material properties to determine how much light coming from a given direction is reflected toward the camera. However, we have not yet considered what happens to the light before and after the instant in time when it hits a solid object. This chapter fills in those gaps by first describing how light travels from its source to the surfaces it illuminates. Because light may be blocked by other objects before it even reaches a surface that we can see, methods for rendering shadows on a large scale are included here. We later visit the subject of fog and describe the changes that light reflected by a surface can experience on its way to the camera.

8.1 Light Sources

A light source can be characterized by two general properties, the color of the light rays it produces and the spatial distribution of those light rays in relation to the light's position and orientation. The color of a light source includes the overall brightness of the light, and it's commonly specified as an RGB color. However, this color may be derived from an emission spectrum in more sophisticated engines by integrating with color matching functions like those described in Chapter 5.

When it comes to spatial distribution, game engines typically support a small variety of light types that each have a fundamentally different purpose. The three most common types are point lights that radiate equally in all directions from one location like a light bulb, spot lights that shine predominantly in one direction like a flashlight, and infinite lights that model distant light sources like the sun. In this section, we discuss the qualities of each of these three types of lights.

8.1.1 Point Lights

A *point light* is a light source that radiates light at the same intensity in every direction from a single position in space. Point lights are sometimes called *omni-directional lights*, or just omni lights, to highlight their isotropic nature. True point lights don't actually exist in the real world, but the idealized model of light being emitted from a mathematical point provides an inexpensive approximation to a wide range of compact light sources.

As first discussed in Section 5.2, the total amount of visual power emitted by a light source is called its luminous flux Φ_V. When we apply our shading calculations in order to render a surface, we need to know how much of that total power is incident upon the surface per unit area, and this is a quantity called *illuminance*. Illuminance E_V is measured in *lux* (lx), which is a unit equivalent to lumens per square meter (lm/m^2), and it generally tells us how brightly a surface is lit.

The *luminous intensity* I_V of a light, often just called the *intensity*, is the amount of power radiated per unit solid angle. It is given its own special unit of measurement, the *candela* (cd), which is equivalent to lumens per steradian (lm/sr). Luminous intensity is generally a directional quantity that can be used to describe how the brightness of a light changes depending on the direction from the light source to an observer, and this is exemplified by the spot lights discussed below. However, for a point light radiating power equally in all directions, the luminous intensity is evenly distributed over the 4π steradians in a sphere, so it is always equal to the quantity

$$I_V = \frac{\Phi_V}{4\pi}, \tag{8.1}$$

which is independent of direction. In all cases, multiplying the intensity I_V by an arbitrary solid angle ω corresponding to a particular direction produces the amount of luminous flux $\varphi = I_V \omega$ radiated only within that angle.

The intensity of a point light can be thought of as an enormous number of rays emanating from the light's position and distributed uniformly over all directions, as shown in Figure 8.1. Each ray represents the luminous flux φ radiating through a small solid angle ω, and the sum over all rays is equal to the total luminous flux Φ_V emitted. As distance from the light source increases, the rays spread out, so the luminous flux incident upon an area of fixed size, the illuminance, is diminished. Imagine a spherical surface of radius R surrounding the light source, and consider that all of the rays must pass through it. The surface area of the sphere is $4\pi R^2$, so the illuminance E_V upon the interior surface of the sphere is given by

$$E_V = \frac{\Phi_V}{4\pi R^2}. \tag{8.2}$$
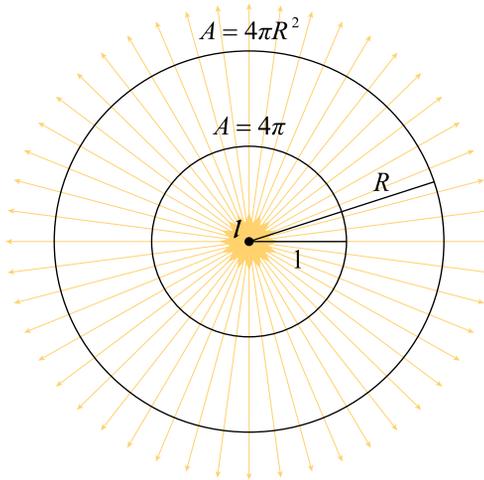
Figure 8.1. The rays emanating from a point light at the position l represent the luminous intensity I_V of the light source. The illuminance upon the interior surface of a sphere is inversely proportional to the square of the radius R. The decrease in brightness as R grows corresponds to the less dense spacing of the rays as they pass through a larger sphere.

The same number of rays always pass through the surface of the sphere, but they are spaced less densely on larger spheres. Equation (8.2) shows that the visual power reaching every unit of area on the surface is proportional to the reciprocal of R^2. This is known as the *inverse square law*, and we must account for it in our shading calculations.

When a point light is placed in a game world at a position l, we specify its intensity I_V instead of its luminous flux Φ_V because doing so will be consistent with the most convenient way of specifying the brightness of a light that does not radiate isotropically. For each point p on a surface being shaded, we subtract the light's position and calculate the distance $r = \|l - p\|$ to the light. The physically correct illuminance $E(r)$ at the distance r is then given by the function

$$E(r) = \frac{I_V}{r^2},\tag{8.3}$$

which implements the inverse square law. The positions l and p must be expressed in the same coordinate system, and it's common practice either to transform vertex positions, from which the shaded point p is derived, from a model's object space to world space or to transform l from world space to the model's object space.

Equation (8.3) presents a problem when it comes to implementing point lights in a game engine. There is no distance r at which the illuminance $E(r)$ decreases to zero, so the light theoretically has an effect on the entire world. Of course, we could calculate the distance r at which the illuminance drops to some value considered to be imperceptible, but that would be dependent on the intensity I_V. It is desirable to have the ability to adjust a light's brightness without affecting its reach, so we replace Equation (8.3) with

$$E(r) = I_V f(r), \tag{8.4}$$

where $f(r)$ is a generic function of our choosing that is guaranteed to be zero at a maximum distance $r = r_{max}$. The function $f(r)$ is commonly called a *distance attenuation* function (or a *distance falloff* function), and it produces values having units of steradians per square meter (sr/m^2). This function turns intensity into illuminance, determining how brightly a point at a distance r from the light is illuminated in relation to the light's intensity I_V.

The attenuation function $f(r)$ may be chosen from numerous possibilities, and a small variety of examples are shown in Figure 8.2. In addition to plots of the attenuation functions, the figure demonstrates how each one causes a surface to be illuminated differently. The first example shows what happens if we stay as close as possible to the inverse square model. In this case, we use a function of the form

$$f_{invsq}(r) = \frac{A}{r^2} + B, \tag{8.5}$$

where A and B are determined by the requirements that $f_{invsq}(r_{max}) = 0$ for the maximum distance r_{max} and $f_{invsq}(r_0) = 1$ for some reference distance r_0 where no attenuation takes place. Solving for the values of A and B produces

$$f_{invsq}(r) = \text{sat}\left[\frac{r_0^2}{r_{max}^2 - r_0^2}\left(\frac{r_{max}^2}{r^2} - 1\right)\right], \tag{8.6}$$

which is shown in Figure 8.2(a) and implemented in Listing 8.1. The saturation operation applied here stops negative values from be generated at distances greater than r_{max}. It also takes care of another practical issue arising from the inverse square model by preventing the illuminance from becoming arbitrarily large as r approaches zero. The illuminance simply stays at its maximum value for any distances less than or equal to the reference distance r_0, which may be regarded as the physical size of the light source itself.
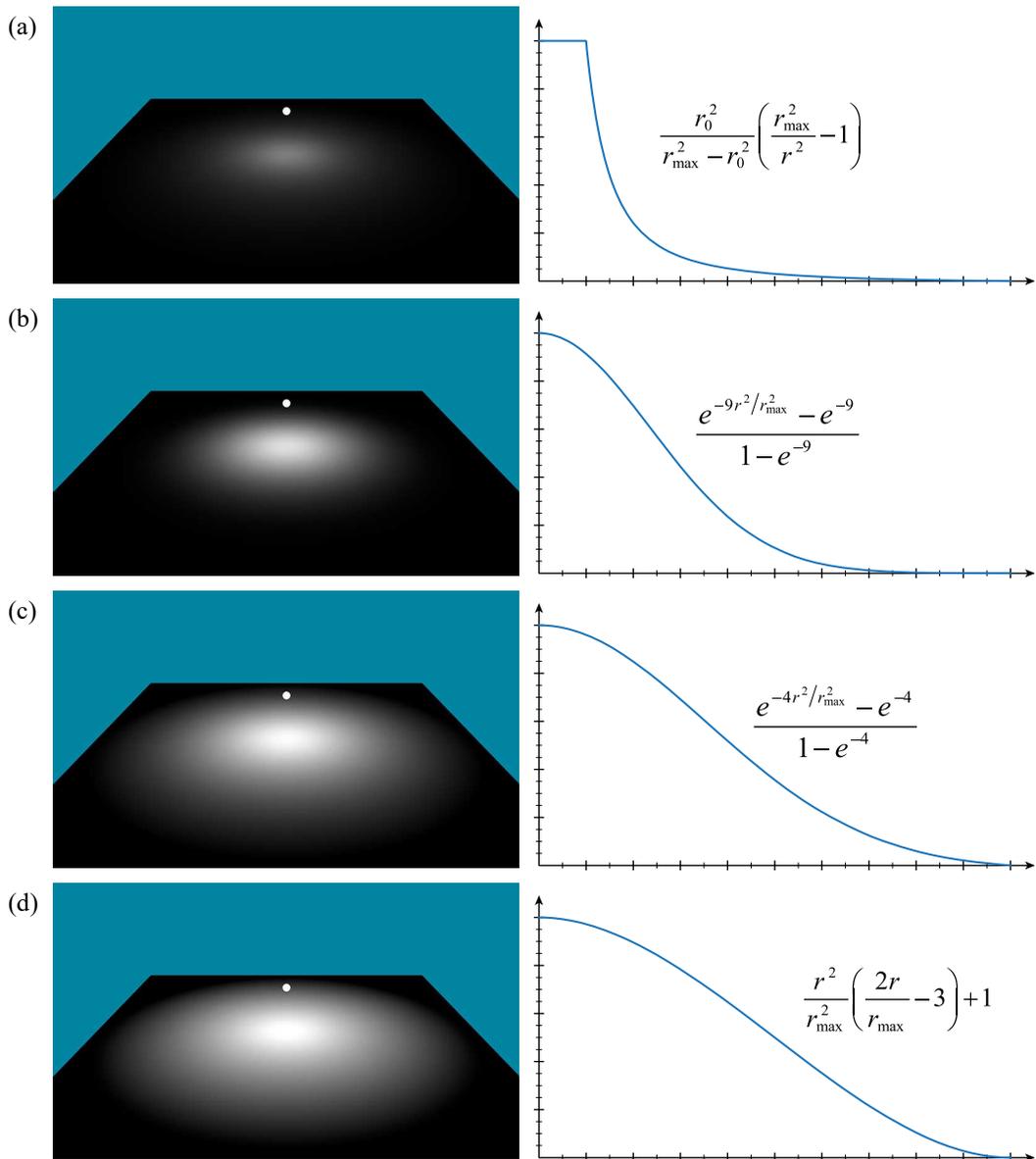
Figure 8.2. A small variety of light attenuation functions are applied to a point light illuminating a flat surface on the left, and their corresponding graphs are shown on the right. The horizontal axis of each graph extends from zero to r_{max}, and the vertical axis extends from zero to one. The specific functions are (a) the nearly physically correct $f_{invsq}(r)$, (b) the exponential function $f_{exp}(r)$ with $k = 3$, (c) the exponential function $f_{exp}(r)$ with $k = 2$, and (d) the cubic polynomial $f_{smooth}(r)$.

Listing 8.1. This pixel shader function calculates the inverse square attenuation given by Equation (8.6). The uniform constant `attenConst` holds the values $r_0^2 / (r_{max}^2 - r_0^2)$ and r_{max}^2.

```
uniform float2  attenConst;  // (r0^2 / (rmax2 - r0^2), rmax2)

float CalculateInverseSquareAttenuation(float3 p, float3 l)
{
    float3 ldir = l - p;
    float r2 = dot(ldir, ldir);
    return (saturate(attenConst.x * (attenConst.y / r2 - 1.0)));
}
```

Since the attenuation function $f(r)$ becomes zero at a specific distance r_{max} from its position, a point light has an effect on a spherical volume of space with a known radius. Any geometries intersecting that volume are at least partially illuminated by the light, and we must therefore include the light's contribution in the shading calculations for those geometries. For performance reasons, we would like to minimize the number of geometries illuminated by each light source, and that is most effectively accomplished by minimizing the radius associated with each point light. However, the inverse square attenuation function given by Equation (8.6) makes this goal difficult to achieve because it decreases quickly for distances near the light source and takes on very small values over most of its domain. As a consequence, most surfaces illuminated by a point light using an inverse square attenuation function are dimly lit. To provide adequate illumination in some areas of a game world, the radius of such a light often needs to be increased substantially beyond the region of influence that we care about. However, the tiny contributions that the light makes to the additional geometries intersecting the outer portions of the light's bounding sphere cost just as much as the larger contributions made close to the sphere's center. This is an inefficient use of the computational resources available on the GPU, so it makes more sense from a practical engineering perspective to choose a nonphysical attenuation function that provides greater illumination over a large portion of the bounding sphere.

A class of exponential attenuation functions that decrease to zero more gradually than the inverse square function is given by

$$f_{exp}(r) = \text{sat}\left(\frac{e^{-k^2 r^2 / r_{max}^2} - e^{-k^2}}{1 - e^{-k^2}}\right). \tag{8.7}$$

The function $f_{exp}(r)$ attains a maximum value of one at $r = 0$ and decreases to zero at $r = r_{max}$, as required. It does not make use of a reference distance r_0 as the inverse

square function did. The saturation operation ensures that no negative values are produced at distances beyond r_{max}, but there is no need for it to clamp values to one since that is already the function's global maximum. The constant k provides some control over the general brightness within a point light's bounding sphere. Smaller values of k cause the illuminance to stay larger until the distance gets closer to r_{max}, as demonstrated by the examples shown in Figures 8.2(b) and 8.2(c) where $k = 3$ and $k = 2$, respectively. As the value of k is lowered, the derivative of $f_{exp}(r)$ becomes larger at $r = r_{max}$, and this can cause the light's brightness to reach zero somewhat abruptly at the light's boundary. Listing 8.2 shows an implementation of the exponential attenuation function.

Listing 8.2. This pixel shader function calculates the exponential attenuation given by Equation (8.7). The uniform constant `attenConst` holds the values $-k^2/r_{max}^2$, $1/\left(1 - e^{-k^2}\right)$, and $e^{-k^2}/\left(1 - e^{-k^2}\right)$.

```
uniform float3  attenConst;  // (-k2/rmax2, 1/(1 - exp(-k2)), exp(-k2)/(1 - exp(-k2)))

float CalculateExponentialAttenuation(float3 p, float3 l)
{
    float3 ldir = l - p;
    float r2 = dot(ldir, ldir);
    return (saturate(exp(r2 * attenConst.x) * attenConst.y - attenConst.z));
}
```

Another possible attenuation function is the cubic polynomial having the form

$$f_{smooth}(r) = a\frac{r^3}{r_{max}^3} + b\frac{r^2}{r_{max}^2} + c\frac{r}{r_{max}} + 1. \tag{8.8}$$

By requiring that $f_{smooth}(r_{max}) = 0$ and that the derivative is zero at both $r = 0$ and $r = r_{max}$, we find that $a = 2$, $b = -3$, and $c = 0$. This attenuation function can thus be written as

$$f_{smooth}(r) = \text{sat}\left[\frac{r^2}{r_{max}^2}\left(\frac{2r}{r_{max}} - 3\right) + 1\right], \tag{8.9}$$

where the saturation operation again prevents negative values beyond $r = r_{max}$. The function $f_{smooth}(r)$ is shown in Figure 8.2(d) and implemented in Listing 8.3. It allows the light to remain a little brighter than $f_{exp}(r)$ for many values of k while still easing out at a slow rate near the light's boundary.

Listing 8.3. This pixel shader function calculates the smooth attenuation given by Equation (8.9). The uniform constant `attenConst` holds the values $1/r_{max}^2$ and $2/r_{max}$.

```
uniform float2  attenConst;  // (1 / rmax2, 2 / rmax)

float CalculateSmoothAttenuation(float3 p, float3 l)
{
    float3 ldir = l - p;
    float r2 = dot(ldir, ldir);
    return (saturate(r2 * attenConst.x * (sqrt(r2) * attenConst.y - 3.0) + 1.0));
}
```

Once the illuminance $E(r) = I_V f(r)$ has been calculated in a pixel shader by applying an attenuation function to a light's intensity, it scales the color of the light C_{light} to produce the quantity

$$C_{illum} = C_{light} I_V f\left(\|\boldsymbol{l} - \boldsymbol{p}\|\right) \tag{8.10}$$

representing the brightness and color of the light reaching the point $\boldsymbol{p}$ from the light's position $\boldsymbol{l}$. Ordinarily, this is the end of the lighting calculation, but an additional factor can be included to project an image on the environment using a cube texture map, as shown in Figure 8.3. Such a texture map holds a prerendered image of nearby geometry to which the point light is rigidly attached, and it can be used to project the shadows of opaque geometry or the colors of transparent geometry. To calculate the coordinates from which a sample is fetched from the cube texture map, the object-space position $\boldsymbol{p}_{object}$ of the point being shaded must be transformed into the light space position $\boldsymbol{p}_{light}$ using the formula

$$\boldsymbol{p}_{light} = \mathbf{M}_{light}^{-1} \mathbf{M}_{object} \boldsymbol{p}_{object}, \tag{8.11}$$

where $\mathbf{M}_{object}$ and $\mathbf{M}_{light}$ are the object-space to world-space transformations for the object being rendered and the light source, respectively. Since the light source is centered at the origin in its own local coordinate system, the value of $\boldsymbol{p}_{light}$ is used directly as the 3D direction vector needed to sample the cube texture map. When the transformation $\mathbf{M}_{light}$ changes because the light moves or rotates in some way, the projection changes along with it to match the coordinate axes of the light space.

8.1.2 Spot Lights

A *spot light* is a type of light source that emits light from a single position in space like a point light does, but its radiation is not equally distributed over all directions. Instead, there is a primary direction in which the intensity is the greatest, and the

(a) (b)
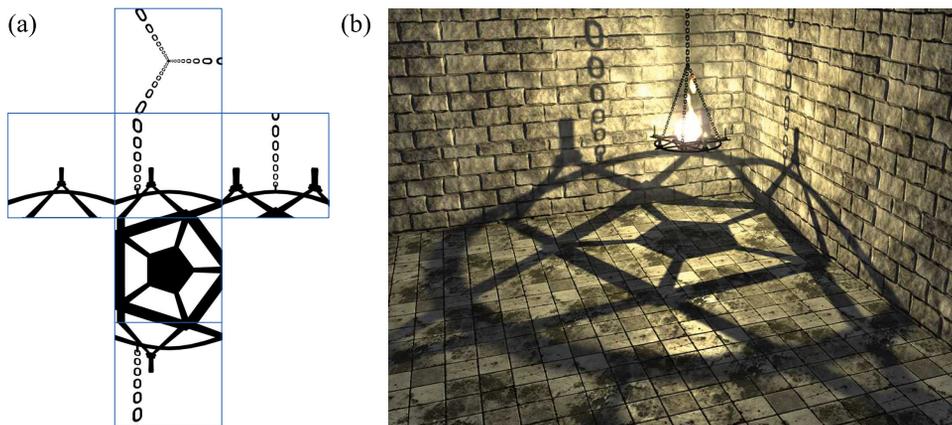
Figure 8.3. (a) A cube texture map captures the nearby geometry to which a point light is attached. (b) The texture is projected onto the environment by transforming shaded points into light space to be used directly as cube texture coordinates.

intensity in other directions becomes lower as the angle made with the primary direction increases. For computational convenience, the primary direction is usually aligned with one of the object-space axes of the light source. We choose to align our spot lights so the greatest intensity is radiated along the positive z axis in the light's local coordinate system.

A distance attenuation function $f(r)$ is applied to a spot light in the same way that it is applied to a point light. However, a spot light has an additional unitless function $h(\theta)$ that controls how the intensity decreases as a function of the angle θ made with the light's positive z axis. The function $h(\theta)$ is called an *angular attenuation* function (or a *angular falloff* function), and it is often expressed in terms of $\cos\theta$ so that an actual angle never needs to be calculated. The value of $\cos\theta$ is particularly easy to obtain because, for a point p that has been transformed into light space where the spot light itself is located at the origin, we have

$$\cos\theta = \frac{p_z}{\|p\|}. \tag{8.12}$$

Using both the distance and angular attenuation functions, the color and brightness of illumination reaching a light-space position p is given by

$$C_{\text{illum}} = C_{\text{light}} I_{\text{V}} h\left(\frac{p_z}{\|p\|}\right) f(\|p\|), \tag{8.13}$$

where C_{light} is the color of the light, and I_{V} is its maximum intensity.

Angular attenuation functions don't really have any physical requirements, so they can be chosen on a purely artistic basis to achieve some desired effect. One possible angular attenuation function is given by

$$h_{\text{power}}(\theta) = [\max(\cos\theta, 0)]^k, \tag{8.14}$$

where k is some constant power to which the cosine is raised. Higher values of k cause the spot light to be more tightly focused along the light's z axis. Because the cosine function is clamped to zero, the value of $h_{\text{power}}(\theta)$ is zero whenever $\theta \geq 90°$. This means that no points behind the light where $p_z \leq 0$ receive any illumination, and thus any object lying entirely on the negative size of the plane $z = 0$ can be skipped altogether.

When using the function $h_{\text{power}}(\theta)$, the entire hemisphere of directions where $p_z > 0$ receives some nonzero illumination, and this volume of space is almost always larger than what actually needs to be lit. Instead, it is common to choose a maximum angle θ_{max} at which the angular attenuation reaches zero and use some monotonically decreasing function $h(\theta)$ that fits our needs and satisfies the requirements that $h(0) = 1$ and $h(\theta_{\text{max}}) = 0$. A minimum angle θ_{min} may also be chosen such that $h(\theta) = 1$ for all $\theta \leq \theta_{\text{min}}$, and the pair $(\theta_{\text{min}}, \theta_{\text{max}})$ then represents the range of angles over which the angular attenuation takes place. Using these values, we can define

$$t(\theta) = \text{sat}\left(\frac{\cos\theta - \cos\theta_{\text{min}}}{\cos\theta_{\text{max}} - \cos\theta_{\text{min}}}\right) \tag{8.15}$$

to provide a parameter value such that $t(\theta_{\text{min}}) = 0$ and $t(\theta_{\text{max}}) = 1$. The angular attenuation function can then be defined as

$$h_{\text{param}}(\theta) = a(t(\theta)), \tag{8.16}$$

where $a(t)$ is any monotonically decreasing function with $a(0) = 1$ and $a(1) = 0$. A spot light using this kind of angular attenuation is said to have an *outer cone angle* θ_{max}, outside of which the intensity is zero, and an *inner cone angle* θ_{min}, inside of which the intensity is the maximum value.

Although the exact volume of space illuminated by a spot light is bounded by a cone, it is more useful to approximate this space by a pyramid having four planar sides. This is particularly convenient for rendering shadows for a spot light, as discussed in Section 8.3, because the pyramidal shape can be used directly as a light-centric view frustum. The planar boundary also allows the same view frustum culling methods described in Chapter 9 to be used for determining what set of objects can be lit by a spot light.

Using a pyramidal shape for a spot light additionally provides a natural boundary for a projected texture map. As shown in Figure 8.4, a rectangular texture map can be applied to a spot light to determine its color as a function of the light-space position $\boldsymbol{p}_{\text{light}}$ of a point being rendered. The coordinates at which the texture is sampled are determined by applying a projection matrix $\mathbf{P}_{\text{spot}}$ having the form

$$\mathbf{P}_{\text{spot}} = \begin{bmatrix} g/2s & 0 & 1/2 & 0 \\ 0 & g/2 & 1/2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \tag{8.17}$$

which transforms the point $\boldsymbol{p}_{\text{light}}$ into the point

$$\boldsymbol{p}_{\text{spot}} = \mathbf{P}_{\text{spot}}\boldsymbol{p}_{\text{light}} = \left(\frac{g}{2s}p_x + \frac{1}{2}p_z, \frac{g}{2}p_y + \frac{1}{2}p_z, 0, p_z \right). \tag{8.18}$$

Here, the meanings of the constants g and s are the same as they were in Chapter 6 for a perspective projection. The value of g is the projection distance given by

$$g = \frac{1}{\tan \theta_{\text{max}}}, \tag{8.19}$$

and the value of s is the aspect ratio of the texture map. The coordinates of $\boldsymbol{p}_{\text{spot}}$ are calculated in the vertex shader and interpolated. (Since the z coordinate is always zero, it is ignored.) In the pixel shader, we divide by the w coordinate to perform the projection and obtain the 2D texture coordinates

$$(u, v) = \left(\frac{g}{2s} \cdot \frac{p_x}{p_z} + \frac{1}{2}, \frac{g}{2} \cdot \frac{p_y}{p_z} + \frac{1}{2} \right). \tag{8.20}$$

These coordinates always fall in the range $[0,1]$ for a point $\boldsymbol{p}_{\text{light}}$ located inside the spot light's pyramidal boundary.

A texture map can be used not only as a projected image, but also as a way to store precomputed angular attenuation. By premultiplying each texel value by the intensity factor determined by a particular function $h(\theta)$, the angular attenuation calculation can be avoided in the pixel shader. As demonstrated in Figure 8.4, the texture map is simply black for any projected points that should not receive any illumination. The texture map's border color should also be set to black, and the wrap modes should be set to clamp to the border. This ensures that no colors along the edges of the texture map are smeared onto parts of illuminated geometries that fall outside the spot light's pyramidal boundary.
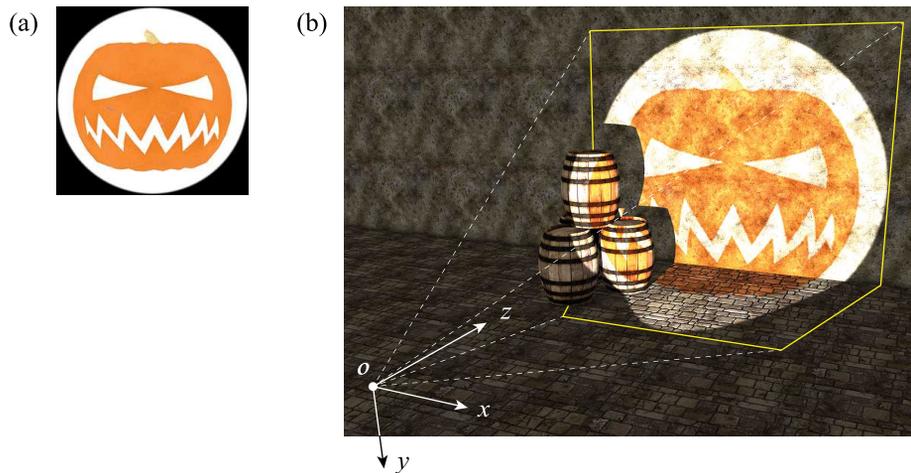
Figure 8.4. A spot light is bounded by a pyramidal shape similar to a view frustum, and a texture map is projected onto the environment. (a) The texture map stores colors and angular attenuation. In this case, the intensity abruptly falls to zero at the outer cone angle θ_{max}. (b) A projection matrix is applied to light-space positions in the vertex shader, and a perspective divide is performed in the pixel shader, producing the final texture coordinates.

8.1.3 Infinite Lights

An *infinite light* is a light source that is extremely far away relative to the size of the world being rendered by the game engine. An infinite light is used to represent the illumination due to a light source such as the sun or moon. Even though the rays emitted by such a light source do spread out and become dimmer over very large distances, they can be considered to be parallel and of constant brightness for all practical purposes inside the comparatively tiny volume of space in which most games take place. Mathematically, we treat these lights as if they are infinitely far away, and this is why they are named infinite lights. The implementation of an infinite light is particularly easy because the direction to the light is a constant vector. For this reason, infinite lights are also called *directional lights*.

It is natural to align the direction in which an infinite light radiates with one of its local coordinate axes. As we did with spot lights, we choose to align this direction with the positive z axis. In a shader, the vector $\mathbf{l}_{infinite}$ representing the direction toward the light source is equal to the opposite direction, the negative z axis. In world space, it is given by

$$\mathbf{l}_{infinite} = -\mathbf{M}_{light[2]}, \tag{8.21}$$

which is the negated third column of the object-space to world-space transform $\mathbf{M}_{light}$ for the light node.

Although the luminous intensity I_V, in candela, of a distant light source like the sun may be a known quantity, it is not generally a useful measure of brightness for infinite lights. When we place the light infinitely far away, the solid angle subtended by any objects we see in a game world is zero, so we cannot calculate any meaningful illuminance from the intensity. Instead, we directly specify a constant illuminance E_V, in lux, for an infinite light. We can consider this to be the result of choosing a representative distance R for the entire game world and setting

$$E_V = \frac{I_V}{R^2}.$$
(8.22)

This accounts for the attenuation up front, so the color and brightness of illumination reaching any point is the constant

$$C_{illum} = C_{light} E_V,$$
(8.23)

where C_{light} is again the color of the light. The value of R would ordinarily have a very large value, such as one astronomical unit, so any differences in position on the relatively small scale of a game world would not change the illuminance by a significant amount. For the sun and earth, the value of E_V is about 100 kilolux (klx) on the clearest sunny days. The moon may also be treated as an infinite light, and the brightest full moon illuminates the earth with about $E_V = 0.3\,\text{lx}$.

8.2 Extent Optimization

When rendering the contribution from a point light or spot light, there are often large geometries that are only partly illuminated by the light but must each still be drawn as a whole triangle mesh. Some examples are small lights illuminating sections of the floor and walls in a long corridor or, as shown in Figure 8.5(a), a campfire in the woods illuminating the terrain and some large trees. The brightly highlighted areas in Figure 8.5(b) show the geometries that are affected by the light source and demonstrate how they can extend far beyond the relatively small radius of the light's bounding sphere. When these geometries are drawn, a lot of time can be wasted filling pixels for which the contribution from the light is zero, and this motivates us to search for techniques that reduce the number of pixels drawn by as much as can be done through practical means.

Fortunately, we can take advantage of the hardware scissor test to easily eliminate almost all of the pixels that can't be illuminated by considering the x and y

Figure 8.5. (a) A campfire at the center of the frame contains a point light source. (b) The bright white areas correspond to the full extents of the individual geometries that intersect the light's bounding sphere. (c) The scissor rectangle enclosing the projected bounds of the light limits the area of the viewport that is rendered. (d) The number of pixels rendered is further reduced by applying the depth bounds test to limit the range of z values that could be illuminated.

extents of the light's bounding sphere in viewport space. Before rendering the contribution from the light source, we calculate the smallest rectangle on the projection plane that encloses the projection of the bounding sphere, and we set the scissor rectangle to the corresponding viewport coordinates. The result is that rendering is restricted to a much smaller area, as shown in Figure 8.5(c), and the pixel shader is executed for far fewer pixels.

By the time contributions from point lights are being rendered, it is ordinarily safe to assume that the final contents of the depth buffer have already been established by earlier rendering passes. This being the case, we can also calculate the z extents of the light's bounding sphere and use them as the minimum and maximum values for the depth bounds test. Many unilluminated pixels will already have

depth values outside that range, so the number of pixels rendered is typically further reduced to those shown in Figure 8.5(d), which is now a considerably smaller number than what we began with.

8.2.1 Scissor Rectangle

Suppose that a point light source having a bounding sphere of radius r is located at a position l in camera space, where the camera is located at the origin o and is pointed in the $+z$ direction. Our goal is to calculate the rectangle on the projection plane shown in Figure 8.6 that minimally encloses the projection of the bounding sphere. We cannot simply add the positive and negative radius r to the x and y coordinates of the center point l and then project the results because doing so would cause small but significant portions of the bounding sphere to be incorrectly chopped off. Instead, we must find planes parallel to the camera-space x and y axes that contain the origin and are tangent to the bounding sphere, and this requires a more involved calculation.

We begin by assuming that $l_z > -r$ because otherwise, the light source would be entirely behind the camera and excluded from the rendering process by the culling methods discussed in Chapter 9. We then have a bounding sphere that is at least partially in front of the x-y plane in camera space, and we'd like to know what ranges of x and y coordinates it covers on the projection plane, which is placed at a distance g from the camera. On the projection plane, the viewport covers the range $[-1, +1]$ in the y direction and the range $[-s, +s]$ in the x direction, where s is the aspect ratio.

The range of x coordinates can be determined by considering the 2D problem shown in Figure 8.7, which lies in a plane parallel to the x-z plane. For this calculation, we translate vertically so that all the y coordinates are zero and then define the point p as

$$p = (l_x + u, 0, l_z).$$

(8.24)

Our goal is to calculate the distance u that causes the line connecting the origin to the point p to be tangent to the sphere. This calculation is described here only for the x extents of the sphere, and a similar calculation in the y-z plane can be used to determine the y extents.

To be tangent to the sphere, the distance between the center l and the parametric line $o + tp$ must be the radius r. Using the square of Equation (3.20), this requirement can be expressed as

$$\frac{(l \times p)^2}{p^2} = r^2.$$

(8.25)
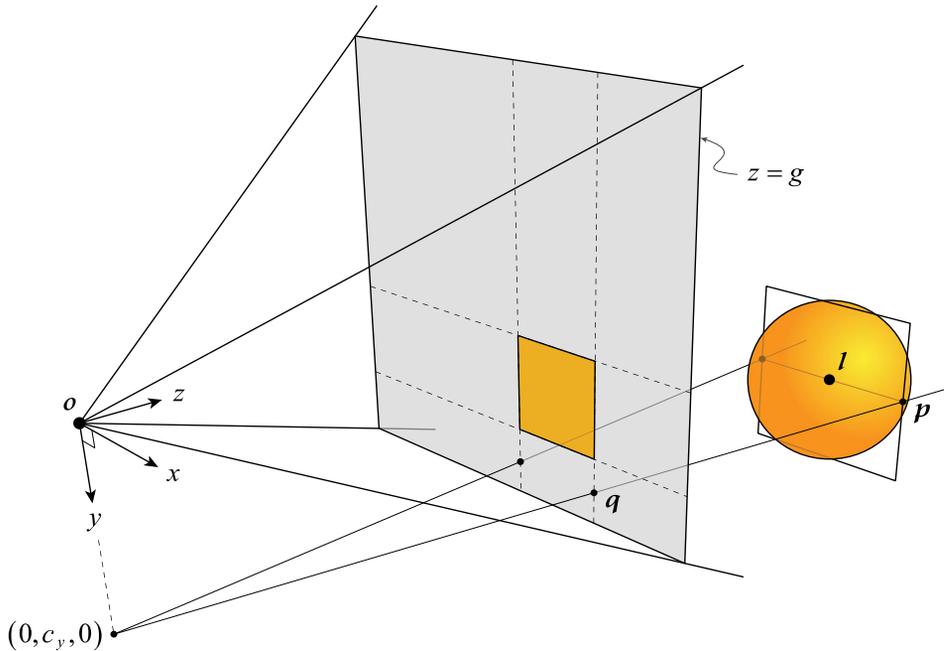
Figure 8.6. The extents of a light source's bounding sphere, centered at the point l in camera space, determine a scissor rectangle on the projection plane at $z = g$. To find the left and right sides of the rectangle, y coordinates are ignored, and two points p to the left and right of the center l are calculated such that lines containing the origin and p are tangent to the bounding sphere. The point q represents the intersection of one of these lines with the projection plane. To top and bottom sides of the rectangle are determined with a similar approach that ignores x coordinates.

Since l and p both lie in the x-z plane, the only nonzero component of their cross product is the y component given by $l_z p_x - l_x p_z$. Substituting the value of p from Equation (8.25) and simplifying the numerator gives us

$$\frac{l_z^2 u^2}{(l_x + u)^2 + l_z^2} = r^2. \tag{8.26}$$

When we expand this and group by powers of u, we arrive at the quadratic equation

$$\left(l_z^2 - r^2\right) u^2 - 2r^2 l_x u - r^2 \left(l_x^2 + l_z^2\right) = 0. \tag{8.27}$$
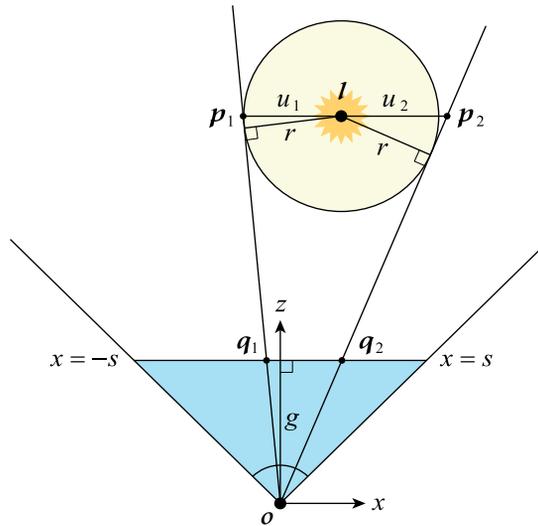
Figure 8.7. The x extents of the light's bounding sphere on the projection plane are determined by calculating distances u_1 and u_2 such that the lines connecting the origin to the points p_1 and p_2 are tangent to the sphere. These points are then projected into the viewport as the points q_1 and q_2.

By making the assignments $a = l_z^2 - r^2$ and $b = l_x r^2$, we can express the solutions to this equation as

$$u = \frac{b \pm \sqrt{b^2 + ar^2\left(l_x^2 + l_z^2\right)}}{a}.$$

(8.28)

This provides the values of u that we can plug back into Equation (8.24) to obtain points p_1 and p_2 that are then projected onto the viewport as the points q_1 and q_2 determining the left and right extents of the optimal scissor rectangle. The projection is performed by calculating

$$q_x = \frac{g}{l_z} p_x,$$

(8.29)

where we need only the x coordinate for each point.

There are several special cases that we must consider. First, if the value of a is near zero, then the bounding sphere is nearly tangent to the x-y plane, and there is only one solution for u given by

$$u = \frac{l_x^2 + r^2}{2l_x}.$$

(8.30)

If $l_x > 0$, then u corresponds to the left edge of the scissor rectangle, and if $l_x < 0$, then u corresponds to the right edge. In the case that l_x is very close to zero, the entire width of the viewport must be included in the scissor rectangle. However, we can still apply the depth bounds test when this happens.

Next, assuming that a is not near zero, the discriminant d under the radical in Equation (8.28), which simplifies to

$$d = r^2 l_z^2 \left(a + l_x^2 \right), \tag{8.31}$$

must be positive. Otherwise, the camera is inside the bounding sphere (or on the surface of the bounding sphere in the case that $d = 0$), and the entire viewport can be affected by the light source. As before, we can still apply the depth bounds test.

Now assuming that $a \neq 0$ and $d > 0$, Equation (8.28) produces two valid solutions, but we still need to detect cases in which a point of tangency lies behind the camera. If $l_z > r$, which is equivalent to having $a > 0$, then the entire sphere lies in front of the x-y plane, then both solutions are valid, and we know that choosing the negative sign in Equation (8.28) yields the smaller value of u as well as the smaller value of q_x.

In the remaining case that $a < 0$, one of the solutions to Equation (8.28) represents a point of tangency behind the camera, and the projection of the sphere extends to infinity in either the positive or negative x direction. If $l_x > 0$, then the larger value of q_x corresponds to the left side of the scissor rectangle. If $l_x < 0$, then the smaller value of q_x corresponds to the right side of the scissor rectangle. (Note that l_x cannot be zero in this case because that would imply that $d < 0$.)

The code in Listing 8.4 implements the scissor rectangle calculation for both the x and y directions and handles all of the special cases. The scissor rectangle is scaled so that the range $[-1, +1]$ represents the full extent of the viewport in both directions, which requires that we divide the projected x coordinates by the aspect ratio s. Any scissor rectangle edges falling outside the viewport are clamped to the range $[-1, +1]$, and this causes the scissor rectangle to have zero area when the bounding sphere lies completely outside the view frustum.

The extents of the scissor rectangle given by the ranges $[x_{\min}, x_{\max}]$ and $[y_{\min}, y_{\max}]$ can be transformed into integer pixel-aligned edges for a viewport of width w and height h by the formulas

$$left = \left\lfloor \frac{w}{2}(x_{\min} + 1) \right\rfloor \qquad right = \left\lceil \frac{w}{2}(x_{\max} + 1) \right\rceil$$

$$top = \left\lfloor \frac{h}{2}(y_{\min} + 1) \right\rfloor \qquad bottom = \left\lceil \frac{h}{2}(y_{\max} + 1) \right\rceil, \tag{8.32}$$

where the floor and ceiling operations ensure the proper rounding.

Listing 8.4. For a light source with a camera-space bounding sphere defined by the `center` and `radius` parameters, this function calculates the extents of the optimal scissor rectangle enclosing the sphere's projection into the viewport, where the shape of the view frustum is defined by the `projectionDistance` and `aspectRatio` parameters. The minimum and maximum extents of the rectangle are returned in the `rectMin` and `rectMax` parameters using normalized coordinates for which the full width and height of the viewport both correspond to the range $[-1, +1]$. The return value of the function indicates whether the rectangle is nonempty.

```cpp
bool CalculateScissorRect(const Point3D& center, float radius, float projectionDistance,
        float aspectRatio, Point2D *rectMin, Point2D *rectMax)
{
    // Initialize the rectangle to the full viewport.
    float xmin = -1.0F, xmax = 1.0F;
    float ymin = -1.0F, ymax = 1.0F;

    // Make sure the sphere isn't behind the camera.
    if (center.z > -radius)
    {
        float lx2 = center.x * center.x;
        float ly2 = center.y * center.y;
        float lz2 = center.z * center.z;
        float r2 = radius * radius;

        // The factors mx and my project points into the viewport.
        float my = projectionDistance / center.z, mx = my / aspectRatio;

        float a = lz2 - r2;
        if (fabs(a) > FLT_MIN)     // Quadratic case when a != 0.
        {
            float f = lz2 * r2;
            float a_inv = 1.0F / a;
            float d = f * (a + lx2);
            if (d > 0.0F)          // Discriminant positive for x extents.
            {
                d = sqrt(d);
                float b = r2 * center.x;
                float x1 = (center.x + (b - d) * a_inv) * mx;
                float x2 = (center.x + (b + d) * a_inv) * mx;

                if (a > 0.0F)      // Both tangencies valid.
                {
                    xmin = fmax(xmin, x1); xmax = fmin(xmax, x2);
                }
                else               // One tangency valid.
                {
                    if (center.x > 0.0F) xmin = fmax(xmin, fmax(x1, x2));
                    else xmax = fmin(xmax, fmin(x1, x2));
                }
            }
        }
```

```
        d = f * (a + ly2);
        if (d > 0.0F)              // Discriminant positive for y extents.
        {
            d = sqrt(d);
            float b = r2 * center.y;
            float y1 = (center.y + (b - d) * a_inv) * my;
            float y2 = (center.y + (b + d) * a_inv) * my;

            if (a > 0.0F)        // Both tangencies valid.
            {
                ymin = fmax(ymin, y1); ymax = fmin(ymax, y2);
            }
            else                      // One tangency valid.
            {
                if (center.y > 0.0F) ymin = fmax(ymin, fmax(y1, y2));
                else ymax = fmin(ymax, fmin(y1, y2));
            }
        }
    }
    else      // Linear case when a == 0.
    {
        if (fabs(center.x) > FLT_MIN)    // Tangency valid for x.
        {
            float x = (center.x - (lx2 + r2) * 0.5F / center.x) * mx;
            if (center.x > 0.0F) xmin = fmax(xmin, x);
            else xmax = fmin(xmax, x);
        }

        if (fabs(center.y) > FLT_MIN)    // Tangency valid for y.
        {
            float y = (center.y - (ly2 + r2) * 0.5F / center.y) * my;
            if (center.y > 0.0F) ymin = fmax(ymin, y);
            else ymax = fmin(ymax, y);
        }
    }
}

rectMin->Set(xmin, ymin); rectMax->Set(xmax, ymax);
return ((xmin < xmax) && (ymin < ymax));
}
```

8.2.2 Depth Bounds

To apply the depth bounds test to a point light, we need to calculate the minimum and maximum values of z_{viewport} for the light's bounding sphere in viewport space. The transformation from device space to viewport space usually does not alter depth values, so we assume for this discussion that z coordinates in the two spaces are equal to each other and calculate the minimum and maximum values of z_{device}.

In the case that the depth range is something other than $[0,1]$, device-space depths would need to be scaled and offset before being specified as the range for the depth bounds test.

Suppose that a point light of radius r is located at the position l in camera space. When using a projection matrix $\mathbf{P}$ that does not have an oblique near plane, which implies $P_{20} = P_{21} = 0$, calculating the depth bounds is rather simple. For any camera-space depth z_{camera}, the projection matrix and perspective divide produce the device-space depth given by

$$z_{\text{device}} = P_{22} + \frac{P_{23}}{z_{\text{camera}}}. \tag{8.33}$$

All we have to do is set z_{camera} to the minimum and maximum extents of the light's bounding sphere in camera space, given by $l_z - r$ and $l_z + r$, and the depth bounds are easily calculated. Functions that perform this calculation for conventional and reversing projection matrices are shown in Listing 8.5. This code handles cases in which the bounding sphere extends outside the view frustum in the z direction by first clamping z_{camera} to a minimum value equal to the near plane distance. The final value of z_{device} is then clamped to one for a conventional projection matrix and to zero for a reversing projection matrix in order to account for a far plane that could be at a finite or infinite distance.

In the case that the projection matrix $\mathbf{P}$ has an oblique near plane, calculating the depth bounds for a point light is vastly more complicated. The third row of $\mathbf{P}$ is now allowed to be any plane $\mathbf{k}$, and the general transformation of a point $\boldsymbol{p}_{\text{camera}}$ in camera space to the coordinates $(z_{\text{clip}}, w_{\text{clip}})$ in clip space is given by

$$\begin{bmatrix} k_x & k_y & k_z & k_w \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_{\text{camera}} \\ y_{\text{camera}} \\ z_{\text{camera}} \\ 1 \end{bmatrix} = \begin{bmatrix} k_x x_{\text{camera}} + k_y y_{\text{camera}} + k_z z_{\text{camera}} + k_w \\ z_{\text{camera}} \end{bmatrix}. \tag{8.34}$$

After performing the perspective divide, the device-space depth z_{device} is given by

$$z_{\text{device}} = \frac{k_x x_{\text{camera}} + k_y y_{\text{camera}} + k_z z_{\text{camera}} + k_w}{z_{\text{camera}}}, \tag{8.35}$$

and it now depends not only on the z coordinate in camera space, but on the x and y coordinates as well. By moving everything to one side of the equation, we can rewrite this as

$$k_x x_{\text{camera}} + k_y y_{\text{camera}} + (k_z - z_{\text{device}}) z_{\text{camera}} + k_w = 0, \tag{8.36}$$

Listing 8.5. For a light source with a camera-space bounding sphere defined by the center and radius parameters, these two functions calculate the minimum and maximum depths in device space for the conventional projection matrix P and the reversing projection matrix R. The depth range is returned in the minDepth and maxDepth parameters. The near plane distance n must have the same value that was used to construct the projection matrix. The return value of the function indicates whether the depth range is nonempty.

```
bool CalculateDepthBounds(const Matrix4D& P, float n, const Point3D& center,
                          float radius, float *minDepth, float *maxDepth)
{
    float zmin = fmin(P(2,2) + P(2,3) / fmax(center.z - radius, n), 1.0F);
    float zmax = fmin(P(2,2) + P(2,3) / fmax(center.z + radius, n), 1.0F);
    *minDepth = zmin;
    *maxDepth = zmax;
    return (zmin < zmax);
}

bool CalculateRevDepthBounds(const Matrix4D& R, float n, const Point3D& center,
                             float radius, float *minDepth, float *maxDepth)
{
    float zmin = fmax(R(2,2) + R(2,3) / fmax(center.z + radius, n), 0.0F);
    float zmax = fmax(R(2,2) + R(2,3) / fmax(center.z - radius, n), 0.0F);
    *minDepth = zmin;
    *maxDepth = zmax;
    return (zmin < zmax);
}
```

which is the dot product between the plane

$$\mathbf{m} = (k_x, k_y, k_z - z_{\text{device}}, k_w) \qquad (8.37)$$

and the point $\boldsymbol{p}_{\text{camera}}$. The plane $\mathbf{m}$ represents the set of points in camera space for which z_{device} has a constant value. It is our goal to find the values of z_{device} such that the corresponding planes $\mathbf{m}$ are tangent to the light's bounding sphere because these supply the range $[z_{\text{min}}, z_{\text{max}}]$ of device-space depths covered by the light.

The geometry of the problem is shown in Figure 8.8, where we again have a point light of radius r is located at the position $\boldsymbol{l}$ in camera space. For a conventional projection matrix, the plane $\mathbf{k}$ is identified as the oblique near plane of the view frustum, and all points in this plane produce a depth of $z_{\text{device}} = 0$. The plane $\mathbf{f}$ is the far plane of the view frustum where all points correspond to a depth of $z_{\text{device}} = 1$. For a reversing projection matrix, the roles of $\mathbf{k}$ and $\mathbf{f}$ are interchanged, but we do not need to make a distinction between the two types of matrices because the derivation of the depth bounds is the same for both.

Figure 8.8. In a view frustum with an oblique near clipping plane **k** and far plane **f**, the depth bounds $[z_{min}, z_{max}]$ for a point light of radius r centered at the position l in camera space are determined by finding planes of constant depth that are tangent to the light's bounding sphere. The heavy gray line is the culling boundary inside which l is assumed to be located. The red circle represents a cylinder of radius r centered on the line where **k** intersects the x-y plane. When l is inside this cylinder, there are no solutions because the discriminant d of the polynomial in Equation (8.40) is negative. When l is inside the green band, the polynomial's quadratic coefficient a is negative because light's bounding sphere intersects the x-y plane. When l is inside the blue band, the polynomial's constant term c is negative because the light's bounding sphere intersects the near plane. The plane **h**, used for classifying solutions, is perpendicular to **k** and intersects the x-y plane at the same line where **k** does.

We assume that $l_z > -r$ so that some part of the light's bounding sphere is in front of the x-y plane in camera space. We also assume that $\mathbf{k} \cdot l > -r \|\mathbf{k}_{xyz}\|$ and $\mathbf{f} \cdot l > -r \|\mathbf{f}_{xyz}\|$ (using the notation $\mathbf{k}_{xyz}$ to mean the 3D vector (k_x, k_y, k_z) without the w component) to exclude any case in which the bounding sphere is completely on the negative side of either the near plane or far plane. These conditions create a region bounded by three planes, drawn as a heavy gray line in Figure 8.8, inside which the light's position must be located for it to have any effect on the visible scene. Any light having a position outside this region can be safely culled.

For the plane **m** in Equation (8.37) to be tangent to the light's bounding sphere, the squared perpendicular distance between it and the point l must be equal to r^2, as expressed by the equation

$$(\mathbf{m} \cdot l)^2 = r^2 m_{xyz}^2. \tag{8.38}$$

Substituting the components of **m** gives us

$$(\mathbf{k} \cdot l - l_z z_{\text{device}})^2 = r^2 \left(k_{xyz}^2 - 2k_z z_{\text{device}} + z_{\text{device}}^2 \right), \tag{8.39}$$

and collecting on powers of z_{device} produces the quadratic equation

$$\left(l_z^2 - r^2 \right) z_{\text{device}}^2 - 2 \left[(\mathbf{k} \cdot l) l_z - r^2 k_z \right] z_{\text{device}} + (\mathbf{k} \cdot l)^2 - r^2 k_{xyz}^2 = 0. \tag{8.40}$$

The solutions to this equation are the minimum and maximum values of z_{device} that we need. After making the usual assignments

$$a = l_z^2 - r^2$$
$$b = (\mathbf{k} \cdot l) l_z - r^2 k_z$$
$$c = (\mathbf{k} \cdot l)^2 - r^2 k_{xyz}^2, \tag{8.41}$$

the solutions are then given by

$$z_{\text{device}} = \frac{b \pm \sqrt{d}}{a}, \tag{8.42}$$

where $d = b^2 - ac$ is the discriminant. The products b^2 and ac tend to be large and close to each other, so calculating their difference is numerically unstable due to the loss of floating-point precision. Fortunately, we can avoid this problem by using the definitions of a, b, and c to expand the discriminant into a different form. After some algebraic simplification, it can be written as

$$d = \left[(k_x l_x + k_y l_y + k_w)^2 + a \left(k_x^2 + k_y^2 \right) \right] r^2, \tag{8.43}$$

which does not suffer from stability issues.

When $a > 0$, the light's position is in front of the green band where $|z_{\text{camera}}| < r$ in Figure 8.8, and its bounding sphere is thus entirely on the front side of the x-y plane. In this region, the discriminant given by Equation (8.43) is always positive, and it is the easiest case to handle. We simply calculate both solutions with Equation (8.42) and clamp them to the range $[0,1]$. Since a is positive, choosing the

minus sign always produces the lesser value of z_{device}, and we can be sure that the minimum and maximum values are given by

$$z_{\min} = \min\left(\max\left(\frac{b - \sqrt{d}}{a}, 0\right), 1\right)$$

$$z_{\max} = \min\left(\max\left(\frac{b + \sqrt{d}}{a}, 0\right), 1\right). \tag{8.44}$$

If the light position is outside the culling boundary, then these calculations result in an empty depth range for which the minimum and maximum values are either both equal to zero or both equal to one.

When a is not a positive value, there are a number of special cases that require careful attention to detail. These cases all occur when the light position is within the distance r to the x-y plane, and the most intricate of them occur when the light position is also within a distance r to the near plane $\mathbf{k}$. In the remainder of this section, we describe these special cases and how to deal with them.

First, if the value of a is very close to zero, then the light's bounding sphere is practically tangent to the x-y plane, and its position lies on the boundary of the green band in Figure 8.8. We exclude the possibility that the position is on the lower boundary where $l_z \approx -r$ because the light would not be visible, and we thus assume that the position lies on the upper boundary where $l_z \approx r$. In this case, one of the solutions diverges to infinity, and the other solution is given by

$$z_{\text{device}} = \frac{c}{2b}. \tag{8.45}$$

If b also happens to be very close to zero, which happens when the light position is the point A in the figure, then there is no solution because all depths are covered by the light. In this case, we set the depth bounds to $[0, 1]$. Otherwise, we must separately handle the cases when b is positive or negative. If $b < 0$, which happens when the light position is left of the point A, then the depth range extends to $-\infty$ in the negative direction, and we set

$$z_{\max} = \min\left(\frac{c}{2b}, 1\right) \tag{8.46}$$

as its limit in the positive direction. If $b > 0$, which happens when the light position is right of the point A, then the depth range extends to $+\infty$ in the positive direction, and we set

$$z_{\min} = \max\left(\frac{c}{2b}, 0\right) \tag{8.47}$$

as its limit in the negative direction. Again, if the light would have been culled, then these calculations produce an empty depth range.

When $a < 0$, it's possible for the discriminant given by Equation (8.43) to be negative. This happens when the light position lies inside a cylinder of radius r centered on the line where the near plane $\mathbf{k}$ intersects the x-y plane, represented by the red circle shown in Figure 8.8. Here, the bounding sphere intersects the line where all planes of constant device-space depth must intersect the x-y plane, making it impossible for any of those planes to be tangent to the sphere. In this case, all depths are covered by the light, and we set the depth bounds to $[0,1]$.

For the remaining cases, we can assume that $a < 0$ and $d > 0$, which means that the light position is inside the green band but outside the red circle. We can classify these cases into those for which $c \leq 0$ and those for which $c > 0$, and these correspond to light positions inside and outside the blue band in the figure, respectively. We examine the case with $c > 0$ first because it is simpler, and we start by making the observation that $ac < 0$ and thus $\sqrt{d} > |b|$. This means that the two solutions given by Equation (8.42) must have different signs. We know that the bounding sphere is completely on the positive side of the plane $\mathbf{k}$, so the negative solution must correspond to a point of tangency behind the x-y plane on the right side of the figure, which can be ignored. The positive solution corresponds to a point of tangency in front of the x-y plane and represents the minimum depth in a range that extends to $+\infty$. Therefore, we set $z_{\max} = 1$ and only calculate

$$z_{\min} = \max\left(\frac{b - \sqrt{d}}{a}, 0\right), \tag{8.48}$$

where we clamp to zero to account for possible floating-point round-off error.

We now examine the more complex case with $c \leq 0$, where the light position is inside both the green and blue bands but outside the red circle under the continued assumptions that $a < 0$ and $d > 0$. This time, $ac \geq 0$ and thus $\sqrt{d} \geq |b|$, so two nonzero solutions to Equation (8.42) must have the same signs. If $b \leq 0$, then both solutions are nonnegative, and if $b \geq 0$, then both solutions are nonpositive. For both of these possibilities, different actions need to be taken depending on whether the light position is on the positive or negative side of the constant-depth plane $\mathbf{h}$ that is perpendicular to the near plane $\mathbf{k}$, as shown in Figure 8.8. The plane $\mathbf{h}$ must have the same form as the plane $\mathbf{m}$ in Equation (8.37), and we calculate the value of the constant depth z_{device} by requiring that the normal vectors are perpendicular with the equation $\mathbf{h}_{xyz} \cdot \mathbf{k}_{xyz} = 0$. This yields a depth of

$$z_{\text{device}} = \frac{k_{xyz}^2}{k_z}. \tag{8.49}$$

To avoid division by zero, we multiply each component of $\mathbf{h}$ by k_z to obtain

$$\mathbf{h} = \left(k_x k_z, k_y k_z, -k_x^2 - k_y^2, k_w k_z \right), \tag{8.50}$$

and this has the additional benefit that it gives $\mathbf{h}$ a consistent orientation such that $h_z \leq 0$ in all cases. Light positions are classified as being on the positive or negative side of $\mathbf{h}$ by calculating the dot product

$$\mathbf{h} \cdot \mathit{l} = k_x l_x k_z + k_y l_y k_z - \left(k_x^2 + k_y^2 \right) l_z + k_w k_z$$
$$= \left(\mathbf{k} \cdot \mathit{l} \right) k_z - k_{xyz}^2 l_z. \tag{8.51}$$

In the case that $b \geq 0$ (with $a < 0$ and $c \leq 0$), there are no positive solutions to Equation (8.40), so we cannot reduce the range of depths. This happens when the light position falls inside the tiny striped region between the points A and B in the figure or inside the same-shaped region between the points C and D. (These regions can be a little larger for different near plane orientations.) We make a distinction between the two by looking at the value of $\mathbf{h} \cdot \mathit{l}$. If $\mathbf{h} \cdot \mathit{l} < 0$, then the light's bounding sphere covers all possible depths, so we set the depth range to $[0,1]$. Otherwise, if $\mathbf{h} \cdot \mathit{l} > 0$, then the light position is inside the region between C and D, and we can cull the light because it cannot extend into the view frustum. If b is exactly zero, then the light position is exactly one of the points B or D because we must have $c = 0$ to avoid a negative discriminant, and thus $\mathbf{h} \cdot \mathit{l} = 0$. In this case, we cull the light when $\mathbf{k} \cdot \mathit{l} < 0$. Note that we cannot use the plane $\mathbf{k}$ to classify light positions in all cases because the signs of $\mathbf{k} \cdot \mathit{l}$ and $\mathbf{h} \cdot \mathit{l}$ are not correlated for all possible orientations of the near plane.

In the final case that $b < 0$ (with $a < 0$ and $c \leq 0$), Equation (8.40) has at least one positive solution, and the other solution cannot be negative. This happens when the light position falls inside the striped parallelogram shown in the figure where the green and blue bands intersect, but excluding the regions between the points A and B and between the points C and D. The solutions have different meanings depending on the value of $\mathbf{h} \cdot \mathit{l}$, which cannot be zero in this case. If $\mathbf{h} \cdot \mathit{l} < 0$, then the depth range extends from zero to the smaller of the two solutions, so we calculate

$$z_{\max} = \frac{b + \sqrt{d}}{a}. \tag{8.52}$$

Otherwise, if $\mathbf{h} \cdot \mathit{l} > 0$, then the depth range extends from the larger of the two solutions to one, so we calculate

$$z_{\min} = \frac{b - \sqrt{d}}{a}. \tag{8.53}$$

It is possible for z_{min} to be larger than one, and this happens when the light position is inside the small region between the points D and E. In this case, the light is not visible and should be culled.

The code in Listing 8.6 implements the depth bounds calculation for a projection matrix having an oblique near plane and handles all of the special cases. If the resulting range of depths $[z_{min}, z_{max}]$ is empty, meaning that $z_{min} \geq z_{max}$, then the function returns `false`, and this indicates that the light should be culled. This code generates the correct results in all situations, including those for which the near plane is not oblique (where $k_x = k_y = 0$), so there is no need to keep track of which kind of projection matrix is active and select one of the functions in Listing 8.5 in the non-oblique cases. (See Exercise 3.)

Listing 8.6. For a light source with a camera-space bounding sphere defined by the `center` and `radius` parameters, this function calculates the minimum and maximum depths in device space for the projection matrix P, which may have an oblique near plane. The depth range is returned in the `minDepth` and `maxDepth` parameters. The return value of the function indicates whether the depth range is nonempty.

```
void CalculateObliqueDepthBounds(const Matrix4D& P, const Point3D& center,
                                 float radius, float *minDepth, float *maxDepth)
{
    float zmin = 0.0F;
    float zmax = 1.0F;

    float kz = P(2,2);
    float kxy2 = P(2,0) * P(2,0) + P(2,1) * P(2,1);
    float kxyz2 = kxy2 + kz * kz;
    float klxyw = P(2,0) * center.x + P(2,1) * center.y + P(2,3);
    float kl = klxyw + kz * center.z;    // k dot l
    float r2 = radius * radius;

    float a = center.z * center.z - r2;
    float b = kl * center.z - r2 * kz;
    float c = kl * kl - r2 * kxyz2;
    if (fabs(a) > FLT_MIN)
    {
        float d = (klxyw * klxyw + a * kxy2) * r2;
        if (d > FLT_MIN)
        {
            // Bounding sphere does not intersect red cylinder in figure.
            float a_inv = 1.0F / a;
            float f = sqrt(d) * a_inv;
            if (a > 0.0F)
            {
                // Bounding sphere is completely in front of x-y plane.
```

```
                // Both roots correspond to valid depth bounds.
                zmin = fmin(fmax(b * a_inv - f, 0.0F), 1.0F);
                zmax = fmin(fmax(b * a_inv + f, 0.0F), 1.0F);
            }
            else if (c > 0.0F)
            {
                // Bounding sphere is completely in front of near plane.
                // Roots must have different signs. Positive root is zmin.
                zmin = fmax(b * a_inv - f, 0.0F);
            }
            else
            {
                float hl = kl * kz - kxyz2 * center.z;    // h dot l
                if (b < -FLT_MIN)
                {
                    // Light position is inside striped parallelogram.
                    if (hl < 0.0F) zmax = b * a_inv + f;
                    else zmin = b * a_inv - f;
                }
                else if (b > FLT_MIN)
                {
                    // If light position is between points C and D, it's not visible.
                    if (hl > 0.0F) zmax = 0.0F;
                }
                else
                {
                    // If light position is at point C, it's not visible.
                    if (kl < 0.0F) zmax = 0.0F;
                }
            }
        }
    }
    else if (center.z > 0.0F)  // The light position is on upper boundary of green band.
    {
        if (fabs(b) > FLT_MIN)
        {
            float zdev = c / b * 0.5F;
            if (b < 0.0F) zmax = fmin(zdev, 1.0F);
            else zmin = fmax(zdev, 0.0F);
        }
    }
    else zmax = 0.0F;   // Light position is on lower boundary of green band.

    *minDepth = zmin;
    *maxDepth = zmax;
    return (zmin < zmax);
}
```

8.3 Shadow Maps

Once we have determined that a surface intersects the volume of space influenced by a light source, we incorporate that light source's contribution into the shading calculations for the surface. In addition to any kind of attenuation that we may need to apply, we must account for the possibility that the incoming light is blocked by another surface, thus casting a shadow, before it can reach the various points on the surface we are rendering. The ability to render shadows is an important component of any game engine because it is necessary for presenting realistic visual cues about the positions of objects to the player.

The most widely used general technique for rendering shadows is known as *shadow mapping*, and it is based on the simple principle that each ray emitted by a light source can strike at most one opaque surface. Whichever surface is closest to the light along a particular ray receives all of the illumination carried by the ray, and any surfaces farther away in the same direction must therefore be in shadow. This concept is comparable to the similar principle upon which the depth buffer operates. Any ray originating from the camera position can strike only one opaque surface, and the depth buffer records the distance, perpendicular to the camera plane, to the nearest surface at each pixel as rendering progresses. The same idea can be applied from the perspective of the light source, and rendering a distinct depth buffer for each light is fundamentally how shadow mapping works. The details are somewhat different for spot lights, point lights, and infinite lights, so we discuss each type separately in this section.

8.3.1 2D Shadow Maps

Of the various types of light sources, a spot light present the simplest situation when it comes to rendering shadow maps. This is due to the similarity between the volume of space illuminated by a spot light and the view frustum visible to an ordinary perspective camera. In both cases, all the information we need about the nearest surfaces is stored in a single two-dimensional depth buffer. To render a shadow map, we essentially add a second view frustum to the world, as shown in Figure 8.9. The shadow map is simply the contents of the depth buffer that we end up with when we render the world using this view frustum. When generating a shadow map, we render *only* the depth buffer, and not a color buffer or any of the special buffers described in Chapter 10. Rendering a shadow map is quite fast because the pixel shader doesn't normally need to do any work, with the exception of cases in which pixels are discarded based on the result of some calculation. Additionally, GPUs are designed to perform better when only the depth is being written to memory after the rasterization stage.
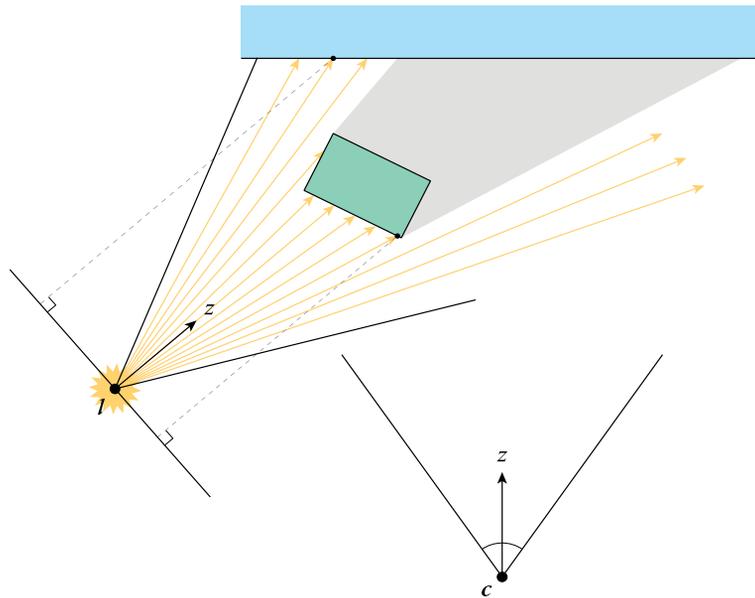
Figure 8.9. A 2D shadow map is generated by rendering a depth buffer from the perspective of the light source. For each ray starting at the light's position l, the shadow map contains the projected depth, parallel to the light's z axis, corresponding to the nearest opaque surface. When the world is rendered from the camera position c, points on visible surfaces are transformed into light space and projected using the same MVP matrix that was used when the shadow map was generated. The projected z coordinate of each point is compared to the value stored in the shadow map to determine whether it is inside a shadow.

The view frustum for the shadow map is aligned with the spot light's local coordinate system, pointing in the direction of the light's positive z axis with its apex at the light's position. The model-view-projection matrix is given by

$$\mathbf{M}_{\text{MVP}} = \mathbf{P}\mathbf{M}_{\text{light}}^{-1}\mathbf{M}_{\text{object}}, \tag{8.54}$$

where the only difference from Equation (5.29), where the MVP matrix was first introduced, is that we are now transforming into light space instead of camera space before the projection is applied. The matrix $\mathbf{P}$ is either a conventional or reversing projection matrix having one of the common forms given by Equation (6.36) or Equation (6.45), respectively, with the distance f to the far plane set to the light's bounding sphere radius r_{max}. We must still have a near plane distance n, and it can be set to some small value representing the minimum distance from the light at which a shadow can be cast. The projection distance g has the value given by

Equation (8.19), corresponding to the angular size of the spot light. Shadow maps are usually square, in which case the aspect ratio s is simply equal to one.

Once the depth information has been rendered into the shadow map, it is used as a special kind of texture map in any pixel shaders that later calculate the contribution from the light source. The shadow map has a single channel containing the depth along the z direction in light space, as illustrated in Figure 8.9. When we render a surface that could be illuminated by the spot light, we transform vertex positions using two different MVP matrices. The first matrix is the ordinary MVP matrix for the camera, and it produces the output vertex positions in clip space. The second matrix is the MVP matrix for the spot light given by Equation (8.54) but with the first two rows of the projection matrix matching the matrix $\mathbf{P}_{\text{spot}}$ given by Equation (8.17). As before, this modification causes the x and y coordinates to fall in the range $[0,1]$ inside the spot light's frustum after the perspective divide so they can be used directly as shadow map texture coordinates. With this change, the conventional projection matrix $\mathbf{P}_{\text{shadow}}$ is given by

$$
\mathbf{P}_{\text{shadow}} = \begin{bmatrix} g/2s & 0 & 1/2 & 0 \\ 0 & g/2 & 1/2 & 0 \\ 0 & 0 & \dfrac{r_{\max}}{r_{\max}-n} & -\dfrac{nr_{\max}}{r_{\max}-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{8.55}
$$

The equivalent reversing projection matrix swaps the values of n and $r_{\max}$.

The 4D homogeneous coordinates resulting from the application of the light's MVP matrix to the object-space vertex position is output by the vertex shader and interpolated over each triangle. In the pixel shader, the perspective divide is applied by multiplying the x, y, and z coordinates by the reciprocal of the w coordinate. At this time, the (x, y) coordinates are the texture coordinates at which we sample the shadow map, and the z coordinate is the projected depth of the point q on the surface we are currently rendering. If, along the direction connecting it to the light's position, the point q happens to be on the surface nearest to the light source, then the depth fetched from the shadow map will be equal to the z coordinate calculated in the pixel shader (ignoring precision issues for the moment). Otherwise, if the point q is not on the nearest surface because it is blocked by another surface, then the depth fetched from the shadow map will be less than the z coordinate for a conventional projection matrix or greater than the z coordinate for a reversing projection matrix. Thus, comparing the two values tells us whether the point q on the surface being rendered is in shadow.

If we were to make only one comparison per pixel against a depth stored in the shadow map, then the resulting shadow would have a sharp boundary with large jagged steps along its edges. We could mitigate this problem by increasing the resolution of the shadow map, but each pixel would still be either completely in shadow or fully illuminated with no smooth transition between those two states. A better solution is to take multiple samples from the shadow map, compare each of them to the light-space projected depth of the point q being rendered, and adjust the brightness of the incoming light based on what portion of the comparisons pass the test. This is called *percentage closer filtering (PCF)* because the amount of light that makes it through is given by the number of samples for which the point q is closer to the light source divided by the total number of samples taken.

To help with shadow mapping, GPUs have texture sampling features that are specifically designed to perform depth comparisons and apply percentage closer filtering. When sampling a 2D shadow map, three texture coordinates are required instead of the usual two. The third coordinate is the depth of the point q being rendered after it has been transformed into light space and projected. The GPU compares the depth in the third coordinate to the depth fetched from the shadow map and generates a value of one if the comparison passes or a value of zero if it fails. More importantly, the comparison happens separately for each sample that participates in bilinear filtering, and the result returned by a single texture fetch is the weighted average of the four outcomes of those comparisons. Without this functionality, conventional filtering would return the weighted average four adjacent depth values stored in the shadow map, and this would not produce correct results if the pixel shader then made a comparison using that average.

Even with the availability of a special bilinear filtering mode, it is usually necessary to fetch multiple samples from the shadow map within a small neighborhood of the projected texture coordinates in order to produce acceptably smooth edges. This is done in Listing 8.7, where samples are fetched from the shadow map at the texture coordinates initially calculated and four additional nearby locations. Offsets between the center sample and the surrounding samples are typically measured in texel-sized units given by the reciprocal of the shadow map's resolution. The averaged result returned by the shader code is always a value in the range $[0,1]$ representing the amount of light that is not blocked by a shadow castor. It should be multiplied by the color C_{illum} calculated with Equation (8.13) to determine the final illumination reaching the surface being rendered. An example scene containing a spot light is shown in Figure 8.10(a). The shadow map itself is shown as a grayscale image in Figure 8.10(b), where lighter shades of gray correspond to greater depths.

Listing 8.7. This pixel shader function samples the 2D shadow map `shadowTexture` that has been rendered for a spot light. The `shadowCoord` parameter contains the interpolated homogenous coordinates produced by the projection matrix given by Equation (8.55). The uniform constant `shadow-Offset` specifies the space between shadow map samples, and it would typically be set to the reciprocal of the shadow map resolution in texels.

```
uniform Texture2DShadow    shadowTexture;
uniform float              shadowOffset;

float CalculateSpotShadow(float4 shadowCoord)
{
    // Project the texture coordinates and fetch the center sample.
    float3 p = shadowCoord.xyz / shadowCoord.w;
    float light = texture(shadowTexture, p);

    // Fetch four more samples at diagonal offsets.
    p.xy -= shadowOffset;
    light += texture(shadowTexture, p.xy, p.z);
    p.x += shadowOffset * 2.0;
    light += texture(shadowTexture, p.xy, p.z);
    p.y += shadowOffset * 2.0;
    light += texture(shadowTexture, p.xy, p.z);
    p.x -= shadowOffset * 2.0;
    light += texture(shadowTexture, p.xy, p.z);

    return (light * 0.2);  // Return average value.
}
```

8.3.2 Cube Shadow Maps

A single 2D shadow map is sufficient for casting shadows from a spot light, but a point light requires something more. To handle omnidirectional shadows, we need to surround a point light with multiple shadow maps storing the depths of shadow-casting geometries, and this is most effectively achieved by using six shadow maps arranged as a cube. This is convenient because the shadow maps can be organized as a single cube texture map that is sampled with direct hardware support when the illumination from the point light is rendered.

Each face of a cube shadow map can be treated as a square 2D shadow map covering a 90-degree field of view from the perspective of the light source. Thus, when each shadow map is rendered, the projection distance g and aspect ratio s used in the projection matrix are both equal to one. As with spot lights, the distance f to the far plane is set to the radius of the point light r_{max}, and the distance n to the near plane is chosen as some minimum shadow-casting distance. To determine the
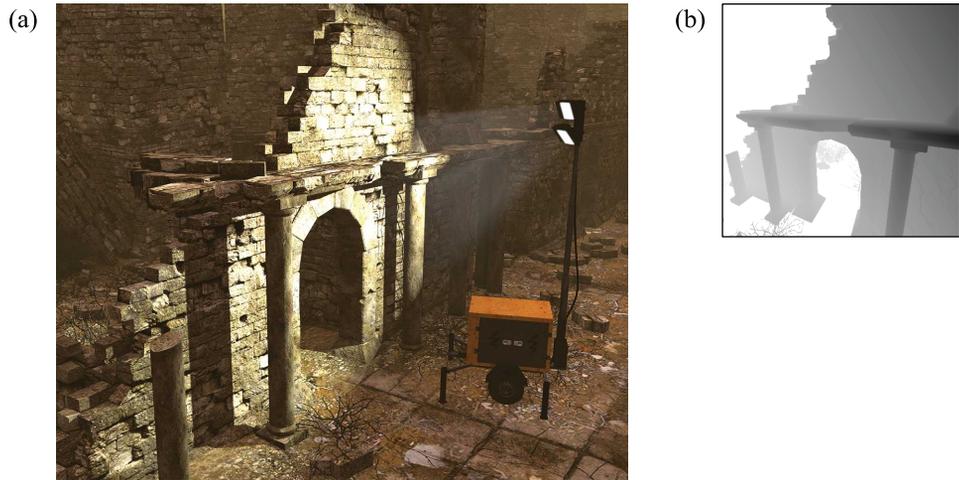
(a)

(b)

Figure 8.10. (a) Shadows are rendered for a spot light using a 2048×2048 shadow map. Five samples are fetched per pixel by the code shown in Listing 8.7, where `shadowOffset` has been set to 1/2048. (b) The depths stored in the shadow map are shown as a grayscale image rendered from the perspective of the light source. Texels become lighter as distance from the camera increases.

orientation of the view frustum, we must account for the directions in which the coordinate axes point for each face of a cube map, as described in Section 7.4.3. The MVP matrix is given by

$$\mathbf{M}_{\mathrm{MVP}} = \mathbf{P}\mathbf{M}_{\mathrm{face}}^{-1}\mathbf{M}_{\mathrm{light}}^{-1}\mathbf{M}_{\mathrm{object}}, \tag{8.56}$$

where $\mathbf{M}_{\mathrm{face}}$ is the matrix listed in Table 7.1 corresponding to the face for which the shadow map is being rendered. This MVP matrix transforms points into light space and then into a face-specific camera space before applying the projection matrix $\mathbf{P}$. Because texture coordinate systems for the cube faces are left handed, the matrix $\mathbf{M}_{\mathrm{face}}$ contains a reflection, and we must be careful to invert the rendering state that specifies the front face winding direction.

After shadow maps have been rendered for all six faces, giving us a complete cube shadow map, we render the contribution from the light source. The 3D texture coordinates at which the cube shadow map is sampled are given by positions in light space, so object-space vertex positions are transformed into light space by the vertex shader and interpolated over each triangle. Unlike the method used for spot lights, the vertex shader does not perform a projection because we don't know at

that point in time which face of the cube map will be sampled. The face is determined in the pixel shader by first selecting the largest component m of the interpolated light-space position q, defined by

$$m = \max\left(|q_x|, |q_y|, |q_z|\right). \tag{8.57}$$

The projected depth z, after the perspective divide, is then given by

$$z = P_{22} + \frac{P_{23}}{m}, \tag{8.58}$$

where P_{22} and P_{23} are the entries in the projection matrix $\mathbf{P}$ used when the shadow maps were rendered.

Once the cube face has been determined, we know the GPU will access it using 2D coordinates given by the corresponding formulas given in Table 7.1. In order to take samples at multiple locations within a small neighborhood, we need to apply offsets to the two coordinates of the position q that do not constitute the major axis. For example, if the major axis is $+y$, then offsets must be added to the x and z coordinates. Any changes we make to the two non-major coordinates are going to be divided by $2m$ when the shadow map is sampled, so if we want the effective offset to be δ, we need to add or subtract $2m\delta$ to each coordinate. We can devise a method for handling all six faces uniformly by making use of the fact that one of the non-major coordinates must be either x or y and the other non-major coordinate must be either y or z. This means we can calculate a pair of two-dimensional vectors o_{xy} and o_{yz} representing the offsets along the two non-major axes. For the three possible values of m, these offsets are given by the following formulas.

$$
\begin{aligned}
m = |q_x|: \quad & o_{xy} = (0, 2m\delta), \quad o_{yz} = (0, 2m\delta) \\
m = |q_y|: \quad & o_{xy} = (2m\delta, 0), \quad o_{yz} = (0, 2m\delta) \\
m = |q_z|: \quad & o_{xy} = (2m\delta, 0), \quad o_{yz} = (2m\delta, 0)
\end{aligned} \tag{8.59}
$$

If we define the values d_{xy} and d_x as

$$
d_{xy} = \begin{cases} 2m\delta, & \text{if } \max\left(|q_x|, |q_y|\right) > |q_z|; \\ 0, & \text{otherwise,} \end{cases}
$$

$$
d_x = \begin{cases} d_{xy}, & \text{if } |q_x| > |q_y|; \\ 0, & \text{otherwise,} \end{cases} \tag{8.60}
$$

then every case shown in Equation (8.59) is given by

$$
\begin{aligned}
o_{xy} &= (2m\delta - d_x, d_x) \\
o_{yz} &= (2m\delta - d_{xy}, d_{xy}).
\end{aligned} \tag{8.61}
$$

When these offsets are applied, we have to make sure that we don't inadvertently cause one of the texture coordinates along a non-major axis to become larger than the value of m because this would cause a different cube face to be sampled, and the depth computed with Equation (8.58) would no longer be valid. This can happen when samples are taken near the edges of the cube shadow map, and it causes lines of incorrectly shadowed pixels to appear. To account for this, we clamp the coordinates that are offset to the range $[-m + 2m\delta\varepsilon, m - 2m\delta\varepsilon]$, where ε is a small positive constant. We can't simply clamp to $[-m, m]$ because tie breaking rules can cause the accessed cube face to change when two or three texture coordinates are equal to each other.

Listing 8.8 demonstrates how a cube shadow map is sampled using the above method. The value 2δ is passed to the shader as a uniform constant, and it is multiplied by the value of m, calculated inside the shader, to obtain the correct offset vectors. The result returned by the shader code represents the amount of light not blocked by a shadow caster. It should be multiplied by the color C_{illum} calculated with Equation (8.10) to determine the final illumination reaching the surface being rendered. As with 2D shadow maps, this code takes five samples from a single face and averages them together. (Even though some GPUs support cube map filtering across face boundaries, we cannot make use of that functionality here because the projected depth we calculate is valid for only the face corresponding to the major axis.) An example scene containing a point light is shown in Figure 8.11(a), and its cube shadow map is shown as a grayscale image in Figure 8.11(b).

Listing 8.8. This pixel shader function samples the cube shadow map `shadowTexture` that has been rendered for a point light. The `lightCoord` parameter contains the interpolated vertex position in light space. The uniform constant `shadowOffset` specifies the space between shadow map samples, and it would typically be set to twice the reciprocal of the shadow map resolution in texels. The uniform constant `depthTransform` contains the $(2,2)$ and $(2,3)$ entries of the projection matrix used when the shadow map was rendered.

```
uniform TextureCubeShadow     shadowTexture;
uniform float                 shadowOffset;     // 2.0 * delta
uniform float2                depthTransform;   // (m22, m23)

float CalculatePointShadow(float3 lightCoord)
{
    float3 absq = abs(lightCoord);
    float mxy = max(absq.x, absq.y);
    float m = max(mxy, absq.z);         // Value of largest component.

    // Calculate offset vectors.
    float offset = shadowOffset * m;
```

```
    float dxy = (mxy > absq.z) ? offset : 0.0;
    float dx = (absq.x > absq.y) ? dxy : 0.0;
    float2 oxy = float2(offset - dx, dx);
    float2 oyz = float2(offset - dxy, dxy);

    float3 limit = float3(m, m, m);
    limit.xy -= oxy * (1.0 / 1024.0);    // Epsilon = 1/1024.
    limit.yz -= oyz * (1.0 / 1024.0);

    // Calculate projected depth and fetch the center sample.
    float depth = depthTransform.x + depthTransform.y / m;
    float light = texture(shadowTexture, float4(lightCoord, depth));

    // Fetch four more samples at diagonal offsets.
    lightCoord.xy -= oxy;
    lightCoord.yz -= oyz;
    light += texture(shadowTexture, clamp(lightCoord, -limit, limit), depth);
    lightCoord.xy += oxy * 2.0;
    light += texture(shadowTexture, clamp(lightCoord, -limit, limit), depth);
    lightCoord.yz += oyz * 2.0;
    light += texture(shadowTexture, clamp(lightCoord, -limit, limit), depth);
    lightCoord.xy -= oxy * 2.0;
    light += texture(shadowTexture, clamp(lightCoord, -limit, limit), depth);

    return (light * 0.2);  // Return average value.
}
```

8.3.3 Cascaded Shadow Maps

Above, we described how shadow maps for spot lights and point lights are each rendered using one or more perspective projections with a camera placed at the light's position. We now consider the best way to render shadow maps for infinite lights. An infinite light has no position, only a direction, and its parallel rays suggest that we should be using an orthographic projection instead of a perspective projection. These are the first indications that the shadow mapping process is going to be different for infinite lights. We must also consider that infinite lights usually illuminate a much larger area than spot lights and point lights do in a typical game world because they are often shining on vast stretches of land in outdoor scenes. It seems natural to cover the illuminated area with one giant 2D shadow map, but turns out not to be the best solution because an extremely high resolution, probably exceeding the capabilities of the GPU, would be required in order to produce shadows having acceptable levels of sharpness. Even of such a resolution were possible, most of it would be wasted on geometry far from the main camera where shadow details appear too small in the viewport to be discernible.
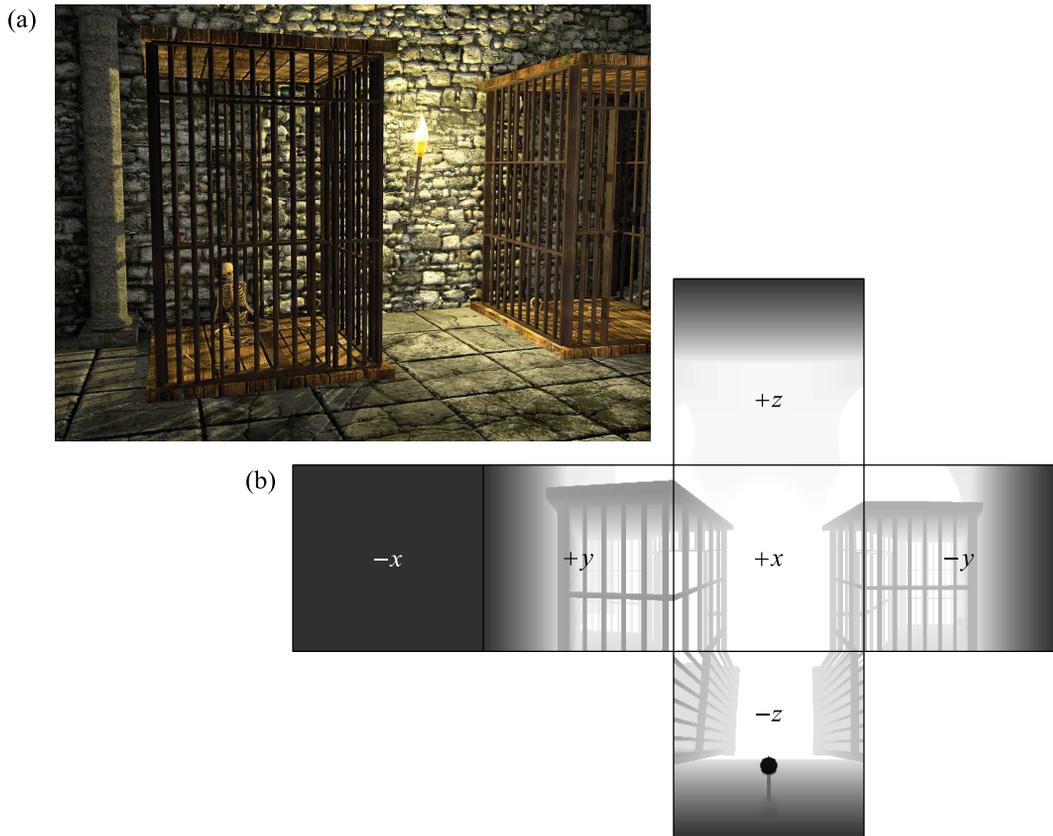
Figure 8.11. (a) Shadows are rendered for a point light using a cube shadow map having faces of size 1024×1024 texels. Five samples are fetched per pixel by the code shown in Listing 8.8, where `shadowOffset` has been set to $2/1024$. (b) The depths stored in the shadow map are shown as six grayscale images rendered from the perspective of the light source. The major axis associated with each face points into the page.

A popular and effective method for rendering shadow maps covering large areas illuminated by infinite lights is called *cascaded shadow maps (CSM)*. This method works by rendering several 2D shadow maps corresponding to volumes of increasing size distributed along the view direction of the main view frustum, as shown in Figure 8.12. Each shadow map is called a *cascade*, and the full set of cascades is organized into a single array texture map having multiple 2D layers. For the purposes of our discussion, we will use four cascades in total, but it is perfectly reasonable to have more or fewer to better fit a particular environment.
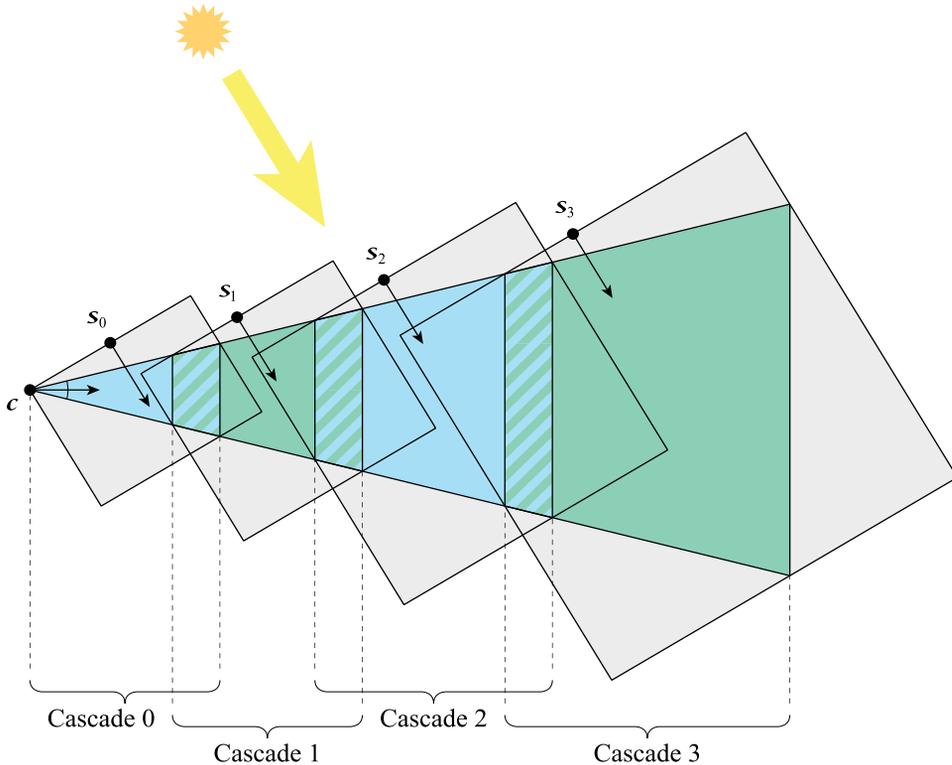
Figure 8.12. A cascaded shadow map is composed of several 2D shadow maps that each cover a range of distances inside the view frustum. The main camera is at the position c, and its view direction points to the right. Each cascade k is a box aligned with the coordinate axes of the light source, and the position s_k represents the location from which the cascade's shadow map is rendered. The actual difference in size between consecutive cascades is typically much larger than shown in the limited space here.

The cascade associated with the space closest to the camera is numbered with an index of zero, and the indices of the other cascades increase with distance. The same number of texels exist in each layer, but higher-numbered cascades cover larger volumes of space where the view frustum is wider. This means that the resolution of the more distant cascades is lower in the sense that each texel corresponds to a larger physical area, which is good because we don't want to waste time rendering excessively detailed shadows farther from the camera.

We define the cascade with index k by the range of distances $[a_k, b_k]$ it covers along the view direction. The first cascade, having index $k = 0$, begins at the camera

plane, so it's always the case that $a_0 = 0$. Otherwise, we can choose any increasing sequences of a_k and b_k that produce good results for the particular world being rendered. When we sample the shadow map as an array texture, we select the layer based on the cascade in which the point being shaded falls. So there isn't an abrupt visible change in shadow resolution on the boundary between two cascades, we allow consecutive cascades to overlap. The minimum distance a_k for one cascade is a little smaller than the maximum distance b_{k-1} for the preceding cascade. For points that fall inside this transition range, we will sample two layers of the shadow map and blend the results together. So that no more than two cascades overlap at any point in space, we require that $a_k > b_{k-2}$ for $k \geq 2$.

In order to fit a 2D shadow map to each cascade, we look at the portion of the view frustum covered by the cascade from the perspective of the light source. We can place the infinite light at any position we choose, but it will be convenient to keep it at the origin. The extents of the cascade in the light-space x and y directions delimit a minimal bounding box for the cascade, as illustrated in Figure 8.12. The extents of the cascade in the light-space z direction tell us what range of physical depths need to be covered by the shadow map. The bounding box is easily calculated by transforming the eight vertices defining each cascade's portion of the view frustum from camera space to light space using the matrix

$$\mathbf{L} = \mathbf{M}_{\text{light}}^{-1} \mathbf{M}_{\text{camera}}, \tag{8.62}$$

where $\mathbf{M}_{\text{camera}}$ and $\mathbf{M}_{\text{light}}$ are the matrices that transform from object space to world space for the camera and light. As previously discussed in Section 6.1, a plane perpendicular to the view direction cuts through the view frustum between the camera-space coordinates $x = \pm us/g$ and $y = \pm u/g$ at the distance u from the camera. Thus, as labeled in Figure 8.13, the four camera-space vertices $\mathbf{v}_{0,1,2,3}^k$ on the boundary plane closer to the camera for cascade k are given by

$$\mathbf{v}_{0,1,2,3}^k = \left(\pm \frac{a_k s}{g}, \pm \frac{a_k}{g}, a_k \right), \tag{8.63}$$

and the four camera-space vertices $\mathbf{v}_{4,5,6,7}^k$ on the boundary plane farther from the camera for cascade k are given by

$$\mathbf{v}_{4,5,6,7}^k = \left(\pm \frac{b_k s}{g}, \pm \frac{b_k}{g}, b_k \right). \tag{8.64}$$

(Here, k is used as a superscript index and not as an exponent.) Given these eight positions, the light-space bounding box for cascade k is determined by the minimum and maximum values of the x, y, and z coordinates of the transformed vertices
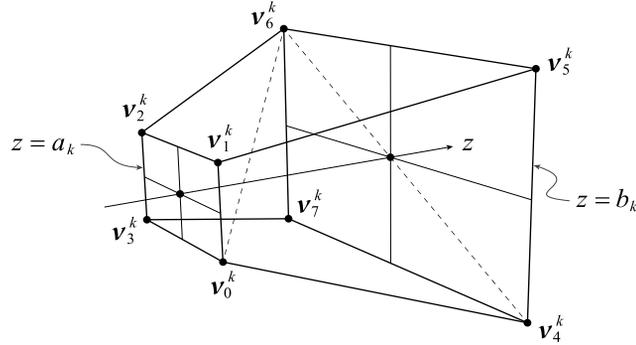
Figure 8.13. The shadow map cascade having index k is the portion of the view frustum bounded by the planes $z = a_k$ and $z = b_k$. The eight camera-space vertex positions v_i^k are given by Equations (8.63) and (8.64). The diameter of the cascade must be the length of the line segment connecting v_6^k to either v_0^k or v_4^k.

Lv_i^k, where i ranges from 0 to 7, which we write as follows.

$$
\begin{aligned}
x_{\min}^k &= \min_{0 \le i \le 7}\left(Lv_i^k\right)_x & x_{\max}^k &= \max_{0 \le i \le 7}\left(Lv_i^k\right)_x \\
y_{\min}^k &= \min_{0 \le i \le 7}\left(Lv_i^k\right)_y & y_{\max}^k &= \max_{0 \le i \le 7}\left(Lv_i^k\right)_y \\
z_{\min}^k &= \min_{0 \le i \le 7}\left(Lv_i^k\right)_z & z_{\max}^k &= \max_{0 \le i \le 7}\left(Lv_i^k\right)_z
\end{aligned}
\tag{8.65}
$$

A cascade's shadow map is rendered using an orthographic projection that encloses the cascade's bounding box. As shown in Figure 8.12, the camera position s_k from which we render the shadow map is the center of the bounding face for which $z = z_{\min}^k$. Its light-space coordinates are given by

$$
s_k = \left(\frac{x_{\max}^k + x_{\min}^k}{2}, \frac{y_{\max}^k + y_{\min}^k}{2}, z_{\min}^k \right),
\tag{8.66}
$$

and the world-space camera position is therefore equal to $M_{\text{light}} s_k$. To avoid clipping shadow-casting geometry where $z < z_{\min}^k$, we must configure the GPU to clamp depths to the near plane when a shadow map is rendered. The orientation of the camera is aligned to the light source, so we can define a matrix M_{cascade}^k that transforms from the cascade-specific camera space to world space as

$$
M_{\text{cascade}}^k = \begin{bmatrix} M_{\text{light}[0]} & M_{\text{light}[1]} & M_{\text{light}[2]} & M_{\text{light}} s_k \end{bmatrix}.
\tag{8.67}
$$

The first three columns of $\mathbf{M}^k_{\text{cascade}}$ are simply the world-space directions in which the coordinate axes of the light source point, and the fourth column is the world-space position of the camera.

The cascade-specific projection matrix, which we call $\mathbf{P}^k_{\text{cascade}}$, is based on Equation (6.49). Since the shadow map is square, the width and height of the volume spanned by the projection need to be equal, so we define

$$d_k = \max\left(x^k_{\text{max}} - x^k_{\text{min}}, y^k_{\text{max}} - y^k_{\text{min}} \right). \tag{8.68}$$

We can now write $\mathbf{P}^k_{\text{cascade}}$ as

$$\mathbf{P}^k_{\text{cascade}} = \begin{bmatrix} 2/d_k & 0 & 0 & 0 \\ 0 & 2/d_k & 0 & 0 \\ 0 & 0 & 1/\left(z^k_{\text{max}} - z^k_{\text{min}} \right) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{8.69}$$

Using the matrices $\mathbf{M}^k_{\text{cascade}}$ and $\mathbf{P}^k_{\text{cascade}}$, the complete MVP matrix used to render an object into the shadow map for cascade k is given by

$$\mathbf{M}^k_{\text{MVP}} = \mathbf{P}^k_{\text{cascade}} \left(\mathbf{M}^k_{\text{cascade}} \right)^{-1} \mathbf{M}_{\text{object}}. \tag{8.70}$$

For a main camera rendering a game world from with any fixed transformation matrix $\mathbf{M}_{\text{camera}}$, cascaded shadow maps are rendered correctly with the transformation matrices $\mathbf{M}^k_{\text{cascade}}$ and $\mathbf{P}^k_{\text{cascade}}$ derived from the values of $\mathbf{s}_k$ and d_k defined by Equations (8.66) and (8.68). However, an enormous problem arises when the main camera moves in any way. The exact set of texels that are covered by rasterization when an object is rendered into a cascade's shadow map is extremely sensitive to the precise position of the camera and the precise physical size of the cascade's bounding box. If the matrix $\mathbf{M}_{\text{camera}}$ changes, then so does the matrix $\mathbf{L}$ used to calculate the light-space extents of the bounding box, and this ultimately affects $\mathbf{s}_k$ and d_k. Allowing these two variables to take on arbitrary values causes the texels along the edges of every shadow to be unstable because the determination as to whether they are illuminated or in shadow is constantly changing.

The first step in stabilizing the shadows is to require that the bounding box for each cascade has a constant physical size in the light-space x and y directions. This ensures that each shadow map has a consistent resolution such that the area corresponding to a single texel never changes. The size we choose is the largest value that could ever be produced by Equation (8.68), which is the diameter of the cascade as defined by the maximal distance between any two points on its boundary.

This diameter must be equal to either the length of the internal diagonal or the length of the diagonal on the face where $z = b_k$. (Exercise 4 asks for a proof.) These two possibilities are illustrated by dashed lines in Figure 8.13. Thus, we can define a new shadow map size d_k as

$$d_k = \left\lceil \max \left(\left\| \mathbf{v}_0^k - \mathbf{v}_6^k \right\|, \left\| \mathbf{v}_4^k - \mathbf{v}_6^k \right\| \right) \right\rceil, \tag{8.71}$$

where we have rounded up to an integer value for reasons explained below. It is important that we carry out this calculation using the original camera-space positions $\mathbf{v}_i^k$ and not the light-space positions $\mathbf{L}\mathbf{v}_i^k$ appearing in Equation (8.65). Even though the results should be mathematically identical, the small measure of floating-point round-off error introduced through the transformation by the matrix $\mathbf{L}$ is enough to change the value of d_k. As long as the view frustum parameters g and s remain constant, d_k could be calculated one time and saved.

Once we have determined d_k using Equation (8.71), we know that every texel in an $n \times n$ shadow map rendered for cascade k corresponds to a physical width and height given by

$$T = d_k / n. \tag{8.72}$$

Shadow edges are stable if the x and y viewport-space coordinates of each vertex belonging to an object rendered into the shadow map have constant fractional parts. A triangle that is moved in viewport space by an integral number of texels left, right, up, or down is always rasterized in the same exact way in the sense that an identical set of texels are filled, just at different locations. Because the distance between adjacent texels in viewport space corresponds to the physical distance T, changing the camera's x or y position by any multiple of T preserves the fractional positions of the vertices. This means that we can achieve shadow stability by requiring that the light-space x and y coordinates of the camera position are always integral multiples of T. We calculate a camera position $\mathbf{s}_k$ satisfying this requirement by replacing Equation (8.66) with

$$\mathbf{s}_k = \left(\left\lfloor \frac{x_{\max}^k + x_{\min}^k}{2T} \right\rfloor T, \left\lfloor \frac{y_{\max}^k + y_{\min}^k}{2T} \right\rfloor T, z_{\min}^k \right). \tag{8.73}$$

For this calculation to be completely effective at eliminating stability problems, the value of T must be exactly representable as a floating-point number. Luckily, the number n would normally be a power of two, so its reciprocal is exactly representable and so are any integer multiples of it. All we have to do is make sure that d_k is an integer, and that's why we applied the ceiling operation in Equation (8.71).

There remains one last source of instability, and it arises in the calculation of the matrix $\mathbf{M}_{\text{cascade}}^{k}$ defined by Equation (8.67) because the position $\mathbf{s}_k$ is transformed into world space by $\mathbf{M}_{\text{light}}$. We only need the inverse of $\mathbf{M}_{\text{cascade}}^{k}$ in the MVP matrix given by Equation (8.70), but floating-point accuracy is lost during the calculation of this inverse. The problem is avoided by instead calculating $\left(\mathbf{M}_{\text{cascade}}^{k}\right)^{-1}$ directly using the formula

$$\left(\mathbf{M}_{\text{cascade}}^{k}\right)^{-1} = \left[\mathbf{M}_{\text{light}[0]}^{\text{T}} \quad \mathbf{M}_{\text{light}[1]}^{\text{T}} \quad \mathbf{M}_{\text{light}[2]}^{\text{T}} \quad -\mathbf{s}_k\right], \tag{8.74}$$

which follows from Equation (2.42) under the assumptions that the upper-left 3×3 portion of $\mathbf{M}_{\text{light}}$ is orthogonal and the translation component of $\mathbf{M}_{\text{light}}$ is zero. Using this definition for $\left(\mathbf{M}_{\text{cascade}}^{k}\right)^{-1}$ with the new camera position $\mathbf{s}_k$ and the definition of $\mathbf{P}_{\text{cascade}}^{k}$ given by Equation (8.69) with the new cascade diameter d_k, highly stable results are produced in each of the shadow maps.

When the contribution from the light source is rendered in the main scene, the cascaded shadow maps are accessed as the separate 2D layers of an array texture map. Because the cascades can overlap, the pixel shader generally needs to fetch samples from two consecutive layers and blend them together. The most difficult task is determining which two layers should be sampled and calculating the correct texture coordinates at which to sample those layers. With a total of four cascades, there are three possible pairs of cascades that can be blended, and the cascade indexes must be 0 and 1, 1 and 2, or 2 and 3.

We can calculate the indices of the cascades as well as their blending weights in the pixel shader with the help of some planar distances calculated in the vertex shader. The plane $\mathbf{f}_k$ at the front of cascade k, where $z = a_k$ in camera space, has an inward-pointing normal vector $\mathbf{n} = \mathbf{M}_{\text{camera}[2]}$ in world space. Since this plane contains the world-space point $\mathbf{c} + a_k \mathbf{n}$, where $\mathbf{c} = \mathbf{M}_{\text{camera}[3]}$ is the camera position, we can express the whole plane as

$$\mathbf{f}_k = \frac{1}{b_{k-1} - a_k}\left[\mathbf{n} \mid -\mathbf{n} \cdot \mathbf{c} - a_k\right]. \tag{8.75}$$

The scale factor $1/(b_{k-1} - a_k)$ has been applied to make the dot product $\mathbf{f}_k \cdot \mathbf{p}$ transition from zero to one precisely in the volume where cascades k and $k-1$ overlap, conveniently providing linear blending weights. In the vertex shader, each vertex position $\mathbf{p}$ is available in object space, so we take dot products with the object-space cascade plane to produce

$$u_k = \mathbf{f}_k \mathbf{M}_{\text{object}} \cdot \mathbf{p}. \tag{8.76}$$

The three scalar results for $k = 1, 2, 3$ are output by the vertex shader and interpolated across triangles.

In the pixel shader, the indices i and j of the two cascades we blend together can be determined by examining the signs of u_2 and u_3. These values tell us whether the point being rendered lies beyond the planes $\mathbf{f}_2$ and $\mathbf{f}_3$ at the front of the third and fourth cascades. The indices are given by

$$i = \begin{cases} 2, & \text{if } u_2 \geq 0; \\ 0, & \text{otherwise,} \end{cases} \quad \text{and} \quad j = \begin{cases} 3, & \text{if } u_3 \geq 0; \\ 1, & \text{otherwise,} \end{cases} \tag{8.77}$$

and these correspond to the layers that will be accessed in the shadow map. Note that in the case that $u_2 \geq 0$ but $u_3 < 0$, the indices $i = 2$ and $j = 1$ are produced in reverse order. The amounts of light L_i and L_j reaching the point being rendered, as determined by sampling layers i and j, are blended together with the simple linear interpolation

$$L = wL_i + (1 - w) L_j, \tag{8.78}$$

where w is the weight given by

$$w = \begin{cases} \text{sat}(u_2) - \text{sat}(u_3), & \text{if } u_2 \geq 0; \\ 1 - \text{sat}(u_1), & \text{otherwise.} \end{cases} \tag{8.79}$$

Outside of any transition range, this weight is always zero or one, and it effectively selects the correct cascade to sample. Inside a transition range, this weight gives the interpolation parameter used in Equation (8.78) for all three possible pairs of the indices i and j. The value of L should be multiplied by the color C_{illum} calculated with Equation (8.23) to determine the final illumination in a manner similar to what was done with the other types of shadow mapping.

As done with the 2D shadow maps associated with spot lights, we determine the texture coordinates at which the cascaded shadow maps are sampled by transforming object-space vertex positions into the coordinate space of the shadow map inside the vertex shader. However, there are now several cascades of different sizes placed at different positions. Fortunately, they are all aligned to the same light-space axes, so the distinct cascades are related by nothing more than scales and offsets. The matrix $\left(\mathbf{M}_{\text{cascade}}^{k} \right)^{-1}$ given by Equation (8.74) transforms from world space to the camera space that was used to render cascade k. The projection matrix $\mathbf{P}_{\text{cascade}}^{k}$ given by Equation (8.69) produces x and y coordinates in the range $[-1, +1]$ over the cascade's bounding box. To produce texture coordinates in the range $[0, 1]$, we scale and offset by $1/2$ to obtain the matrix

$$\mathbf{P}_{\text{shadow}}^{k} = \begin{bmatrix} 1/d_k & 0 & 0 & 1/2 \\ 0 & 1/d_k & 0 & 1/2 \\ 0 & 0 & 1/(z_{\text{max}}^{k} - z_{\text{min}}^{k}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{8.80}$$

The full transformation from world space to the texture coordinates specific to cascade k is then

$$\mathbf{M}_{\text{shadow}}^{k} = \mathbf{P}_{\text{shadow}}^{k} \left(\mathbf{M}_{\text{cascade}}^{k} \right)^{-1}, \tag{8.81}$$

and this includes the proper scaling for the depth in the shadow map. We do not calculate this matrix for each cascade. Instead, we calculate only $\mathbf{M}_{\text{shadow}}^{0}$ for the first cascade and multiply it by $\mathbf{M}_{\text{object}}$ to construct a single transformation from object space to shadow map space for the object being rendered. This matrix is supplied as a uniform input to the vertex shader, which uses it to transform object-space vertex positions and interpolate the results. Given the 3D texture coordinates in cascade 0, the texture coordinates corresponding to the same point in space for a different cascade k are found by applying the transformation

$$\mathbf{C}_{k} = \mathbf{M}_{\text{shadow}}^{k} \left(\mathbf{M}_{\text{shadow}}^{0} \right)^{-1}. \tag{8.82}$$

When we insert the definitions of each of the matrices and multiply everything out, we end up with the matrix

$$\mathbf{C}_{k} = \begin{bmatrix} \dfrac{d_0}{d_k} & 0 & 0 & \dfrac{(\mathbf{s}_0)_x - (\mathbf{s}_k)_x}{d_k} - \dfrac{d_0}{2d_k} + \dfrac{1}{2} \\ 0 & \dfrac{d_0}{d_k} & 0 & \dfrac{(\mathbf{s}_0)_y - (\mathbf{s}_k)_y}{d_k} - \dfrac{d_0}{2d_k} + \dfrac{1}{2} \\ 0 & 0 & \dfrac{z_{\text{max}}^{0} - z_{\text{min}}^{0}}{z_{\text{max}}^{k} - z_{\text{min}}^{k}} & \dfrac{(\mathbf{s}_0)_z - (\mathbf{s}_k)_z}{z_{\text{max}}^{k} - z_{\text{min}}^{k}} \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{8.83}$$

which consists of the 3D scale and offset that were promised above. These six numbers are precalculated for each cascade other than the first and supplied as uniform inputs to the pixel shader. They allow us to easily convert the texture coordinates computed in the vertex shader from cascade 0 to any other cascade.

A pixel shader function that samples two layers in a cascaded shadow map is shown in Listing 8.9. The `cascadeCoord0` parameter passed to the function contains the texture coordinates given by the interpolated result of transforming object-space vertex positions with the matrix $\mathbf{M}_{\text{shadow}}^{0}\mathbf{M}_{\text{object}}$ in the vertex shader. The `cascadeBlend` parameter contains the interpolated values of u_1, u_2, and u_3 given by Equation (8.76) in its x, y, and z components. The uniform constant `cascade-Scale`$[k-1]$ contains the three scale entries along the diagonal of the matrix $\mathbf{C}_k$, and `cascadeOffset`$[k-1]$ contains the offset entries in the fourth column of $\mathbf{C}_k$. These are used to calculate the texture coordinates corresponding to cascades 1, 2, and 3. The code then determines the layer indices i and j using Equation (8.77) and selects the two sets of texture coordinates that will be used to sample from those layers. The depth for the last cascade is clamped to [0,1] so that geometry lying beyond the farthest cascade plane is not darkened due to its depth being greater than the depth to which the shadow map is cleared. The final calculation before sampling begins produces the weight w given by Equation (8.79).

The example code in Listing 8.9 takes four samples from each of the two cascades that could be blended together. The locations of the samples are specified in relation to the base texture coordinates by the (x, y) and (z, w) components of the uniform constants `shadowOffset[0]` and `shadowOffset[1]`. These offsets would typically form a simple pattern in a small neighborhood of a few texels in diameter, and they can be chosen by experimentation to produce satisfactory results. The four samples from each cascade are multiplied by $1/4$ to take an average and then blended together using Equation (8.78). A scene rendered with this pixel shader is shown in Figure 8.14 along with the four shadow map cascades. The sample offsets used in this image are $(-\delta, -3\delta)$, $(3\delta, -\delta)$, $(\delta, 3\delta)$, and $(-3\delta, \delta)$, where δ has been set to $3/16r$, and r is the reciprocal of the shadow map resolution.

Listing 8.9. This pixel shader function samples the cascaded shadow map `shadowTexture` that has been rendered with four layers for an infinite light. The `cascadeCoord0` parameter contains the interpolated texture coordinates for cascade 0, and the `cascadeBlend` parameter contains the interpolated values of u_1, u_2, and u_3 given by Equation (8.76). The uniform constant `shadowOffset` contains four sets of texture coordinate offsets that determine where samples are taken. The uniform constants `cascadeScale` and `cascadeOffset` contain the nontrivial entries of the matrices $\mathbf{C}_1$, $\mathbf{C}_2$, and $\mathbf{C}_3$ given by Equation (8.83).

```
uniform Texture2DArrayShadow  shadowTexture;
uniform float4                shadowOffset[2];
uniform float3                cascadeScale[3];
uniform float3                cascadeOffset[3];
```

```
float CalculateInfiniteShadow(float3 cascadeCoord0, float3 cascadeBlend)
{
    float3    p1, p2;

    // Apply scales and offsets to get texcoords in all four cascades.
    float3 cascadeCoord1 = cascadeCoord0 * cascadeScale[0] + cascadeOffset[0];
    float3 cascadeCoord2 = cascadeCoord0 * cascadeScale[1] + cascadeOffset[1];
    float3 cascadeCoord3 = cascadeCoord0 * cascadeScale[2] + cascadeOffset[2];

    // Calculate layer indices i and j.
    bool beyondCascade2 = (cascadeBlend.y >= 0.0);
    bool beyondCascade3 = (cascadeBlend.z >= 0.0);
    p1.z = float(beyondCascade2) * 2.0;
    p2.z = float(beyondCascade3) * 2.0 + 1.0;

    // Select texture coordinates.
    float2 shadowCoord1 = (beyondCascade2) ? cascadeCoord2.xy : cascadeCoord0.xy;
    float2 shadowCoord2 = (beyondCascade3) ? cascadeCoord3.xy : cascadeCoord1.xy;
    float depth1 = (beyondCascade2) ? cascadeCoord2.z : cascadeCoord0.z;
    float depth2 = (beyondCascade3) ? saturate(cascadeCoord3.z) : cascadeCoord1.z;

    // Calculate blend weight w.
    float3 blend = saturate(cascadeBlend);
    float weight = (beyondCascade2) ? blend.y – blend.z : 1.0 – blend.x;

    // Fetch four samples from the first cascade.
    p1.xy = shadowCoord1 + shadowOffset[0].xy;
    float light1 = texture(shadowTexture, p1, depth1);
    p1.xy = shadowCoord1 + shadowOffset[0].zw;
    light1 += texture(shadowTexture, p1, depth1);
    p1.xy = shadowCoord1 + shadowOffset[1].xy;
    light1 += texture(shadowTexture, p1, depth1);
    p1.xy = shadowCoord1 + shadowOffset[1].zw;
    light1 += texture(shadowTexture, p1, depth1);

    // Fetch four samples from the second cascade.
    p2.xy = shadowCoord2 + shadowOffset[0].xy;
    float light2 = texture(shadowTexture, p2, depth2);
    p2.xy = shadowCoord2 + shadowOffset[0].zw;
    light2 += texture(shadowTexture, p2, depth2);
    p2.xy = shadowCoord2 + shadowOffset[1].xy;
    light2 += texture(shadowTexture, p2, depth2);
    p2.xy = shadowCoord2 + shadowOffset[1].zw;
    light2 += texture(shadowTexture, p2, depth2);

    // Return blended average value.
    return (lerp(light2, light1, weight) * 0.25);
}
```
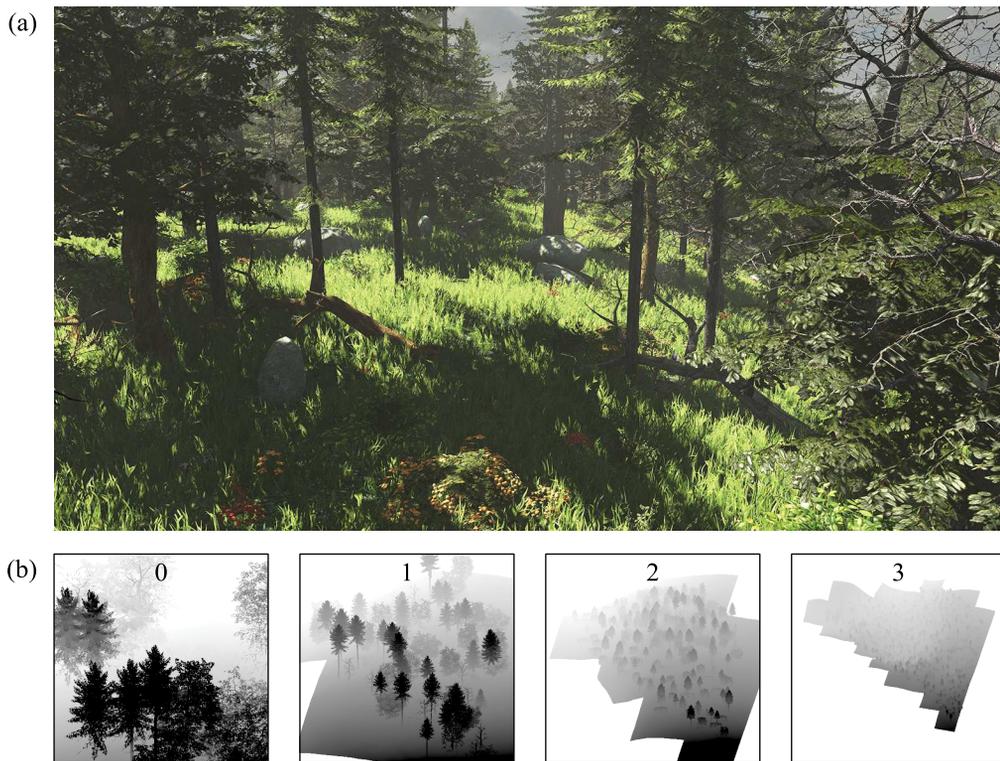
Figure 8.14. (a) Shadows are rendered with a cascaded shadow map having four layers of size 1024×1024 texels. Cascade 0 extends from the camera position to 15 m along the view direction, cascade 1 extends from 10 m to 50 m, cascade 2 extends from 40 m to 120 m, and cascade 3 extends from 100 m to 320 m. (b) The depths stored in the separate cascades are shown as grayscale images rendered from the perspective of the light source. The area covered by the shadow map is considerably larger as the cascade index increases. In each cascade, the main camera position is near the upper-right corner of the image, and the geometry rendered in the shadow map roughly approximates the shape of the cascade boundary.

8.3.4 Shadow Depth Offset

When a shadow map is rendered from the perspective of the light source, it records depths of the scene geometry at a set of positions corresponding to the centers of the texels in the shadow map. When the scene is later rendered from the perspective of the main camera, depths relative to the light source are calculated for a *different* set of positions corresponding to the centers of the pixels in the frame buffer. Consequently, for a pixel belonging to any surface that does not have a shadow cast

upon it, the depth fetched from the shadow map and the calculated depth are very close to the same value, but they are not an exact match. Either of the values can be less or greater than the other, and this is true even in the absence of floating-point precision limits. The result is one of the well-known artifacts for which shadow maps are notorious. Surfaces tend to cast shadows on themselves in essentially random patterns that move and flicker due to their highly sensitive dependence on the camera transform.

The specific type of self-shadowing just described is rather easy to eliminate. (A related but different artifact discussed below is more difficult.) All we need to do is add a small value to all the depths rendered into every shadow map. This causes all the surfaces to be pushed away from the light source by a little bit so that their depths are no longer close enough to the values calculated for the main camera to cause a problem. A reliable way to offset the depths in a shadow map is to modify the projection matrix applied when the shadow map is rendered by adding the offset δ to the appropriate matrix entry.

For the perspective projection matrix $\mathbf{P}_{\text{shadow}}$ given by Equation (8.55), which we use to render the shadow maps associated with spot and point lights, we add δ to the $(2, 2)$ entry because that entry becomes the constant term in the depth calculation after the perspective divide. (If the matrix is a reversing projection, then we instead subtract δ to increase the distance from the camera.) For the orthographic projection matrix $\mathbf{P}^{k}_{\text{cascade}}$ given by Equation (8.69), which we use to render cascaded shadow maps, we simply set the $(2, 3)$ entry to δ because there is no perspective divide. In either case, a value of $\delta = 2^{-19}$ produces excellent results and removes the random self-shadowing artifact in all practical situations. Large values of δ should be avoided because they can cause surfaces facing the light source to be pushed beyond the back side of a thin object, and this can lead to the appearance of detached shadows.

There is a second well-known artifact of shadow mapping called *shadow acne*, and although good methods exist for getting rid of it, it often cannot be eliminated completely. As illustrated in Figure 8.15, shadow acne generally arises from a significant and ordinarily unavoidable mismatch between the resolution of the shadow map and the resolution of the frame buffer. On any illuminated surface, a single texel in the shadow map can cover an area that fills many pixels when the main scene is rendered. The texel holds just one depth that represents the entire area it covers, but for any surface not perpendicular to the light direction, the true depth of the geometry isn't actually constant over that area. As a result, the depth sampled from the shadow map is too small for some pixels and too large for others, demonstrated by the stair-step pattern in the Figure 8.15. The pixels for which the depth is too small are mistakenly classified as being inside a shadow, and this produces

a periodic darkening on the surface. When many samples are taken for percentage closer filtering, the effect tends to smooth out to become the repeating gradient pattern that can be seen in Figure 8.16(a).

The maximum error δ between the depth sampled from the shadow map and the true depth of a flat surface depends on the angle θ between the normal vector $\mathbf{n}$ and the direction $\mathbf{l}$ to the light source. As shown in Figure 8.15, we have the trigonometric relationship

$$\delta = \tfrac{1}{2} \tan \theta. \tag{8.84}$$

If the depth could be offset away from the light source by at least this amount when the shadow map is rendered, then we could avoid many self-shadowing artifacts. To account for all possible alignments of pixels and texels, we generally need to double the value of δ. Furthermore, a larger depth offset must be used to account for surfaces that curve away from the light source and thus extend to greater depths within the footprint of a single texel. In practice, a depth offset of around $3 \tan \theta$ works well in typical scenes.
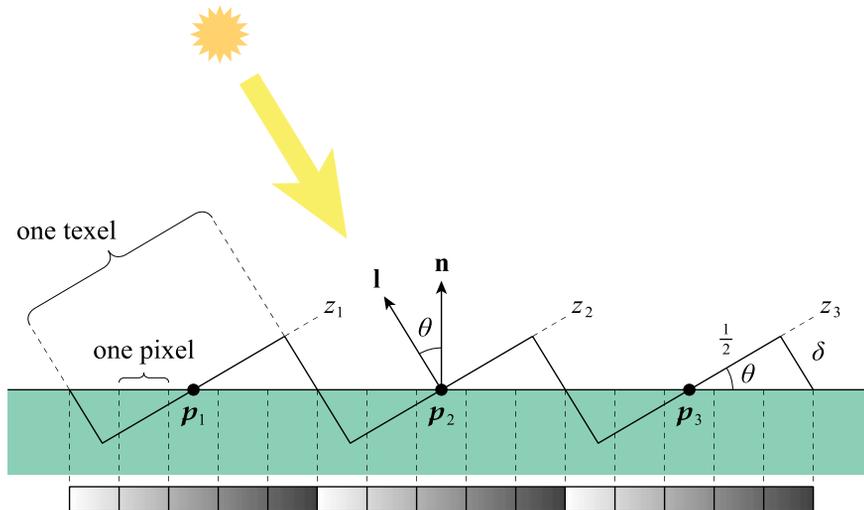
Figure 8.15. The points $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$ represent the centers of texels at which the depths z_1, z_2, and z_3 are rendered in a shadow map. A single texel of the shadow map may cover many pixels when the main scene is rendered, but the same depth value is not accurate over that entire area. This causes self-shadowing artifacts similar to the repeating gradient shown below the surface, and they get worse as the angle θ between the normal vector $\mathbf{n}$ and the direction to light $\mathbf{l}$ increases.

Figure 8.16. (a) A shadow map for an outdoor scene is rendered without any slope-based depth offset, and severe shadow acne is visible on most surfaces. (b) The GPU has been configured to apply depth offset with a slope factor of 3.0 multiplying the value of m given by Equation (8.85), and the shadow acne has been eliminated. (c) Without depth offset clamping, surfaces nearly parallel to the light direction can be pushed beyond shadow-receiving surfaces, and this allows light to leak through. (d) The depth offset has been clamped to a maximum value of $1/128$, and the artifacts have been removed.

Fortunately, the polygon offset functionality built into most GPUs is designed to perform the type of angle-dependent depth offset calculation that we need to remove shadow acne. As the shadow map is being rendered, the GPU can calculate the value

$$m = \max\left(\left| \frac{\Delta z}{\Delta x} \right|, \left| \frac{\Delta z}{\Delta y} \right| \right) \tag{8.85}$$

corresponding to the largest amount by which the depth z changes for a one-texel change in the x or y directions. This maximum slope is precisely the value of 2δ that we want because m is basically equal to the larger value of $\tan \theta$ calculated in

horizontal and vertical planes. All we have to do is specify what multiple of $\tan\theta$ that we would like that hardware to use, which should be positive for conventional projection matrices and negative for reversing projection matrices. An outdoor scene in which shadow acne is removed with a slope-based depth offset of $3m$ is shown in Figures 8.16(a) and 8.16(b).

There is one problem with the value of m given by Equation (8.85). As the angle between the normal vector $\mathbf{n}$ and direction to light $\mathbf{l}$ approaches 90 degrees, the tangent function becomes arbitrarily large. This can cause the depth offset to be large enough that the shadow-casting surface is pushed beyond the surface that should receive the shadow. This causes artifacts like those shown in Figure 8.16(c), where light leaks through an I-beam because it has interior surfaces that are nearly parallel to the light direction. The solution to this problem is to limit the value of m to some maximum offset, and this functionality is also provided by most GPUs. In Figure 8.16(d), the maximum offset has been set to 1/128, and the artifact has been eliminated.

8.4 Stencil Shadows

The *stencil shadow* algorithm is a method for rendering highly accurate shadows that was popular for several years beginning in the late 1990s. The method has since gone out of fashion for a variety of reasons, some of which are mentioned below, but we nevertheless include a complete discussion of stencil shadows here with the purpose of illustrating a clever procedure that could be an inspiration to those designing new techniques in computer graphics. The concept of a silhouette, introduced in this section, also has important applications in Chapter 9.

As its name suggests, the stencil shadow algorithm uses the stencil buffer to mask off areas of the scene that should not receive light. Before the contribution from any particular light source is rendered, the stencil buffer is cleared to zero, and the volumes of space blocked by shadow-casting objects are drawn in such a way that nonzero values are left in the stencil buffer wherever shadows fall on other objects in the scene. When the contribution from the light source is rendered, the stencil test is configured so that light is added to the frame buffer only for pixels still having a stencil value of zero. Since the stencil buffer can be multisampled, this produces a nicely antialiased shadow outline, and unlike shadow maps, its precision always matches the full resolution of the frame buffer.

8.4.1 Rendering Algorithm

The stencil shadow algorithm works by constructing a triangle mesh called a *shadow volume* for each object that can cast shadows. A shadow volume is drawn

only into the stencil buffer, so it is never directly visible when a scene is rendered. The exact shape of the shadow volume depends on the position of the light source, so an object simultaneously casting shadows for *n* different lights during the same frame needs to have *n* different shadow volumes to be available at certain times during that frame. The stencil shadow algorithm is not generally amenable to combining multiple lights into a single rendering pass, so the process we describe here has to be repeated independently for each light that affects the visible scene. Our discussion considers only a single light source with the understanding that additional light sources are handled in a loop.

A shadow volume is constructed by locating all the edges in an object's triangle mesh having the property that one of the two triangles sharing the edge faces toward the light source and the other triangle faces away from the light source. The set of all such edges forms the *silhouette* of the object with respect to the light source, and it represents the boundary between the lit and unlit parts of the object. By extruding the silhouette edges away from the light position, we create a surface that encloses the volume of space where light is blocked by the object's shadow. Shadow volumes are built a little differently for point lights and infinite lights, but the rendering operations are the same for both types of light source. Spot lights are treated exactly the same as point lights, so we make no distinction between them in this section.

An object is not required to be convex to cast a stencil shadow, and silhouettes can be rather complicated. This is not a problem, however, as long as every silhouette is composed of one or more closed loops. This condition is never verified directly because it is guaranteed by requiring each object's triangle mesh to be a *closed orientable manifold*. What this means is that every triangle is adjacent to exactly three other triangles across its three edges and that the two triangles in every pair sharing a common edge have consistent winding directions. There can be no dangling edges belonging to only one triangle because that would imply the mesh has a hole in it through which the interior is visible. Also, adjacent triangles must have normal vectors that either both point outward or both point inward. The requirement of a closed orientable manifold forces every object to be solid, with well-defined interior and exterior spaces, and it makes the stencil shadow algorithm unable to tolerate sloppy modeling.

Techniques for rendering stencil shadows come in two variants called *Z-pass* and *Z-fail*, and both of these variants make use of the two-sided stencil functionality available on most GPUs. Shadow volumes are rendered with face culling disabled so that different stencil operations can be applied to front faces and back faces in a single draw call. The Z-pass variant of the algorithm modifies the stencil buffer only when the depth test passes, and the Z-fail variant of the algorithm modifies

the stencil buffer only when the depth test fails. The appropriate variant to use depends on the relationship among the positions of the camera, the light source, and the object casting a shadow. We first describe how each variant works and then discuss the precise conditions under which we choose one variant or the other.

Before we can begin rendering shadows, the contents of the depth buffer must already have been established. This is usually accomplished by rendering the scene with shading contributions only from ambient light and other effects that are not dependent on any direct source of light, such as emission and environment mapping. Once this ambient pass has been completed, we add the contribution from each light source in a separate rendering pass, but only after we have constructed shadow masks in the stencil buffer. These additional passes must be rendered with the depth test configured so that equal depth values pass because the triangles will be the same as those rendered in the ambient pass.

The shadow rendering process for a single light is initiated by clearing the stencil buffer to zero. We then render the silhouette extrusion for every object that could be casting a shadow for that light using the same camera and view frustum used for the main scene. The stencil shadow algorithm does not render from different camera positions as would be done in shadow mapping. While rendering silhouette extrusions into the stencil buffer, it is necessary to configure the depth test so that equal depth values fail to avoid self-shadowing.

The Z-pass variant of the stencil shadow algorithm is the simpler of the two types, and it's the one that we are able to use in most situations. For the Z-pass variant, each shadow volume is composed only of the polygons created by extruding the silhouette edges of an object away from the light source. These polygons are rendered with the stencil operation configured such that the value in the stencil buffer is incremented for front faces and decremented for back faces whenever the depth test passes, as illustrated in Figure 8.17(a). No change is made to the value in the stencil buffer when the depth test fails. It's important to specify the increment and decrement operations that allow wrapping so that shadow volume faces can be rendered in any order.

Along any ray starting at the camera position c, if both the front side and back side of the silhouette extrusion are visible, because neither fails the depth test, then the stencil value is decremented for every time it is incremented, resulting in no net change. These rays, such as the ray labeled B in Figure 8.17(a), pass completely through the shadow volume and ultimately hit some point on a surface farther away that is not in the object's shadow. However, for rays along which the front side of the silhouette extrusion passes the depth test, but the back side fails the depth test, the stencil value is only incremented. Without a balancing decrement operation, a nonzero value is left in the stencil buffer. These are precisely the rays for which a
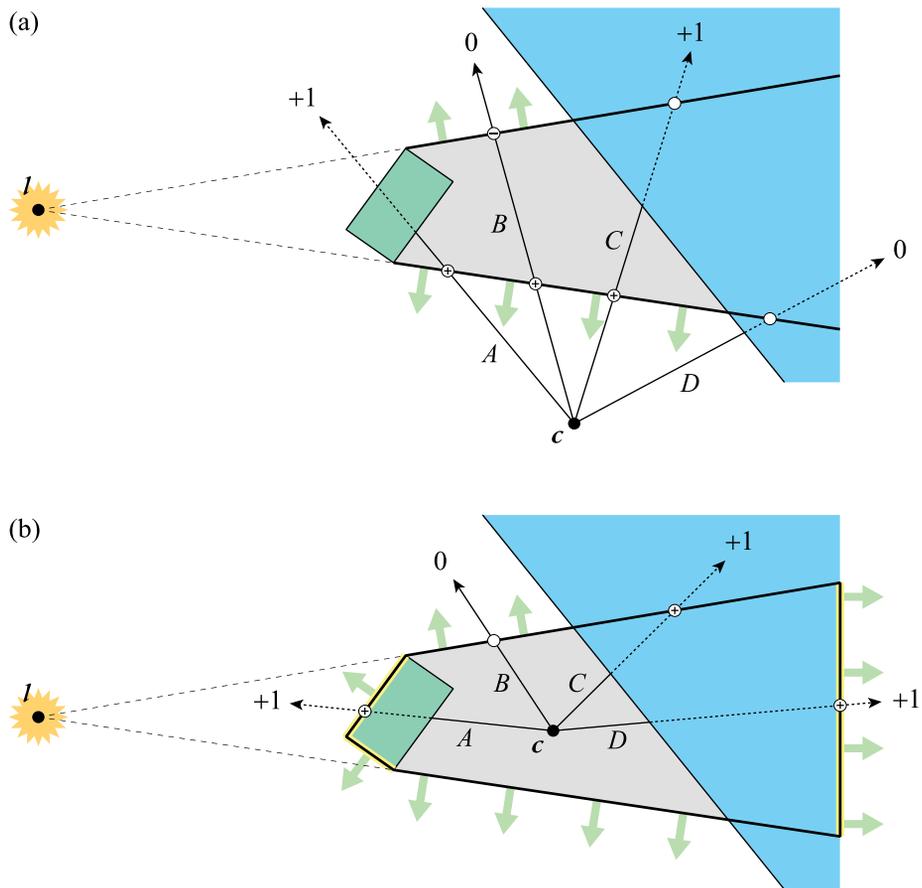
Figure 8.17. A shadow volume for an object represented by the green box is extruded away from the light position *l*, and it intersects solid geometry represented by the blue region. The green arrows show the normal direction for the polygons making up the shadow volume. Rays emanating from the camera position *c* demonstrate how stencil operations applied to the front and back faces of the shadow volume cause a mask to be rendered in the stencil buffer. Circles containing plus signs and minus signs indicate where the stencil value is incremented and decremented. No change is made to the stencil value for empty circles. (a) The extruded polygons of the object's silhouette edges are rendered with the Z-pass variant of the stencil shadow algorithm. The stencil value is incremented and decremented when *front* faces and *back* faces of the shadow volume *pass* the depth test, respectively. (b) Light and dark caps, highlighted in yellow, are added to the shadow volume to create a closed surface for the Z-fail variant of the algorithm. The stencil value is incremented and decremented when *back* faces and *front* faces *fail* the depth test, respectively.

shadow-receiving surface passes between the front and back sides of the silhouette extrusion, as exemplified by the ray labeled C in Figure 8.17(a). A similar effect occurs along the ray labeled A, but in this case, the increment operation is not balanced by a decrement because there is no back side to the silhouette extrusion along that direction. In the remaining case, as shown by the ray labeled D, the entire shadow volume passes behind solid geometry, so the value in the stencil buffer is neither incremented nor decremented, and the point where the ray intersects the surface is not in shadow.

The Z-pass variant works extremely well as long as the camera is *outside* of every object's silhouette extrusion. More specifically, if a shadow volume intersects the near plane of the view frustum inside the viewport rectangle, then the correct values will not be written to the stencil buffer because, in any particular ray direction, there may be no front faces relative to the camera position where the stencil value would be incremented. The solution to this problem is to use the Z-fail variant of the algorithm, which always produces the correct results regardless of the camera position. The Z-fail variant is more complicated than the Z-pass variant, and it generally requires that we operate on more pixels in the stencil buffer, so it is more expensive. For these reasons, we do not use Z-fail all the time but instead choose to use it on a per-object basis only when we have detected that there exists a possible intersection between the shadow volume and the visible part of the near plane, as described below.

There are two differences between the Z-pass and Z-fail variants of the algorithm. First, the Z-fail variant requires that each shadow volume be a closed surface, which means we must do more than just extrude the silhouette edges of an object. We also add a *light cap*, which consists of all the triangles in the object that face toward the light source, and we add a *dark cap*, which consists of all the triangles in the object that face away from the light source. The light cap is rendered in place without any modification. The dark cap, however, is projected away from the light source to the far end of the silhouette extrusion. With the addition of these caps, the shadow volume is now bounded by a closed surface. The second difference is that the stencil operations are applied only when the depth test fails, and they are inverted with respect to which side of a shadow volume polygon is visible. The stencil operations are configured such that the value in the stencil buffer is incremented for back faces and decremented for front faces when the depth test fails, as illustrated in Figure 8.17(b). In the Z-fail variant, no change is made to the value in the stencil buffer when the depth test passes.

As demonstrated by the rays labeled A, C, and D in Figure 8.17(b), the value in the stencil buffer is incremented whenever a back facing polygon belonging to the shadow volume lies beyond a solid surface in a particular ray's direction. Such

a back facing polygon can be part of the silhouette extrusion, or it can belong to either the light cap or dark cap. In the case that the ray hits a back facing polygon before it intersects a solid surface, as demonstrated by the ray labeled B, the value in the stencil buffer is not changed. These operations cause the correct masks to be drawn in the stencil buffer when the camera position c lies inside the shadow volume. By decrementing the stencil value when a ray hits a front facing polygon behind solid geometry, shadows are also rendered correctly when the camera position is outside the shadow volume.

The procedure for rendering stencil shadows is summarized by the following list. Before shadows are rendered for the first light source, an ambient pass of some kind must be rendered to fill the depth buffer with its final values. Then these steps are repeated for each light source that casts shadows.

- Clear the stencil buffer to zero.
- Disable writes to the color buffer and the depth buffer. Shadow volumes are rendered only into the stencil buffer.
- Set the depth test function to *less* for a conventional projection matrix or to *greater* for a reversing projection matrix. Any fragment having a depth equal to the value in the depth buffer should fail the depth test.
- Disable face culling. Front faces and back faces of each shadow volume must both be rendered.
- Enable the stencil test, and configure the stencil test so that it always passes. Stencil operations are based only on the outcome of the depth test.
- For each shadow volume, configure two-sided stencil operations as follows:

  - For the Z-pass variant, set the stencil operations to *increment with wrap* for front faces that pass the depth test and *decrement with wrap* for back faces that pass the depth test.
  - For the Z-fail variant, set the stencil operations to *decrement with wrap* for front faces that fail the depth test and *increment with wrap* for back faces that fail the depth test.

- After all shadow volumes have been rendered, enable writes to the color buffer so that the contribution from the light can be added to the frame buffer.
- Enable blending, set the blend operation to *add*, and set the blend factors for both the source and destination colors to *one*.
- Set the depth test function to *less or equal* for a conventional projection matrix or to *greater or equal* for a reversing projection matrix. Fragments having depths identical to those in the depth buffer must pass the depth test in the lighting pass.

- Enable face culling so that back faces of solid geometry are not rendered.
- Set the stencil test function to *equal*, and set the stencil reference value to *zero*. Configure all stencil operations so that they *keep* the values that are stored in the stencil buffer.
- Render the lighting pass. The stencil test will prevent changes from being made in shadows where the stencil value is nonzero.

8.4.2 Variant Selection

The Z-fail variant of the stencil shadow algorithm always works, regardless of whether the camera is inside or outside each shadow volume, but it is more expensive than the Z-pass variant. We now answer the question of how to decide when it is safe to use the Z-pass variant and when we must use the Z-fail variant to guarantee correct results. Let R represent any ray that starts at the light position and passes through the near plane inside the four sides of the view frustum. The exact condition that determines which variant to use for a particular shadow-casting object is whether there exists a ray R that intersects the object at any point. If such a ray exists, then we must render the shadow volume with the Z-fail variant because either it includes the camera position or its surface is clipped by the near plane (or both). Before we can use the Z-pass variant, we have to be sure that the object does not intersect the volume of space formed by the complete set of all rays R. This volume is called the *Z-fail region*, and it has the shape of a thin oblique pyramid, as shown in Figure 8.18.

As illustrated in Figure 8.18(a), the Z-fail region is defined in world space by the light position $\boldsymbol{l}$ and the four points $\boldsymbol{q}_i$ at the corners of the rectangle where the near plane intersects the lateral planes of the view frustum. The points $\boldsymbol{q}_i$ are given by Equation (6.6) with $u = n$. Combining them with the light position $\boldsymbol{l}$, we can calculate four world-space planes $\mathbf{f}_i$ with inward-pointing normal vectors that, together with the near plane, bound the Z-fail region. The signs of the bounding planes depend on the location of the light source with respect to the near plane. Assuming for the moment that the light position $\boldsymbol{l}$ lies beyond the near plane in the view direction $\mathbf{z}$, the world-space near plane $\mathbf{f}_{\text{near}}$ is given by

$$\mathbf{f}_{\text{near}} = \left[\mathbf{z} \mid -\mathbf{z} \cdot \boldsymbol{c} - n\right]. \tag{8.86}$$

For the four planes $\mathbf{f}_i$, we first calculate a normal vector $\mathbf{n}_i$ using the formula

$$\mathbf{n}_i = \left(\boldsymbol{q}_{(i+1)\bmod 4} - \boldsymbol{q}_i\right) \times \left(\boldsymbol{l} - \boldsymbol{q}_i\right), \tag{8.87}$$

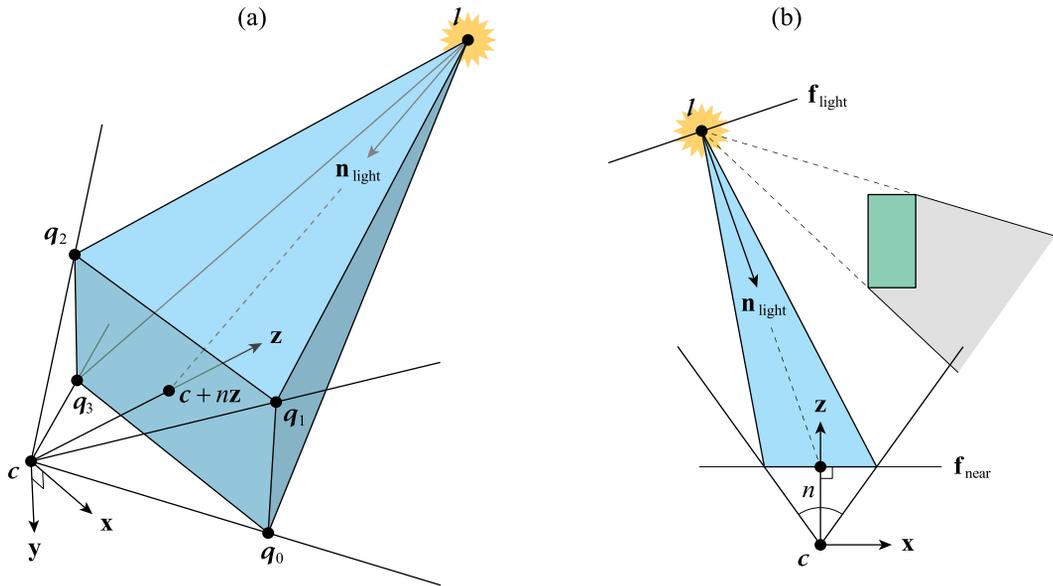and then the normalized plane $\mathbf{f}_i$ is given by

Figure 8.18. The Z-fail region has the shape of an oblique rectangular pyramid, and it is used to determine whether a shadow volume can be safely rendered with the Z-pass variant of the stencil shadow algorithm. (a) The edges of the pyramid connect the world-space light position l to the points q_0, q_1, q_2, and q_3 where the near plane intersects the lateral planes of the view frustum. (b) Any object that does not intersect the Z-fail region, such as the green box, cannot cast a shadow into the visible part of the near plane, and thus its shadow volume can be rendered with the Z-pass variant. To prevent false intersection results near the sharp point at the light position, the plane $\mathbf{f}_{light}$ is added to the Z-fail region. Its normal direction points to the center of the near plane at $c + n\mathbf{z}$, where c is the world-space camera position, $\mathbf{z}$ is the view direction, and n is the distance to the near plane.

$$\mathbf{f}_i = \frac{1}{\|\mathbf{n}_i\|}[\mathbf{n}_i \mid -\mathbf{n}_i \cdot q_i]. \tag{8.88}$$

If $(l - c) \cdot \mathbf{z} < n$, then the light position l does not lie beyond the near plane. In this case, all five of the planes defined by Equations (8.86) and (8.88) must be negated so that their normal vectors point toward the interior of the Z-fail region.

To determine whether a shadow-casting object intersects the Z-fail region, we test its bounding volume against the convex region of space enclosed by the planes $\mathbf{f}_0$, $\mathbf{f}_1$, $\mathbf{f}_2$, $\mathbf{f}_3$, and $\mathbf{f}_{near}$ using the same methods described in Section 9.4 that we use for frustum culling. For example, we know that an object bounded by a sphere having center p and radius r does not intersect the Z-fail region if $\mathbf{f} \cdot p < -r$ for any

one of the planes. When there is no intersection, we render the object's shadow volume with the Z-pass variant. Otherwise, if an intersection is likely based on the outcome of the test, then we render the object's shadow volume with the Z-fail variant.

Because the distance n to the near plane is usually very small, the points q_i are close together, and this causes the bounding planes of the Z-fail region to meet at a sharp point at the light position l. This shape can lead to false intersection results for objects that are near the light source but have a bounding volume that lies completely outside the Z-fail region. To remedy this situation, we add a sixth plane $\mathbf{f}_{\text{light}}$ passing through the light position, as shown in Figure 8.18(b). The normal vector $\mathbf{n}_{\text{light}}$ for this plane points toward the center of the near plane, which is the point $c + n\mathbf{z}$. Thus, $\mathbf{n}_{\text{light}}$ is given by

$$\mathbf{n}_{\text{light}} = c + n\mathbf{z} - l, \tag{8.89}$$

and the normalized plane $\mathbf{f}_{\text{light}}$ is calculated as

$$\mathbf{f}_{\text{light}} = \frac{1}{\|\mathbf{n}_{\text{light}}\|}\left[\mathbf{n}_{\text{light}} \mid -\mathbf{n}_{\text{light}} \cdot l\right]. \tag{8.90}$$

This plane is added to the boundary of the Z-fail region for light positions on both sides of the near plane.

When rendering shadows for an infinite light, we can consider the vector $\mathbf{l}_{\text{infinite}}$ pointing in the direction toward the light to be a point $(l_x, l_y, l_z, 0)$ at infinity having a w coordinate of zero. This allows us to incorporate infinite lights into the calculation of the Z-fail region's bounding planes by revising Equation (8.87) to produce normal vectors using the formula

$$\mathbf{n}_i = \left(q_{(i+1)\bmod 4} - q_i\right) \times \left(\mathbf{l}_{xyz} - l_w q_i\right), \tag{8.91}$$

where $\mathbf{l}$ is now a generalized 4D homogeneous vector representing the position of the light source. For a point light, $l_w = 1$ and nothing changes. For an infinite light, $l_w = 0$ and the normal vector $\mathbf{n}_i$ is always perpendicular to $\mathbf{l}$. In this case, the Z-fail region extends to infinity, and the plane $\mathbf{f}_{\text{light}}$ is never included.

If the light position happens to be close to the near plane, then the Z-fail region becomes very flat, and floating-point precision issues can arise in Equation (8.91). We can detect this case for both point lights and infinite lights by calculating

$$d = \left(\mathbf{l}_{xyz} - l_w c\right) \cdot \mathbf{z} - n \tag{8.92}$$

and comparing it to a small distance ε. If $|d| \geq \varepsilon$, then the light source is not too close to the near plane, and we proceed to calculate the bounding planes of the Z-fail region as previously described, remembering to negate the planes produced by Equations (8.86) and (8.88), but not Equation (8.90), if d is negative. Otherwise, if $|d| < \varepsilon$, then the light source is close to the near plane, and we define the Z-fail region by the two planes

$$\mathbf{f}_A = \begin{bmatrix} \mathbf{z} \,|\, -\mathbf{z} \cdot \mathbf{c} - n + \varepsilon \end{bmatrix}$$
$$\mathbf{f}_B = \begin{bmatrix} -\mathbf{z} \,|\, \mathbf{z} \cdot \mathbf{c} + n + \varepsilon \end{bmatrix}. \tag{8.93}$$

The shadow volume for any object not intersecting the thin but infinitely wide volume of space bounded by these two planes can be safely rendered using the Z-pass variant.

8.4.3 Shadow Volumes

A shadow volume is composed of up to three distinct components: the extrusion of the silhouette edges, the light cap, and the dark cap. The extrusion is the only part that is always rendered because the caps are not necessary in the Z-pass variant of the algorithm. In order to construct the silhouette extrusion for a triangle mesh, we need to precompute some additional information beyond the arrays of vertex positions and per-triangle vertex indices. We also need a list of data structures that tell us how triangles share their edges so we can identify the silhouette edges between triangles facing toward the light source and facing away from it. To store this information, we use the Edge structure shown in Listing 8.10.

An Edge structure is associated with each unique pair of vertices a and b that forms the edge of any triangle in the mesh. The indices referring to those two vertices must be stored in a consistent order with respect to the two triangles that share the edge so that it's possible to create an extrusion whose polygons have a consistent winding direction. The rule we follow is that the indices of the vertices a

Listing 8.10. For each edge in a triangle mesh, the Edge structure stores the indices of the two vertices connected by the edge and the indices of the two faces that share the edge. The first face is always the one for which the two vertex indices correspond to a counterclockwise winding direction.

```
struct Edge
{
    uint16    vertexIndex[2];
    uint16    faceIndex[2];
};
```

and **b** are stored in whichever order is counterclockwise with respect to the front side of the first triangle referenced by the Edge structure. For example, consider the two triangles sharing a common edge in Figure 8.19. Written in counterclockwise order, the first triangle, shown on the right in the figure, has the vertex indices (i_0, j_0, k_0), and the second triangle has the vertex indices (i_1, j_1, k_1). Because the triangles share the edge connecting **a** and **b**, the indices i_0 and k_1 are equal and refer to the vertex **a**. Likewise, the indices j_0 and j_1 are equal and refer to the vertex **b**. Enforcing our vertex order rule, the vertex indices for the edge must be specified as (i_0, j_0) because that follows the counterclockwise direction for the first triangle.

As demonstrated in Listing 8.11, the complete list of edges belonging to a triangle mesh can be constructed by making two passes over the list of triangles. In the first pass, every triangle is visited, and a new Edge structure is initialized for each edge satisfying the condition that the first vertex index for the edge is *less* than the second index in counterclockwise order. For a triangle having indices (i, j, k), an edge is created whenever $i < j$, $j < k$, or $k < i$. Either one or two of these inequalities is true for every triangle. (See Exercise 6.) Each of these new edges is stored in a temporary linked list associated with the lesser of its two vertex indices so it can be found quickly later. When the first pass completes, an Edge structure exists for every edge in the triangle mesh because the vertex indices defining any particular edge always have increasing numerical values in counterclockwise order for exactly one of the two triangles sharing that edge. The job of the second pass is to match the adjacent triangle to every edge that has already been created. We again
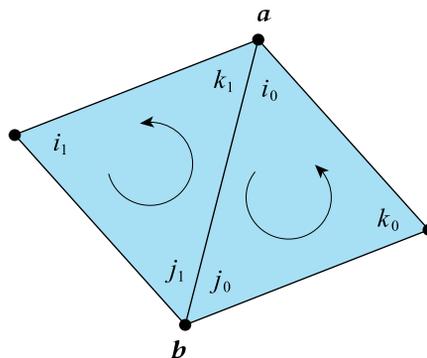
Figure 8.19. Two triangles having vertex indices (i_0, j_0, k_0) and (i_1, j_1, k_1), wound in counterclockwise order, share the edge connecting vertices **a** and **b**. By the ordering rule for an edge, the vertex indices must be specified as (i_0, j_0) or the equivalent (k_1, j_1) so they follow a counterclockwise direction for the first triangle.

visit every triangle, but this time we look for edges for which the first vertex index *i* is *greater* than the second index *j* in counterclockwise order. When such an edge is encountered, we search the linked list associated with the lesser index *j* for the Edge structure having the same greater index *i*. For the matching edge, we fill in the index of the current triangle, which must be adjacent to the triangle for which the edge was originally created.

Listing 8.11. This function builds an array of Edge data structures for an arbitrary triangle mesh having vertexCount vertices and triangleCount triangles. The triangleArray parameter points to an array of Triangle data structures each holding three vertex indices. The edgeArray parameter must point to storage for the maximum number of edges that could be produced, which can never exceed twice the number of triangles in the mesh. The return value is the number of Edge structures that were written to edgeArray.

```
int32 BuildEdgeArray(int32 vertexCount, int32 triangleCount,
                     const Triangle *triangleArray, Edge *edgeArray)
{
    // Initialize all edge lists to empty.
    uint16 *firstEdge = new uint16[vertexCount + triangleCount * 2];
    uint16 *nextEdge = firstEdge + vertexCount;
    for (int32 k = 0; k < vertexCount; k++) firstEdge[k] = 0xFFFF;

    int32 edgeCount = 0;
    const Triangle *triangle = triangleArray;

    // Identify all edges that have increasing vertex indices in CCW direction.
    for (int32 k = 0; k < triangleCount; k++)
    {
        uint16 i1 = triangle[k].index[2];
        for (int32 v = 0; v < 3; v++)
        {
            uint16 i2 = triangle[k].index[v];
            if (i1 < i2)
            {
                Edge *edge = &edgeArray[edgeCount];
                edge->vertexIndex[0] = i1; edge->vertexIndex[1] = i2;
                edge->faceIndex[0] = uint16(k); edge->faceIndex[1] = uint16(k);

                // Add the edge to the front of the list for the first vertex.
                nextEdge[edgeCount] = firstEdge[i1];
                firstEdge[i1] = edgeCount++;
            }

            i1 = i2;
        }
    }
```

```
    // Match all edges to the triangles for which they are wound clockwise.
    for (int32 k = 0; k < triangleCount; k++)
    {
        uint16 i1 = triangle[k].index[2];
        for (int32 v = 0; v < 3; v++)
        {
            uint16 i2 = triangle[k].index[v];
            if (i1 > i2)
            {
                for (uint16 e = firstEdge[i2]; e != 0xFFFF; e = nextEdge[e])
                {
                    Edge *edge = &edgeArray[e];
                    if ((edge->vertexIndex[1] == i1) &&
                        (edge->faceIndex[0] == edge->faceIndex[1]))
                    {
                        edge->faceIndex[1] = uint16(k);
                        break;
                    }
                }
            }

            i1 = i2;
        }
    }

    delete[] firstEdge;
    return (edgeCount);
}
```

Given the set of all edges belonging to a triangle mesh, a shadow volume is constructed by determining which subset of those edges constitute the silhouette with respect to a specific light position. We first classify each triangle as either facing toward the light or away from the light, and store the results in an array of boolean values. For a triangle having vertex positions v_0, v_1, and v_2, we can pre-compute the outward-pointing normal vector $\mathbf{n} = (v_1 - v_0) \times (v_2 - v_0)$ and store the plane $\mathbf{f} = [\mathbf{n} \,|\, -\mathbf{n} \cdot v_0]$ in an array. To classify the triangles in a mesh with respect to a 4D homogeneous light position $\mathbf{l}$ (where $l_w = 1$ for point lights, and $l_w = 0$ for infinite lights) we simply calculate $\mathbf{f} \cdot \mathbf{l}$ for each one. A triangle is facing toward the light source if $\mathbf{f} \cdot \mathbf{l} > 0$ and away from the light source otherwise. To determine the silhouette, we then visit each edge in the mesh and look up the classifications of the two triangles it references. The edge is part of the silhouette precisely when the triangles have different classifications.

To ensure that a silhouette extrusion always extends beyond any geometry that could receive shadows, the silhouette edges are extruded all the way to infinity. To accommodate this, scenes containing stencil-based shadows are always rendered

with an infinite projection matrix, either the conventional matrix $\mathbf{P}^*_{\text{infinite}}$ given by Equation (6.42) or the reversing matrix $\mathbf{R}^*_{\text{infinite}}$ given by Equation (6.47). The extrusion calculation itself is performed in the vertex shader.

The triangle mesh composing the silhouette extrusion is built on the CPU using an array of 4D vertex positions. The x, y, and z coordinates of each position are always equal to the exact object-space coordinates of one of the vertices a and b making up an edge on the silhouette. The w coordinate is set to one for vertices that remain on the silhouette and zero for vertices that will be extruded away from the light source. The vertex shader uses this w coordinate to decide what it's supposed to do with the vertex position.

For a point light, each silhouette edge becomes a four-sided polygon composed of two triangles in the extrusion mesh. Two of the vertices stay put at the positions a and b, and the other two are moved infinitely far away from the light position. However, for an infinite light, the extrusion of any vertex ends up at the same point at infinity opposite the direction toward the light, so each silhouette edge becomes a single triangle. The exact vertex positions created for both cases are shown in Table 8.1. All of the triangles composing the silhouette extrusion need to be consistently wound counterclockwise, but the direction from a to b could follow either winding order. If the first triangle referenced by the edge connecting a and b faces the light source, then the vertices must occur in the opposite order in the edge's extruded polygon for it to be wound counterclockwise. If the first triangle faces away from the light source, then the vertices stay in the same order. Table 8.1 accounts for these two possibilities.

First Triangle Facing	Point Light Extrusion	Infinite Light Extrusion
Toward light	$\mathbf{v}_0 = (b_x, b_y, b_z, 1)$ $\mathbf{v}_1 = (a_x, a_y, a_z, 1)$ $\mathbf{v}_2 = (a_x, a_y, a_z, 0)$ $\mathbf{v}_3 = (b_x, b_y, b_z, 0)$	$\mathbf{v}_0 = (b_x, b_y, b_z, 1)$ $\mathbf{v}_1 = (a_x, a_y, a_z, 1)$ $\mathbf{v}_2 = (0, 0, 0, 0)$
Away from light	$\mathbf{v}_0 = (a_x, a_y, a_z, 1)$ $\mathbf{v}_1 = (b_x, b_y, b_z, 1)$ $\mathbf{v}_2 = (b_x, b_y, b_z, 0)$ $\mathbf{v}_3 = (a_x, a_y, a_z, 0)$	$\mathbf{v}_0 = (a_x, a_y, a_z, 1)$ $\mathbf{v}_1 = (b_x, b_y, b_z, 1)$ $\mathbf{v}_2 = (0, 0, 0, 0)$

Table 8.1. For a silhouette edge connecting two vertices a and b, the extruded polygon has the object-space homogeneous vertex positions shown in this table. The order of the vertices depends on whether the first triangle referenced by the edge, for which the direction from a to b is winds counterclockwise, faces toward or away from the light source.

Given a homogeneous object-space light position $\mathbf{l}$, the vertex shader shown in Listing 8.12 performs the calculation

$$\mathbf{v}'_{xyz} = (1 - v_w)(l_w \mathbf{v}_{xyz} - \mathbf{l}_{xyz}) + v_w \mathbf{v}_{xyz}$$
$$v'_w = v_w \tag{8.94}$$

for each vertex $\mathbf{v}$ and transforms the result $\mathbf{v}'$ with the MVP matrix. This calculation has the effect of leaving any vertex with $v_w = 1$ where it is and extruding any vertex with $v_w = 0$ away from the light source. In the case of a point light, we have $l_w = 1$, and the extruded vertex position given by Equation (8.94) is $\mathbf{v}' = (\mathbf{v}_{xyz} - \mathbf{l}_{xyz} \mid 0)$. In the case of an infinite light, we have $l_w = 0$, and the extruded vertex position is $\mathbf{v}' = (-\mathbf{l}_{xyz} \mid 0)$, which is independent of the original vertex position.

Listing 8.12. This vertex shader code performs silhouette extrusion for both point lights and infinite lights. The homogeneous object-space position of the light source is specified in the uniform constant `lightPosition`. Vertices having a w coordinate of one are not moved, and vertices having a w coordinate of zero are extruded to infinity in the direction away from the light.

```
uniform float4  mvp[4];
uniform float4  lightPosition;

float4 ExtrudeSilhouette(float4 position)
{
    float3 v = lerp(position.xyz * lightPosition.w - lightPosition.xyz,
                    position.xyz, position.w);

    return (float4(dot(v, mvp[0].xyz) + position.w * mvp[0].w,
                   dot(v, mvp[1].xyz) + position.w * mvp[1].w,
                   dot(v, mvp[2].xyz) + position.w * mvp[2].w,
                   dot(v, mvp[3].xyz) + position.w * mvp[3].w));
}
```

When rendering a shadow volume using the Z-fail variant of the algorithm, we draw light and dark caps in addition to the silhouette edge extrusion. For both caps, we render with the unmodified 3D vertex positions that belong to the shadow castor's triangle mesh. All we need to do is build two index lists during the process of classifying triangles with respect to the light source. One index list references the vertices of all triangles facing toward the light source, and the other index list references the vertices of all triangles facing away from the light source. Since every triangle in the mesh is included in one list or the other, we can allocate a single

buffer large enough to hold every triangle in the mesh and then build the two lists from opposite ends so they meet in the middle. The light cap is rendered without any change being made to the vertex positions in the vertex shader. The dark cap is rendered with a vertex shader that extrudes every vertex position away from the light source, as demonstrated in Listing 8.13. The dark cap is rendered only for point lights, so we assume $l_w = 1$ and always calculate $\mathbf{v}' = (\mathbf{v}_{xyz} - \mathbf{l}_{xyz} \mid 0)$.

Figure 8.20(a) shows an example of a stencil shadow rendered for a character model illuminated by a point light source. Light is not added to the ground surface inside the shadow because that is where nonzero values were left in the stencil buffer after the shadow volume was rendered. The shadow volume itself is visualized as a wireframe in Figure 8.20(b), and it includes both the silhouette extrusion and the dark cap. The extruded vertices of the silhouette edges and the vertices belonging to the dark cap are rendered at infinity.

Listing 8.13. This vertex shader code extrudes vertices belonging to the dark cap of a shadow volume away from a point light. The object-space position of the light source is specified in the uniform constant `lightPosition`.

```
uniform float4   mvp[4];
uniform float4   lightPosition;

float4 ExtrudeDarkCap(float3 position)
{
    float3 v = position - lightPosition.xyz;
    return (float4(dot(v, mvp[0].xyz), dot(v, mvp[1].xyz),
                   dot(v, mvp[2].xyz), dot(v, mvp[3].xyz)));
}
```

8.4.4 Optimizations

One of the well-known drawbacks of the stencil shadow algorithm is its tendency to consume an enormous amount of triangle filling power. Any shadow volume can cover a large portion of the viewport depending on the spatial relationships among the object casting a shadow, the camera position, and the light position. This is taken to the extreme if the camera happens to be inside the shadow volume because in that case, the shadow volume surrounds the camera and fills the entire screen. Some geometrical shapes make the problem even worse because their shadow volumes consist of multiple extrusions that can be stacked up along the view direction. For example, consider the cages shown in Figure 8.11. Each vertical bar would be extruded away from the torch, which is a point light, and every pixel covered by
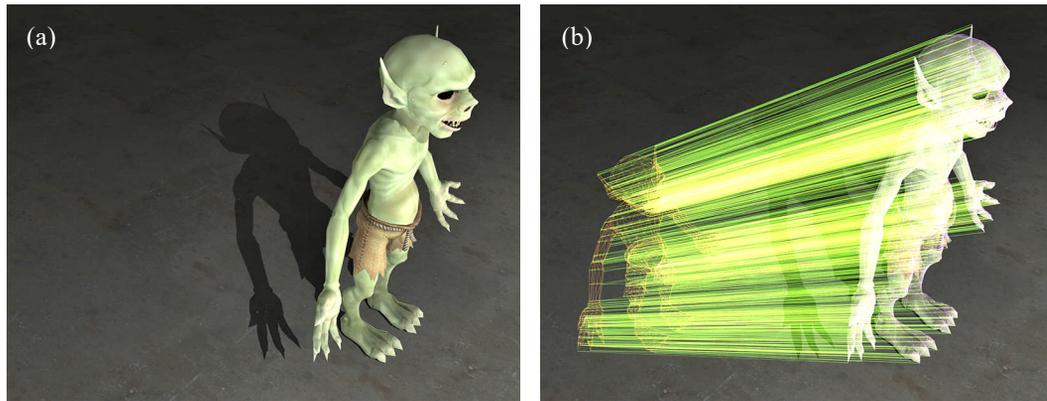
Figure 8.20. A stencil shadow is drawn for a character model illuminated by a point light source. (a) The shadow appears where nonzero values were left in the stencil buffer after the shadow volume was rendered. (b) A wireframe view of the shadow volume shows the extruded silhouette edges and the triangles belonging to the dark cap.

the shadow volume would be rendered twice *per bar*, once for front faces and once for back faces. Fortunately, very little work is being done at each pixel because no pixel shader is executed, the color buffer is not involved in any way, and nothing is written to the depth buffer. Today's GPUs are able blast through shadow volumes extremely quickly, but the rendering costs can add up in complex scenes, so we would still like to find ways to reduce the number of pixels that we have to process.

For point lights, the extent optimizations discussed in Section 8.2 provide highly effective ways to reduce the number of pixels covered by a shadow volume. Consider the character model illuminated by a point light near the upper-left corner of the image in Figure 8.21(a). The model's shadow volume would normally extend to the right and bottom edges of the viewport, but as shown by the wireframe view in Figure 8.21(b), the area actually rendered has been reduced a little by using the scissor rectangle for the light source. The size of the reduction can be very significant depending on the position and orientation of the camera, but the optimization works only when the camera is outside the light's radius.

A separate optimization makes use of the depth bounds test, which compares the depth already stored in the depth buffer against a given range. The depth bounds test was designed to work the way it does specifically to optimize shadow volume rendering. When the minimum and maximum depths are set to the projected range covered by the light source, the depth bounds test passes only for pixels corresponding to a surface that could receive shadows in that range. If, for any particular
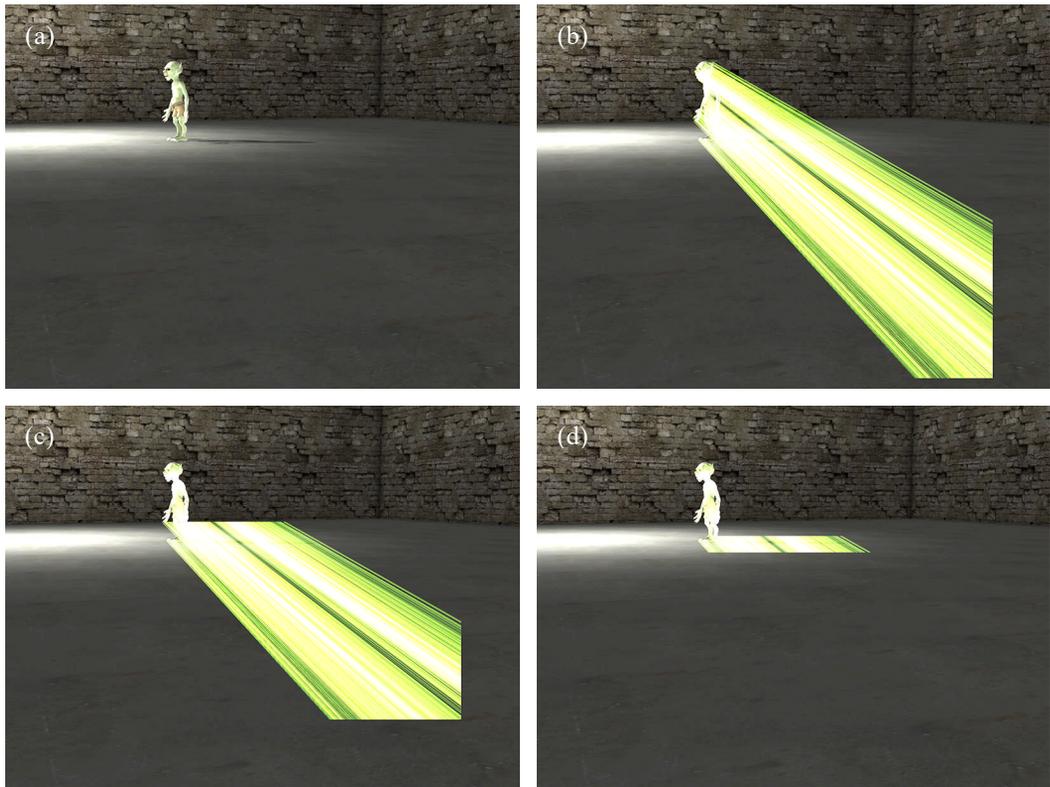
Figure 8.21. Stencil shadow optimizations can reduce the area rendered for a shadow volume by a significant amount. (a) A character model is illuminated by a point light near the upper-right corner of the image. (b) The model's shadow volume is drawn only inside the scissor rectangle calculated for the light source. (c) The depth bounds test has also been enabled, and it limits rendering to the range of depths covered by the light source. (d) A scissor rectangle and depth bounds that take the bounding volume of the shadow-casting geometry into account can reduce the filled area to a small fraction of its original size.

pixel, the existing surface is too far away or too close to the camera to be illuminated, then there's no reason to draw the shadow volume at that pixel because we already know that the net change to the stencil buffer will be zero. As shown in Figure 8.21(c), enabling the depth bounds test further reduces the area rendered for the shadow volume. In this case, the wall in the background is too far away to fall inside the depth bounds, and the floor at the bottom of the image is too close.

The depth bounds test provides a remarkably fast optimization because it can make use of a hierarchical depth buffer implemented by the GPU. Suppose that the

depth bounds have been set to $[z_{min}, z_{max}]$. If the minimum depth for an entire tile is greater than z_{max} or the maximum depth for an entire tile is less than z_{min}, then the whole tile can be rejected at once. This allows the GPU to quickly eliminate large portions of a shadow volume during the rasterization stage.

When they are based only on the radius of the light source, the scissor rectangle and depth bounds are often much larger than necessary, and this limits their effectiveness. The scissor rectangle and depth bounds shown in Figure 8.22(a) correspond to the extents of the light source as they would be calculated using the techniques described in Section 8.2, and we can always limit rendering to these ranges. However, as shown in Figure 8.22(b), it is frequently possible to do much better by also considering the bounding volume of the shadow-casting geometry. We can easily determine the silhouette of an object's bounding box, which contains at most six edges, and extrude it away from the light source to a plane lying at the maximum reach of the light's bounding sphere. This extrusion is a bounding volume for the shadow itself. If we transform each of its vertices into device space,
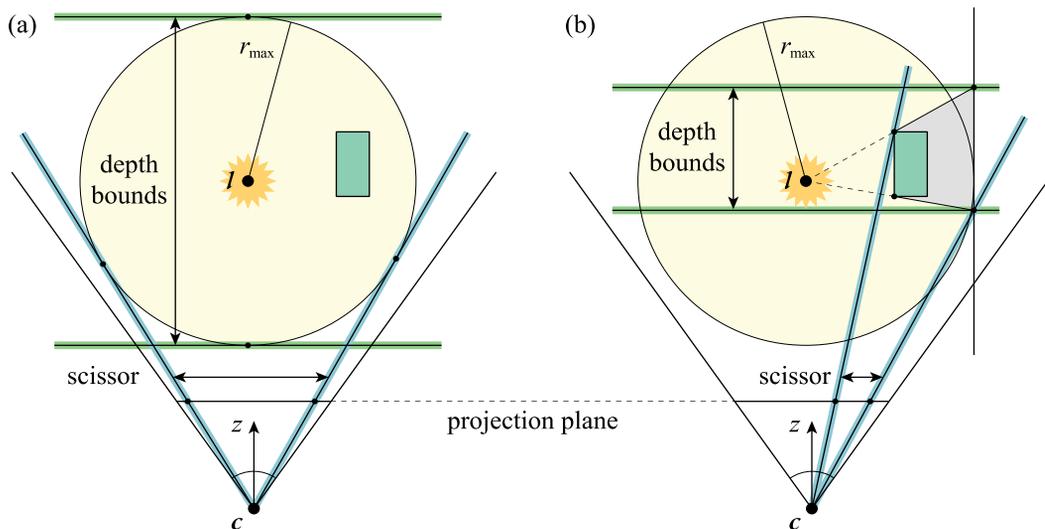
Figure 8.22. Extents for the scissor rectangle and depth bounds are calculated for the camera at the position c and the point light of radius r_{max} at the position l. Lines highlighted in blue correspond to the planes bounding the scissor rectangle, and lines highlighted in green represent the minimum and maximum depth bounds. (a) The extents are calculated based only on the bounding sphere of the light source. (b) The silhouette edges of the green bounding box surrounding a shadow-casting object are extruded away from the light source to a boundary plane. After being transformed into device space, the vertices of this extrusion provide a much tighter scissor rectangle and depth bounds.

then minimum and maximum values of the x, y, and z coordinates provide a scissor rectangle and depth bounds that are much tighter than those calculated for the light source. The reduced shadow volume shown in Figure 8.21(d) demonstrates that the results can be quite impressive, and this is a typical case. It's possible for the co-ordinate ranges calculated for the extruded bounding volume to extend outside the scissor rectangle and depth bounds calculated for the light source, so we still need the light-based extents so they can be used to clamp the geometry-based extents.

An infinite light has no scissor rectangle or depth bounds by itself, but it is still possible to calculate per-geometry extents for a shadow volume extruded away from an infinite light. Doing so requires that we have at least one plane that repre-sents a boundary for all shadow-receiving geometry. For example, in a large out-door world, a horizontal plane could be established at the lowest point on the ground. The silhouette edges of each object's bounding box can then be extruded to this plane, and the extents of the vertex coordinates in device space still provide an equally effective scissor rectangle and depth bounds.

8.5 Fog

The bulk of this chapter has dealt with the journey that a ray of light takes between its source and a point on the surface we are rendering. A light's color C_{light} and its intensity are transformed into a combined color and illuminance C_{illum}. The pixel shader then applies a reflection model to produce the combined color and lumi-nance C_{shaded} that are directed toward the camera from the surface. This section fo-cuses on the final transformation to a combined color and luminance C_{camera} that takes place after the light leaves the surface and before it reaches the camera, which is generally known as the application of *fog*. We consider the effects on the light due to its interaction with *participating media*, a term that collectively refers to the gas, liquid, or microscopic particles floating around in the open space through which the light is traveling.

8.5.1 Absorption and Scattering

The composition of light is affected by multiple phenomena as it travels through a participating medium, and these are illustrated in Figure 8.23. A beam of light is represented by a thin cylinder containing particles with which the light can interact. The light leaving a surface enters one end, and a different composition of light reaches the camera at the other end. Although the interactions are shown separately in the figure, it should be understood that they are mixed together, and each one occurs continuously from one end to the other. (In unusual cases, the particles themselves could also be emitting light, but we do not account for that here.)
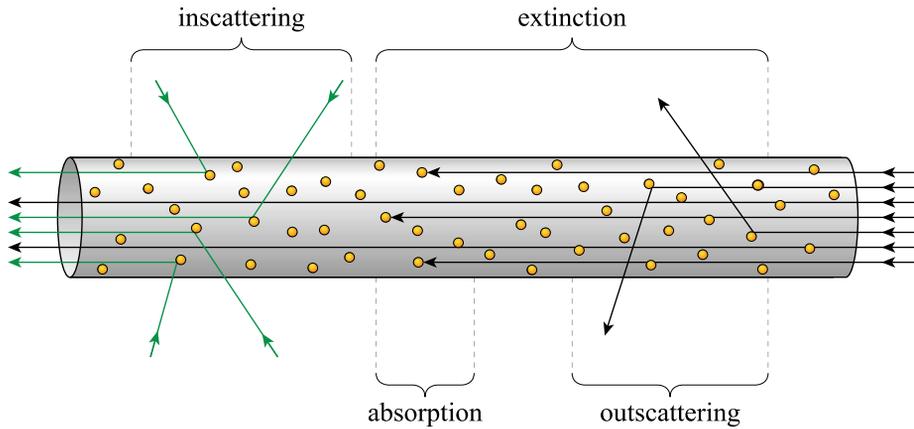
Figure 8.23. A long, thin cylinder represents a beam of light leaving a surface on the right end and reaching the camera on the left end. The participating medium is made up of randomly distributed particles illustrated as orange spheres. Some fraction of the original light, shown as black lines, is absorbed by the participating medium, and more of it is outscattered to other directions that do not reach the camera. Extinction refers to the combined effect of absorption and outscattering. Some light from environmental sources, shown as green lines, is added to the beam by inscattering, and this changes the color composition of the light reaching the camera.

Some of the light in the beam can be *absorbed* by the medium and turned into heat, and this causes the light's original brightness to be diminished. Some of the light can also be *scattered* when it collides with particles in the medium, and this allows the light to continue traveling, but in a different direction. When light no longer reaches the camera because it has been scattered to a different direction outside the beam, the effect is called *outscattering*, and it also causes the light's original brightness to be diminished. The decrease in brightness observed by the camera due to the combination of absorption and outscattering is commonly called *extinction* or *attenuation*.

Scattering by the medium can also cause light originating from environmental sources to be redirected into the beam toward the camera. This phenomenon is called *inscattering*, and it causes light of a different color to be added to the beam that originally left the surface. For example, inscattered light accounts for the color of fog or haze in the atmosphere as well as the color of murky water. As explained below, inscattered light is subject to the same absorption and outscattering effects as the original light in the beam, just over a shorter distance. Inscattered light is

generally dependent on the angle at which external light is incident upon the beam. In this section, however, we make the assumption that the source of inscattered light is isotropic. In more advanced settings, the fog calculations made here could be applied only to ambient light, and the angle-dependent inscattering contribution from a bright directional light source can be added by using the atmospheric shadowing technique presented in Section 10.6.

Absorption has a multiplicative effect over the distance d that a beam of light travels. Suppose that a beam has an original luminance L_0, and consider what happens when the beam travels through one unit of distance inside a uniform medium. The outgoing luminance of the beam is reduced to some fraction F of its incoming luminance, so after one unit of distance, the luminance has been reduced to $L_0 F$. If the beam now travels through one more unit of distance, then the same fraction F of the light makes it through, but we started with a luminance of $L_0 F$ this time, so the luminance after two units of distance has been reduced to $L_0 F^2$. Generalizing to any distance d, the outgoing luminance L is given by

$$L = L_0 F^d. \tag{8.95}$$

We can rewrite this equation in terms of the base of the natural logarithm e as

$$L = L_0 e^{d \ln F}. \tag{8.96}$$

Because F is a fraction less than one, the logarithm appearing in the exponent is negative. To create a quantity that provides a measure of how much a medium tends to absorb light, we define

$$\alpha_{ab} = -\ln F \tag{8.97}$$

and instead write Equation (8.96) as

$$L = L_0 \exp(-\alpha_{ab} d). \tag{8.98}$$

The constant α_{ab} is called the *absorption coefficient*, and it is a positive value having units of inverse distance. The minimum value of $\alpha_{ab} = 0$ means that a medium absorbs no light at all. There is no limit to how large the absorption coefficient can be, but practically speaking, the amount of light passing through a medium with a fairly high value of α_{ab} will be imperceptible.

Outscattering has the same kind of multiplicative effect as absorption. We therefore define another positive constant α_{sc} called the *scattering coefficient*. This constant also has units of inverse distance, and it provides a measure of how much light traveling in any direction is scattered by a medium. Outscattering and absorp-

tion reduce the luminance of the incoming light L_0 independently, and we can express their combined effect as

$$L = L_0 \exp\left(-(\alpha_{ab} + \alpha_{sc})d\right). \tag{8.99}$$

The *extinction coefficient* α_{ex} is defined as the sum

$$\alpha_{ex} = \alpha_{ab} + \alpha_{sc}, \tag{8.100}$$

and this allows us to write

$$\boxed{L = L_0 \exp(-\alpha_{ex}d).} \tag{8.101}$$

The extinction coefficient is often referred to as the *fog density* because it represents the number of particle interactions that are likely to take place per unit distance.

The *optical depth* or *optical thickness* τ along a path of length d through a medium having the extinction coefficient α_{ex} is defined as

$$\tau = \alpha_{ex}d, \tag{8.102}$$

and it has no physical units. The optical depth provides a direct measure of how much the luminance is attenuated over a given path, and it allows us to further simplify the form of Equation (8.101) to

$$\boxed{L = L_0 \exp(-\tau).} \tag{8.103}$$

When the extinction coefficient is not a constant throughout a medium, the optical depth becomes a function of the position on the surface being rendered. This is discussed for media having a linearly varying density below.

Unlike absorption and outscattering, inscattering is an additive effect. For each unit of distance along the beam, the same amount of external light is scattered toward the camera, and we simply add up the contributions over the beam's entire length. We express the quantity of inscattered light per unit distance as $k\alpha_{sc}$, where α_{sc} is the same scattering coefficient that we used for outscattering, and k represents the fraction of scattered light that is redirected toward the camera. Then, the amount of light inscattered along the full length d of the beam is simply $k\alpha_{sc}d$. As mentioned earlier, however, the inscattered light is also absorbed and outscattered by the medium in the same way that the light originating from the surface is according to Equation (8.101). Thus, to calculate the correct amount of inscattered light L_{in} that reaches the camera, we integrate the attenuated contributions from every differential length ds over the entire beam. This yields the quantity

$$L_{\text{in}} = L_{\text{ambient}} \int_0^d k\alpha_{\text{sc}} \exp(-\alpha_{\text{ex}}s)\,ds$$

$$= L_{\text{ambient}} \frac{k\alpha_{sc}}{\alpha_{ex}}[1 - \exp(-\alpha_{\text{ex}}d)], \tag{8.104}$$

where L_{ambient} represents the luminance of the ambient light incident on the beam.

Taking both extinction and inscattering effects into account, the final luminance L reaching the camera is given by

$$L = L_0 \exp(-\alpha_{\text{ex}}d) + L_{\text{ambient}} \frac{k\alpha_{\text{sc}}}{\alpha_{\text{ex}}}[1 - \exp(-\alpha_{\text{ex}}d)]. \tag{8.105}$$

The constant $k\alpha_{\text{sc}}/\alpha_{\text{ex}}$ and ambient luminance L_{ambient} are typically combined with an ambient light color to create a *fog color* C_{fog}, and the separate factors are never specified individually. The initial luminance L_0 is part of the color C_{shaded} output from the pixel shader. We can now express the color and brightness C_{camera} of the light reaching the camera after the application of fog as

$$C_{\text{camera}} = C_{\text{shaded}} f + C_{\text{fog}}(1 - f), \tag{8.106}$$

where f is the *fog factor* given by

$$f = \exp(-\alpha_{\text{ex}}d) = \exp(-\tau), \tag{8.107}$$

and d is the distance between the point p on the surface being rendered and the camera position c. To calculate d in a pixel shader, we simply take the magnitude of the unnormalized view vector $\mathbf{v} = c - p$ that is typically calculated by the vertex shader and interpolated. A simple implementation of Equation (8.106) is shown in Listing 8.14.

8.5.2 Halfspace Fog

There are many situations in which a nonconstant fog density is desirable in order to achieve effects such as layered fog in a mountain valley or increasing murkiness at greater water depths. The simplest nonconstant density is given by a linear function of distance along a single direction, and this gives rise to a boundary plane at which the density becomes zero. This is called *halfspace fog* because fog exists everywhere on one side of the plane, filling half of all space, but fog is completely

Listing 8.14. This pixel shader code applies fog using Equation (8.106). The distance d is given by the magnitude of the view vector v, which should be the unnormalized output of the vertex shader. The uniform constant fogDensity holds the extinction coefficient α_{ex}.

```
uniform float3      fogColor;
uniform float       fogDensity;

float3 ApplyFog(float3 shadedColor, float3 v)
{
    float f = exp(-fogDensity * length(v));
    return (lerp(fogColor, shadedColor, f));
}
```

absent from the other side where the density would be negative. An example scene containing halfspace fog is shown in Figure 8.24. Here, a horizontal fog plane has been placed near the top of a small canyon, and the fog density increases linearly with the vertical depth inside the canyon.

Let **f** be an arbitrary plane having a unit-length normal vector pointing out of the volume in which we want fog to be present. To create halfspace fog behind this plane, on its negative side, we define the extinction coefficient α_{ex} no longer as a constant, but as the function

$$\alpha_{ex}(q) = -\alpha_0(\mathbf{f} \cdot q), \tag{8.108}$$

for any point q in space. The negative sign accounts for the fact that $\mathbf{f} \cdot q < 0$ inside the fog. The constant α_0 corresponds to the reference fog density at one unit of distance behind the boundary plane, and it controls how quickly the fog gets thicker. Because the density increases without bound, an object that is deep enough inside the halfspace fog can easily become *fogged out*, which means that the fog factor is so close to zero that light originating from the object is too dim to appear on the display. This opens up culling opportunities that are discussed in Section 9.9.

In order to formulate an optical depth $\tau(p)$ as a function of the point p on the surface being rendered, we must integrate $\alpha_{ex}(q)$ along the path that light follows between p and the camera position c. But we have to be careful not to integrate over any part of the path that lies on the positive side of the fog plane **f**. Assuming for the moment that $\mathbf{f} \cdot p$ and $\mathbf{f} \cdot c$ are both negative, meaning that both p and c are inside the fog, we can express the integral for $\tau(p)$ as

$$\tau(p) = \int_p^c \alpha_{ex}(q)\, dq = -\alpha_0 \int_p^c (\mathbf{f} \cdot q)\, dq. \tag{8.109}$$

Figure 8.24. Halfspace fog is rendered with a linear density function in a canyon. (a) The camera is placed at a small distance above a horizontal fog plane. The fog becomes thicker as the depth of the geometry in the canyon increases. (b) The camera is placed well beneath the fog plane almost directly below its previous position. The fog appears thinner in directions that look upward because the density decreases in those directions, reaching zero at the fog plane.

The point q represents a position on the path connecting the points p and c, which can be expressed parametrically as

$$q(t) = p + t\mathbf{v}, \tag{8.110}$$

where $\mathbf{v}$ is again the unnormalized view vector given by $\mathbf{v} = c - p$. By substituting $q(t)$ for q in Equation (8.109), we can rewrite the integral as

$$\tau(p) = -\alpha_0 \int_0^1 \mathbf{f} \cdot (p + t\mathbf{v}) \|\mathbf{v}\| \, dt, \tag{8.111}$$

which also makes use of the relationship $dq = \|\mathbf{v}\| \, dt$. This integral is readily evaluated to obtain the formula

$$\boxed{\tau(p) = -\frac{\alpha_0}{2} \|\mathbf{v}\| (\mathbf{f} \cdot c + \mathbf{f} \cdot p).} \tag{8.112}$$

As illustrated in Figure 8.25, there are three separate cases in which fog is applied to at least part of the path connecting the points p and c. Equation (8.112) corresponds to the case shown in Figure 8.25(a), where the entire path lies inside the fogged halfspace. In the remaining two cases, where $\mathbf{f} \cdot p$ and $\mathbf{f} \cdot c$ have opposite signs, we must exclude the parts of the paths that lie on the positive side of the fog plane. The parameter t_0 at which the ray $q(t)$ intersects the fog plane is given by

$$t_0 = -\frac{\mathbf{f} \cdot p}{\mathbf{f} \cdot \mathbf{v}}, \tag{8.113}$$

and this replaces one of the limits of integration in Equation (8.111). In the case that only $\mathbf{f} \cdot p$ is negative, as shown in Figure 8.25(b), we have

$$\tau(p) = -\alpha_0 \int_0^{t_0} \mathbf{f} \cdot (p + t\mathbf{v}) \|\mathbf{v}\| \, dt$$
$$= \frac{\alpha_0}{2} \|\mathbf{v}\| \frac{(\mathbf{f} \cdot p)^2}{\mathbf{f} \cdot \mathbf{v}}. \tag{8.114}$$

In the case that only $\mathbf{f} \cdot c$ is negative, as shown in Figure 8.25(c), we have

$$\tau(p) = -\alpha_0 \int_{t_0}^1 \mathbf{f} \cdot (p + t\mathbf{v}) \|\mathbf{v}\| \, dt$$
$$= -\frac{\alpha_0}{2} \|\mathbf{v}\| \left(\mathbf{f} \cdot c + \mathbf{f} \cdot p + \frac{(\mathbf{f} \cdot p)^2}{\mathbf{f} \cdot \mathbf{v}} \right). \tag{8.115}$$
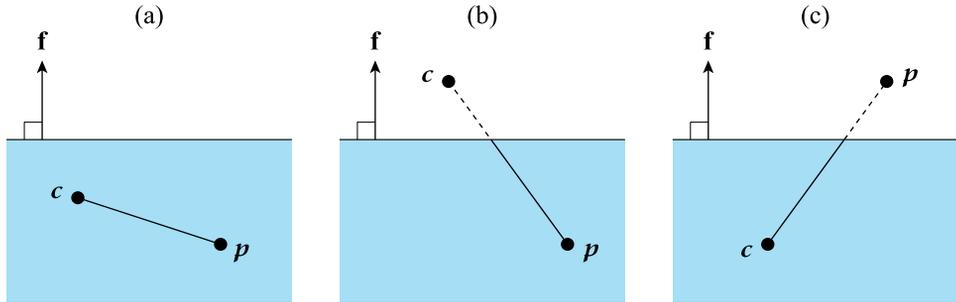
Figure 8.25. For a point p on the surface being rendered and the camera position c, there are three separate cases in which halfspace fog bounded by the plane $\mathbf{f}$ must be applied to at least part of the path connecting p and c.

The formulas for the optical depth $\tau(p)$ in all possible cases, including a fourth case in which neither $\mathbf{f} \cdot p$ nor $\mathbf{f} \cdot c$ is negative, are summarized in Table 8.2. These formulas share enough similarities that we can devise a single calculation to handle all four cases without conditional code in the pixel shader. First, we note that the quantity $\mathbf{f} \cdot c + \mathbf{f} \cdot p$ appears only when $\mathbf{f} \cdot c$ is negative, so we define a constant

$$m = \begin{cases} 1, & \text{if } \mathbf{f} \cdot c < 0; \\ 0, & \text{otherwise.} \end{cases} \tag{8.116}$$

We can also make use of the fact that $\mathbf{f} \cdot \mathbf{v}$ is always positive in case B and always negative in case C. By taking the absolute value of $\mathbf{f} \cdot \mathbf{v}$ and using the constant m, we can unify cases B and C into the single formula

$$\tau(p) = -\frac{\alpha_0}{2} \|\mathbf{v}\| \left[m(\mathbf{f} \cdot c + \mathbf{f} \cdot p) - \frac{(\mathbf{f} \cdot p)^2}{|\mathbf{f} \cdot \mathbf{v}|} \right]. \tag{8.117}$$

In order to incorporate case A into this formula, we need to eliminate the last term inside the brackets whenever $\mathbf{f} \cdot p$ and $\mathbf{f} \cdot c$ have the same sign. This can be accomplished by replacing $\mathbf{f} \cdot p$ with $\min((\mathbf{f} \cdot p)\operatorname{sgn}(\mathbf{f} \cdot c), 0)$ in the last term to arrive at the formula

$$\tau(p) = -\frac{\alpha_0}{2} \|\mathbf{v}\| \left[m(\mathbf{f} \cdot c + \mathbf{f} \cdot p) - \frac{[\min((\mathbf{f} \cdot p)\operatorname{sgn}(\mathbf{f} \cdot c), 0)]^2}{|\mathbf{f} \cdot \mathbf{v}| + \varepsilon} \right]. \tag{8.118}$$

Case	$\mathbf{f} \cdot \boldsymbol{p}$	$\mathbf{f} \cdot \boldsymbol{c}$	Optical Depth $\tau(\boldsymbol{p})$
A	negative	negative	$-\dfrac{\alpha_0}{2}\|\mathbf{v}\|(\mathbf{f} \cdot \boldsymbol{c} + \mathbf{f} \cdot \boldsymbol{p})$
B	negative	positive	$\dfrac{\alpha_0}{2}\|\mathbf{v}\|\dfrac{(\mathbf{f} \cdot \boldsymbol{p})^2}{\mathbf{f} \cdot \mathbf{v}}$
C	positive	negative	$-\dfrac{\alpha_0}{2}\|\mathbf{v}\|\left(\mathbf{f} \cdot \boldsymbol{c} + \mathbf{f} \cdot \boldsymbol{p} + \dfrac{(\mathbf{f} \cdot \boldsymbol{p})^2}{\mathbf{f} \cdot \mathbf{v}}\right)$
D	positive	positive	0

Table 8.2. The formulas for the optical depth $\tau(\boldsymbol{p})$ are listed for all four possible combined signs of $\mathbf{f} \cdot \boldsymbol{p}$ and $\mathbf{f} \cdot \boldsymbol{c}$. The similarities among these formulas allow us to devise a single calculation that can be implemented in a pixel shader to handle every possibility uniformly.

This formula also works for case D because both terms are eliminated when $\mathbf{f} \cdot \boldsymbol{p}$ and $\mathbf{f} \cdot \boldsymbol{c}$ are both positive. Equation (8.118) thus provides a single unified optical depth that is valid in all cases. We have added a small constant ε to the denominator of the last term to prevent a zero divided by zero operation in the event that $\mathbf{f} \cdot \mathbf{v} = 0$. Although it may look complicated, it is inexpensive to calculate because $\boldsymbol{p}$ and $\mathbf{v}$ are the only values that are not constants.

The pixel shader function shown in Listing 8.15 implements the optical depth calculation given by Equation (8.118) and applies halfspace fog to a shaded color. The fog plane $\mathbf{f}$ and camera position $\boldsymbol{c}$ are supplied to the vertex shader in object space along with the constants m, $\mathbf{f} \cdot \boldsymbol{c}$, and $\operatorname{sgn}(\mathbf{f} \cdot \boldsymbol{c})$. For each vertex position $\boldsymbol{p}$, the vertex shader calculates the vector $\mathbf{v}$, the scalar $\mathbf{f} \cdot \mathbf{v}$, and the two additional scalars

$$u_1 = m(\mathbf{f} \cdot \boldsymbol{c} + \mathbf{f} \cdot \boldsymbol{p})$$
$$u_2 = (\mathbf{f} \cdot \boldsymbol{p})\operatorname{sgn}(\mathbf{f} \cdot \boldsymbol{c}). \tag{8.119}$$

These four values are interpolated and used in what amounts to a rather short calculation in the pixel shader. The negative sign at the front of Equation (8.118) and the negative sign in the exponent of the fog factor cancel each other out, so neither appears in the shader.

Listing 8.15. This pixel shader code applies halfspace fog using Equation (8.118), where v is the unnormalized view vector, fv is the product $\mathbf{f} \cdot \mathbf{v}$, and the values u1 and u2 are given by Equation (8.119). The uniform constant fogDensity holds the reference fog density α_0.

```
uniform float3      fogColor;
uniform float       fogDensity;

float3 ApplyHalfspaceFog(float3 shadedColor, float3 v, float fv, float u1, float u2)
{
    const float kFogEpsilon = 0.0001;

    float x = min(u2, 0.0);
    float tau = 0.5 * fogDensity * length(v) * (u1 - x * x / (abs(fv) + kFogEpsilon));
    return (lerp(fogColor, shadedColor, exp(tau)));
}
```

Exercises for Chapter 8

1. Suppose the sun shines on the surface of the earth with an illuminance of exactly $E_V = 100$ klx. Assuming the sun radiates equally in all directions, calculate its luminous intensity in candelas. For the distance from the earth to the sun, use the value 1.5×10^{11} m, which is about one astronomical unit.

2. Suppose a point light with range r is located at the position l in camera space. Consider the extents of its bounding sphere on the projection plane as shown in Figure 8.7, and assume that $l_z > g + r$. Let $\mathbf{f}$ be a normalized plane passing through the origin that is parallel to the y axis and tangent to the bounding sphere. Calculate the points t of tangency where $\mathbf{f} \cdot t = r$, and show that the x coordinates of their projections onto the plane $z = g$ are equal to the values of q_x given by Equation (8.29).

3. Suppose that $\mathbf{P}$ is a conventional infinite projection matrix having a third row equal to $(0, 0, 1, -n)$. Show that the solutions to Equation (8.40), giving the depth bounds for a point light of radius r at the position l, simplify to

$$z_{\text{device}} = 1 - \frac{n}{l_z \pm r}.$$

4. Prove that the maximum diameter of a frustum representing a shadow cascade, as shown in Figure 8.13, is either the interior diagonal or the diagonal on the far plane.

5. Show that the cascade sample weights given by Equation (8.79) produce the correct linear blending for all possible values of u_1, u_2, and u_3.

6. Let a triangle have vertex indices (i, j, k). Prove that either one or two of the inequalities $i < j$, $j < k$, and $k < i$ are always true, but never zero or three.

7. Let $\mathbf{f}$ be the boundary plane for halfspace fog in which the extinction coefficient α_{ex} is given by Equation (8.108). Suppose that the camera position $\mathbf{c}$ is inside the fog such that $\mathbf{f} \cdot \mathbf{c} < 0$. Calculate the optical depth $\tau(\mathbf{d})$ along a path from the camera position to infinity in the direction $\mathbf{d}$. Assume that $\mathbf{d}$ points out of the fog such that $\mathbf{f} \cdot \mathbf{d} > 0$.

Chapter 9

Visibility and Occlusion

GPUs are able to quickly decide that rasterization can be skipped for triangles that lie completely outside the viewport, but this decision can be made only after the vertex shader has run. When all of the triangles making up an object happen to be off screen, a significant amount of time can be wasted processing a mesh that ends up not being drawn at all. If we simply rendered every object in the world during each frame, then we would run into limits on the amount of geometry a world could contain very quickly. In order to support large and complex environments, game engines have several layers of culling functionality designed to efficiently determine what parts of the world can be seen from the current camera position.

This chapter discusses a range of techniques commonly used to perform visibility and occlusion operations at various granularities on the CPU. The overall performance of a game engine is often determined by the quality and depth of its culling code, and that makes these topics extremely important. As higher-level visibility structures are implemented, an engine generally gains the ability to handle much larger worlds. In addition to identifying the objects that are visible to the camera, these structures can be used to identify what objects are illuminated by a light source or need to have shadows rendered.

9.1 Polygon Clipping

Polygon clipping is used in such a wide variety of situations that it can be considered a fundamental operation in computer graphics. We introduce it here because we will need it for specific purposes later in this chapter and in Chapter 10. Given a flat polygon defined by three or more vertices and an arbitrary plane in 3D space, *polygon clipping* is the process of cutting off any portion of the polygon that lies

on the negative side of the plane. There are three possible outcomes to this process depending on the geometric relationship between the polygon and the plane. If all of the vertices lie on the positive side of the plane, then no change is made to the polygon. If all of the vertices lie on the negative side of the plane, then the polygon is completely clipped away, leaving nothing behind. Otherwise, the plane intersects the polygon, and only the portion lying on the positive side of the plane remains after the clipping operation. In this last case, new vertices are calculated where edges of the original polygon cross the clipping plane.

We limit our discussion to convex polygons because they are sufficient for all situations in which clipping is applied in this book. A *convex* polygon is one having no interior angle greater than 180 degrees. Equivalently, a convex polygon is one such that for any two points p and q inside the polygon, every point on the line segment connecting p and q is also inside the polygon. These definitions admit polygons having three or more collinear vertices, and while such polygons are best avoided, we don't disallow them because the most harm they can usually do is create a computational redundancy.

Let $p_0, p_1, \ldots, p_{n-1}$ be the vertices of an n-sided convex polygon, and let $\mathbf{k}$ be a clipping plane that is normalized with $k_{xyz}^2 = 1$. The vertices must be stored in the proper order so that each edge has the consecutive endpoints p_i and $p_{(i+1)\bmod n}$, where the modulo operation simply means that one of the edges has the endpoints p_{n-1} and p_0. We use the convention that the vertices are wound in counterclockwise order when the polygon is viewed from the front side of its own plane, as determined by the direction of its normal vector. For each vertex p_i, we calculate the distance d_i to the clipping plane with

$$d_i = \mathbf{k} \cdot p_i, \tag{9.1}$$

and this allows us to classify each vertex as lying on the positive side of the plane, on the negative side of the plane, or in the plane itself. In practice, we consider any vertex for which $|d_i| \leq \varepsilon$ to be lying in the plane, where ε is a small distance such as 0.001. This helps us deal with floating-point round-off error, and it prevents any edge in the clipped polygon from having a length less than ε. We can thus think of the clipping plane as having a small thickness 2ε, as shown in Figure 9.1.

A polygon is clipped by visiting each vertex p_i, in order, and examining the distance d_i together with the distance $d_{(i-1)\bmod n}$ of its predecessor to determine the state of each edge. There are three possible cases that must be handled in different ways. First, whenever $d_i \geq -\varepsilon$ and $d_{(i-1)\bmod n} \geq -\varepsilon$, indicating that neither vertex lies on the negative side of the clipping plane, the unmodified edge becomes part of the output polygon. This is the case for the edges (p_0, p_1) and (p_4, p_0) in Figure 9.1.
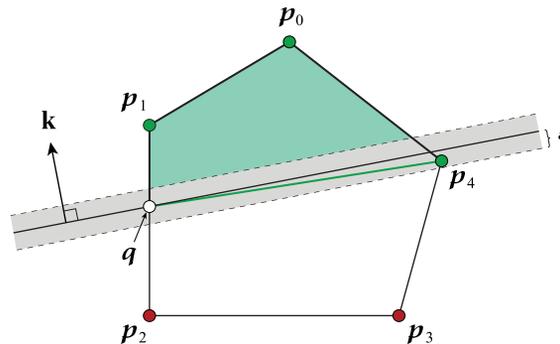
Figure 9.1. A 5-sided convex polygon defined by vertices p_0 through p_4 is clipped by a plane **k**. The vertices p_0 and p_1 lie on the positive side of the plane. The vertices p_2 and p_3 lie on the negative side of the plane. The vertex p_4 is considered to be lying in the plane because it falls within a distance ε of the plane. Vertices p_0, p_1, and p_4, colored green, become part of the clipped polygon. Vertices p_2 and p_3, colored red, are discarded. A new vertex q, colored white, is calculated where the edge (p_1, p_2) intersects the plane. The vertices composing the final clipped polygon are p_0, p_1, q, and p_4.

Second, whenever $d_i \leq \varepsilon$ and $d_{(i-1) \bmod n} \leq \varepsilon$, indicating that neither vertex lies on the positive side of the clipping plane, the edge is removed in its entirety. This is the case for the edges (p_2, p_3) and (p_3, p_4) in the figure. (This is also the case if both vertices lie in the plane, but a new identical edge is implicitly formed by the endpoints of the adjacent edges that are not removed.) Finally, whenever one of the distances is greater than ε, and the other distance is less than $-\varepsilon$, we calculate the position of a new vertex where the edge crosses the clipping plane, and the edge connecting the original vertex on the positive side of the plane to the new vertex on the plane becomes part of the output polygon. This is the case for edge (p_1, p_2) in the figure. A polygon has at most two such edges that require a new vertex position to be calculated, one where a transition is made from the positive side of the plane to the negative side and another where a transition is made back to the positive side. The maximum number of vertices that the output polygon could have is $n + 1$. This occurs when none of the original vertices lie in the clipping plane and exactly one of the original vertices lies on the negative side.

When an edge needs to be clipped, we calculate the intersection of the parametric line $\mathcal{L}(t) = p_{\text{pos}} + t(p_{\text{neg}} - p_{\text{pos}})$ and the plane **k** in the manner described in Section 3.4.4. The vertex p_{pos} refers to the endpoint that lies on the positive side of the plane at the distance d_{pos}, and the vertex p_{neg} refers to the endpoint that lies on the negative side of the plane at the distance d_{neg}. We intentionally label the

endpoints this way instead of keeping them in polygon order because it's important that we perform the intersection calculation using a consistent line direction with respect to the plane. In Section 10.1, this will be necessary when clipping adjacent polygons in a triangle mesh in order to avoid seams that could occur due to floating-point round-off error.

Since we have already calculated $d_{pos} = \mathbf{k} \cdot \boldsymbol{p}_{pos}$ and $d_{neg} = \mathbf{k} \cdot \boldsymbol{p}_{neg}$ during the classification step, the parameter t at which the line $\mathcal{L}(t)$ intersects the clipping plane can be expressed as

$$t = \frac{d_{pos}}{d_{pos} - d_{neg}}. \tag{9.2}$$

The new vertex $\boldsymbol{q}$ on the clipping plane is then given by

$$\boxed{\boldsymbol{q} = (1-t)\,\boldsymbol{p}_{pos} + t\,\boldsymbol{p}_{neg}.} \tag{9.3}$$

The vertex $\boldsymbol{q}$ is inserted into the output polygon following the vertex $\boldsymbol{p}_{pos}$. The vertex following $\boldsymbol{q}$ in the output polygon is either an original vertex lying in the plane or another vertex calculated with Equation (9.3).

The ClipPolygon() function shown in Listing 9.1 clips a convex polygon against a given plane. Because information about the maximum number of vertices that could compose an input polygon resides with the caller, it is the caller's responsibility to provide storage space for the output polygon as well as an array used to hold the distances between the original vertices and the clipping plane. This allows the function to be applied to polygons having an unbounded number of vertices without allocating its own temporary storage.

Listing 9.1. This function clips a convex polygon specified by vertexCount and the points stored in the vertex array against the plane given by the plane parameter. The location array must point to storage for at least vertexCount distance values, and the result array must point to storage for at least one greater than vertexCount points. The return value is the number of vertices in the output polygon that were written to the result array.

```
int32 ClipPolygon(int32 vertexCount, const Point3D *vertex,
                  const Plane& plane, float *location, Point3D *result)
{
    const float kPolygonEpsilon = 0.001F;
    int32 positiveCount = 0, negativeCount = 0;

    // Calculate the signed distance to plane for all vertices.
    for (int32 a = 0; a < vertexCount; a++)
```

```
    {
        float d = Dot(plane, vertex[a]);
        location[a] = d;

        if (d > kPolygonEpsilon) positiveCount++;
        else if (d < -kPolygonEpsilon) negativeCount++;
    }

    if (negativeCount == 0)
    {
        // No vertices on negative side of plane. Copy original polygon to result.
        for (int32 a = 0; a < vertexCount; a++) result[a] = vertex[a];
        return (vertexCount);
    }
    else if (positiveCount == 0) return (0);   // No vertices on positive side of plane.

    // Loop through all edges, starting with edge from last vertex to first vertex.
    int32 resultCount = 0;
    const Point3D *p1 = &vertex[vertexCount - 1]; float d1 = location[vertexCount - 1];
    for (int32 index = 0; index < vertexCount; index++)
    {
        const Point3D *p2 = &vertex[index]; float d2 = location[index];
        if (d2 < -kPolygonEpsilon)
        {
            // Current vertex is on negative side of plane.
            if (d1 > kPolygonEpsilon)
            {
                // Preceding vertex is on positive side of plane.
                float t = d1 / (d1 - d2);
                result[resultCount++] = *p1 * (1.0F - t) + *p2 * t;
            }
        }
        else
        {
            // Current vertex is on positive side of plane or in plane.
            if ((d2 > kPolygonEpsilon) && (d1 < -kPolygonEpsilon))
            {
                // Current vertex on positive side, and preceding vertex on negative side.
                float t = d2 / (d2 - d1);
                result[resultCount++] = *p2 * (1.0F - t) + *p1 * t;
            }

            result[resultCount++] = *p2;
        }

        p1 = p2; d1 = d2;
    }

    return (resultCount);
}
```

9.2 Polyhedron Clipping

Later in this chapter, we will construct convex polyhedra representing regions of space that can be seen by the camera or directly illuminated by a light source, and it will be necessary to clip these polyhedra against arbitrary planes. We can think of a convex polyhedron as a collection of faces that all share a set of vertices and edges satisfying the Euler formula $V - E + F = 2$, where V is the number of vertices, E is the number of edges, and F is the number of faces. By associating an inward-pointing plane with each face, we can define the interior of the polyhedron as the set of points lying on the positive side of every plane at once. When a clipping plane intersects the polyhedron, as shown in Figure 9.2, a new face is created in the plane, and the clipping plane itself is added to the set of planes bounding the polyhedron's volume.

As with polygon clipping, our method of polyhedron clipping centers around the identification of edges that cross the clipping plane and the calculation of new vertices at the points of intersection. However, we now also require a significant
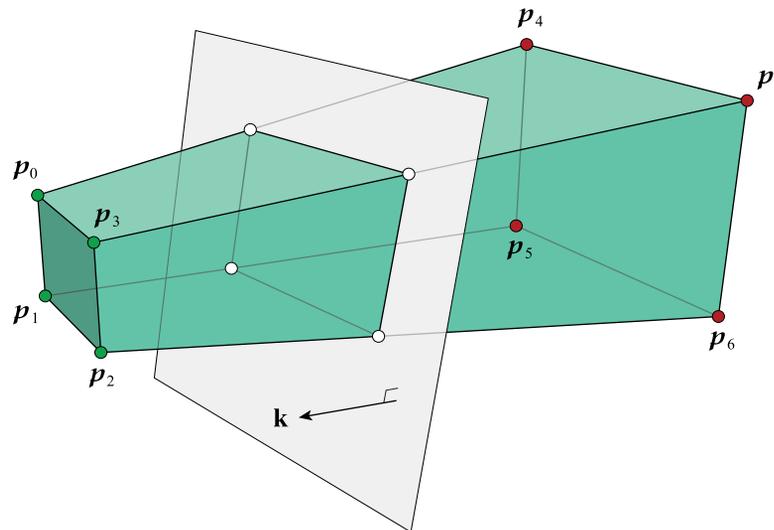
Figure 9.2. A 6-sided convex polyhedron having vertices p_0 through p_7 is clipped by a plane $\mathbf{k}$. The vertices p_0, p_1, p_2, and p_3, colored green, lie on the positive side of the plane and become part of the clipped polyhedron. The rest of the vertices, colored red, lie on the negative side of the plane and are discarded. A new face is created in the clipping plane, and its vertices, colored white, are placed at the locations where edges such as (p_0, p_4) intersect the plane.

amount of bookkeeping code that maintains face connectivity information. In a manner very much like the stencil shadow algorithm described in Section 8.4, techniques discussed later in this chapter will need to find the silhouettes of polyhedra with respect to the position of a light source. Thus, each edge must carry information about the two faces it joins together in addition to the two vertices at its endpoints. Each face is defined by the set of edges that make up its boundary.

The data structures that we use to store a polyhedron are shown in Listing 9.2. The `Polyhedron` structure contains an array of vertex positions numbered 0 to $V - 1$, an array of `Edge` structures numbered 0 to $E - 1$, an array of `Face` structures

Listing 9.2. All of the geometric information defining a convex polyhedron is stored in the `Polyhedron` data structure. All of the vertices, edges, and faces, up to the limits defined at the beginning of the code, are stored in the `vertex`, `edge`, and `face` arrays in any order. The plane containing each face is stored at the same offset in the `plane` array. The `Face` structure contains an array of edge indices, also in any order, referencing entries in the `edge` array of the `Polyhedron` structure. The `Edge` structure contains two vertex indices referencing entries in the `vertex` array and two face indices referencing entries in the `face` array of the `Polyhedron` structure. The face indices must be ordered such that the edge's vertices are wound counterclockwise for the first face when the polyhedron is viewed from its exterior.

```cpp
constexpr int32 kMaxPolyhedronVertexCount   = 28;
constexpr int32 kMaxPolyhedronFaceCount     = 16;
constexpr int32 kMaxPolyhedronEdgeCount     = (kMaxPolyhedronFaceCount - 2) * 3;
constexpr int32 kMaxPolyhedronFaceEdgeCount = kMaxPolyhedronFaceCount - 1;

struct Edge
{
    uint8       vertexIndex[2];
    uint8       faceIndex[2];
};

struct Face
{
    uint8       edgeCount;
    uint8       edgeIndex[kMaxPolyhedronFaceEdgeCount];
};

struct Polyhedron
{
    uint8       vertexCount, edgeCount, faceCount;
    Point3D     vertex[kMaxPolyhedronVertexCount];
    Edge        edge[kMaxPolyhedronEdgeCount];
    Face        face[kMaxPolyhedronFaceCount];
    Plane       plane[kMaxPolyhedronFaceCount];
};
```

numbered 0 to $F-1$, and an array of planes for which each entry corresponds to the face with the same index. Each `Edge` structure stores the indices of its two vertex positions and the indices of the two faces that share it. (Note that the indices in the `Edge` structure we use here are only 8 bits wide, where they were 16 bits wide in Listing 8.10.) The index of the first face must be the one for which the edge's vertices are wound counterclockwise when viewed from the exterior of the polyhedron, as was also the case in Section 8.4. Each `Face` structure stores the number of edges belonging to a face and an array of indices referencing that many specific `Edge` structures. The indices of adjacent edges belonging to a face do not need to be stored consecutively in the `Face` structure. Edges for each face can be referenced in any order as long as the full set forms a complete convex polygon.

Our algorithm begins by classifying all of the vertices belonging to a polyhedron with respect to the clipping plane $\mathbf{k}$. As with polygon clipping, we think of the clipping plane as having some thickness such that any points within a small distance ε of the plane are considered to be lying in the plane. For each vertex $\boldsymbol{p}_i$, we again calculate the distance d_i to the clipping plane with $d_i = \mathbf{k} \cdot \boldsymbol{p}_i$, but we now also assign an integer code c_i to the vertex that will be used to quickly classify edges. The code is defined as

$$c_i = \begin{cases} 0, & \text{if } d_i \leq -\varepsilon; \\ 1, & \text{if } -\varepsilon \geq d_i \geq \varepsilon; \\ 3, & \text{if } d_i > \varepsilon. \end{cases} \qquad (9.4)$$

That is, any vertex on the negative side of the plane is assigned code 0, any vertex lying in the plane is assigned code 1, and any vertex on the positive side of the plane is assigned code 3.

The values of the vertex codes are chosen so that the sum of any pair of codes is unique. This lets us calculate a meaningful classification code for each edge in the polyhedron by simply adding the codes assigned to its two endpoints. The following six outcomes are possible, and they are illustrated in Figure 9.3.

- **Codes 0 and 1.** These edges have at least one endpoint on the negative side of the clipping plane and no endpoint on the positive side. They cannot be part of the output polyhedron and are therefore deleted.
- **Code 2.** These edges lie completely in the clipping plane and become part of the result as long as the entire polyhedron is not clipped away.
- **Code 3.** These edges have endpoints on opposite sides of the clipping plane, and they are the only edges that need to be modified. The original vertex on the positive side of the plane and the newly created vertex where the edge intersects the plane define a new edge in the output polyhedron.
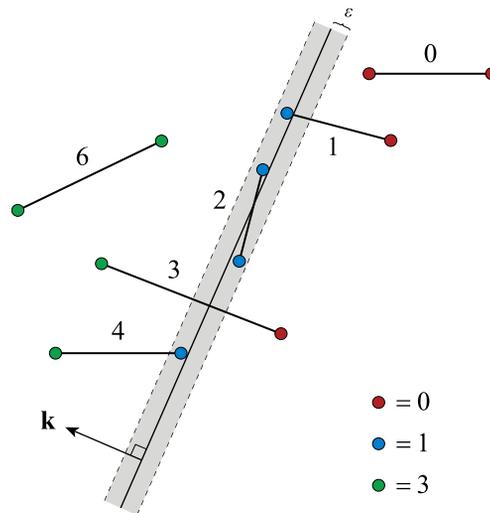
Figure 9.3. Each edge belonging to a polyhedron is classified into one of six possible types based on the locations of its endpoints. The classification code assigned to each edge is the sum of the classification codes assigned to its two vertices. A vertex is assigned code 0 (red) if it lies on the negative side of the clipping plane, code 1 (blue) if it lies within a distance ε of the clipping plane, and code 3 (green) if it lies on the positive side of the clipping plane. The six possible sums of these codes are 0, 1, 2, 3, 4, and 6.

- ■ **Codes 4 and 6.** These edges have at least one endpoint on the positive side of the clipping plane and no endpoint on the negative side. They are always part of the output polyhedron without modification.

The `ClipPolyhedron()` function shown in Listing 9.3 performs the classification step. After classification codes are determined for all of the vertices, a quick check of the minimum and maximum code values tells us whether the polyhedron is either unclipped or completely clipped away. If there are no vertices on the negative side of the clipping plane with code 0, then the input polyhedron is returned unchanged. If there are no vertices on the positive side of the clipping plane with code 3, then the entire polyhedron must be clipped away, and no polyhedron is returned. Otherwise, some kind of clipping will take place, and the function proceeds to calculate the edge classification codes.

The function then examines each face of the original polyhedron to determine which faces have at least one vertex on the positive side of the clipping plane by looking for edges with a classification code of 3 or higher. Each such face becomes

part of the output polyhedron after possible clipping. The surviving faces are re-numbered so they are stored contiguously, and the mapping from the original face indices to the new face indices is stored in the `faceRemap` array. The maximum 8-bit value of `0xFF` is used in this array to indicate that the original face is completely clipped away and thus will not be part of the result.

Listing 9.3. This function clips the polyhedron specified by the `polyhedron` parameter against the plane given by the `plane` parameter and returns a clipped polyhedron through the `result` parameter. The return value is `true` if the output polyhedron is nonempty and `false` if it is completely clipped away. The portion of the function shown in this listing classifies the vertices and edges of the poly-hedron and then determines which faces become part of the result. (Continued in Listing 9.4.)

```cpp
bool ClipPolyhedron(const Polyhedron *polyhedron, const Plane& plane, Polyhedron *result)
{
    float    vertexLocation[kMaxPolyhedronVertexCount];
    int8     vertexCode[kMaxPolyhedronVertexCount];
    int8     edgeCode[kMaxPolyhedronEdgeCount];
    uint8    vertexRemap[kMaxPolyhedronVertexCount];
    uint8    edgeRemap[kMaxPolyhedronEdgeCount];
    uint8    faceRemap[kMaxPolyhedronFaceCount];
    uint8    planeEdgeTable[kMaxPolyhedronFaceEdgeCount];

    const float kPolyhedronEpsilon = 0.001F;
    int32 minCode = 6, maxCode = 0;

    // Classify vertices.
    uint32 vertexCount = polyhedron->vertexCount;
    for (uint32 a = 0; a < vertexCount; a++)
    {
        vertexRemap[a] = 0xFF;
        float d = Dot(plane, polyhedron->vertex[a]);
        vertexLocation[a] = d;

        int8 code = (d > -kPolyhedronEpsilon) + (d > kPolyhedronEpsilon) * 2;
        minCode = min(minCode, code); maxCode = max(maxCode, code);
        vertexCode[a] = code;
    }

    if (minCode != 0)
    {
        *result = *polyhedron;     // No vertices on negative side of clip plane.
        return (true);
    }

    if (maxCode <= 1) return (false);   // No vertices on positive side of clip plane.

    // Classify edges.
```

```
    uint32 edgeCount = polyhedron->edgeCount;
    for (uint32 a = 0; a < edgeCount; a++)
    {
        edgeRemap[a] = 0xFF;
        const Edge *edge = &polyhedron->edge[a];
        edgeCode[a] = int8(vertexCode[edge->vertexIndex[0]]
                            + vertexCode[edge->vertexIndex[1]]);
    }

    // Determine which faces will be in result.
    uint32 resultFaceCount = 0;
    uint32 faceCount = polyhedron->faceCount;
    for (uint32 a = 0; a < faceCount; a++)
    {
        faceRemap[a] = 0xFF;
        const Face *face = &polyhedron->face[a];
        uint32 faceEdgeCount = face->edgeCount;
        for (uint32 b = 0; b < faceEdgeCount; b++)
        {
            if (edgeCode[face->edgeIndex[b]] >= 3)
            {
                // Face has a vertex on the positive side of the plane.
                result->plane[resultFaceCount] = polyhedron->plane[a];
                faceRemap[a] = uint8(resultFaceCount++);
                break;
            }
        }
    }
}
```

The next major step in the ClipPolyhedron() function, shown in Listing 9.4, determines which edges belonging to the original polyhedron become part of the output polyhedron and clips those edges that cross the clipping plane. Only edges having a classification code of 3 are clipped, and the position of the new vertex is calculated using Equations (9.2) and (9.3). During this process, vertices and edges are renumbered, and mappings from the original vertex and edge indices to the new vertex and edge indices are stored in the vertexRemap and edgeRemap arrays. As before, the value 0xFF indicates that an original vertex or edge does not become part of the result. At this point, the function has determined the entire set of vertices belonging to the output polyhedron, and this includes original vertices that do not lie on the negative side of the clipping plane and new vertices that were created where original edges intersect the clipping plane. The function has also determined which of the original edges and faces will not be completely clipped away. Later, one new face and possibly several new edges will be created in the clipping plane to close the hole made by the clipping process.

Listing 9.4. This piece of the `ClipPolyhedron()` function loops over the edges in the original polyhedron and identifies those that will become part of the result. Edges having a classification code of 0 or 1 are eliminated. Edges having a classification code of 3 are clipped to produce new vertices where they intersect the clipping plane. (Continued in Listing 9.5.)

```
uint32 resultVertexCount = 0, resultEdgeCount = 0;
for (uint32 a = 0; a < edgeCount; a++)
{
    if (edgeCode[a] >= 2)
    {
        // The edge is not completely clipped away.
        const Edge *edge = &polyhedron->edge[a];
        Edge *resultEdge = &result->edge[resultEdgeCount];
        edgeRemap[a] = uint8(resultEdgeCount++);

        resultEdge->faceIndex[0] = faceRemap[edge->faceIndex[0]];
        resultEdge->faceIndex[1] = faceRemap[edge->faceIndex[1]];

        // Loop over both vertices of edge.
        for (machine i = 0; i < 2; i++)
        {
            uint8 vertexIndex = edge->vertexIndex[i];
            if (vertexCode[vertexIndex] != 0)
            {
                // This vertex on positive side of plane or in plane.
                uint8 remappedVertexIndex = vertexRemap[vertexIndex];
                if (remappedVertexIndex == 0xFF)
                {
                    remappedVertexIndex = resultVertexCount++;
                    vertexRemap[vertexIndex] = remappedVertexIndex;
                    result->vertex[remappedVertexIndex] = polyhedron->vertex[vertexIndex];
                }

                resultEdge->vertexIndex[i] = remappedVertexIndex;
            }
            else
            {
                // This vertex on negative side, and other vertex on positive side.
                uint8 otherVertexIndex = edge->vertexIndex[1 - i];
                const Point3D& p1 = polyhedron->vertex[vertexIndex];
                const Point3D& p2 = polyhedron->vertex[otherVertexIndex];
                float d1 = vertexLocation[vertexIndex];
                float d2 = vertexLocation[otherVertexIndex];
                float t = d2 / (d2 - d1);
                result->vertex[resultVertexCount] = p2 * (1.0F - t) + p1 * t;
                resultEdge->vertexIndex[i] = uint8(resultVertexCount++);
            }
        }
    }
}
```

The step that remains, shown in Listing 9.5, is to assign edges to every face and construct a new face in the clipping plane itself. The vertices belonging to this new face are any original vertices that lie in the clipping plane and all new vertices that were created by clipping edges with a classification code of 3. The new face incorporates any original edges for which both endpoints lie in the clipping plane, thus having a classification code of 2. Each original face having any clipped edges contributes one new edge to the new face, and these are created in arbitrary order as the code visits each of the original faces.

The code in Listing 9.5 loops over all of the original faces of the polyhedron. For each face that is not completely clipped away, the code loops over the edges forming the boundary of that face. Any original edge having a classification code that is nonzero and even (code 2, 4, or 6) does not cross onto the negative side of the clipping plane at any point, so it is added to the boundary of the corresponding face in the output polyhedron. In the case that the classification code is 2, the edge is also added to the new face in the clipping plane.

Any original edge having a classification code that is odd (code 1 or 3) triggers the creation of a new edge in the clipping plane. There must be either zero or two original edges having an odd classification code. A new edge is created only when the first one is found, but information is added to the new edge in both instances when an original edge having an odd classification code is encountered. A new edge is always wound counterclockwise with respect to the original face, so following the convention we have established, the new edge's first face index is the renumbered index of the original face. For each original edge, we need to determine whether the endpoint lying in the clipping plane (for code 1) or the new vertex where the edge crosses the clipping plane (for code 3) becomes the first or second vertex for the new edge. If we consider the entire boundary of the original face to be a counterclockwise loop as required locally for the new edge, then the first vertex corresponds to the location where the boundary enters the negative side of the clipping plane and the second vertex corresponds to the location where the boundary returns from the negative side. An original edge contributes the first vertex of the new edge if (a) it is wound counterclockwise for the original face and its first vertex is not on the negative side of the plane or (b) it is wound clockwise for the original face and its first vertex is on the negative side of the plane. An original edge contributes the second vertex of the new edge if neither of those pairs of conditions are satisfied. We always insert the new edge into the new face at the same time that the first vertex is contributed to the new edge.

After a polyhedron has been clipped by several planes, it is possible for floating-point round-off error to have accumulated to the point that some mathematically valid assumptions are no longer true. This can happen if the clipping plane is

nearly coincident with a face for which the vertices are not coplanar within an acceptable margin of error. The result is often that one of the original faces has exactly one edge with an odd classification code, making it impossible to construct the new edge in the clipping plane. When this case is detected in the final lines of Listing 9.5, the code simply copies the original polyhedron to the result because doing so is the safest option for the visibility regions used later in this chapter.

All original edges lying in the clipping plane, as well as any new edges created on clipped faces, are added to the `planeEdgeTable` array in Listing 9.5. The final piece of the `ClipPolyhedron()` function, shown in Listing 9.6, assembles these edges into a new face that lies in the clipping plane. Original edges can have either winding direction with respect to the new face, but new edges are always wound clockwise. For each edge, the code determines which face index corresponds to the new face by simply checking which index still has the uninitialized value `0xFF`.

Listing 9.5. This piece of the `ClipPolyhedron()` function loops over the faces in the original poly-hedron and for each face, loops over the edges forming the boundary of that face. When original edges having an odd classification code are encountered, it means that a new edge needs to be cre-ated in the clipping plane to close the original face. The new edge is always wound counterclockwise with respect to the original face. (Continued in Listing 9.6.)

```
uint32 planeEdgeCount = 0;
for (uint32 a = 0; a < faceCount; a++)
{
    uint8 remappedFaceIndex = faceRemap[a];
    if (remappedFaceIndex != 0xFF)
    {
        // The face is not completely clipped away.
        Edge *newEdge = nullptr; uint8 newEdgeIndex = 0xFF;

        const Face *face = &polyhedron->face[a];
        uint32 faceEdgeCount = face->edgeCount;
        Face *resultFace = &result->face[remappedFaceIndex];
        uint32 resultFaceEdgeCount = 0;

        for (uint32 b = 0; b < faceEdgeCount; b++)    // Loop over face's original edges.
        {
            uint8 edgeIndex = face->edgeIndex[b];
            int32 code = edgeCode[edgeIndex];
            if (code & 1)
            {
                // One endpoint on negative side of plane, and other either
                // on positive side (code == 3) or in plane (code == 1).
                if (!newEdge)
                {
```

```
                    // At this point, we know we need a new edge.
                    newEdgeIndex = resultEdgeCount;
                    newEdge = &result->edge[resultEdgeCount];
                    planeEdgeTable[planeEdgeCount++] = uint8(resultEdgeCount++);
                    *newEdge = Edge{{0xFF, 0xFF}, {remappedFaceIndex, 0xFF}};
                }

                const Edge *edge = &polyhedron->edge[edgeIndex];
                bool ccw = (edge->faceIndex[0] == a);
                bool insertEdge = ccw ^ (vertexCode[edge->vertexIndex[0]] == 0);

                if (code == 3)  // Original edge has been clipped.
                {
                    uint8 remappedEdgeIndex = edgeRemap[edgeIndex];
                    resultFace->edgeIndex[resultFaceEdgeCount++] = remappedEdgeIndex;
                    const Edge *resultEdge = &result->edge[remappedEdgeIndex];
                    if (insertEdge)
                    {
                        newEdge->vertexIndex[0] = resultEdge->vertexIndex[ccw];
                        resultFace->edgeIndex[resultFaceEdgeCount++] = newEdgeIndex;
                    }
                    else newEdge->vertexIndex[1] = resultEdge->vertexIndex[!ccw];
                }
                else    // Original edge has been deleted, code == 1.
                {
                    if (insertEdge)
                    {
                        newEdge->vertexIndex[0] = vertexRemap[edge->vertexIndex[!ccw]];
                        resultFace->edgeIndex[resultFaceEdgeCount++] = newEdgeIndex;
                    }
                    else newEdge->vertexIndex[1] = vertexRemap[edge->vertexIndex[ccw]];
                }
            }
            else if (code != 0)
            {
                // Neither endpoint is on the negative side of the clipping plane.
                uint8 remappedEdgeIndex = edgeRemap[edgeIndex];
                resultFace->edgeIndex[resultFaceEdgeCount++] = remappedEdgeIndex;
                if (code == 2) planeEdgeTable[planeEdgeCount++] = remappedEdgeIndex;
            }
        }

        if ((newEdge) && (max(newEdge->vertexIndex[0], newEdge->vertexIndex[1]) == 0xFF))
        {
            *result = *polyhedron; return (true);  // The input polyhedron was invalid.
        }

        resultFace->edgeCount = uint8(resultFaceEdgeCount);
    }
}
```

Listing 9.6. This final piece of the `ClipPolyhedron()` function assembles all of the edges in the clipping plane to form a new face covering the clipped portion of the polyhedron.

```cpp
    if (planeEdgeCount > 2)
    {
        result->plane[resultFaceCount] = plane;
        Face *resultFace = &result->face[resultFaceCount];
        resultFace->edgeCount = uint8(planeEdgeCount);

        for (uint32 a = 0; a < planeEdgeCount; a++)
        {
            uint8 edgeIndex = planeEdgeTable[a];
            resultFace->edgeIndex[a] = edgeIndex;

            Edge *resultEdge = &result->edge[edgeIndex];
            uint8 k = (resultEdge->faceIndex[1] == 0xFF);
            resultEdge->faceIndex[k] = uint8(resultFaceCount);
        }

        resultFaceCount++;
    }

    result->vertexCount = uint8(resultVertexCount);
    result->edgeCount = uint8(resultEdgeCount);
    result->faceCount = uint8(resultFaceCount);
    return (true);
}
```

9.3 Bounding Volumes

A single object in a game world can be made up of many thousands of triangles sharing many thousands of vertices. It would be horribly inefficient and impractical to determine whether such an object is visible inside the view frustum by performing any calculation that operates directly on its triangle mesh. Instead, each object is given a *bounding volume* defined by a simple convex geometric shape that surrounds the triangle mesh as tightly as possible. Calculations performed on these basic shapes are very fast, and they allow large numbers of objects to be tested for visibility at high speed. Although support for a wider variety of shapes could be implemented by a game engine, we focus exclusively on spheres and boxes in this chapter. The rewards for supporting additional shapes are often too small to justify the greater complexity and impact on code cleanliness.

When fitting a bounding volume to a triangle mesh, it is sufficient to enclose all of the vertices because convexity then guarantees that every point inside a tri-

angle is also included in the bounding volume. We measure the quality of a bounding volume, relative to other bounding volumes of the same basic shape, by its surface area because the likelihood that a shape intersects another geometric object is determined by its cross section, not its volume. For every triangle mesh, there exist an optimal sphere and box constituting the bounding volumes with the smallest possible surface areas, but these are surprisingly difficult to calculate in general. To avoid complexity that often yields a depressingly small return on investment, we settle for simple methods that produce acceptably good approximations to the best bounding volumes possible.

9.3.1 Bounding Spheres

Given a set of n vertices $v_0, v_1, \ldots, v_{n-1}$, the task of calculating a bounding sphere having a tight fit involves finding a satisfactory center position p and radius r. It may be tempting to simply use the average position of v_i as the center of the sphere, but vertex clustering can have a negative impact on the quality of the result. The best bounding sphere is not determined by the vertices in its interior, but only those on its surface. Considering that fact, a basic strategy that may immediately come to mind is to search for the two vertices v_a and v_b that are furthest apart and place the center p halfway between them. The radius is then set to the greatest distance between p and any vertex v_i, which could be larger than half the distance between v_a and v_b. There are two problems with this approach, however. First, finding the most separated vertices v_a and v_b is an $O(n^2)$ process, which can be time consuming for large triangle meshes. Second, the point halfway between v_a and v_b isn't necessarily a good choice for the center. For example, the best center position for a sphere surrounding a pyramid is not located on its base or any of its edges. We will address both of these issues by taking a slightly more sophisticated approach.

Instead of directly finding the two vertices v_a and v_b such that $\|v_b - v_a\|$ is maximized, we project the vertex positions onto a small set of m fixed directions $\mathbf{u}_j$ and, for each direction, select the vertices producing the minimum and maximum values of the dot product $\mathbf{u}_j \cdot v_i$. As illustrated in Figure 9.4, this process identifies the pair of vertices v_a and v_b for which tangent planes perpendicular to the direction $\mathbf{u}_j$ are furthest apart. After finding the extents of the triangle mesh along every one of the m directions $\mathbf{u}_j$, we choose the direction for which the actual distance $\|v_b - v_a\|$, not just the difference in dot products, is maximized and place the center of the sphere halfway between those two vertices. Since m is a constant and each step requires only one pass through the array of vertices, the search runs in $O(n)$ time. This procedure calculates the approximate *diameter* of a triangle mesh, defined as the maximal distance between any two points on its surface. The accuracy
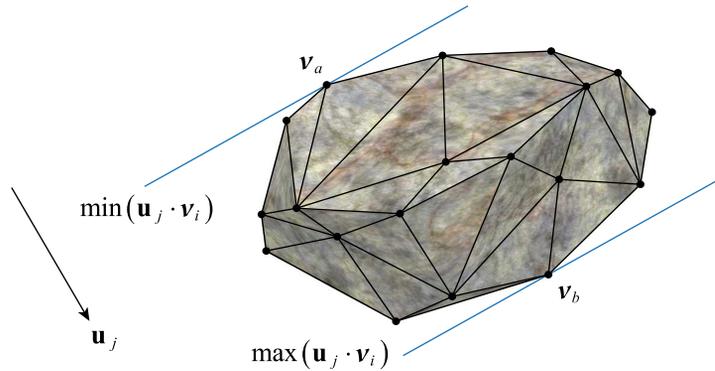
Figure 9.4. The extents of a triangle mesh with respect to a direction $\mathbf{u}_j$ are determined by the indices $i = a$ and $i = b$ for which the dot product $\mathbf{u}_j \cdot \mathbf{v}_i$ is minimized and maximized.

of the diameter increases with the number of directions m that are considered. The implementation shown in Listing 9.7 uses a total of $m = 13$ directions corresponding to the vertices, edge midpoints, and face centers of a cube. (It is not necessary that the directions be normalized to unit length.) The code returns the indices a and b of the vertices $\mathbf{v}_a$ and $\mathbf{v}_b$ that were found to be furthest apart as well as the squared distance between them. This information is used as a starting point for calculating a bounding sphere, and it will later be used to determine the orientation of a bounding box.

Once the vertices $\mathbf{v}_a$ and $\mathbf{v}_b$ representing the diameter of a triangle mesh have been determined, we set the center $\boldsymbol{p}$ and radius r of the bounding sphere to

$$\boldsymbol{p} = \frac{\boldsymbol{v}_a + \boldsymbol{v}_b}{2} \quad \text{and} \quad r = \frac{\left\| \boldsymbol{v}_b - \boldsymbol{v}_a \right\|}{2}. \tag{9.5}$$

Even if we have found the true diameter of the triangle mesh, this sphere may not contain every vertex. The simplest example is that of an equilateral triangle. In that case, the initial radius r is at most half the length of an edge, but it does not produce a sphere large enough to contain all three vertices for any center position. We need to make one more pass through the vertex array and adjust the center and radius of the sphere as necessary to guarantee inclusion.

During the final pass over the vertices, we maintain a current center $\boldsymbol{p}$ and radius r that are initially set to the values given by Equation (9.5). If we encounter a vertex $\boldsymbol{v}$ such that $(\boldsymbol{v} - \boldsymbol{p})^2 > r^2$, then we have found a vertex outside the current bounding sphere. To remedy the situation, we calculate the center $\boldsymbol{p}'$ and radius r'

Listing 9.7. This function calculates the approximate diameter of a set of `vertexCount` vertices with positions stored in the array specified by the `vertex` parameter. The pair of vertices corresponding to the minimum and maximum extents of the mesh are calculated for 13 directions, and the indices of the vertices in the pair found to be furthest apart are returned in the a and b parameters. The return value of the function is the squared distance between these two vertices.

```cpp
float CalculateDiameter(int32 vertexCount, const Point3D *vertex, int32 *a, int32 *b)
{
    constexpr int32 kDirectionCount = 13;
    static const float direction[kDirectionCount][3] =
    {
      {1.0F, 0.0F, 0.0F}, {0.0F, 1.0F, 0.0F}, {0.0F, 0.0F, 1.0F},
      {1.0F, 1.0F, 0.0F}, {1.0F, 0.0F, 1.0F}, {0.0F, 1.0F, 1.0F},
      {1.0F, -1.0F, 0.0F}, {1.0F, 0.0F, -1.0F}, {0.0F, 1.0F, -1.0F},
      {1.0F, 1.0F, 1.0F}, {1.0F, -1.0F, 1.0F}, {1.0F, 1.0F, -1.0F}, {1.0F, -1.0F, -1.0F}
    };

    float     dmin[kDirectionCount], dmax[kDirectionCount];
    int32     imin[kDirectionCount], imax[kDirectionCount];

    // Find min and max dot products for each direction and record vertex indices.
    for (int32 j = 0; j < kDirectionCount; j++)
    {
        const float *u = direction[j];
        dmin[j] = dmax[j] = u[0] * vertex[0].x + u[1] * vertex[0].y + u[2] * vertex[0].z;
        imin[j] = imax[j] = 0;

        for (int32 i = 1; i < vertexCount; i++)
        {
            float d = u[0] * vertex[i].x + u[1] * vertex[i].y + u[2] * vertex[i].z;
            if (d < dmin[j]) {dmin[j] = d; imin[j] = i;}
            else if (d > dmax[j]) {dmax[j] = d; imax[j] = i;}
        }
    }

    // Find direction for which vertices at min and max extents are furthest apart.
    float d2 = SquaredMagnitude(vertex[imax[0]] - vertex[imin[0]]); int32 k = 0;
    for (int32 j = 1; j < kDirectionCount; j++)
    {
        float m2 = SquaredMagnitude(vertex[imax[j]] - vertex[imin[j]]);
        if (m2 > d2) {d2 = m2; k = j;}
    }

    *a = imin[k];
    *b = imax[k];
    return (d2);
}
```

of a new bounding sphere of minimal size that encloses the current bounding sphere and the point v. As shown in Figure 9.5, the new sphere is tangent to the current sphere at the point q directly opposite the point v through the center p. We can calculate the point q in terms of the direction vector $v - p$ as

$$q = p - r\frac{v - p}{\|v - p\|}. \tag{9.6}$$

The new center p' is then placed halfway between the points v and q, and the new radius r' is given by the distance between p' and either v or q, as expressed by

$$p' = \frac{v + q}{2} \quad \text{and} \quad r' = \|q - p'\|. \tag{9.7}$$

After every vertex has been checked, we end up with a bounding sphere that adequately surrounds the entire triangle mesh. During this process, the center of the bounding sphere tends to shift toward vertices that were not initially included.

The code shown in Listing 9.8 calculates a bounding sphere for an arbitrary set of vertices. It first calls the `CalculateDiameter()` function given by Listing 9.7 and applies Equation (9.5) to determine the initial center and radius. It then makes a pass through all of the vertices and adjusts the bounding sphere as necessary using Equation (9.7).
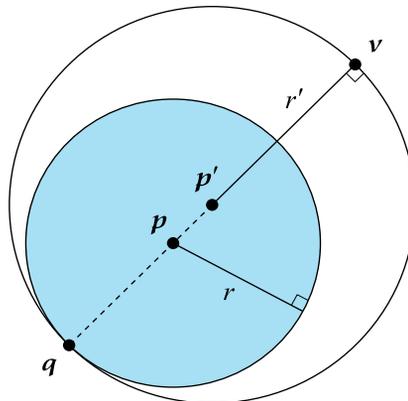
Figure 9.5. A bounding sphere defined by the center p and radius r may not initially include every vertex v in a triangle mesh. By calculating the point q directly opposite v through the center of the sphere, we find the diameter of a larger sphere with center p' and radius r' that includes the original sphere and the point v.

Listing 9.8. This function calculates the bounding sphere for a set of `vertexCount` vertices with positions stored in the array specified by the `vertex` parameter. The center p of the sphere is returned in the `center` parameter, and the radius r is the return value of the function.

```
float CalculateBoundingSphere(int32 vertexCount, const Point3D *vertex, Point3D *center)
{
    int32    a, b;

    // Determine initial center and radius.
    float d2 = CalculateDiameter(vertexCount, vertex, &a, &b);
    *center = (vertex[a] + vertex[b]) * 0.5F;
    float radius = sqrt(d2) * 0.5F;

    // Make pass through vertices and adjust sphere as necessary.
    for (int32 i = 0; i < vertexCount; i++)
    {
        Vector3D pv = vertex[i] - *center;
        float m2 = SquaredMagnitude(pv);
        if (m2 > radius * radius)
        {
            Point3D q = *center - (pv * (radius / sqrt(m2)));
            *center = (q + vertex[i]) * 0.5F;
            radius = Magnitude(q - *center);
        }
    }

    return (radius);
}
```

9.3.2 Bounding Boxes

A bounding box can be defined by a center p and three half-extents h_x, h_y, and h_z. The half-extents are equal to the distances from p to the planes coincident with the box's faces along three mutually orthogonal axes. Due to the additional freedoms provided by having three independent dimensions, a box is usually the superior choice for a bounding volume compared to a sphere. Although a box and sphere surrounding the same triangle mesh will have similar diameters, a box can often be shrunk to a much tighter fit along various directions derived from the direction between the vertices with the greatest separation. This allows a bounding box to have a smaller cross-sectional area from some perspectives, whereas the cross-sectional area of a bounding sphere remains the same.

Bounding boxes come in two different varieties whose names reflect their relationships to the underlying coordinate system. An *axis-aligned bounding box*

(AABB) is a box having sides that are aligned to the x, y, and z coordinate axes. An axis-aligned bounding box is particularly easy to calculate by simply making one pass through a set of vertices $\{v_i\}$ and recording the minimum and maximum x, y, and z coordinate values, as shown in Figure 9.6(a). An axis-aligned bounding box is frequently chosen for computational simplicity in situations where the speed of calculating a bounding box or the ease of placing it in some larger data structure is more important than whether the box has the best possible size. The function shown in Listing 9.9 calculates the center and half-extents of an AABB.

An *oriented bounding box (OBB)* is a box that has been aligned to a set of perpendicular directions **s**, **t**, and **u**, as shown in Figure 9.6(b). Choosing the directions **s**, **t**, and **u** optimally is a difficult problem, so we will again make an approximation to keep the implementation simple. Once the directions **s**, **t**, and **u** have been selected, the extents of an oriented bounding box are easy to calculate by recording minimum and maximum dot products $s \cdot v_i$, $t \cdot v_i$, and $u \cdot v_i$ over a set of vertices $\{v_i\}$. When **s**, **t**, and **u** coincide with the x, y, and z axes, this reduces to the process of selecting the minimum and maximum coordinates described for an axis-aligned bounding box.

A triangle mesh is typically created with some care taken to align its geometric features to the x, y, and z axes in its local coordinate system. In this case, an AABB
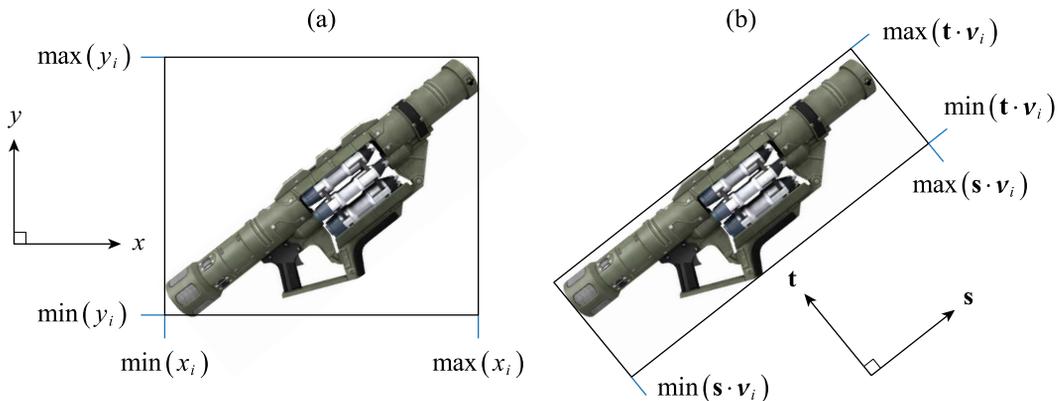
Figure 9.6. Bounding boxes are calculated for a set of vertices $\{v_i\}$ making up the triangle mesh for a rocket launcher. (a) The axis-aligned bounding box (AABB) is determined by the minimum and maximum x, y, and z coordinates over all vertices $v_i = (x_i, y_i, z_i)$. (The z axis points out of the page.) (b) The oriented bounding box (OBB) aligned to the directions **s**, **t**, and **u** is determined by the minimum and maximum values of the dot products $s \cdot v_i$, $t \cdot v_i$, and $u \cdot v_i$. (The direction **u** points out of the page.)

Listing 9.9. This function calculates the axis-aligned bounding box (AABB) for a set of `vertex-Count` vertices with positions stored in the array specified by the `vertex` parameter. The center p of the box is returned in the `center` parameter, and the half-extents h_x, h_y, and h_z are returned in the `size` parameter.

```
void CalculateAxisAlignedBoundingBox(int32 vertexCount, const Point3D *vertex,
                                     Point3D *center, Vector3D *size)
{
    Point3D vmin = vertex[0], vmax = vertex[0];
    for (int32 i = 1; i < vertexCount; i++)
    {
        vmin = Min(vmin, vertex[i]);
        vmax = Max(vmax, vertex[i]);
    }

    *center = (vmin + vmax) * 0.5F;
    *size = (vmax - vmin) * 0.5F;
}
```

is the appropriate type of bounding box in object space, and it becomes an OBB when transformed into world space by a matrix $\mathbf{M}_{\text{object}}$. In world space, the directions $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{u}$ are not calculated but are given by the first three columns of $\mathbf{M}_{\text{object}}$, and the half-extents acquire any scale contained in $\mathbf{M}_{\text{object}}$.

In cases where there may not be an obvious natural alignment, we would like to be able to create an oriented bounding box that fits tightly around a triangle mesh. The strategy for quickly calculating good directions $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{u}$ for an OBB is to derive a small set of candidate directions from the vertices $\{v_i\}$ to serve as the primary axis $\mathbf{s}$. For each direction $\mathbf{s}$, we also derive a small set of candidate directions perpendicular to $\mathbf{s}$ to serve as the secondary axis $\mathbf{t}$. The third axis is always set to $\mathbf{u} = \mathbf{s} \times \mathbf{t}$. Then we simply test each combination of primary and secondary axes by computing the extents of the bounding box in those directions, and we select the pair of axes that produces the box with the least surface area.

The candidates for the primary axis are determined by a heuristic method that systematically identifies five vertices representing extremal points of the triangle mesh. The first two vertices are the points v_a and v_b corresponding to the approximate diameter of the triangle mesh, and they are calculated using the same method that was used for a bounding sphere with the code in Listing 9.7. A third point v_c is then identified as the vertex having the greatest perpendicular distance to the line containing v_a and v_b. As shown in Figure 9.7(a), these three points define a plane that cuts through the triangle mesh. The final two points v_e and v_f are set to the vertices having the least and greatest dot products with that plane. The vertices v_a,
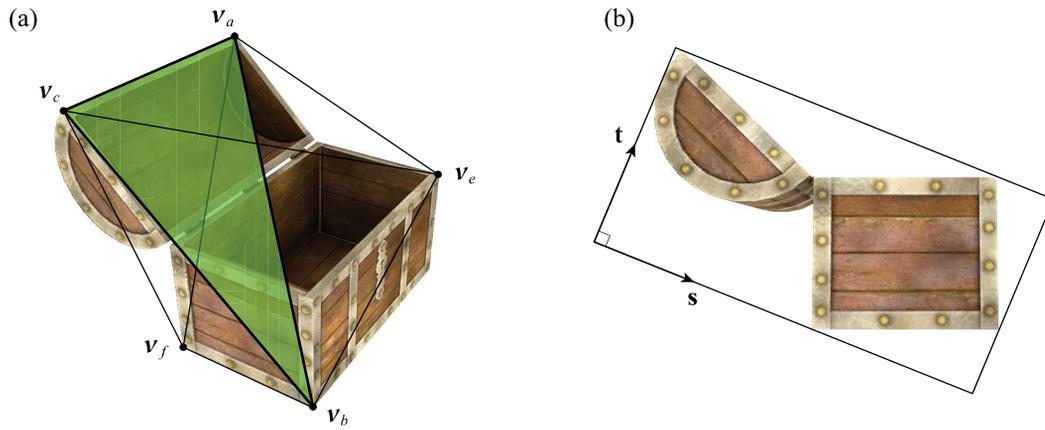
Figure 9.7. An oriented bounding box (OBB) is calculated for an open treasure chest. (a) Candidates for the primary axis of the OBB are given by the nine possible differences between pairs of systematically derived extremal points v_a, v_b, v_c, v_e, and v_f. (b) The direction from v_a to v_e was chosen as the best primary axis $\mathbf{s}$, and a perpendicular direction in the vertical plane was chosen as the best secondary axis $\mathbf{t}$.

v_b, v_c, v_e, and v_f form a pair of tetrahedra that share a common base drawn as a green triangle in the figure. The nine unique edges of these tetrahedra are the candidates for the primary axis, and chances are that at least one of them represents a natural orientation for the triangle mesh. Figure 9.7(b) shows the primary axis $\mathbf{s}$ that was chosen for an example model.

For each possible primary axis $\mathbf{s}$, we build a set of candidates for the secondary axis $\mathbf{t}$ in the plane perpendicular to $\mathbf{s}$ by applying a two-dimensional analog of the same heuristic method. First, a vector basis is established for the space orthogonal to $\mathbf{s}$ by selecting a perpendicular vector $\mathbf{x}$ using the code in Listing 9.10. This function produces a vector in the plane perpendicular to any vector $\mathbf{v}$ by taking a cross product with whichever axis-aligned unit vector $\mathbf{i}$, $\mathbf{j}$, or $\mathbf{k}$ corresponds to the smallest component of $\mathbf{v}$. After the first perpendicular vector $\mathbf{x}$ is determined, the 2D basis is completed by calculating $\mathbf{y} = \mathbf{s} \times \mathbf{x}$.

In the space spanned by $\mathbf{x}$ and $\mathbf{y}$, we calculate a secondary diameter for the triangle mesh using a method analogous to the method used to calculate the primary diameter in Listing 9.7. As before, we search through a small set of fixed directions $\mathbf{u}_j$ and find the one for which the minimum and maximum dot products $\mathbf{u}_j \cdot v_i$ have the largest difference. This time, however, each direction $\mathbf{u}_j$ is specified relative to the basis $\{\mathbf{x}, \mathbf{y}\}$, and the diameter is chosen as whichever pair of extremal vertices are separated by the greatest distance after rejecting the component

parallel to the primary axis **s**. If the minimum dot product occurs for the direction $\mathbf{u}_j$ when $i = a$, and the maximum dot product occurs when $i = b$, then the secondary diameter is given by the pair of vertices $\mathbf{v}_a$ and $\mathbf{v}_b$ for which $\left\| (\mathbf{v}_b - \mathbf{v}_a)_{\perp \mathbf{s}} \right\|$ has the largest value over all directions $\mathbf{u}_j$. This process is implemented in Listing 9.11 for a set of four directions spaced 45 degrees apart.

Listing 9.10. This function calculates a vector that is perpendicular to the vector v. The return value is not normalized to unit length.

```
Vector3D MakePerpendicularVector(const Vector3D& v)
{
    float x = Fabs(v.x), y = Fabs(v.y), z = Fabs(v.z);
    if (z < Fmin(x, y)) return (Vector3D(v.y, -v.x, 0.0F));
    if (y < x) return (Vector3D(-v.z, 0.0F, v.x));
    return (Vector3D((0.0F, v.z, -v.y));
}
```

Listing 9.11. This function calculates the secondary diameter of a set of `vertexCount` vertices with positions stored in the array specified by the `vertex` parameter. The pair of vertices corresponding to the minimum and maximum extents of the mesh are calculated for four directions in the 2D space perpendicular to the primary axis specified by `axis`, which must have unit length. The indices of the vertices in the pair found to be furthest apart after rejecting the component parallel to the primary axis are returned in the `a` and `b` parameters.

```
void CalculateSecondaryDiameter(int32 vertexCount, const Point3D *vertex,
                                const Vector3D& axis, int32 *a, int32 *b)
{
    constexpr int32 kDirectionCount = 4;
    static const float direction[kDirectionCount][2] =
        {{1.0F, 0.0F}, {0.0F, 1.0F}, {1.0F, 1.0F}, {1.0F, -1.0F}};

    float    dmin[kDirectionCount], dmax[kDirectionCount];
    int32    imin[kDirectionCount], imax[kDirectionCount];

    // Create vectors x and y perpendicular to the primary axis.
    Vector3D x = MakePerpendicularVector(axis), y = Cross(axis, x);

    // Find min and max dot products for each direction and record vertex indices.
    for (int32 j = 0; j < kDirectionCount; j++)
    {
        Vector3D t = x * direction[j][0] + y * direction[j][1];
        dmin[j] = dmax[j] = Dot(t, vertex[0]);
        imin[j] = imax[j] = 0;
```

```
    for (int32 i = 1; i < vertexCount; i++)
    {
        float d = Dot(u, vertex[i]);
        if (d < dmin[j]) {dmin[j] = d; imin[j] = i;}
        else if (d > dmax[j]) {dmax[j] = d; imax[j] = i;}
    }
}

// Find diameter in plane perpendicular to primary axis.
Vector3D dv = vertex[imax[0]] - vertex[imin[0]]; int32 k = 0;
float d2 = SquaredMagnitude(dv - axis * Dot(dv, axis));
for (int32 j = 1; j < kDirectionCount; j++)
{
    dv = vertex[imax[j]] - vertex[imin[j]];
    float m2 = SquaredMagnitude(dv - axis * Dot(dv, axis));
    if (m2 > d2) {d2 = m2; k = j;}
}

*a = imin[k]; *b = imax[k];
}
```

Given the vertices v_a and v_b corresponding to the secondary diameter, we form the plane containing those two points and the direction **s** of the primary axis. In a manner similar to how two tetrahedra were created in 3D space, we create two triangles in 2D space by finding the vertices v_e and v_f having the least and greatest dot products with that plane. After rejecting components parallel to **s**, the unique edges of these two triangles provide five candidates for the secondary axis **t**. It is likely that one of those candidates represents a natural orientation for the bounding box in the space orthogonal to the primary axis **s**. Code that calculates all candidate directions for primary and secondary axes is shown in Listing 9.12.

The code shown in Listing 9.13 uses all of the procedures discussed so far to finally calculate an oriented bounding box for a triangle mesh. It first calculates the diameter and establishes the set of nine candidates for the primary axis **s**. For each of these candidates, the code calculates the secondary diameter in the space perpendicular to **s** and establishes the set of five candidates for the secondary axis **t**. For every one of the 45 combinations of these axis directions, the code calculates the minimum and maximum dot products between each vertex position v_i and the three axes **s**, **t**, and **u**, where $\mathbf{u} = \mathbf{s} \times \mathbf{t}$, producing the values

$$s_{\min} = \min(\mathbf{s} \cdot \mathbf{v}_i) \qquad s_{\max} = \max(\mathbf{s} \cdot \mathbf{v}_i)$$
$$t_{\min} = \min(\mathbf{t} \cdot \mathbf{v}_i) \qquad t_{\max} = \max(\mathbf{t} \cdot \mathbf{v}_i)$$
$$u_{\min} = \min(\mathbf{u} \cdot \mathbf{v}_i) \qquad u_{\max} = \max(\mathbf{u} \cdot \mathbf{v}_i). \tag{9.8}$$

Listing 9.12. These functions calculate the candidate directions for the primary and secondary axes of an oriented bounding box. The candidates for the primary axis correspond to the edges of the two tetrahedra shown in Figure 9.7. For each specific primary axis **s**, the candidates for the secondary axis **t** correspond to the edges of an analogous pair of triangles in the space perpendicular to **s**.

```cpp
void FindExtremalVertices(int32 vertexCount, const Point3D *vertex, const Plane& plane,
                          int32 *e, int32 *f)
{
    *e = 0; *f = 0;
    float dmin = Dot(plane, vertex[0]), dmax = dmin;
    for (int32 i = 1; i < vertexCount; i++)
    {
        float m = Dot(plane, vertex[i]);
        if (m < dmin) {dmin = m; *e = i;}
        else if (m > dmax) {dmax = m; *f = i;}
    }
}

void GetPrimaryBoxDirections(int32 vertexCount, const Point3D *vertex,
                             int32 a, int32 b, Vector3D *direction)
{
    int32 c = 0;
    direction[0] = vertex[b] – vertex[a];
    float dmax = DistPointLine(vertex[0], vertex[a], direction[0]);
    for (int32 i = 1; i < vertexCount; i++)
    {
        float m = DistPointLine(vertex[i], vertex[a], direction[0]);
        if (m > dmax) {dmax = m; c = i;}
    }

    direction[1] = vertex[c] – vertex[a]; direction[2] = vertex[c] – vertex[b];
    Vector3D normal = Cross(direction[0], direction[1]);
    Plane plane(normal, –Dot(normal, vertex[a]));

    int32 e, f;
    FindExtremalVertices(vertexCount, vertex, plane, &e, &f);
    direction[3] = vertex[e] – vertex[a]; direction[4] = vertex[e] – vertex[b];
    direction[5] = vertex[e] – vertex[c]; direction[6] = vertex[f] – vertex[a];
    direction[7] = vertex[f] – vertex[b]; direction[8] = vertex[f] – vertex[c];
}

void GetSecondaryBoxDirections(int32 vertexCount, const Point3D *vertex,
                               const Vector3D& axis, int32 a, int32 b, Vector3D *direction)
{
    direction[0] = vertex[b] – vertex[a];
    Vector3D normal = Cross(axis, direction[0]);
    Plane plane(normal, –Dot(normal, vertex[a]));
```

```
    int32 e, f;
    FindExtremalVertices(vertexCount, vertex, plane, &e, &f);
    direction[1] = vertex[e] - vertex[a]; direction[2] = vertex[e] - vertex[b];
    direction[3] = vertex[f] - vertex[a]; direction[4] = vertex[f] - vertex[b];
    for (int32 j = 0; j < 5; j++) direction[j] -= axis * Dot(direction[j], axis);
}
```

Considering these values, the half-extents h_x, h_y, and h_z of a bounding box aligned to the directions **s**, **t**, and **u** are then given by

$$h_x = \frac{s_{max} - s_{min}}{2}, \quad h_y = \frac{t_{max} - t_{min}}{2}, \quad \text{and} \quad h_z = \frac{u_{max} - u_{min}}{2}. \tag{9.9}$$

Since we want the bounding box having the smallest surface area, we calculate

$$A = h_x h_y + h_y h_z + h_z h_x, \tag{9.10}$$

which is only one-eighth of the surface area but provides an accurate measurement for comparison. The best bounding box is produced by the pair of candidate axes **s** and **t** for which the area A is minimized. As the code iterates through the 45 possibilities, the center position is calculated and recorded along with the half-extents whenever a better box is found. In terms of the extents along the axes, the center position p is given by

$$\mathbf{p} = \frac{s_{min} + s_{max}}{2}\mathbf{s} + \frac{t_{min} + t_{max}}{2}\mathbf{t} + \frac{u_{min} + u_{max}}{2}\mathbf{u}. \tag{9.11}$$

Listing 9.13. This function calculates an oriented bounding box for a set of `vertexCount` vertices with positions stored in the array specified by the `vertex` parameter. The center p is returned in the `center` parameter, and the half-extents h_x, h_y, and h_z are returned in the `size` parameter. The `axis` parameter must provide storage for an array of three vectors to which the **s**, **t**, and **u** axes are written.

```
void CalculateOrientedBoundingBox(int32 vertexCount, const Point3D *vertex,
                                  Point3D *center, Vector3D *size, Vector3D *axis)
{
    int32       a, b;
    Vector3D    primaryDirection[9], secondaryDirection[5];

    CalculateDiameter(vertexCount, vertex, &a, &b);
    GetPrimaryBoxDirections(vertexCount, vertex, a, b, primaryDirection);

    float area = FLT_MAX;
```

```
    for (int32 k = 0; k < 9; k++)    // Loop over all candidates for primary axis.
    {
        Vector3D s = Normalize(primaryDirection[k]);
        CalculateSecondaryDiameter(vertexCount, vertex, s, &a, &b);
        GetSecondaryBoxDirections(vertexCount, vertex, s, a, b, secondaryDirection);

        for (int32 j = 0; j < 5; j++)    // Loop over all candidates for secondary axis.
        {
            Vector3D t = Normalize(secondaryDirection[j]), u = Cross(s, t);
            float smin = Dot(s, vertex[0]), smax = smin;
            float tmin = Dot(t, vertex[0]), tmax = tmin;
            float umin = Dot(u, vertex[0]), umax = umin;

            for (int32 i = 1; i < vertexCount; i++)
            {
                float ds = Dot(s, vertex[i]), dt = Dot(t, vertex[i]), du = Dot(u, vertex[i]);
                smin = Fmin(smin, ds); smax = Fmax(smax, ds);
                tmin = Fmin(tmin, dt); tmax = Fmax(tmax, dt);
                umin = Fmin(umin, du); umax = Fmax(umax, du);
            }

            float hx = (smax - smin) * 0.5F, hy = (tmax - tmin) * 0.5F,
                  hz = (umax - umin) * 0.5F;

            // Calculate one-eighth surface area and see if it's better.
            float m = hx * hy + hy * hz + hz * hx;
            if (m < area)
            {
                *center = (s * (smin + smax) + t * (tmin + tmax) + u * (umin + umax)) * 0.5F;
                size->Set(hx, hy, hz); axis[0] = s; axis[1] = t; axis[2] = u;
                area = m;
            }
        }
    }
}
```

9.4 Frustum Culling

Frustum culling is the process of rejecting objects that are known to lie outside the view frustum before they are needlessly sent to the GPU to be rendered. Frustum culling is applied to whole objects that would each be rendered as a single triangle mesh on the GPU, and this typically represents the finest granularity at which culling is performed on the CPU. In cases when an object is only partially visible, its entire triangle mesh is submitted to the GPU where any portions outside the view frustum are then rejected on a per-triangle basis.

The frustum culling process takes only an object's bounding volume into consideration. Whatever might be inside the bounding volume, whether it's a triangle mesh, a light source, or some kind of special effect, is never examined directly. If we can prove that an object's bounding volume lies completely outside the view frustum, then we know with certainty that the object itself cannot be visible, and we cull it. Because a bounding volume is almost always larger in some way than the object it surrounds, there can be instances in which frustum culling fails to reject an object that is actually out of view by a small distance, but these occurrences are rare enough that we don't worry about them.

A complex model may have several bounding volumes that are organized in a tree structure called a *bounding volume hierarchy (BVH)*. In such an arrangement, each piece of the model has its own bounding volume, but those pieces are also grouped together and surrounded by a larger bounding volume. This grouping can happen once or continue for several levels until there is ultimately one bounding volume for the entire model that contains all of the smaller bounding volumes beneath it in the hierarchy. The advantage to using a BVH is that once a particular node in the tree is found to be invisible, because it's completely outside the view frustum, then its subnodes are known to be invisible as well and can therefore be skipped altogether. Conversely, if a node in the tree is found to lie entirely *inside* the view frustum, then there is no need to test its subnodes because we already know they must be visible as well.

9.4.1 Visibility Regions

We often need to test objects for visibility not only against the view frustum, but against other volumes of space that could be generated from another perspective, such as the position of a light source. The geometric description of each such volume can be generalized to a convex shape bounded by a set of inward-facing planes that we call a *visibility region*. There are many types of visibility regions that we will encounter in this chapter, and they are used for a variety of purposes beyond determining whether an object is visible inside the view frustum. They can also be used to figure out whether an object is illuminated by a light source or whether an object that may itself be invisible still has a shadow that enters the view frustum. Another example is the Z-fail region introduced in Section 8.4.2 used to determine which variant of the stencil shadow algorithm should be applied.

The view frustum is the visibility region defined by the six inward-facing planes listed in Table 6.1. To demonstrate that an object is not visible, all we have to do is show that its bounding volume lies completely on the negative side of at least one of the region's boundary planes, as illustrated in Figure 9.8. It is often the case that an infinite projection matrix is being used or the far plane is so far away
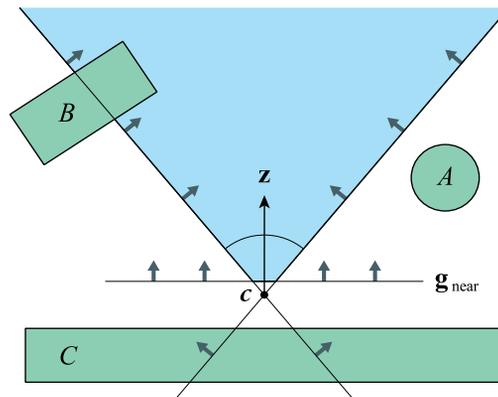
Figure 9.8. The view frustum for a camera at the position c pointing in the direction $\mathbf{z}$, shaded blue, is a visibility region defined by the near plane and four lateral planes, all of which have inward-pointing normal vectors. (The top and bottom lateral planes are not shown.) Object A can be culled because it lies completely on the negative side of one of the region's boundary planes, but object B cannot be culled because part of it lies on the positive side of every boundary plane. Object C could lie well behind the camera without falling on the negative side of any one of the lateral planes, but it is easily culled against the near plane $\mathbf{g}_{near}$.

that nothing will ever lie beyond it. For this reason, we usually perform visibility tests only against the near plane and the four lateral planes of the view frustum. It may seem like the near plane is redundant because the lateral planes all meet at the camera position a short distance away, but it's important to include the near plane because large objects behind the camera often do not fall completely on the negative side of any lateral plane. For example, the object labeled C in Figure 9.8 could represent a wall that lies several meters behind the camera. It is clearly not visible, but without the near plane to test against, it is not culled because it crosses all four of the lateral planes.

The methods used to test a bounding sphere and bounding box for intersection with a visibility region are described below. In all cases, we make the assumption that the boundary planes of a visibility region and the bounding volume of an object are provided in world space. This means that the planes specified in Table 6.1 need to be transformed from camera space to world space with Equation (6.3) before they are used for culling.

In Sections 9.6 and 9.7, we will perform operations on visibility regions that require information about the region's vertices and edges in addition to its boundary planes. All of this data can be concisely stored inside the `Polyhedron` structure

defined by Listing 9.2 earlier in this chapter. The polyhedron corresponding to a view frustum (with the far plane included) has the same topology as a cube, consisting of six faces, twelve edges, and eight vertices on the near and far planes having world-space positions provided by Equation (6.6). This polyhedron is constructed by the code shown in Listing 9.14 for a given camera node transformation matrix $\mathbf{M}_{\text{camera}}$, projection distance g, aspect ratio s, and distances n and f to the near and far planes. When rendering cascaded shadow maps, the values of n and f can be set to the range $[a_k, b_k]$ covered by a single cascade in order to construct a smaller polyhedron useful for shadow culling.

Listing 9.14. This function generates a polyhedron corresponding to a view frustum having projection distance g and aspect ratio s. The values of n and f provide the distances to the near plane and far plane, which could also represent a cascade range. The Mcam parameter specifies the object-space to world-space transformation for the camera. The results are stored in the Polyhedron data structure specified by the polyhedron parameter.

```
void BuildFrustumPolyhedron(const Transform4D& Mcam, float g, float s, float n, float f,
                            Polyhedron *polyhedron)
{
    polyhedron->vertexCount = 8; polyhedron->edgeCount = 12; polyhedron->faceCount = 6;

    // Generate vertices for the near side.
    float y = n / g, x = y * s;
    polyhedron->vertex[0] = Mcam * Point3D(x, y, n);
    polyhedron->vertex[1] = Mcam * Point3D(x, -y, n);
    polyhedron->vertex[2] = Mcam * Point3D(-x, -y, n);
    polyhedron->vertex[3] = Mcam * Point3D(-x, y, n);

    // Generate vertices for the far side.
    y = f / g; x = y * s;
    polyhedron->vertex[4] = Mcam * Point3D(x, y, f);
    polyhedron->vertex[5] = Mcam * Point3D(x, -y, f);
    polyhedron->vertex[6] = Mcam * Point3D(-x, -y, f);
    polyhedron->vertex[7] = Mcam * Point3D(-x, y, f);

    // Generate lateral planes.
    Transform4D inverse = Inverse(Mcam);
    float mx = 1.0F / sqrt(g * g + s * s), my = 1.0F / sqrt(g * g + 1.0F);
    polyhedron->plane[0] = Plane(-g * mx, 0.0F, s * mx, 0.0F) * inverse;
    polyhedron->plane[1] = Plane(0.0F, g * my, my, 0.0F) * inverse;
    polyhedron->plane[2] = Plane(g * mx, 0.0F, s * mx, 0.0F) * inverse;
    polyhedron->plane[3] = Plane(0.0F, -g * my, my, 0.0F) * inverse;

    // Generate near and far planes.
    float d = Dot(Mcam[2], Mcam[3]);
    polyhedron->plane[4].Set(Mcam[2], -(d + n));
```

```
    polyhedron->plane[5].Set(-Mcam[2], d + f);

    // Generate all edges and lateral faces.
    Edge *edge = polyhedron->edge; Face *face = polyhedron->face;
    for (int32 i = 0; i < 4; i++, edge++, face++)
    {
        edge[0].vertexIndex[0] = uint8(i);
        edge[0].vertexIndex[1] = uint8(i + 4);
        edge[0].faceIndex[0] = uint8(i);
        edge[0].faceIndex[1] = uint8((i - 1) & 3);

        edge[4].vertexIndex[0] = uint8(i);
        edge[4].vertexIndex[1] = uint8((i + 1) & 3);
        edge[4].faceIndex[0] = 4;
        edge[4].faceIndex[1] = uint8(i);

        edge[8].vertexIndex[0] = uint8(((i + 1) & 3) + 4);
        edge[8].vertexIndex[1] = uint8(i + 4);
        edge[8].faceIndex[0] = 5;
        edge[8].faceIndex[1] = uint8(i);

        face->edgeCount = 4;
        face->edgeIndex[0] = uint8(i);
        face->edgeIndex[1] = uint8((i + 1) & 3);
        face->edgeIndex[2] = uint8(i + 4);
        face->edgeIndex[3] = uint8(i + 8);
    }

    // Generate near and far faces.
    face[0].edgeCount = face[1].edgeCount = 4;
    face[0].edgeIndex[0] = 4; face[0].edgeIndex[1] = 5;
    face[0].edgeIndex[2] = 6; face[0].edgeIndex[3] = 7;
    face[1].edgeIndex[0] = 8; face[1].edgeIndex[1] = 9;
    face[1].edgeIndex[2] = 10; face[1].edgeIndex[3] = 11;
}
```

9.4.2 Sphere Visibility

A bounding sphere of radius r that has a world-space center position $\boldsymbol{p}$ intersects a visibility region precisely when there exists a point inside the sphere that is on the positive side of *all* of the region's boundary planes. Thus, we can conclude that the sphere is not visible if we can show that for any *one* of the boundary planes $\mathbf{g}$, no point inside the sphere could possibly be on the positive side of $\mathbf{g}$. Since the boundary planes face inward, the sphere must be outside the visibility region if

$$\mathbf{g} \cdot \boldsymbol{p} \leq -r, \tag{9.12}$$

assuming that **g** is normalized. When this inequality is satisfied, it means the sphere is far enough on the negative side of **g** that no point within its radius could be inside the visibility region, as illustrated in Figure 9.9(a). This leads us to the short procedure implemented in Listing 9.15 for determining whether an object's bounding sphere is visible. We simply loop over all boundary planes belonging to a visibility region and immediately cull the object as soon as we find a plane for which Equation (9.12) is true. If no such plane is found, then we conclude that the bounding sphere is visible.

Determining whether a sphere is visible by checking to see if it's center p is never located at a distance of at least r on the negative side of any boundary plane is equivalent to moving all of the boundary planes outward by the distance r and checking to see if the point p is inside the expanded visibility region. This larger volume is shown as the lighter blue area in Figure 9.9(a), and it is the volume in which the center of the sphere must lie for it to be considered visible by the method described above. However, if we were to construct the exact set of center positions representing all possible visible spheres of radius r, then the edges and corners of this volume would be rounded off. There are always small volumes of space for which center positions are a little further away than the radius r from the visibility
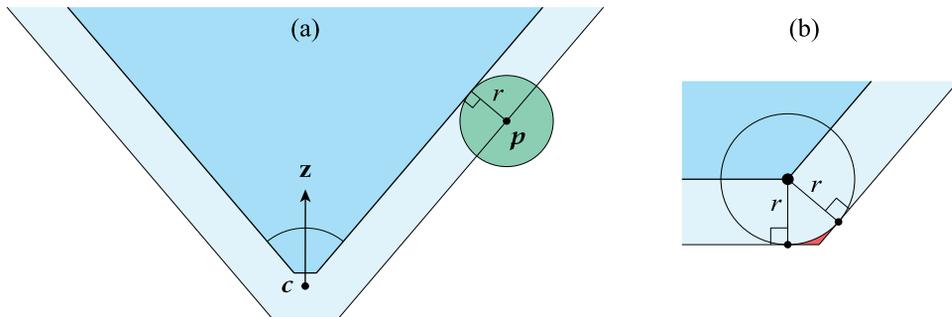
Figure 9.9. (a) The darker blue area represents the visibility region corresponding to the view frustum for a camera at the position c. A bounding sphere of radius r is invisible if its center p falls on the negative side of any boundary plane by at least the distance r. The lighter blue area represents the volume of space in which the sphere's center can be located without causing the object it contains to be culled. (b) The red area represents a small volume of space where false positives are generated. If the center of a bounding sphere is located here, then it lies completely outside the visibility region but not on the negative side of any boundary plane.

region, as exemplified by the tiny red area in Figure 9.9(b). If the center of a bounding sphere happens to fall in one of these extra volumes, then it will be classified as visible even though it actually falls entirely outside the visibility region. This is more likely to happen for spheres of larger radii, but it is generally uncommon enough that we accept the false positives for the sake of simplicity in our methods.

Listing 9.15. This function determines whether the bounding sphere specified by the `center` and `radius` parameters is visible in the region defined by the `planeCount` elements of the array specified by the `planeArray` parameter. The function returns `false` if the sphere can be culled.

```
bool SphereVisible(int32 planeCount, const Plane *planeArray,
                   const Point3D& center, float radius)
{
    float negativeRadius = -radius;
    for (int32 i = 0; i < planeCount; i++)
    {
        if (Dot(planeArray[i], center) <= negativeRadius) return (false);
    }

    return (true);
}
```

9.4.3 Box Visibility

The method that tests a bounding box for visibility is based on the same idea as the method used for bounding spheres. For a bounding box with a world-space center position p, we take the dot product with each boundary plane $\mathbf{g}$ belonging to the visibility region and compare the result to a radius representative of the box's size. The difference is that the radius is no longer a constant, but instead depends on the relationship between the orientations of the box and the boundary plane. As shown in Figure 9.10, the *effective radius* $r_{\mathbf{g}}$ of a bounding box with respect to the plane $\mathbf{g}$ is equal to half of the total extent of the box when it is projected onto the direction of the plane's normal vector. If the box has the half-extents h_x, h_y, and h_z oriented to the axes $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{u}$, then we can project each of the vectors $h_x\mathbf{s}$, $h_y\mathbf{t}$, and $h_z\mathbf{u}$ onto the normal vector $\mathbf{g}_{xyz}$ and simply add up their lengths. The effective radius $r_{\mathbf{g}}$ is thus given by

$$r_{\mathbf{g}} = |h_x\mathbf{g}\cdot\mathbf{s}| + |h_y\mathbf{g}\cdot\mathbf{t}| + |h_z\mathbf{g}\cdot\mathbf{u}|. \tag{9.13}$$

Once this radius has been calculated, the procedure for determining whether an object's bounding box is visible is the same as that for a bounding sphere. As
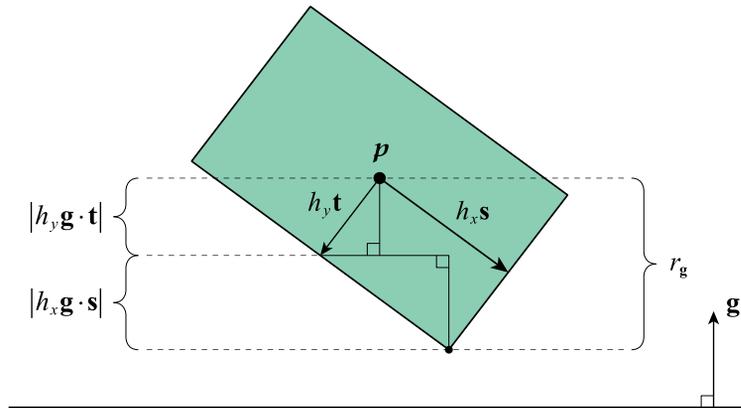
Figure 9.10. A bounding box has the center p and half-extents h_x, h_y, and h_z oriented to the axes **s**, **t**, and **u**. The box's effective radius r_g with respect to the plane **g** is given by the sum of the projected lengths of its half-extents onto the normal direction of the plane.

implemented in Listing 9.16, we loop over all the boundary planes of a visibility region and cull the object if $\mathbf{g} \cdot p \leq -r_g$ for any one of them. In the case of an axis-aligned bounding box, the vectors **s**, **t**, and **u** are aligned with the coordinate axes, and Equation (9.13) reduces to

$$r_g = |h_x g_x| + |h_y g_y| + |h_z g_z|. \tag{9.14}$$

A separate function implementing this equation for testing the visibility of AABBs is also included in Listing 9.16.

Listing 9.16. For a visibility region defined by the `planeCount` elements of the array specified by the `planeArray` parameter, these functions determine whether bounding boxes are visible and return `false` if they can be culled. In both cases, the center p of the box is specified by the `center` parameter, and the vector (h_x, h_y, h_z) of half-extents is specified by the `size` parameter. In the case of an oriented bounding box, the three unit vectors **s**, **t**, and **u** are passed in through the array specified by the `axis` parameter.

```
bool OrientedBoxVisible(int32 planeCount, const Plane *planeArray,
                    const Point3D& center, const Vector3D& size, const Vector3D *axis)
{
    for (int32 i = 0; i < planeCount; i++)
    {
        const Plane& g = planeArray[i];
        float rg = fabs(Dot(g, axis[0]) * size.x) + fabs(Dot(g, axis[1]) * size.y)
```

```
                    + fabs(Dot(g, axis[2]) * size.z);
        if (Dot(g, center) <= -rg) return (false);
    }

    return (true);
}

bool AxisAlignedBoxVisible(int32 planeCount, const Plane *planeArray,
                           const Point3D& center, const Vector3D& size)
{
    for (int32 i = 0; i < planeCount; i++)
    {
        const Plane& g = planeArray[i];
        float rg = fabs(g.x * size.x) + fabs(g.y * size.y) + fabs(g.z * size.z);
        if (Dot(g, center) <= -rg) return (false);
    }

    return (true);
}
```

9.5 Light Culling

In addition to determining what objects are visible, a game engine must be able to decide which light sources penetrate the view frustum. For each light source that happens to be visible, the engine must also have a way of determining what subset of visible objects is actually illuminated. We don't want to make unnecessary lighting calculations in the shaders for any objects that lie beyond a light's maximum range. Furthermore, if the light source is located outside the view frustum, then there may be objects that are not directly visible but still cast shadows onto objects that are visible, and the engine must be able to determine where that happens. All of these tasks can be handled by using various kinds of visibility regions and intersection tests.

Suppose we have a point light with a maximum range r_{max} at the world-space position l. Since this clearly provides us with a natural bounding sphere, we can test it against a visibility region without modification to easily determine whether the volume of space illuminated by the light is visible. It would also be possible to determine whether a spot light is visible using the its position l and range r_{max}, but that would not be a good test because at least half of the space inside that bounding sphere is not illuminated. A better test would ascertain whether the pyramidal boundary of a spot light, shown in Figure 8.4, intersects the view frustum. One method that can quickly cull a spot light in some situations is to calculate the world-space positions of the five vertices of the pyramid and take their dot products

with every plane of a visibility region. If we find a plane for which no dot product is positive, then the whole interior of the pyramid lies on the negative side of that plane, and the spot light is not visible. This test is easy to implement, but for spot lights with a large range, it can produce a lot of false positives. The most accurate way to test a spot light for visibility is to create a polyhedron with the five pyramid vertices and clip it against each of a visibility region's boundary planes using the `ClipPolyhedron()` function developed in Section 9.2. If the polyhedron is reduced to nothing after clipping it against any one of the planes, then the spot light is not visible.

After it has been decided that a particular light source illuminates at least part of the space visible to the camera, we have the task of figuring out which objects do not receive a contribution from that light so they can be excluded. Everything is lit by an infinite light, but for a point light or spot light, it's often the case that a large portion of the visible set of objects is not affected, especially if the light is far from the camera or has a short range. We consider spot lights first because the task of culling unlit objects is virtually identical to frustum culling. We then discuss intersection tests for point lights.

Since a spot light has its own frustum, we can use the same exact bounding volume tests already used to determine whether an object was not visible to now determine whether it is not illuminated. We just need to construct a visibility region using the five world-space boundary planes coinciding with the pyramidal boundary of the spot light. As with a camera's view frustum, these are given in object space by the far plane and four lateral planes listed in Table 6.1. To avoid problems near the apex of the pyramid, it's a good idea to include a near plane passing through the spot light's position. This plane is simply $\begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}$ in object space, so it is given by the third row of $\mathbf{M}_{\mathrm{light}}^{-1}$ in world space, where $\mathbf{M}_{\mathrm{light}}$ is the light's object-space to world-space transformation.

To determine whether an object is not illuminated by a point light at the position $\boldsymbol{l}$, we must be able to tell whether the object's bounding volume lies entirely outside the sphere of radius r_{max} surrounding the light source. In the case that the object has a bounding sphere with center $\boldsymbol{p}$ and radius r, the test is very simple. If the distance between the centers is greater than or equal to the sum of the radii, then the object is too far away to be illuminated. To avoid taking square roots, we implement this test by evaluating

$$(\boldsymbol{p} - \boldsymbol{l})^2 \geq (r + r_{\mathrm{max}})^2. \tag{9.15}$$

Now suppose that an object has an oriented bounding box with center $\boldsymbol{p}$, half-extents h_x, h_y, and h_z, and unit-length axes $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{u}$. Using Equation (9.13), we

can calculate the effective radius r_v of this box with respect to the vector $\mathbf{v} = \mathbf{p} - \mathbf{l}$ between the center of the box and the position of the light source. As shown in Figure 9.11(a), this allows us to make a comparison similar to the sphere-sphere test performed by Equation (9.15). If the magnitude of $\mathbf{v}$ is greater than or equal to the sum of the radii r_v and r_{max}, then we know the bounding box is out of range. Thus, one test for culling an object surrounded by a box is given by

$$v^2 \geq (r_v + r_{max})^2 = (|h_x \mathbf{v} \cdot \mathbf{s}| + |h_y \mathbf{v} \cdot \mathbf{t}| + |h_z \mathbf{v} \cdot \mathbf{u}| + r_{max})^2. \qquad (9.16)$$

This test works well for objects that are small in comparison to range of the point light, but it can have problems culling objects when the light range is smaller in relative size. Figure 9.11(b) demonstrates how the effective radius r_v is too large for the box to be culled by the test in Equation (9.16). However, we can still cull the box based on its distance from the sphere's center along one of its own axes $\mathbf{s}$, $\mathbf{t}$, or $\mathbf{u}$. The effective radii of the box in these directions are simply the half-extents h_x, h_y, and h_z. If the projected length of $\mathbf{v}$ onto $\mathbf{s}$, $\mathbf{t}$, or $\mathbf{u}$ is greater than or equal to the sum of r_{max} and the corresponding half-extent, then we know the box is out of range. This gives us three additional sphere-box tests that we can express as

$$|\mathbf{v} \cdot \mathbf{s}| \geq h_x + r_{max}, \quad |\mathbf{v} \cdot \mathbf{t}| \geq h_y + r_{max}, \quad \text{and} \quad |\mathbf{v} \cdot \mathbf{u}| \geq h_z + r_{max}. \qquad (9.17)$$

(a)

(b)
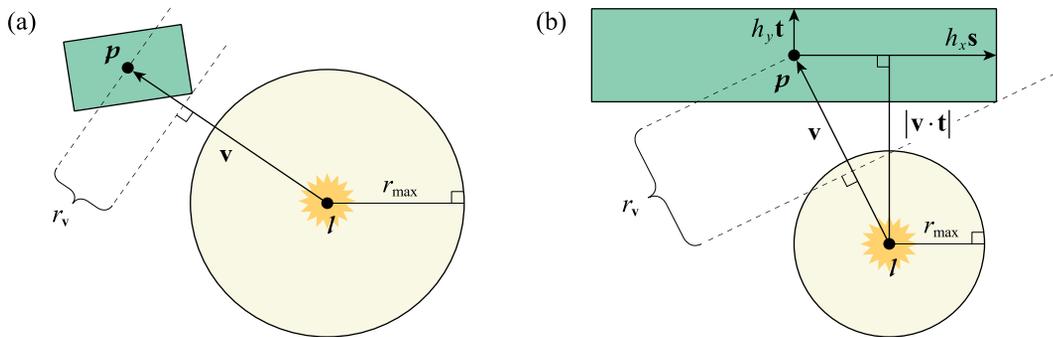
Figure 9.11. An oriented bounding box with the center $\mathbf{p}$, half-extents h_x, h_y, and h_z, and unit-length axes $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{u}$ is tested against the bounding sphere surrounding a point light of range r_{max} at the position $\mathbf{l}$. (a) The magnitude of the vector $\mathbf{v}$ between the centers $\mathbf{p}$ and $\mathbf{l}$ is greater than the sum of the effective radius r_v of the box and the radius r_{max} of the sphere, so the box can be culled by the test in Equation (9.16). (b) The effective radius is too large in this case, but the length of the projection of $\mathbf{v}$ onto the axis $\mathbf{t}$ is greater than the sum of the half-extent h_y along the direction $\mathbf{t}$ and the radius r_{max} of the sphere, so the box can be culled by the test in Equation (9.18).

If any one of these three inequalities is true, then the box can be culled. To avoid performing three separate comparisons, these tests can be consolidated into a single inequality by writing

$$\max\left(\left|\mathbf{v}\cdot\mathbf{s}\right|-h_x,\left|\mathbf{v}\cdot\mathbf{t}\right|-h_y,\left|\mathbf{v}\cdot\mathbf{u}\right|-h_z\right)\geq r_{\max}. \tag{9.18}$$

Neither one of Equations (9.16) and (9.18) subsume the other, so we perform both tests to detect as many culling opportunities as possible, as shown in Listing 9.17. There are still instances in which both tests fail even though the bounding box is actually beyond the reach of a point light, but these cases are uncommon and occur only when an edge of the bounding box is very close to the light's bounding sphere.

Listing 9.17. For a point light at the position `lightPosition` with range `rmax`, this function determines whether a bounding box is illuminated and returns `false` if it can be culled. The center $\boldsymbol{p}$ of the box is specified by the `center` parameter, and the vector $\left(h_x,h_y,h_z\right)$ of half-extents is specified by the `size` parameter. The three unit vectors $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{u}$ are passed in through the array specified by the `axis` parameter.

```cpp
bool OrientedBoxIlluminated(const Point3D& lightPosition, float rmax,
                    const Point3D& center, const Vector3D& size, const Vector3D *axis)
{
    Vector3D v = center – lightPosition;
    float vs = fabs(Dot(v, axis[0]));
    float vt = fabs(Dot(v, axis[1]));
    float vu = fabs(Dot(v, axis[2]));

    float m = size.x * vs + size.y * vt + size.z * vu + rmax;
    if (Dot(v, v) >= m * m) return (false);

    return (fmax(fmax(vs – size.x, vt – size.y), vu – size.z) < rmax);
}
```

9.6 Shadow Culling

Once we have determined that a visible object is illuminated by a light source, we know that it has to be included when shadows are rendered for that light because at very least, it may cast shadows on itself. If the light source lies inside the view frustum, then the set of illuminated objects and the set of shadow-casting objects are the same. However, if the light source lies outside the view frustum, then the full set of shadow-casting objects is generally larger than the set of illuminated

objects. When rendering shadows in this case, we also have to consider objects that are positioned between the view frustum and the light source where they are not directly visible, but they still cast shadows onto other objects that we can see. If we didn't take this possibility into consideration, then an object's shadow would disappear as soon as the object moved outside the view frustum, and this obviously is not correct.

For a point on a surface to be covered by a shadow, the line segment connecting that point to the light position l must intersect a shadow-casting object. The set of all such line segments connecting any point inside the view frustum to the light position forms a convex volume of space called a *shadow region*. Only objects that intersect the shadow region can possibly cast shadows into the view frustum, so we test bounding volumes against it and cull any objects that are discovered to fall outside. As shown in Figure 9.12, a shadow region can be recognized as the volume enclosed by the convex hull formed by the view frustum and the light position. It will be convenient for us to generalize this concept in two ways. First, the view frustum can be replaced by an arbitrary visibility region. We will demonstrate below how clipping the view frustum to a smaller volume can drastically reduce the size of the shadow region. Second, we treat the light position as a 4D homogeneous point $\mathbf{l}$ having a w coordinate of one for point lights and a w coordinate of zero for infinite lights. In the case of an infinite light, the shadow region extends to infinity in the direction toward the light $\mathbf{l}_{\text{infinite}}$ given by Equation (8.21).
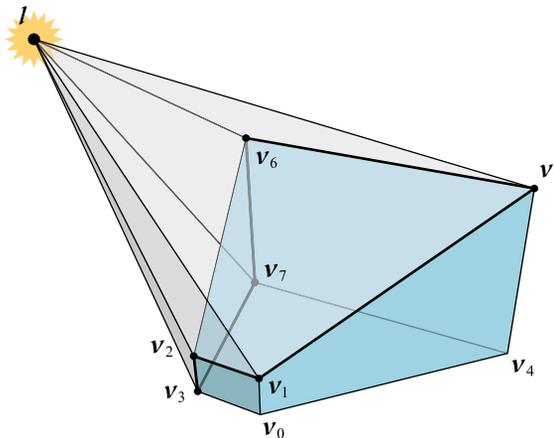
Figure 9.12. A shadow region is the convex volume of space containing the view frustum and the light position l. In addition to the back faces of the view frustum, the set of planes bounding a shadow region are derived from the silhouette edges of the view frustum with respect to the light position, shown by the bold lines.

A shadow region is constructed by identifying the *silhouette edges* of a visibility region. An edge belongs to the silhouette if the two planes intersecting at the edge have the property that the light source is on the positive side of one plane and the negative side of the other plane. This is the same meaning of silhouette that was introduced for stencil shadows in Section 8.4. For each silhouette edge, we calculate the plane that contains both the edge and the light position and add it to the shadow region. Because the planes of a visibility region have normal vectors that point inward, any plane $\mathbf{g}$ on the back side with respect to the light position $\mathbf{l}$ is actually facing toward the light and satisfies $\mathbf{g} \cdot \mathbf{l} > 0$. These back planes are also added to the shadow region to complete the convex hull.

To find silhouette edges, we first need the geometric information about our visibility region to be organized in a `Polyhedron` structure, which is accomplished by Listing 9.14. That code generates a polyhedron for which the vertices are numbered as shown in Figure 9.12. When creating a shadow region for a point light or spot light, we generally start with a polyhedron enclosing the entire view frustum. Even if an infinite projection matrix is being used, we need to choose some reasonable finite distance to the far plane so the polyhedron is closed. If we are rendering cascaded shadow maps for an infinite light, then we can build a separate polyhedron for the portion of the view frustum covered by each cascade by setting the near and far plane distances to the cascade range $[a_k, b_k]$. This allows us to create smaller cascade-specific shadow regions that minimize the number of objects rendered in each shadow map.

For a given polyhedron enclosing a visibility region, the code in Listing 9.18 constructs the set of boundary planes of the shadow region associated with a given 4D homogeneous light position $\mathbf{l}$. This code first classifies each plane $\mathbf{g}$ of the visibility region as being either on the front side of the polyhedron or back side. Since the plane normals point inward, $\mathbf{g} \cdot \mathbf{l} < 0$ for planes on the front side. Any planes that are classified as belonging to the back side of the polyhedron are added to the output shadow region. The code then loops over all of the polyhedron's edges and looks for silhouette edges, which are identified by having the property that they are shared by faces having different front or back classifications. A new shadow region plane is created for each silhouette edge, but we have to make sure its normal direction points inward. The vertices $\mathbf{v}_0$ and $\mathbf{v}_1$ referenced by an edge follow the rule that the are wound in counterclockwise order with respect to the first face referenced by the edge. If the first face happens to be classified as belonging to the front side of the polyhedron, then the plane normal $\mathbf{n}$ is given by

$$\mathbf{n} = \left(\mathbf{l}_{xyz} - l_w \mathbf{v}_0 \right) \times \left(\mathbf{v}_1 - \mathbf{v}_0 \right). \tag{9.19}$$

Listing 9.18. This function constructs a shadow region for the polyhedron and world-space light position specified by the `polyhedron` and `lightPosition` parameters. The boundary planes of the shadow region are stored in the array specified by `shadowPlane`, which must be large enough to hold `kMaxPolyhedronFaceCount` elements. The return value of the function is the number of planes that were generated.

```cpp
int32 CalculateShadowRegion(const Polyhedron *polyhedron, const Vector4D& lightPosition,
                            Plane *shadowPlane)
{
    const float kShadowRegionEpsilon = 1.0e-6F;
    bool frontArray[kMaxPolyhedronFaceCount];

    // Classify faces of polyhedron and record back planes.
    int32 shadowPlaneCount = 0;
    int32 cameraPlaneCount = polyhedron->planeCount;
    for (int32 i = 0; i < cameraPlaneCount; i++)
    {
        const Plane& plane = cameraPolyhedron->plane[i];
        frontArray[i] = (Dot(plane, lightPosition) < 0.0F);
        if (!frontArray[i]) shadowPlane[shadowPlaneCount++] = plane;
    }

    // Construct planes containing silhouette edges and light position.
    const Edge *edge = polyhedron->edge;
    int32 edgeCount = polyhedron->edgeCount;
    for (int32 i = 0; i < edgeCount; i++, edge++)
    {
        bool front = frontArray[edge->faceIndex[0]];
        if (front ^ frontArray[edge->faceIndex[1]])
        {
            // This edge is on the silhouette.
            const Point3D& v0 = polyhedron->vertex[edge->vertexIndex[0]];
            const Point3D& v1 = polyhedron->vertex[edge->vertexIndex[1]];
            Vector3D n = Cross(lightPosition.xyz() - v0 * lightPosition.w, v1 - v0);

            // Make sure plane is not degenerate.
            float m = SquaredMagnitude(n);
            if (m > kShadowRegionEpsilon)
            {
                // Normalize and point inward.
                n *= ((front) ? 1.0F : -1.0F) / sqrt(m);
                shadowPlane[shadowPlaneCount].Set(n, -Dot(n, v0));
                if (++shadowPlaneCount == kMaxPolyhedronFaceCount) break;
            }
        }
    }

    return (shadowPlaneCount);
}
```

Otherwise, if the first face is classified as belonging to the back side of the polyhedron, then the normal is negated. The plane $\mathbf{g}_{shadow}$ added to the shadow region is then equal to

$$\mathbf{g}_{shadow} = \frac{1}{\|\mathbf{n}\|}[\mathbf{n} \mid -\mathbf{n} \cdot \mathbf{v}_0]. \tag{9.20}$$

If the magnitude of $\mathbf{n}$ is very small because the points $\mathbf{v}_0$, $\mathbf{v}_1$, and $\mathbf{l}$ are nearly collinear, then the code skips the plane. This never causes a problem because planes for adjacent silhouette edges always meet near the same line.

When an infinite light source is behind the camera so that the view direction is pointing roughly the same way that the light is shining, the silhouette of the visibility region is made up of the four edges of the view frustum on the far plane or the far side of the most distance shadow cascade. Since an infinite light usually represents the sun or moon, this alignment occurs when the camera is looking in a generally downward direction, in which case the ground is likely to fill a large portion of the viewport. As shown in Figure 9.13(a), the associated shadow region can be much larger than necessary because it is accounting for a lot of space beneath the ground that is not visible. The magnitude of the problem is actually far greater than what can be portrayed in the limited space of the figure, and it would not be unexpected for the shadow region to include every shadow-casting object in the entire world. This, of course, creates a performance problem if it is not addressed in some way. Fortunately, there is an easy solution that comes to the rescue if we happen to know what the lowest point in the visible part of the world is. This lowest point can be set manually, or it can be derived from the most distant object found to be visible in the view frustum. If a horizontal plane $\mathbf{k}$ is established at the lowest point, then it can be used to clip the polyhedron corresponding to the view frustum's visibility region, as shown in Figure 9.13(b). The shadow region is then constructed from this clipped polyhedron, and it can be considerably smaller. When polyhedra representing separate shadow mapping cascades are clipped, some of them may be eliminated completely, saving us from rendering shadow maps that won't actually be used in a subsequent lighting pass. If the camera is not looking downward, and the plane $\mathbf{k}$ is not visible, then the polyhedron clipping process quickly determines that no work needs to be done, and no change is made to the visibility region.
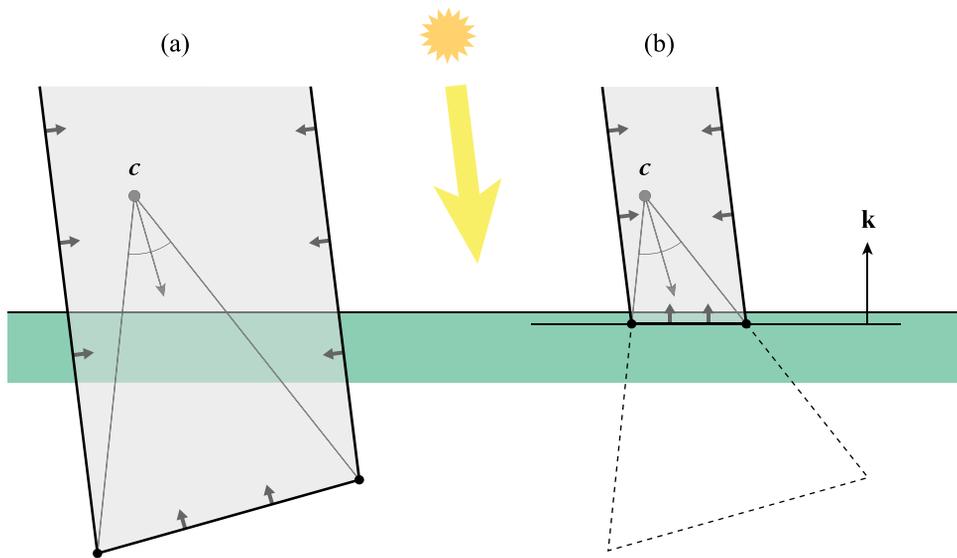
Figure 9.13. The camera at position c is looking roughly in the same direction as an infinite light shines. (a) Because the camera is pointed in a downward direction, the ground makes most of the space inside view frustum invisible, but the associated shadow region still accounts for all of that space. This can make it excessively large to the point that most or all of the objects in the world are eligible for shadow casting. (b) By clipping the view frustum's visibility region to a plane $\mathbf{k}$ representing the lowest point in the visible part of the world, the shadow region is reduced in size by a considerable amount.

9.7 Portal Systems

While testing bounding volumes for visibility is a fast operation, the time spent culling all of the objects outside the view frustum adds up, and this becomes a serious performance problem in large worlds that potentially contain millions of objects. Organizing the world into some kind of hierarchical structure such as an octree can usually reduce visibility testing from an $O(n)$ operation to an $O(\log n)$ operation, but there are many situations in which we can do better. Whenever a world is naturally divided into discrete enclosed areas, it's usually the case that adjacent areas are visible only through relatively small openings. This is certainly true for indoor environments where individual rooms and corridors are connected by doorways. The same kind of structure can often be found in outdoor environments as well on a significantly larger scale. If we supply information to our game engine about what objects belong to each area and how multiple areas are linked

together, then we can take advantage of that knowledge to avoid processing parts of the world that are completely extraneous for the current location of the camera. This leads to the development of a visibility determination technique known as a *portal system*.

9.7.1 Zones and Portals

In a portal system, the world is divided into areas called *zones*, and zones are connected to other zones through *portals*. This is demonstrated in Figure 9.14 for a small part of an indoor environment. The shape of each zone is allowed to be any convex volume, but simple boxes are usually adequate. Zones may be organized into a hierarchy so that a large zone can contain several smaller subzones. In this case, there is a root zone that encompasses the entire world, and all other zones are its descendants in a tree structure. An object belongs to a zone if it intersects the volume enclosed by the zone, and this is usually determined using the object's bounding volume.

When rendering with a portal system, the engine first identifies the zone that contains the camera. The objects belonging to that zone are tested for visibility using ordinary frustum culling methods. No consideration is given to objects outside the zone at this point, so the number of objects in the rest of the world has no effect whatsoever on the amount of processing that has to be done. The engine then goes through a list of portals that exit the current zone and determines if any of them are visible. For each portal that can be seen from the camera position, the zone to which the portal leads becomes qualified for rendering. All of the objects in this new zone are also tested for visibility, but they are tested against the smaller region of space produced by considering the boundary of the portal. The process continues recursively through additional portals exiting the connected zone until no more portals are visible. Through this algorithm, a portal system provides the capability of rendering enormously complex worlds because the time spent processing any parts that are not directly visible through a series of portals is zero.

Zones are not required to be disjoint, and it is often practical for them to overlap. Any object intersecting two different zones in the region where they overlap belongs to *both* zones. This is useful for geometry such as walls and doorways that make up the barrier between adjacent zones. In the case that a large zone enclosing an outdoor area has a subzone corresponding to an indoor area, objects making up the exterior of the structure belong to both the outdoor zone and the indoor zone. In general any object not fully contained in the union of zones at one level in the hierarchy must also belong to the parent zone. Because objects belonging to multiple zones can be visited multiple times during the portal traversal, they need to

(a)

(b)
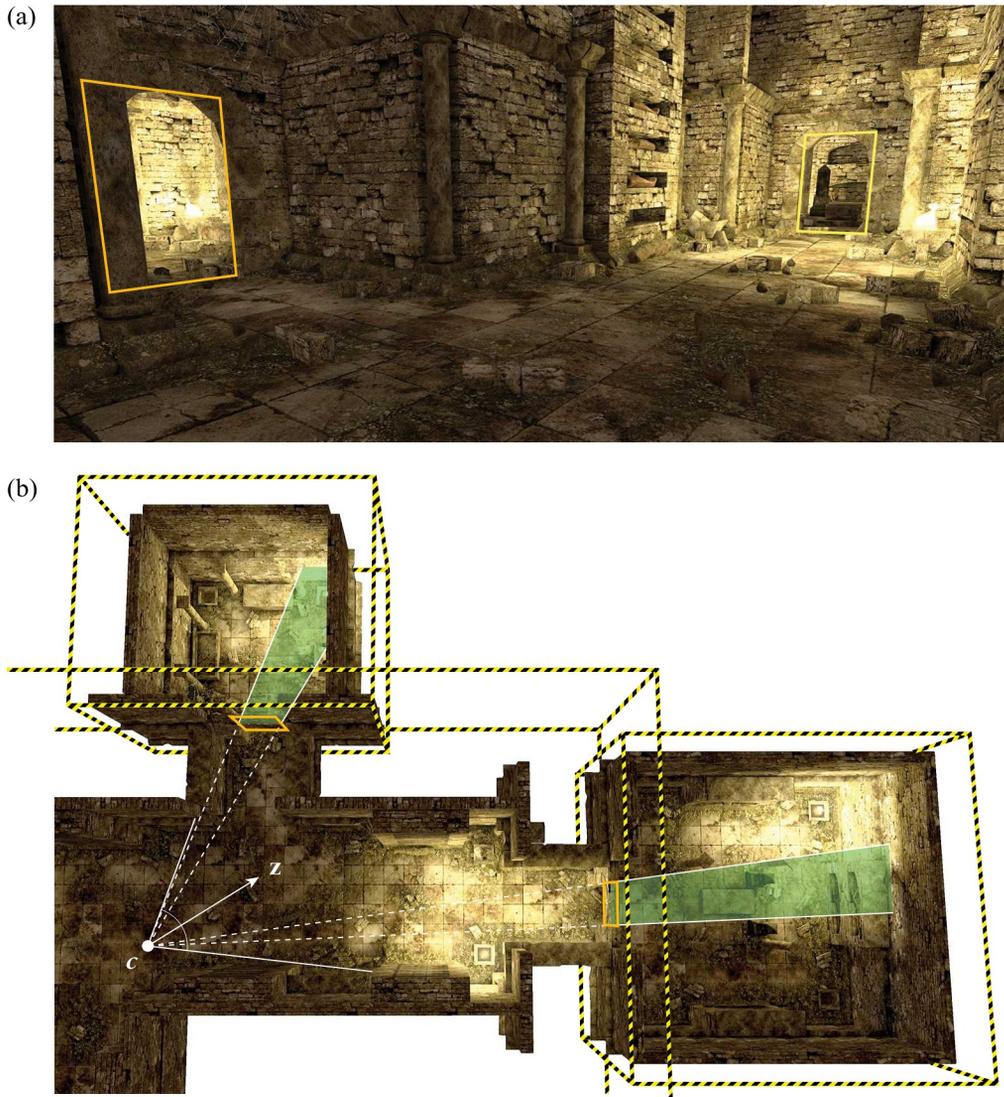
Figure 9.14. A portal system is used to determine what areas are visible in a crypt that has been divided into multiple zones. (a) From the perspective of the camera, two outgoing portals are visible, and their polygons are outlined in orange. (b) In a top view of the same scene, visibility regions created for the zones to which the portals lead are highlighted in green. The boundary planes of these regions are extrusions of the portal's edges from the camera position c. The yellow and black striped lines correspond to the edges of the zones, which overlap by a distance approximately equal to the thickness of the walls.

be tagged with some kind of index to prevent them from being rendered more than once. The index can simply be a counter that is incremented for each new zone traversal, and any object already tagged with the current index is skipped. It is also possible for the camera to be located in the space where two zones overlap. When this happens, traversals need to be initiated for both zones, but we set a flag for each one that prevents us from following a portal into one of the starting zones.

A portal is defined by a planar convex polygon with its vertices wound in the counterclockwise direction from its front side. This polygon represents the approximate boundary of the opening through which another zone is visible, as shown by two portals surrounding doorways in Figure 9.14(a). Portals are one-way windows, so if one portal belonging to zone *A* provides a link to zone *B*, then there must be a corresponding portal belonging to zone *B* that links back to zone *A*. This is the case because the two portals do not have to be in the same exact location. When zones overlap to contain a shared wall, for example, it's best to place the outgoing portals as far as possible into the opening between the two zones.

For each zone visited during a traversal, we create a visibility region representing the volume of space that can be seen from the camera position. Two of these visibility regions are shaded green in Figure 9.14(b), and their boundary planes are extruded from the portals through which they are visible. As a traversal proceeds, the *current* visibility region refers to the one created for the zone that we are currently processing. For the zones containing the camera position, the current visibility region is the whole view frustum. For other zones, the current visibility region is the smaller volume associated with the portal that led us there.

To determine whether a portal whose vertices lie in the world-space plane $\mathbf{f}_{portal}$ is visible from a camera position c, we first do a quick check that $\mathbf{f}_{portal} \cdot c > 0$ to make sure the camera is on the front side of the portal. If this succeeds, then we can test a bounding volume for the portal, either a sphere or a box of zero thickness, against the current visibility region. If the portal isn't culled by this test, then we clip the portal's polygon against all of the *lateral* planes of the current visibility region using the ClipPolygon() function in Listing 9.1. It's important that we exclude the near plane in this process so the clipped polygon is not too small, and the far plane can be omitted simply because it doesn't matter. This means that we must keep track of which planes are the near and far planes in each visibility region. They are always the last two planes in the visibility regions generated by code in this chapter.

If a portal is completely clipped away, then we ignore it. Otherwise, we have found a visible opening to another zone that needs to be processed. The new visibility region for that zone is created by extruding the clipped portal's vertices away from the camera position c to a plane $\mathbf{f}_{back}$ that is either the far plane or a plane

placed at the most distant boundary of the zone. Suppose the portal has n vertices $v_0, v_1, \ldots, v_{n-1}$ after clipping, and define the vector $\mathbf{u}_k = v_k - c$. For each portal vertex v_k, the corresponding extruded vertex w_k is then given by

$$w_k = c - \frac{\mathbf{f}_{\text{back}} \cdot c}{\mathbf{f}_{\text{back}} \cdot \mathbf{u}_k} \mathbf{u}_k, \tag{9.21}$$

which is produced by Equation (3.36) for the intersection of the line $\mathcal{L}(t) = c + t\mathbf{u}_k$ with the plane $\mathbf{f}_{\text{back}}$. Each pair of vertices v_k and w_k forms an edge between two of the n lateral planes of the new visibility region. The negation of the plane $\mathbf{f}_{\text{portal}}$ containing the portal defines the front side of the region, and the plane $\mathbf{f}_{\text{back}}$, oriented so its normal vector is $-\mathbf{z}$, defines the back side.

The function shown in Listing 9.19 constructs a polyhedron for the visibility region by extruding a portal. This polyhedron has $2n$ vertices, where half are the portal vertices $\{v_k\}$ and the other half are the extruded positions $\{w_k\}$. There are $3n$ edges, where the first group of n corresponds to the set of extrusions from v_j and w_j, the second group of n corresponds to the set of edges from v_j to $v_{(j+1) \bmod n}$ wound counterclockwise for the front side of the portal, and the third group of n corresponds to the set of edges from $w_{(j+1) \bmod n}$ to w_j wound counterclockwise for the back side of the region. There are $n + 2$ faces made up of n lateral faces, the front face, and the back face. Each lateral plane has an inward-pointing normal vector $\mathbf{n}$ given by

$$\mathbf{n} = \text{nrm}\left(\mathbf{u}_k \times \mathbf{u}_{(k-1) \bmod n}\right), \tag{9.22}$$

and since the plane must contain the camera position, it is given by $[\mathbf{n} \,|\, -\mathbf{n} \cdot c]$. To avoid exceeding the maximum number of faces m allowed in a polyhedron, a clipped portal must be limited to $m - 2$ vertices. The maximum number of edges defined in Listing 9.2 has been set to $3(m - 2)$ so enough space exists to handle any portal extrusion.

The reason we create a polyhedron for each zone's visibility region, instead of simply using an array of planes, is to make it possible to clip it against additional planes of external origin. As discussed in Section 9.6, clipping a visibility region against a ground plane can drastically reduce the size of a shadow region, and this idea can be extended to indoor environments. The part of a shadow cascade covering the range $[a_k, b_k]$ that intersects a zone can also be determined by clipping the zone's visibility region against planes perpendicular to the view direction at distances a_k and b_k from the camera. We use the clipped polyhedra for visibility testing and for the construction of shadow regions, but we do not use them to clip portals to avoid problems caused by the arbitrary orientations of the clipping

planes. A portal is always clipped using the original lateral planes of the visibility region generated by a preceding portal or, for a zone containing the camera, the lateral planes of the view frustum.

Listing 9.19. This function generates a polyhedron corresponding to a clipped portal with vertex-Count vertices with positions stored in the array specified by portalVertex. The portal plane $\mathbf{f}_{portal}$ and back plane $\mathbf{f}_{back}$ are specified by the portalPlane and backPlane parameters, and both should face the camera position specified by the cameraPosition parameter. The results are stored in the Polyhedron data structure supplied by the polyhedron parameter.

```cpp
void BuildPortalRegion(int32 vertexCount, const Point3D *portalVertex,
                       const Plane& portalPlane, const Plane& backPlane,
                       const Point3D& cameraPosition, Polyhedron *polyhedron)
{
    polyhedron->vertexCount = vertexCount * 2;
    polyhedron->edgeCount = vertexCount * 3;
    polyhedron->faceCount = vertexCount + 2;

    Point3D *v = polyhedron->vertex, *w = v + vertexCount;
    Plane *plane = polyhedron->plane;

    // Calculate lateral planes and vertex positions.
    float bc = Dot(backPlane, cameraPosition);
    Vector3D u0 = portalVertex[vertexCount - 1] - cameraPosition;
    for (int32 k = 0; k < vertexCount; k++)
    {
        Vector3D u1 = portalVertex[k] - cameraPosition;
        Vector3D normal = Normalize(Cross(u1, u0));
        plane[k].Set(normal, -Dot(normal, cameraPosition));

        v[k] = portalVertex[k];
        w[k] = cameraPosition - u1 * (bc / Dot(backPlane, u1));
        u0 = u1;
    }

    // Generate front and back planes.
    plane[vertexCount] = -portalPlane;
    plane[vertexCount + 1] = backPlane;

    // Generate all edges and lateral faces.
    int32 i = vertexCount - 1;
    Edge *edge = polyhedron->edge; Face *face = polyhedron->face;
    for (int32 j = 0; j < vertexCount; i = j++, edge++, face++)
    {
        int32 k = j + 1; k &= (k - vertexCount) >> 8;   // if j + 1 >= n, then k = 0.

        edge[0].vertexIndex[0] = uint8(j);
```

```
        edge[0].vertexIndex[1] = uint8(j + vertexCount);
        edge[0].faceIndex[1] = uint8(j);
        edge[0].faceIndex[0] = uint8(k);

        edge[vertexCount].vertexIndex[0] = uint8(j);
        edge[vertexCount].vertexIndex[1] = uint8(k);
        edge[vertexCount].faceIndex[1] = uint8(k);
        edge[vertexCount].faceIndex[0] = uint8(vertexCount);

        edge[vertexCount * 2].vertexIndex[0] = uint8(k + vertexCount);
        edge[vertexCount * 2].vertexIndex[1] = uint8(j + vertexCount);
        edge[vertexCount * 2].faceIndex[1] = uint8(k);
        edge[vertexCount * 2].faceIndex[0] = uint8(vertexCount + 1);

        face->edgeCount = 4;
        face->edgeIndex[0] = uint8(j);
        face->edgeIndex[1] = uint8(i + vertexCount * 2);
        face->edgeIndex[2] = uint8(i);
        face->edgeIndex[3] = uint8(i + vertexCount);
    }

    // Generate front and back faces.
    face[0].edgeCount = vertexCount;
    face[1].edgeCount = vertexCount;
    for (int32 k = 0; k < vertexCount; k++)
    {
        face[0].edgeIndex[k] = uint8(k + vertexCount);
        face[1].edgeIndex[k] = uint8(k + vertexCount * 2);
    }
}
```

9.7.2 Light Regions

By starting in a zone containing the camera and recursively following portals to additional zones, we naturally create a tree structure that reflects the relationships among the visibility regions. Every time a portal is visible in region R, the new region created on the other side of the portal becomes a child node of region R in the tree. The same kind of structure can be built for any position, and in particular, we can construct a region tree for a light source. The resulting collection of *light regions* tells us what volumes of space are illuminated due to the fact that each point they contain has a direct line of sight to the light's position through a series of portals. The illumination tree can be saved for each light source and rebuilt only when the light actually moves.

An example set of light regions for a point light at the position l is shown in Figure 9.15, and the volume of space directly visible to the light source is shaded
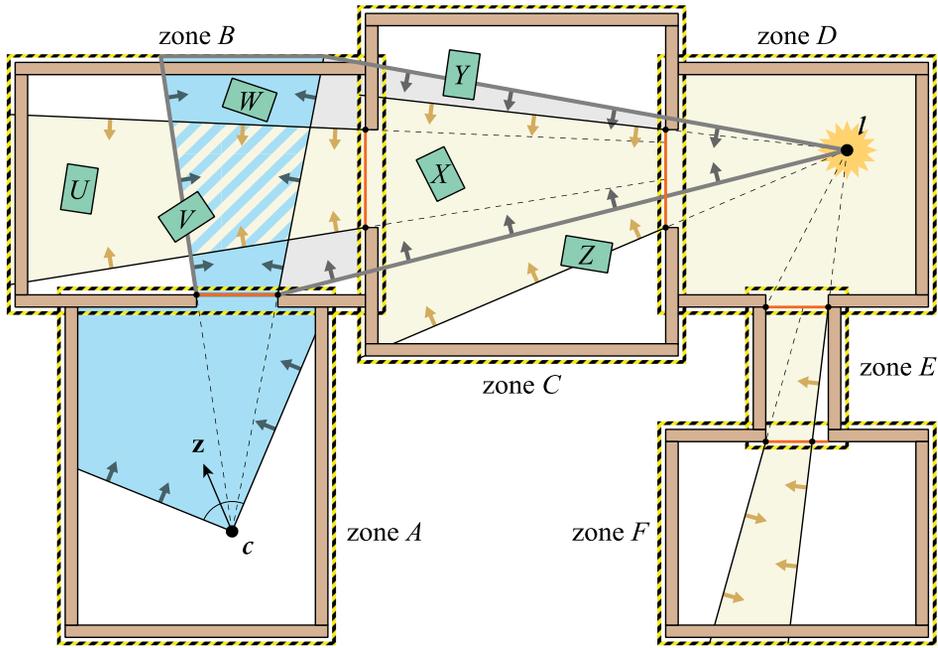
Figure 9.15. Visibility regions and light regions interact in a world that has been divided into several zones. The visibility regions corresponding to the camera at position c are shaded blue, and the light regions corresponding to the point light at position l are shaded yellow. The light is known to have an effect on the visible scene because a visibility region and light region intersect in zone B. Object U is not visible, object V is visible and illuminated, and object W is visible but not illuminated. The dark gray outline is the boundary of the shadow region. In zone C, object X casts a shadow, but objects Y and Z do not.

yellow. Since a point light is omnidirectional, there is no boundary to the volume it illuminates in zone D where it is located, and the same would be true for an infinite light. We can consider the root light region in this zone to be empty with no boundary planes. In the case of a spot light, the planes of the root light region are set to the pyramidal boundary of the light, as described in Section 9.5. To build the rest of the illumination tree, we proceed in the same way that we did for a visibility tree. We iterate over the portals exiting the current zone, clip them to the lateral planes of the current light region, and for any portals that are still visible, create new light regions in the connected zone by extruding the clipped polygons away from the light position. For point lights and spot lights, the extrusion can be performed by Equation (9.21) after substituting the light position l for the camera position c. To accommodate infinite lights, we can replace this equation with

$$\mathbf{w}_k = \mathbf{v}_k - \frac{\mathbf{f}_{back} \cdot \mathbf{v}_k}{\mathbf{f}_{back} \cdot \mathbf{u}_k} \mathbf{u}_k, \tag{9.23}$$

where $\mathbf{u}_k$ is now defined as

$$\mathbf{u}_k = l_w \mathbf{v}_k - \mathbf{l}_{xyz}, \tag{9.24}$$

and $\mathbf{l}$ is now a 4D homogeneous light position. For point and spot lights, it's possible for a portal to be out of range, which is something that doesn't happen when building a visibility tree. When considering a portal with the plane $\mathbf{f}_{portal}$, we first make sure that $\mathbf{f}_{portal} \cdot \mathbf{l} < r_{max}$, where r_{max} is the maximum range of the light source, because it is otherwise too far away for geometry on the other side of the portal receive any light.

Every time a new light region is created, it can be added to a list of light regions belonging to the zone for which it was made. This provides instant access to the full set of lights that have any effect on each zone, which limits the processing required to determine which light sources throughout the entire world actually need to be considered for the visible geometry. In Figure 9.15, the camera is located at the point c in zone A, and its visibility regions are shaded blue. There is only one light source in this example, and we know it does not affect any objects in zone A because there are no light regions in that zone. However, part of zone B is visible to the camera, and it does contain a light region, so we have to consider the possibility that visible objects in zone B are illuminated by the light. Only objects that intersect *both* the visibility region and the light region need to include a shading contribution from the light when they are rendered. Object V in the figure satisfies this condition and is therefore lit. Object W is visible to the camera, but it does not intersect the light region in zone B, so no lighting contribution is necessary.

For any visibility region that interacts with a light region, a shadow region should be constructed as described in Section 9.6 for the visibility region's entire boundary polyhedron and not for the smaller polyhedron corresponding to the volume where the two regions intersect. Using the full extent of the visibility region allows any geometry making up inter-zone barriers that happen to lie between the light source and the visibility region to cast shadows properly. In Figure 9.15, the shadow region constructed for the visibility region in zone B is depicted by the dark gray outline. If the smaller polyhedron had been used instead, then the walls between zones B and C may not have been included in the shadow. This could cause an object that extends outside the light region to unlit areas of the visibility region to receive light on surfaces that should be in shadow.

A zone that is not visible to the camera may still contain objects that cast shadows into areas that we can see. All of these objects must belong to zones occurring

on the branch of the illumination tree between the zone visible to the camera and the zone containing the light source. To determine what objects need to cast shadows, we walk up the tree until we reach the root node and consider the objects in each zone visited along the way. In the example of Figure 9.15, we would visit zones C and D during this process. In each of these zones, we look for objects that intersect the shadow region *and* any light region generated for the light source. (There could be more than one light region in the zone for the same light source if light can enter through multiple portals.) Objects that intersect both regions are added to the set of shadow castors. In the figure, object X satisfies these conditions and must cast a shadow. However, even though object Y intersects the shadow region, it does not need to cast a shadow because it does not intersect the light region in zone C. Object Z does intersect the light region, but it lies outside the shadow region, so it also does not need to cast a shadow.

9.8 Occluders

Some game worlds, especially those with wide open outdoor areas, contain many places with irregular shapes that are not a good fit for zones. Although it may not be practical to use a portal system everywhere in these worlds, it is often still the case that there are large geometric structures blocking the visibility of objects behind them. These structures provide culling opportunities that we can take advantage of by placing invisible *occluders* inside things like buildings and terrain.

An occluder is generally any simple shape that can be used by the game engine to quickly determine whether objects are completely blocked from view. One of the most basic examples of an occluder is a convex polygon called an *antiportal*. An antiportal is extruded away from the camera in exactly the same way that a portal is to form a convex volume bounded by inward-pointing planes, but now we call the extrusion an *occlusion region* instead of a visibility region. Whereas an object is visible if any part of it intersects a visibility region, an object is occluded only if it lies completely inside an occlusion region. An antiportal can be designed to function as an occluder for camera positions on either side of the plane containing its vertices, and we just have to remember to account for the different winding directions. To avoid culling objects that are closer to the camera than the antiportal, the plane must also be added to the occlusion region's boundary, after negating if necessary to make it point away from the camera.

Since an antiportal has no thickness, it is not a very effective occluder unless it is viewed from a direction largely perpendicular to its plane. Better results are usually attained by filling the volume inside view-blocking geometry to the greatest extent possible with a *box occluder*. The advantage to using a box occluder is

that it can often provide a sizeable cross section from any view direction. To construct an occlusion region for a box occluder, we identify the edges on the box's silhouette from the perspective of the camera. As shown in Figure 9.16, those edges are extruded away from the camera position to form the lateral bounding planes of the occlusion region. The planes corresponding to any front faces of the box must also be added to the boundary of the occlusion region to prevent culling in front of the box. To test whether an object is occluded, we calculate dot products between each plane $\mathbf{g}$ belonging to the occlusion region and the center $\mathbf{p}$ of the object's bounding volume. If $\mathbf{g} \cdot \mathbf{p} > r_{\mathbf{g}}$ for *every* plane $\mathbf{g}$, where $r_{\mathbf{g}}$ is the effective radius of the bounding volume with respect to $\mathbf{g}$, then the object is completely inside the occlusion region, and it can thus be culled.

Before a box occluder is used for culling, it should be tested for visibility against the view frustum, or if portals are in use, against the visibility regions for the zone containing it. If the box occluder is not visible, then any objects that it could occlude would be very likely to fail the visibility test themselves, so any work spent culling them through the occluder would be redundant. Another advantage to using box occluders is that they can be tested for visibility using the same methods applied to the bounding boxes of ordinary geometric objects.

For the purposes of rendering a stencil shadow in Section 8.4 and constructing a shadow region in Section 9.6, the silhouette edges of an object were identified by classifying each face as pointing toward or away from the camera and then collecting the edges between faces having opposite classifications. Such a general method
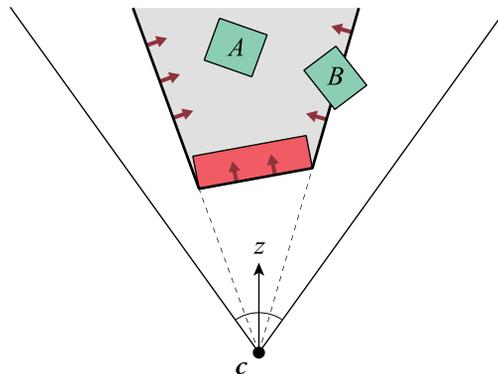
Figure 9.16. The occlusion region, shown in gray, is generated from a box occluder, shown in red. The inward-pointing planes bounding the occlusion region are derived from the front faces of the occluder and the extrusion of the occluder's silhouette with respect to the camera position c. Object A is culled because it lies completely inside the occlusion region, but object B is not culled because it is only partially inside.

was necessary for the arbitrary triangle meshes and convex polyhedra that could arise in those settings, but the consistent geometry of a box occluder allows us to find the silhouette by faster and more clever means. As illustrated in Figure 9.17, there are only three possible types of silhouette for a box, and these correspond to one, two, or three sides of the box facing toward the camera. In the case that only one side faces the camera, as shown in Figure 9.17(a), the silhouette is always composed of the four edges belonging to that one side. In the cases that two or three sides face the camera, as shown in Figures 9.17(b) and 9.17(c), the silhouette is always composed of six edges. An exhaustive list of all possible silhouettes can be assembled by observing that there are exactly 6 choices for the single front-facing side in Figure 9.17(a), exactly 12 choices for the shared edge between the two front-facing sides in Figure 9.17(b), and exactly 8 choices for the shared vertex among the three front-facing sides in Figure 9.17(c). The actual silhouette with respect to any given camera position is always one of these 26 cases.

(a)   (b)   (c)
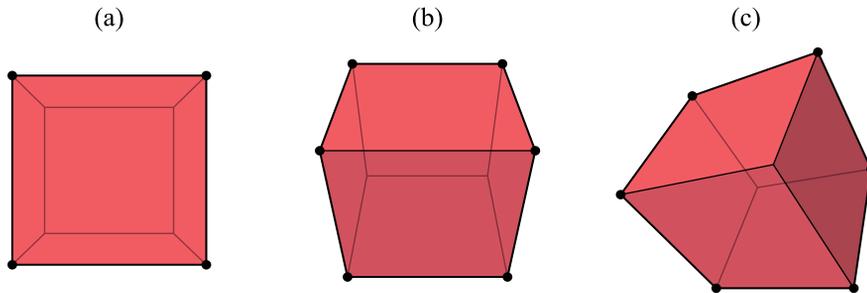
Figure 9.17. A box occluder can have one, two, or three front faces, and its silhouette can have either four or six sides. (a) There is one front face, and its edges form the four sides of the silhouette. (b) There are two front faces and six silhouette edges. (c) There are three front faces and six silhouette edges.

If we can efficiently determine which of the 26 cases applies, then we can simply fetch the corresponding silhouette from a table. Fortunately, this is easy to do in the object-space coordinate system of the box occluder where it is aligned to the axes and has one corner at the origin, as shown in Figure 9.18. We assign a size $\mathbf{s}$ to the box so that it extends in the positive directions along the x, y, and z axes to the maximum coordinates s_x, s_y, and s_z. Given transformation matrices $\mathbf{M}_{camera}$ and $\mathbf{M}_{occluder}$ from object space to world space for the camera and the box occluder, the position c of the camera in the occluder's object space is

$$c = \mathbf{M}_{occluder}^{-1} \mathbf{M}_{camera[3]}. \tag{9.25}$$
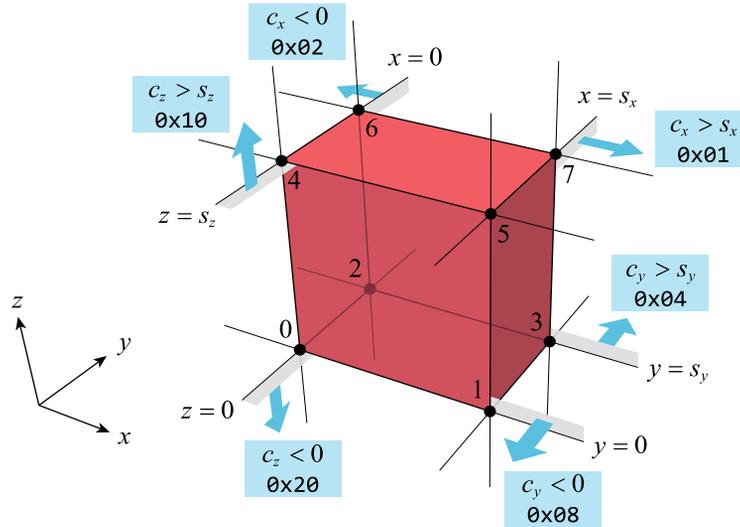
Figure 9.18. A box occluder of size **s** is aligned to the coordinate axes, and its vertices are numbered 0 to 7 as shown. A 6-bit occlusion code N used as a lookup table index is calculated by adding the one-bit constants associated with each condition satisfied by the camera position c.

Each of the three coordinates c_i the camera position falls into one of three distinct states in comparison to the extents of the box: $c_i > s_i$, $c_i < 0$, or neither. In combination, this gives rise to a total of $3^3 = 27$ possibilities, 26 of which correspond to the 26 possible silhouettes. The remaining case corresponds to the interior of the box, but we do not perform any occlusion if the camera happens to be there.

In order to calculate an occlusion code N that serves an index for a lookup table, we assign the single-bit numerical codes shown in Figure 9.18 to each of the camera position states to first obtain

$$n_x = \begin{cases} 1, & \text{if } c_x > s_x; \\ 2, & \text{if } c_x < 0; \\ 0, & \text{otherwise,} \end{cases} \quad n_y = \begin{cases} 4, & \text{if } c_y > s_y; \\ 8, & \text{if } c_y < 0; \\ 0, & \text{otherwise,} \end{cases} \quad n_z = \begin{cases} 16, & \text{if } c_z > s_z; \\ 32, & \text{if } c_z < 0; \\ 0, & \text{otherwise.} \end{cases} \quad (9.26)$$

The full 6-bit occlusion code is then produced by adding these values together (or logically ORing them together) to get

$$N = n_x + n_y + n_z. \quad (9.27)$$

The number of bits set in N is equal to the number of sides that face toward the camera. Because the bit associated with one side of the box is mutually exclusive with the bit associated with the opposite side, not all 6-bit codes can occur. If we sum the larger of the pairs of values that could be assigned to n_x, n_y, and n_z, then we see that the maximum index is 42, so our lookup table only needs to contain 43 entries of which 26 correspond to valid cases.

The lookup table provides a vertex count, which must be either four or six, and a list of the vertices making up the box occluder's silhouette. Vertices are identified by the numbering shown in Figure 9.18, and a separate table provides normalized vertex coordinates for a unit cube. Those coordinates are scaled by the size **s** of the box and transformed into camera space. From the perspective of the camera, the vertices making up the silhouette form a convex polygon. Even though the polygon is not necessarily planar, we can safely clip it against the four lateral frustum planes listed in Table 6.1 to remove extraneous off-screen edges. The edges of the resulting polygon are extruded away from the origin in camera space and finally transformed into world space.

As shown in Figure 9.19(a), the clipping process can introduce silhouette edges that make the occlusion region smaller than it would have been if no clipping had been performed. However, we do not need create an occlusion region boundary plane for any edges that lie in one of the lateral frustum planes because any part of an object that extends past such a plane would also extend outside the view frustum and thus would not be visible anyway. Eliminating boundary planes for these edges actually allows us to construct a much larger occlusion region, as demonstrated in Figure 9.19(b). The additional volume can cause objects that are partially outside the view frustum to be culled even though they would not be culled if the whole box occluder was visible.

The code in Listing 9.20 constructs the occlusion region for a box occluder of a given size by classifying the camera position and looking up the box's silhouette in a table. As the occlusion code is being calculated, the planes corresponding to the front faces of the box are output in world space using the rows of the matrix $\mathbf{M}_{occluder}^{-1}$. (See Exercise 8.) The occlusion code is then used to look up an 8-bit value from a small table. The high three bits contain a vertex count, which is always four or six in valid cases, and the low five bits contain the silhouette index, which is in the range 0 to 25. The silhouette index is used to look up an array of vertex indices for the silhouette's polygon in a separate table. The vertex indices are assigned to the corners of the box as shown in Figure 9.18, and they are used to look up the corresponding vertex coordinates of a unit cube in yet another table. Each vertex is finally transformed into camera space through multiplication by the matrix $\mathbf{M}_{camera}^{-1}\mathbf{M}_{occluder}$.
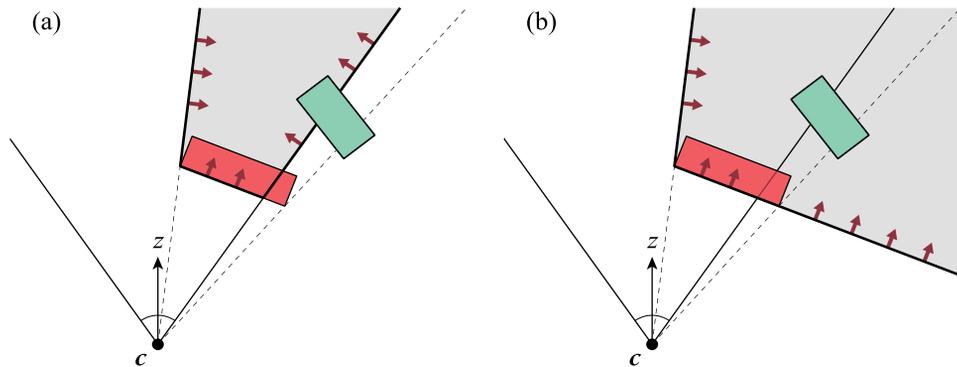
Figure 9.19. (a) A box occluder, shown in red, has a silhouette that is clipped by the right frustum plane. If the entire clipped polygon is extruded to form the occlusion region, then the green object is not culled even though the part of it inside the view frustum is occluded. (b) The edge of the clipped silhouette lying in the right frustum plane is eliminated and does not generate a boundary plane for the occlusion region. The occlusion region now includes a much larger off-screen volume, and the green object is culled.

The silhouette's polygon is clipped in camera space against the four lateral planes of the view frustum using the `ClipPolygon()` function in Listing 9.1. The polygon can begin with six vertices, and it's possible that up to four more are added during the clipping process, so the final clipped polygon has a maximum of ten vertices. For each pair of consecutive vertices v_1 and v_2, we calculate an inward-pointing normal direction

$$\mathbf{n} = \mathrm{nrm}(v_2 \times v_1). \tag{9.28}$$

The occlusion region bounding plane generated by the edge is then $[\mathbf{n} \,|\, 0]$ in camera space. The signed distances from both v_1 and v_2 to all four of the lateral frustum planes are calculated so we can eliminate edges that lie in any one of those planes. If the distances from v_1 and v_2 to the same frustum plane $\mathbf{f}_{camera}$ are both less than a small positive constant ε, then the edge connecting those two vertices may be rejected. However, we must be careful to reject such an edge only when $\mathbf{f}_{camera} \cdot \mathbf{n} > 0$ because this ensures that the occlusion region is being extended outside the view frustum. If $\mathbf{f}_{camera} \cdot \mathbf{n} \leq 0$, then only a small sliver of the silhouette polygon lies inside the view frustum, and removing the edge would incorrectly extend the occlusion region into areas that should be visible.

For each edge of the clipped silhouette that is not rejected, the extruded occlusion region bounding plane $\mathbf{g}$ is given by

$$\mathbf{g} = [\mathbf{n} \,|\, 0] \, \mathbf{M}_{camera}^{-1} \tag{9.29}$$

in world space. Theoretically, there should be no more than six extruded planes because edges introduced by clipping should be rejected. But to account for all possible floating-point precision shenanigans, we must be prepared to handle up to ten extruded planes. There can also be up to three front-facing planes generated for the sides of the box occluder, so the total number of bounding planes output can be as large as 13. At the opposite extreme, if the entire viewport is filled by one side of a box occluder, then no extruded planes are generated, and we are left with only one front-facing plane in the output. In the final case that the silhouette polygon is completely clipped away or the camera is inside the box, zero planes are generated, and no occlusion should be performed.

Listing 9.20. This function constructs the occlusion region for the box occluder having the dimensions given by the `size` parameter. The `frustumPlane` parameter must point to an array holding the four camera-space lateral frustum planes listed in Table 6.1 (in any order). The matrix `Mocc` transforms from the object space of the occluder to world space, and the matrix `Mcam` transforms from camera space to world space. The world-space planes bounding the occlusion region are written to the storage pointed to by `occluderPlane`, which must be large enough to hold 13 elements. The return value is the number of planes that were generated.

```
const uint8 occluderPolygonIndex[43] =     // 3-bit vertex count, 5-bit polygon index.
{
    0x00, 0x80, 0x81, 0, 0x82, 0xC9, 0xC8, 0, 0x83, 0xC7, 0xC6, 0, 0, 0, 0, 0,
    0x84, 0xCF, 0xCE, 0, 0xD1, 0xD9, 0xD8, 0, 0xD0, 0xD7, 0xD6, 0, 0, 0, 0, 0,
    0x85, 0xCB, 0xCA, 0, 0xCD, 0xD5, 0xD4, 0, 0xCC, 0xD3, 0xD2
};

const uint8 occluderVertexIndex[26][6] =  // Vertex indices for all 26 polygons.
{
    {1, 3, 7, 5}, {2, 0, 4, 6}, {3, 2, 6, 7}, {0, 1, 5, 4}, {4, 5, 7, 6}, {1, 0, 2, 3},
    {2, 0, 1, 5, 4, 6}, {0, 1, 3, 7, 5, 4}, {3, 2, 0, 4, 6, 7}, {1, 3, 2, 6, 7, 5},
    {1, 0, 4, 6, 2, 3}, {5, 1, 0, 2, 3, 7}, {4, 0, 2, 3, 1, 5}, {0, 2, 6, 7, 3, 1},
    {0, 4, 5, 7, 6, 2}, {4, 5, 1, 3, 7, 6}, {1, 5, 7, 6, 4, 0}, {5, 7, 3, 2, 6, 4},
    {3, 1, 5, 4, 6, 2}, {2, 3, 7, 5, 4, 0}, {1, 0, 4, 6, 7, 3}, {0, 2, 6, 7, 5, 1},
    {7, 6, 2, 0, 1, 5}, {6, 4, 0, 1, 3, 7}, {5, 7, 3, 2, 0, 4}, {4, 5, 1, 3, 2, 6}
};

const float occlusionVertexPosition[8][3] =  // Normalized vertex coords for unit cube.
{
    {0.0F, 0.0F, 0.0F}, {1.0F, 0.0F, 0.0F}, {0.0F, 1.0F, 0.0F}, {1.0F, 1.0F, 0.0F},
    {0.0F, 0.0F, 1.0F}, {1.0F, 0.0F, 1.0F}, {0.0F, 1.0F, 1.0F}, {1.0F, 1.0F, 1.0F}
};

int32 MakeOcclusionRegion(const Vector3D& size, const Plane *frustumPlane,
                const Transform4D& Mocc, const Transform4D& Mcam, Plane *occluderPlane)
{
```

```cpp
Point3D      polygonVertex[2][10];
float        vertexDistance[2][4];
float        vertexLocation[10];

const float kOccluderEpsilon = 0.002F;
uint32 occlusionCode = 0;
int32 planeCount = 0;

// Transform the camera position into the occluder's object space.
Transform4D m = Inverse(Mocc);
Point3D cameraPosition = m * Mcam.GetTranslation();

// Calculate 6-bit occlusion code and generate front planes.
uint32 axisCode = 0x01;
for (int32 i = 0; i < 3; i++, axisCode <<= 2)
{
    if (cameraPosition[i] > size[i])
    {
        occlusionCode |= axisCode;
        occluderPlane[planeCount++].Set(-m(i,0), -m(i,1), -m(i,2), size[i] - m(i,3));
    }
    else if (cameraPosition[i] < 0.0F)
    {
        occlusionCode |= axisCode << 1;
        occluderPlane[planeCount++].Set(m(i,0), m(i,1), m(i,2), m(i,3));
    }
}

// Look up silhouette polygon with occlusion code.
uint32 polygonIndex = occlusionPolygonIndex[occlusionCode];
const uint8 *vertexIndex = occlusionVertexIndex[polygonIndex & 0x1F];
int32 vertexCount = polygonIndex >> 5;

// Generate silhouette vertices in camera space.
Transform4D McamInverse = Inverse(Mcam);
Transform4D t = McamInverse * Mocc;
for (int32 i = 0; i < vertexCount; i++)
{
    const float *p = occlusionVertexPosition[vertexIndex[i]];
    polygonVertex[1][i] = t * Point3D(p[0] * size.x, p[1] * size.y, p[2] * size.z);
}

// Clip silhouette to lateral planes of view frustum.
const Point3D *vertex = polygonVertex[1];
for (int32 k = 0; k < 4; k++)
{
    const Plane& plane = frustumPlane[k];
    Point3D *result = polygonVertex[k & 1];
    vertexCount = ClipPolygon(vertexCount, vertex, plane, vertexLocation, result);
    vertex = result;
}
```

```
    if (vertexCount < 3) return (0);

    // Generate occlusion region planes in world space.
    const Vector3D *v1 = &vertex[vertexCount - 1];
    for (int32 k = 0; k < 4; k++)
        vertexDistance[0][k] = Dot(frustumPlane[k].GetNormal(), *v1);

    for (int32 i = 0; i < vertexCount; i++)
    {
        bool cull = false; int32 j = i & 1;
        const Vector3D *v2 = &vertex[i];
        Vector3D planeNormal = Normalize(Cross(*v2, *v1));
        for (int32 k = 0; k < 4; k++)
        {
            const Vector3D& frustumNormal = frustumPlane[k].GetNormal();
            float d = Dot(frustumNormal, *v2);
            vertexDistance[j ^ 1][k] = d;

            // Cull edge lying in frustum plane, but only if its extrusion points inward.
            if ((fmax(d, vertexDistance[j][k]) < kOccluderEpsilon) &&
                (Dot(planeNormal, frustumNormal) > 0.0F)) cull = true;
        }

        if (!cull) occluderPlane[planeCount++] = Plane(planeNormal, 0.0F) * McamInverse;
        v1 = v2;
    }

    return (planeCount);
}
```

9.9 Fog Occlusion

When fog is applied to a scene using the methods presented in Section 8.5, the fog factor can become so small that the light originating from an object is no longer perceptible when it reaches the camera. Objects that are completely fogged out in this way do not need to be rendered, and it would be advantageous to cull them before they consume any GPU resources. First, we need to define what it is that we consider to be imperceptible. We can certainly say that a color C_{shaded} cannot be displayed if it is multiplied by a fog factor f that is smaller than the smallest non-zero intensity level that could be stored in the frame buffer. For example, if each channel in the frame buffer uses eight bits of precision, then any fog factor less than $f_{\text{min}} = 1/256$ will cause the product $C_{\text{shaded}}f$ to become zero. If the fog color C_{fog} is bright, then we can get away with using larger values of f_{min} because the human eye cannot discern small variations in bright colors as well as it can in dark

colors. In any case, we simply choose a value of f_{min} that works well and solve Equation (8.107) to determine the corresponding maximum optical depth

$$\tau_{max} = -\ln f_{min}. \tag{9.30}$$

For fog having a constant density α_{ex}, we then know that any object further from the camera than the distance $d = \tau_{max}/\alpha_{ex}$ is not visible. For simplicity, we can place a single occlusion plane at the distance d in front of the camera and cull objects having a bounding volume that lies completely beyond it.

In the case of halfspace fog using a linear density function, the optical depth $\tau(\mathbf{p})$ inside the fog is established by Equation (8.112). After making the definitions $z_0 = \mathbf{f} \cdot \mathbf{c}$ and $z = \mathbf{f} \cdot \mathbf{p}$, this equation becomes

$$\tau(z) = -\frac{\alpha_0}{2}\|\mathbf{v}\|(z_0 + z), \tag{9.31}$$

and it now expresses τ as a function of the signed distance z between the point $\mathbf{p}$ being rendered and the fog plane $\mathbf{f}$. Suppose that we have chosen a maximum value τ_{max} of the optical depth and that we know the vertical position z_0 where the camera is located beneath the fog plane. We can determine a distance function $d(\varphi)$ from the camera to a vertical position z on the occlusion boundary, where φ is the angle of depression relative to z_0, by first expressing z as

$$z = z_0 - d(\varphi)\sin\varphi, \tag{9.32}$$

and then plugging it into Equation (9.31) to get

$$\tau_{max} = -\frac{\alpha_0}{2}d(\varphi)(2z_0 - d(\varphi)\sin\varphi). \tag{9.33}$$

Solving this for a positive value of $d(\varphi)$ gives us

$$d(\varphi) = \frac{z_0 + \sqrt{z_0^2 + \sigma\sin\varphi}}{\sin\varphi}, \tag{9.34}$$

where we have defined the constant

$$\sigma = \frac{2\tau_{max}}{\alpha_0}. \tag{9.35}$$

Plotting the function $d(\varphi)$ for various values of z_0 produces the blue curves shown in Figure 9.20, and they can be classified into three groups that we discuss
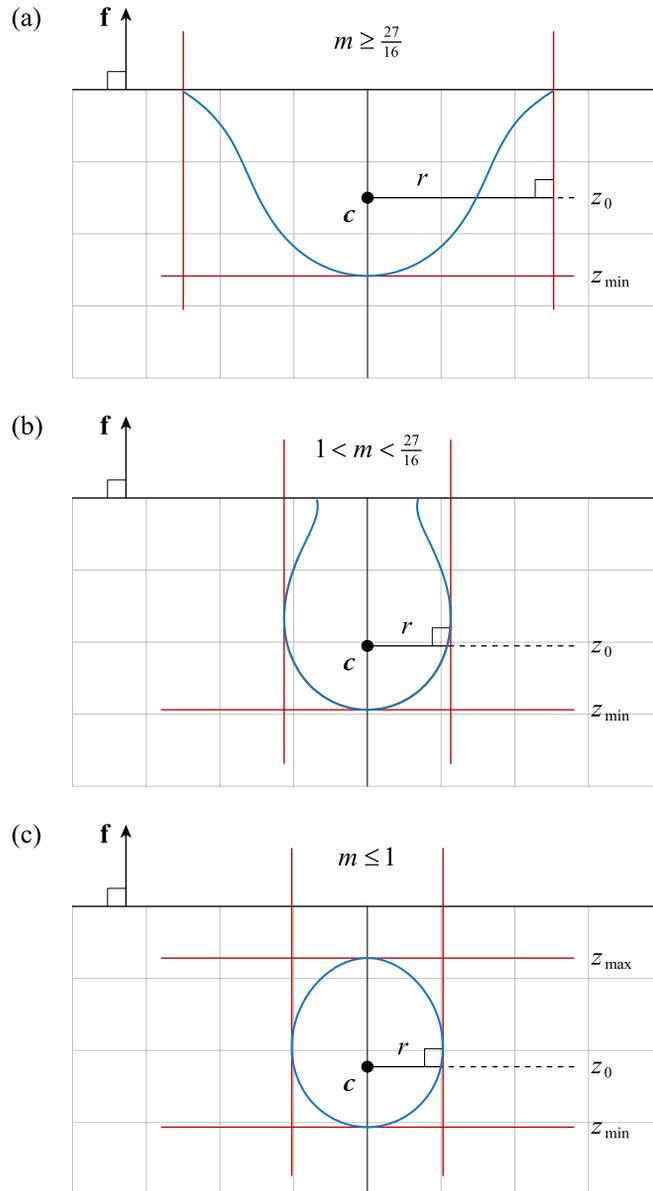
Figure 9.20. For a camera having a vertical position z_0 beneath the fog plane **f**, the blue curves represent the exact surfaces beyond which objects are fully occluded by fog having an optical depth determined by a linear density function. The red lines represent the planes parallel and perpendicular to **f** coinciding with the maximum extents of the surfaces.

below. The 2D surface at which the optical depth τ_{max} is reached is represented by the revolution of any one of these curves about the vertical axis, and an object that lies completely beyond that surface is fully occluded by the fog. Culling objects directly against this analytical occlusion surface would be impractical, however, so we instead make it our goal to determine planes parallel and perpendicular to the fog plane that coincide with the maximum extents of the occlusion surface, as shown by the red lines in the figure.

We first consider the planes that are tangent to the occlusion surface and parallel to the fog plane. One of these planes always exists beneath the camera position, as shown for all three cases in Figure 9.20, because the fog always becomes dense enough to occlude objects when we are looking downward. If the camera is sufficiently deep inside the fog volume, then there is also a tangent plane above the camera, as shown only for case (c). To determine the positions of these tangent planes for a given optical depth τ_{max}, we write Equation (9.31) as

$$\tau_{max} = -\frac{\alpha_0}{2}|z_0 - z|(z_0 + z),\tag{9.36}$$

where $\|\mathbf{v}\|$ has been replaced by the absolute vertical distance between the camera position c and the point p being rendered. We assume that the camera is located inside the fog volume and thus $z_0 \le 0$. If the camera is located above the fog plane, then the position of the occluding plane beneath the camera is the same as it would be for $z_0 = 0$ because cameras in both positions would look downward through the same amount of fog.

For the plane below the camera, the value of $z_0 - z$ is always positive, so we can remove the absolute value in Equation (9.36) and solve for z to obtain

$$z_{min} = -\sqrt{z_0^2 + \sigma},\tag{9.37}$$

where σ is the same constant that was previously defined by Equation (9.35). We have labeled the result as the minimum value of z because it represents the most negative z coordinate at which any object could be visible. For the plane above the camera, the value of $z_0 - z$ is always negative, so the right side of Equation (9.36) must be negated when we remove the absolute value. Solving for z in this case gives us

$$z_{max} = -\sqrt{z_0^2 - \sigma},\tag{9.38}$$

and this result corresponds to the maximum z coordinate at which any object could be visible. This maximum value and the associated plane above the camera exist

only when $z_0^2 \geq \sigma$ because we would otherwise have a negative value inside the radical. Following the convention that occlusion planes have normals that point into the occluded region of space, the components of the planes corresponding to $z_{\min}$ and $z_{\max}$ are given by

$$\mathbf{g}_{\min} = \left(-f_x, -f_y, -f_z, z_{\min} - f_w\right)$$
$$\mathbf{g}_{\max} = \left(f_x, f_y, f_z, f_w - z_{\max}\right). \tag{9.39}$$

The code in Listing 9.21 calculates these planes for a given camera position c, fog attenuation coefficient α_0, and maximum optical depth $\tau_{\max}$.

We now consider an occlusion plane perpendicular to the fog plane that represents the maximum horizontal distance beyond which any object is completely fogged out. We use the letter r for this distance because it actually defines a cylinder of radius r outside which nothing in the fog volume can be seen. The derivation below is almost entirely dedicated to calculating the correct value of r, and it may

Listing 9.21. For a world-space fog plane given by `fogPlane` and a world-space camera position given by `cameraPosition`, this function calculates the occlusion planes that are parallel to the fog plane and stores them in the array specified by `occlusionPlane`, which must provide storage for two planes. The linear fog attenuation coefficient α_0 is given by `fogDensity`, and the maximum optical depth $\tau_{\max}$ is given by `maxOpticalDepth`. The return value of the function is the number of occlusion planes and is always one or two.

```
int32 CalculateParallelFogOcclusionPlanes(const Plane& fogPlane,
                        const Point3D& cameraPosition, float fogDensity,
                        float maxOpticalDepth, Plane *occlusionPlane)
{
    float z0 = Dot(fogPlane, cameraPosition);
    float z02 = z0 * z0;
    float sigma = 2.0F * maxOpticalDepth / fogDensity;

    // Calculate the plane below the camera, which always exists.
    float zmin = -sqrt(z02 + sigma);
    occlusionPlane[0].Set(-fogPlane.GetNormal(), zmin - fogPlane.w);

    // Return if the plane above camera does not exist.
    if (z02 < sigma) return (1);

    // Calculate the plane above the camera.
    float zmax = -sqrt(z02 - sigma);
    occlusionPlane[1].Set(fogPlane.GetNormal(), fogPlane.w - zmax);
    return (2);
}
```

be practical for some applications to test for occlusion directly against the resulting cylinder. However, we will ultimately determine an occlusion plane that is tangent to the cylinder and perpendicular to the projection of the camera's view direction onto the fog plane. In any case, the camera position must be beneath the fog plane with $z_0 < 0$ in order for a maximum distance r to exist.

The calculation of r works out best when we express z as a multiple of z_0 and make the substitution

$$z = uz_0. \tag{9.40}$$

For a given maximum optical depth τ_{max}, Equation (9.31) is then written as

$$\tau_{max} = -\frac{\alpha_0}{2} \|\mathbf{v}\| (u+1) z_0. \tag{9.41}$$

The distance $\|\mathbf{v}\|$ between the camera position c and the point p being rendered forms the hypotenuse of the right triangle shown in Figure 9.21, and the lengths of the other two sides are the distance r and the difference $|z_0 - z|$ of the vertical positions relative to the fog plane. By using the relationship

$$\|\mathbf{v}\|^2 = r^2 + (1-u)^2 z_0^2, \tag{9.42}$$

we can write the optical depth as

$$\tau_{max} = -\frac{\alpha_0}{2} (u+1) z_0 \sqrt{r^2 + (1-u)^2 z_0^2}. \tag{9.43}$$
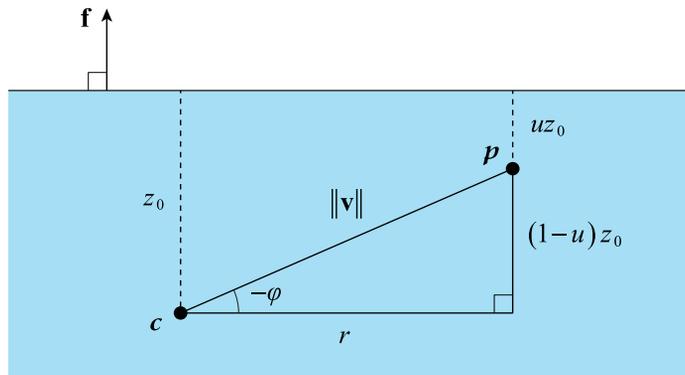
Figure 9.21. The distance $\|\mathbf{v}\|$ between the camera position c and the point p being rendered forms the hypotenuse of a right triangle in which $\|\mathbf{v}\|^2 = r^2 + (1-u)^2 z_0^2$.

Solving Equation (9.43) for r^2 gives us

$$r^2 = \frac{\sigma^2}{(u+1)^2 z_0^2} - (1-u)^2 z_0^2, \tag{9.44}$$

and this is an equation relating the distance r at which full occlusion occurs directly to the factor u corresponding to the value of z at the point being rendered. To find the maximum value of r attained over all values of u, we take a derivative of the right side of this equation and set it equal to zero. After some algebraic simplification, doing so brings us to the quartic polynomial function

$$h(u) = u^4 + 2u^3 - 2u + m - 1 = 0, \tag{9.45}$$

where the constant m is defined as

$$m = \frac{\sigma^2}{z_0^4} = \frac{4\tau_{max}^2}{\alpha_0^2 z_0^4}. \tag{9.46}$$

A typical plot of $h(u)$ is shown in Figure 9.22. If it exists, the largest root of $h(u)$ corresponds to the vertical position at which r^2 reaches a local maximum. To determine the value of this root, we start by making some observations about the function $h(u)$. First, the value of $h(1)$ is always m, which is a positive number. Second, if we take a derivative of $h(u)$ to get

$$h'(u) = 4u^3 + 6u^2 - 2, \tag{9.47}$$

then we see that $h'\left(\frac{1}{2}\right) = 0$ regardless of the value of m. This means that $h(u)$ always has a local minimum at $u = \frac{1}{2}$, and if that local minimum happens to be a negative value, then there must be a root in the interval $\left(\frac{1}{2}, 1\right)$ because $h(1)$ is positive. By plugging the value $\frac{1}{2}$ into $h(u)$, we can calculate that $h\left(\frac{1}{2}\right) < 0$ precisely when $m < \frac{27}{16}$, and this condition corresponds to cases (b) and (c) in Figure 9.20.

It may be tempting to calculate the roots of $h(u)$ using algebraic methods for the solution to a quartic equation, but in this case, the root in which we are interested can be found more efficiently and accurately by using Newton's method with an initial value of $u_0 = 1$. Refined approximations to the root are calculated using the formula

$$u_{i+1} = u_i - \frac{h(u_i)}{h'(u_i)}. \tag{9.48}$$
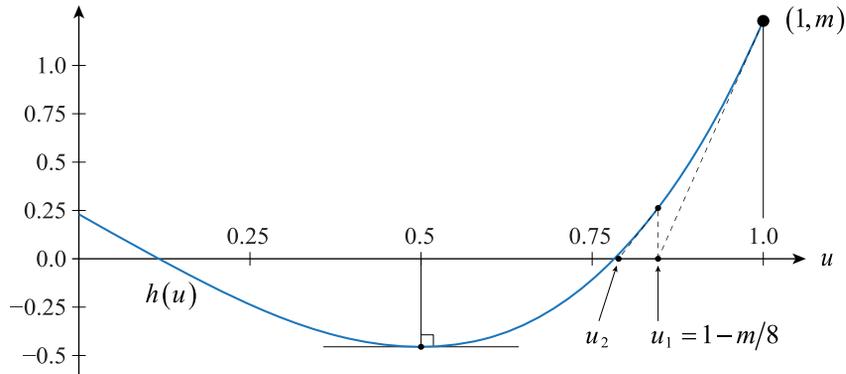
Figure 9.22. This is a plot of the function $h(u) = u^4 + 2u^3 - 2u + m - 1$ with $m \approx 1.23$ (corresponding to the values $\tau_{max} = \ln 256$, $\alpha_0 = 0.1$, and $z_0 = -10$). In this example, the first two iterations of Newton's method produce $u_1 \approx 0.846$ and $u_2 \approx 0.791$.

We know right at the beginning that $h(1) = m$ and $h'(1) = 8$, so the first iteration of Newton's method can easily be precalculated to obtain

$$u_1 = 1 - \frac{m}{8},\tag{9.49}$$

and this is shown in Figure 9.22. One or two more iterations of Newton's method are all that are needed to produce an extremely precise value of u that can be plugged back into Equation (9.44) to calculate the occlusion radius r.

If $m > 1$, then the exact occlusion surface intersects the fog plane, and it is possible that the largest horizontal distance to the surface occurs on the fog plane itself where $u = 0$. This corresponds to cases (a) and (b) in Figure 9.20. When we plug the value zero into Equation (9.44), the radius r is given by

$$\boxed{r = -z_0 \sqrt{m - 1},}\tag{9.50}$$

where the negative square root has been chosen to produce a positive distance.

The code in Listing 9.22 considers all three of the cases shown in Figure 9.20 for a given camera position c, fog attenuation α_0, and maximum optical depth τ_{max}. In case (a) where $m \geq \frac{27}{16}$, the occlusion surface has no local maxima, and the only possible fog-out distance is given by Equation (9.50). In case (c) where $m \leq 1$, the occlusion surface does not intersect the fog plane, and the fog-out distance is given

by Equation (9.44) for the largest root of the function $h(u)$. In case (b) where $1 < m < \frac{27}{16}$, both solutions are valid, so we calculate both fog-out distances and simply take whichever one is larger. Once the fog-out distance has been determined, an occlusion plane perpendicular to the horizontal component of the camera's view direction is returned.

Listing 9.22. For a world-space fog plane given by `fogPlane`, a world-space camera position given by `cameraPosition`, and a world-space view direction given by `viewDirection`, this function calculates the occlusion plane that is perpendicular to the fog plane and returns it in `occlusionPlane`. The linear fog attenuation coefficient α_0 is given by `fogDensity`, and the maximum optical depth τ_{max} is given by `maxOpticalDepth`. The return value is `true` if the view direction has a nonzero horizontal component relative to the fog plane. Otherwise, the return value is `false`, and no occlusion plane is generated.

```
bool CalculatePerpendicularFogOcclusionPlane(const Plane& fogPlane,
        const Point3D& cameraPosition, const Vector3D& viewDirection,
        float fogDensity, float maxOpticalDepth, Plane *occlusionPlane)
{
    // Project view direction onto fog plane.
    Vector3D normal = Reject(viewDirection, fogPlane.GetNormal());
    float n2 = Dot(normal, normal);
    if (n2 > FLT_MIN)
    {
        // View direction has nonzero horizontal component.
        float z0 = Dot(fogPlane, cameraPosition), z02 = z0 * z0, z0_inv2 = 1.0F / z02;
        float sigma = 2.0F * maxOpticalDepth / fogDensity, sigma2 = sigma * sigma;
        float m = sigma2 * z0_inv2 * z0_inv2;
        float r = 0.0F;

        if (m < 1.6875F)
        {
            // When m < 27/16, a root of h(u) exists.
            float u = 1.0F - m * 0.125F, u2 = u * u;

            // Apply second iteration of Newton's method.
            u -= (((u + 2.0F) * u2 - 2.0F) * u + m - 1.0F) /
                ((u * 4.0F + 6.0F) * u2 - 2.0F);

            // Plug root into Equation (9.44).
            float up1 = u + 1.0F, um1 = u - 1.0F;
            r = sqrt(sigma2 * z0_inv2 / (up1 * up1) - um1 * um1 * z02);
        }

        if (m > 1.0F)
        {
```

```
            // Calculate occlusion distance with Equation (9.50).
            // Take larger value of r when both solutions are valid.
            r = fmax(-z0 * sqrt(m - 1.0F), r);
        }

        // Construct occlusion plane perpendicular to fog plane.
        normal *= 1.0F / sqrt(n2);
        occlusionPlane->Set(normal, -Dot(normal, cameraPosition) - r);
        return (true);
    }

    return (false);
}
```

Exercises for Chapter 9

1. Given a sphere A with center p_1 and radius r_1 and another sphere B with center p_2 and radius r_2, calculate the center and radius of the smallest sphere enclosing both A and B. Be sure to handle all cases in which the spheres intersect.

2. Determine the optimal bounding sphere for an equilateral triangle, and calculate the differences between its center and radius and the center and radius of the bounding sphere produced by the method described in Section 9.3.1.

3. Write simplified versions of Equations (9.16) and (9.18) that apply specifically to axis-aligned bounding boxes.

4. Suppose that an object is bounded by a circular cylinder of radius r with its axis aligned to the unit-length direction $\mathbf{u}$. Place the center halfway between the bases, and let h be half of the cylinder's height. Calculate the effective radius $r_{\mathbf{g}}$ of this cylinder with respect to a normalized plane $\mathbf{g}$.

5. Suppose that an object is bounded by an ellipsoid with semiaxis lengths h_x, h_y, and h_z oriented to the unit-length directions $\mathbf{s}$, $\mathbf{t}$, and $\mathbf{u}$. Calculate the effective radius $r_{\mathbf{g}}$ of this ellipsoid with respect to a normalized plane $\mathbf{g}$.

6. Suppose a view frustum for a camera at the position c pointing in the view direction $\mathbf{z}$ has an aspect ratio s and a projection distance g. Refer to Exercise 4 in Chapter 6, and find an expression for $\sin \alpha$ corresponding to the cone circumscribing this view frustum. Given a normalized plane $\mathbf{g}$, and assuming that

$\mathbf{g} \cdot \mathbf{c} < 0$, determine the minimum value of $\mathbf{g} \cdot \mathbf{z}$ for which the plane intersects the infinite extension of the cone in terms of $\sin \alpha$.

7. Modify Listing 9.14 so that it constructs a five-sided polyhedron corresponding to the boundary pyramid of a spot light.

8. Let $\mathbf{M}_{occluder}$ be the matrix that transforms a box occluder of size $\mathbf{s}$ from object space to world space. Express the world-space bounding planes of the box occluder in terms of the rows of the matrix $\mathbf{M}_{occluder}^{-1}$.

9. Suppose that a maximum optical depth τ_{max} has been calculated for halfspace fog with a reference density α_0. Given a negative camera depth z_0, determine a formula for the minimum value of the angle of depression φ at which fog occlusion is possible relative to the camera position. Identify the condition under which no minimum angle exists.

Chapter **10**

Advanced Rendering

The field of game engine development is abundant with advanced rendering techniques that produce a wide range of visual effects. In this chapter, we have selected several graphical objectives, some of which are considered basic necessities in many engines, and discuss a particular solution to each of the rendering problems they present. While we acknowledge that for many of these objectives, there are multiple valid methods for achieving similar high-quality results, we avoid the inescapable vagueness of a survey and instead concentrate on the precise implementation details of one proven approach.

10.1 Decals

As interaction takes place among various entities and the environment, a game engine often needs to draw new objects like bullet holes, scorch marks, or footprints. This is done by creating special triangle meshes called *decals* and applying them to existing surfaces. These meshes are built from the triangles belonging to the underlying surfaces by clipping those triangles against a bounding box surrounding the decal. An example is shown in Figure 10.1, where the edges of a bright yellow blast mark are clearly visible within the triangle mesh for the stone wall to which it is applied.

Because the triangles that make up a decal are always coincident with the triangles belonging to another surface, they need to be rendered with an offset toward the camera to avoid Z fighting artifacts. This can be accomplished by using the hardware polygon offset functionality or by modifying the projection matrix with the method described by Exercise 14 in Chapter 6. For decals that could be far away from the camera, both of these options work best with a reversing projection matrix because it provides much greater depth precision.
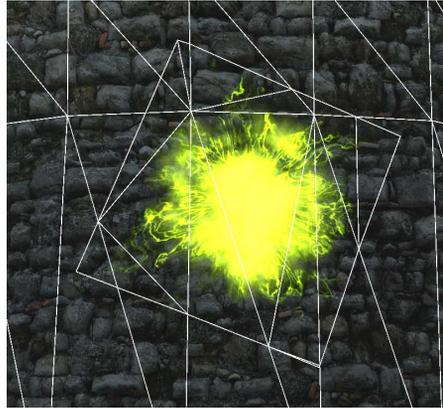
Figure 10.1. A blast mark is applied to a surface as a decal by clipping the polygons belonging to the surface against the bounding box of the decal.

To create a decal, we first need to construct the six planes of its bounding box. We assume that the center of the decal p and normal vector $\mathbf{n}$ at the center are already available. This information is usually supplied either by a ray intersection function or by the physics system in response to a collision of some kind. We choose a tangent vector $\mathbf{t}$ to be perpendicular to the normal vector $\mathbf{n}$, perhaps using Listing 9.10. The tangent vector can be randomly rotated, or it can be aligned to a specific orientation such as part of a character's skeleton in the case of footprints being applied to the ground. Once the tangent vector has been determined, we complete a right-handed coordinate system by calculating a bitangent vector with $\mathbf{b} = \mathbf{n} \times \mathbf{t}$. If necessary, we normalize all three vectors to unit length.

Suppose we want our decal to have a radius r_x in the tangent direction and a radius r_y in the bitangent direction, as shown in Figure 10.2. Its overall width and height are then $2r_x$ and $2r_y$. We also choose a distance d by which the decal extends above and below the surface in the normal direction. This distance generally needs to be large enough to account for any amount by which the height of the surface might change inside the decal's boundary, but not so large that the decal is unintentionally applied to other surfaces that could be in front of or behind the surface on which the point p lies. Using these measurements, the six inward-facing planes defining the bounding box of the decal are given by

$$
\begin{aligned}
\mathbf{f}_{\text{left}} &= \begin{bmatrix} \mathbf{t} \mid r_x - \mathbf{t} \cdot p \end{bmatrix} & \mathbf{f}_{\text{right}} &= \begin{bmatrix} -\mathbf{t} \mid r_x + \mathbf{t} \cdot p \end{bmatrix} \\
\mathbf{f}_{\text{bottom}} &= \begin{bmatrix} \mathbf{b} \mid r_y - \mathbf{b} \cdot p \end{bmatrix} & \mathbf{f}_{\text{top}} &= \begin{bmatrix} -\mathbf{b} \mid r_y + \mathbf{b} \cdot p \end{bmatrix} \\
\mathbf{f}_{\text{back}} &= \begin{bmatrix} \mathbf{n} \mid d - \mathbf{n} \cdot p \end{bmatrix} & \mathbf{f}_{\text{front}} &= \begin{bmatrix} -\mathbf{n} \mid d + \mathbf{n} \cdot p \end{bmatrix}.
\end{aligned}
\tag{10.1}
$$
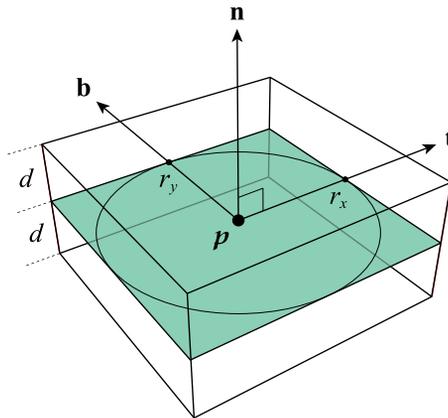
Figure 10.2. A decal centered at the point p has a right-handed coordinate system defined by the normal vector $\mathbf{n}$, the tangent vector $\mathbf{t}$, and the bitangent vector $\mathbf{b}$. The dimensions of the decal are described by the radius r_x in the tangent direction, the radius r_y in the bitangent direction, and a maximum offset d in the normal direction.

To build the mesh for a decal, we search the world for other geometries that intersect the decal's bounding box. For each geometry that we find, we clip its triangles against all six of the planes given by Equation (10.1) using the method described in Section 9.1. The bounding planes must first be transformed into the geometry's object-space coordinate system where its vertex positions are defined. During the clipping process, many of the triangles will be discarded because they are completely clipped away. Those that remain form the triangle mesh for the decal. As mentioned in Section 9.1, it is important that we use a consistent vertex ordering with respect to each plane so that any new vertices created on an edge shared by two triangles has exactly the same coordinates when each triangle is clipped, ensuring that the decal mesh is watertight.

We may choose to combine the clipped triangles from multiple geometries into a single decal mesh, or we may decide to create a separate decal mesh for each geometry that is clipped. The latter option is useful when some geometries may be moving, and we want the decals to move with them. In this case, we can simply make each decal mesh a subnode of the geometry from which it was derived, and this will cause it to automatically inherit the geometry's transform.

As a geometry is being clipped, we are likely to encounter triangles that face away from the normal direction $\mathbf{n}$, and those triangles can be discarded without clipping. Suppose a triangle has the vertex positions v_0, v_1, and v_2 wound in the counterclockwise direction, and let $\mathbf{m}$ be the normal vector for the plane of the

triangle, given by $\mathbf{m} = (\boldsymbol{v}_1 - \boldsymbol{v}_0) \times (\boldsymbol{v}_2 - \boldsymbol{v}_0)$. We discard any triangles for which $\mathbf{n} \cdot \mathbf{m} < \varepsilon \|\mathbf{m}\|$, where ε is a small positive constant that allows us to ignore triangles that are nearly perpendicular to the surface as well as those that face away from it.

For each vertex of a decal mesh having the position $\boldsymbol{q}$, the associated texture coordinates (u, v) are typically calculated in terms of perpendicular distances to the bounding planes using the formulas

$$u = \frac{1}{2r_x}\mathbf{f}_{\text{left}} \cdot \boldsymbol{q} \quad \text{and} \quad v = \frac{1}{2r_y}\mathbf{f}_{\text{bottom}} \cdot \boldsymbol{q}. \tag{10.2}$$

This causes the texture coordinates to cover the range from zero to one across the full width and height of the decal. When texture coordinates are generated in this manner, the tangent and bitangent vectors that may be required by the decal's material are simply the constant values of $\mathbf{t}$ and $\mathbf{b}$.

Depending on what kind of material is going to be assigned to a decal, it may be necessary to interpolate vertex normals in addition to positions during the clipping process. This can be done using a formula similar to Equation (9.3) for the normal vectors and then rescaling to unit length. Tangent vectors do not need to be interpolated unless the decal's material makes use of the underlying geometry's texture coordinates (which would then also need to be interpolated), perhaps using a sophisticated technique that incorporates the same normal map that's applied to the geometry.

In some cases, it may be desirable for a decal to fade out in areas where its vertex normals get close to being perpendicular to the surface normal $\mathbf{n}$ at the center. If we have interpolated the vertex normals of the underlying geometry during the clipping process, then we can examine the dot product $\mathbf{n} \cdot \mathbf{z}$ at each clipped vertex, where $\mathbf{z}$ is the interpolated normal vector scaled to unit length. An opacity α could then be obtained using a formula such as

$$\alpha = \max\left(\frac{\mathbf{n} \cdot \mathbf{z} - \varepsilon}{1 - \varepsilon}, 0\right), \tag{10.3}$$

where ε is the same constant used to discard triangles above. The value of α is one when $\mathbf{n} \cdot \mathbf{z} = 1$ and zero when $\mathbf{n} \cdot \mathbf{z} \leq \varepsilon$.

10.2 Billboards

There are many situations in which some kind of object needs to look like it has volume, but rendering it in a truly volumetric manner would either be prohibitively expensive or would not significantly improve its appearance. In these cases, it is

often acceptable to render such an object on a flat rectangle called a *billboard*, also known as a *sprite*, that always faces toward the camera so the illusion of volume is created as the viewer moves around. Billboards are frequently used to draw each of the particles in a particle system because they are often small in size, short lived, and moving fast enough that their lack of true volume goes unnoticed. (Particle systems will be discussed in detail in Volume 3.) Various types of billboards are also used to render effects like fire, light halos, laser beams, and lightning bolts. A special type of billboard is used to render *impostors*, which are flat images substituted for complex models like trees when they are far from the camera.

Because billboards are flat objects, they do have their limits. For example, if a billboard intersects solid geometry, perhaps because smoke particles have drifted too close to a wall, then the flatness of the billboard becomes visible at the boundary where part of it fails the depth test. A method for mitigating this problem is discussed in Section 10.3. Another limitation of billboards occurs when the camera and a billboard move through each other in the view direction because the billboard suddenly disappears as soon as it crosses the near plane. A trick often used to hide this problem is to gradually increase the transparency of the billboard as it gets close to the camera. Handling this kind of interaction correctly without tricks requires a volumetric technique like those described in Section 10.4.

In this section, we describe two general types of billboards called spherical billboards and cylindrical billboards. These names are not meant to imply any geometrical shape beyond the flat polygon of which all billboards are made. They instead reflect the ways in which the billboards are oriented with respect to the camera. Figures 10.3(a) and 10.3(b) show examples in which these types of billboards are used to render smoke particles and a fire effect. We also discuss polyboards, which consist of multiple billboards joined together to follow a path, as shown in Figure 10.3(c). Finally, we introduce polygon trimming as an optimization that often applies to objects rendered as billboards when they contain large transparent regions, such as the tree impostors shown in Figure 10.3(d).

10.2.1 Spherical Billboards

A *spherical billboard* is defined by a center point p and two radii r_x and r_y equal to half its width and half its height. The billboard can rotate about its center to any orientation so that its normal vector points toward the camera, and the union of all possible orientations fills a sphere of radius $\sqrt{r_x^2 + r_y^2}$. To determine the actual orientation of a billboard and calculate its vertex positions, we transform information about the camera into the billboard's object space. Let $\mathbf{M}_{\text{camera}}$ be the transform from camera space to world space, and let $\mathbf{M}_{\text{object}}$ be the transform from the bill-

Figure 10.3. In these examples, different kinds of billboards are shown in wireframe. (a) The black smoke is rendered as a particle system containing many spherical billboards. (b) The flames from these torches are rendered as cylindrical billboards. (c) Several bolts of electricity are rendered as polyboards. (d) Tree imposters are rendered as cylindrical billboards, and their corners have been trimmed to reduce the area that they fill.

board's object space to world space. Then vectors $\mathbf{x}$ and $\mathbf{y}$ in the billboard's object space corresponding to the positive x and y axes of the camera are given by

$$
\mathbf{x} = \mathbf{M}_{\text{object}}^{-1} \mathbf{M}_{\text{camera}[0]}
$$
$$
\mathbf{y} = \mathbf{M}_{\text{object}}^{-1} \mathbf{M}_{\text{camera}[1]},
\tag{10.4}
$$

where $\mathbf{M}_{\text{camera}[j]}$ denotes column j of the matrix $\mathbf{M}_{\text{camera}}$. If we place the vertices of the billboard in the plane determined by the point $\boldsymbol{p}$ and the directions $\mathbf{x}$ and $\mathbf{y}$, then the resulting polygon is always parallel to the projection plane, as shown in Figure 10.4(a). The four vertex positions $\boldsymbol{v}_0$, $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, and $\boldsymbol{v}_3$ are given by

$$
\begin{aligned}
\boldsymbol{v}_0 &= \boldsymbol{p} - r_x\mathbf{x} + r_y\mathbf{y} & \boldsymbol{v}_1 &= \boldsymbol{p} + r_x\mathbf{x} + r_y\mathbf{y} \\
\boldsymbol{v}_2 &= \boldsymbol{p} + r_x\mathbf{x} - r_y\mathbf{y} & \boldsymbol{v}_3 &= \boldsymbol{p} - r_x\mathbf{x} - r_y\mathbf{y}.
\end{aligned}
\tag{10.5}
$$

If the matrix $\mathbf{M}_{object}$ contains a scale, then $\mathbf{x}$ and $\mathbf{y}$ first need to be renormalized to unit length. For objects made of a single billboard, it is often the case that $\boldsymbol{p}$ lies at the object-space origin, so the vertex positions are simply sums and differences of $r_x\mathbf{x}$ and $r_y\mathbf{y}$. But for objects like particle systems, which contain a large number of independent billboards, there is a different point $\boldsymbol{p}$ representing the center of each particle.

The vertex positions given by Equation (10.5) produce billboards that face only the x-y plane of the camera and not the actual position of the camera except in those cases when the billboard lies at the center of the viewport. However, this orientation is usually good enough from a visual standpoint, especially when billboard sizes are small. When we want the orientation toward the camera to be exact, a little more computation is required. First, we need the camera position c in the billboard's object space, which is given by

$$c = \mathbf{M}_{object}^{-1}\mathbf{M}_{camera[3]}. \tag{10.6}$$

(Remember that $\mathbf{M}_{camera[3]}$ is a point having a w coordinate of one, unlike $\mathbf{M}_{camera[0]}$ and $\mathbf{M}_{camera[1]}$, which are vectors having a w coordinate of zero.) Our goal is to place the vertices in a plane containing the billboard's center $\boldsymbol{p}$ that has the normal vector

$$\mathbf{n} = \mathrm{nrm}\,(c - \boldsymbol{p}). \tag{10.7}$$
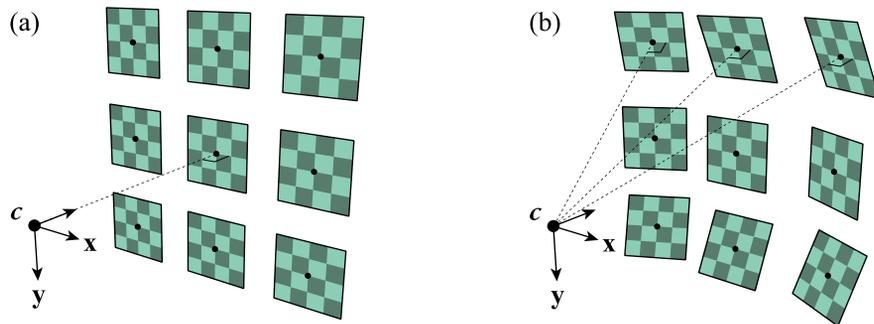
Figure 10.4. Billboards are rendered so they face toward the camera at the position c, which has been transformed into object space. (a) The billboards are oriented to lie in planes parallel to the projection plane by directly using the camera's right direction $\mathbf{x}$ and down direction $\mathbf{y}$ as the basis for their local coordinate systems. (b) Each billboard is oriented so it is perpendicular to the normal direction $c - \boldsymbol{p}$ between the camera position and the billboard's center $\boldsymbol{p}$.

A tangent vector $\mathbf{a}$ for the billboard can now be calculated as

$$\mathbf{a} = \mathrm{nrm}\,(\mathbf{n} \times \mathbf{y}). \tag{10.8}$$

Because the vector $\mathbf{y}$ corresponds to the direction pointing downward in camera space, the tangent vector $\mathbf{a}$ points to the right from the camera's perspective. Another tangent vector $\mathbf{b}$ that points upward is given by $\mathbf{b} = \mathbf{n} \times \mathbf{a}$, and this completes a local set of axes for the billboard. Now, the four vertex positions $\boldsymbol{v}_0$, $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, and $\boldsymbol{v}_3$ are given by

$$\boldsymbol{v}_0 = \boldsymbol{p} - r_x \mathbf{a} - r_y \mathbf{b} \qquad \boldsymbol{v}_1 = \boldsymbol{p} + r_x \mathbf{a} - r_y \mathbf{b}$$
$$\boldsymbol{v}_2 = \boldsymbol{p} + r_x \mathbf{a} + r_y \mathbf{b} \qquad \boldsymbol{v}_3 = \boldsymbol{p} - r_x \mathbf{a} + r_y \mathbf{b}, \tag{10.9}$$

and this produces the billboard orientations shown in Figure 10.4(b). The vertex $\boldsymbol{v}_0$ is the lower-left corner of the billboard, where the texture coordinates would typically be $(0,1)$, and the rest follow in the counterclockwise direction. Note that the vector $\mathbf{b}$ in Equation (10.9) has the opposite sign of the vector $\mathbf{y}$ in Equation (10.5).

Once a billboard has been oriented with its normal vector pointing toward the camera position, we can still rotate the billboard about that normal vector without breaking the alignment constraint. This additional degree of freedom is often used to draw a group of billboards with random rotations so they don't all look the same. The rotation can also be animated to produce effects like swirling smoke. Suppose that a billboard is rotated through an angle θ about its normal vector, where an angle of zero corresponds to the tangent direction $\mathbf{a}$. Then rotated tangent vectors $\mathbf{a}'$ and $\mathbf{b}'$ can be expressed as

$$\mathbf{a}' = \mathbf{a} \cos \theta + \mathbf{b} \sin \theta$$
$$\mathbf{b}' = \mathbf{b} \cos \theta - \mathbf{a} \sin \theta, \tag{10.10}$$

as illustrated in Figure 10.5. When we substitute these into Equation (10.9) and collect terms containing the vectors $\mathbf{a}$ and $\mathbf{b}$, we get

$$\boldsymbol{v}_0 = \boldsymbol{p} - (r_x \cos \theta - r_y \sin \theta)\,\mathbf{a} - (r_y \cos \theta + r_x \sin \theta)\,\mathbf{b}$$
$$\boldsymbol{v}_1 = \boldsymbol{p} + (r_x \cos \theta + r_y \sin \theta)\,\mathbf{a} - (r_y \cos \theta - r_x \sin \theta)\,\mathbf{b}$$
$$\boldsymbol{v}_2 = \boldsymbol{p} + (r_x \cos \theta - r_y \sin \theta)\,\mathbf{a} + (r_y \cos \theta + r_x \sin \theta)\,\mathbf{b}$$
$$\boldsymbol{v}_3 = \boldsymbol{p} - (r_x \cos \theta + r_y \sin \theta)\,\mathbf{a} + (r_y \cos \theta - r_x \sin \theta)\,\mathbf{b}. \tag{10.11}$$

We call the scalar values multiplying the vectors $\mathbf{a}$ and $\mathbf{b}$ in Equation (10.11) *billboard coordinates*. Disregarding signs, there are four distinct coordinate values,
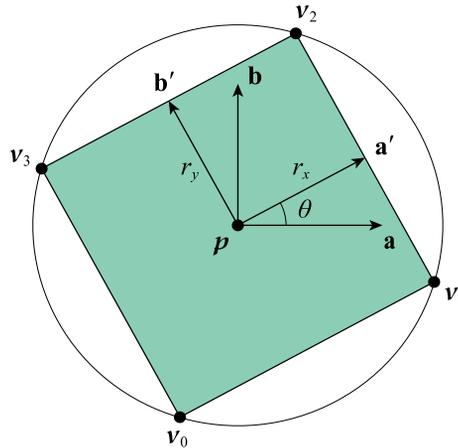
Figure 10.5. The vertices v_0, v_1, v_2, and v_3 of a billboard are given by $p \pm r_x a' \pm r_y b'$, where a' and b' are rotated versions of the tangent vectors a and b.

and this is reduced to only two if $r_x = r_y$. The billboard coordinates can be precalculated for each vertex and stored in a two-component attribute array. A billboard typically has a vertex attribute holding the position, which is the same center point p for all four vertices, another vertex attribute holding the billboard coordinates, a vertex attribute holding the texture coordinates, and possibly one more vertex attribute holding a color. As demonstrated in Listing 10.1, the vectors a and b are calculated in the vertex shader using the camera position and down direction. The object-space vertex position is determined by adding the vectors a and b to the point p after scaling them by the precalculated billboard coordinates. The result would later be transformed into clip space with the MVP matrix as usual.

10.2.2 Cylindrical Billboards

A *cylindrical billboard* is defined by a point p, a radius r, and a height h, as shown in Figure 10.6. This type of billboard can rotate only about its object-space z axis to face the camera, and the union of all possible orientations thus fills a cylinder. Although the z axis is usually considered to be aligned with the vertical direction in object space, the billboard can be transformed into any arbitrary orientation in world space. For example, a cylindrical billboard could be used to render a laser beam that runs horizontally, parallel to the ground. The fire effects and impostors shown in Figures 10.3(b) and 10.3(d) are rendered with cylindrical billboards.

Listing 10.1. This vertex shader function calculates the object-space position of a spherical billboard vertex using the position p given by `center` and the billboard coordinates given by `billboard`, both of which are supplied by vertex attributes. The uniform constant `cameraPosition` holds the object-space camera position c, and `cameraDown` holds the objects-space camera down direction $\mathbf{y}$.

```
uniform float3 cameraPosition;
uniform float3 cameraDown;

float3 CalculateSphericalBillboardVertexPosition(float3 center, float2 billboard)
{
    float3 n = normalize(cameraPosition - center);
    float3 a = normalize(cross(n, cameraDown));
    float3 b = cross(n, a);
    return (center + a * billboard.x + b * billboard.y);
}
```

We place the point p defining the position of the billboard at the center of the cylinder's base. The billboard always contains the line segment connecting the point p and the point $(p_x, p_y, p_z + h)$ at the opposite end of the cylinder. The one tangent direction that we are able to choose must be perpendicular to the z axis, so it always lies in the x-y plane. We also want the tangent direction to be perpendicular to the direction to the camera given by $c - p$, where c is again the object-space camera position. The tangent vector $\mathbf{a}$ must therefore be given by the cross product

$$\mathbf{a} = (0,0,1) \times (c - p). \tag{10.12}$$
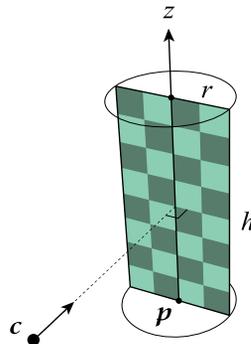
Figure 10.6. A cylindrical billboard is defined by a radius r and height h. It is constrained to rotate only about its z axis in object space to face toward the camera at the position c.

When we write out the components of this cross product and normalize the result to unit length, we get

$$\mathbf{a} = \frac{(p_y - c_y, c_x - p_x, 0)}{\sqrt{\max\left((c_x - p_x)^2 + (c_y - p_y)^2, \varepsilon\right)}}. \tag{10.13}$$

This basically takes the x and y components of the vector $\mathbf{c} - \mathbf{p}$ and rotates them 90 degrees counterclockwise about the z axis. The max function in the denominator uses a small constant ε to ensure that division by zero doesn't occur in the case that the camera position differs from the point $\mathbf{p}$ only in the z direction. Using the tangent vector $\mathbf{a}$, the billboard's four vertex positions $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$ are given by

$$\begin{aligned}
\mathbf{v}_0 &= \mathbf{p} - r\mathbf{a} \\
\mathbf{v}_1 &= \mathbf{p} + r\mathbf{a} \\
\mathbf{v}_2 &= (p_x, p_y, p_z + h) - r\mathbf{a} \\
\mathbf{v}_3 &= (p_x, p_y, p_z + h) + r\mathbf{a}.
\end{aligned} \tag{10.14}$$

As with spherical billboards, a cylindrical billboard is typically rendered using vertex attribute arrays containing positions, billboard coordinates, texture coordinates, and possibly colors. This time, however, the vertex position can be either the point $\mathbf{p}$ or the point $(p_x, p_y, p_z + h)$. Also, there is only one billboard coordinate for each vertex, and it has the value $+r$ or $-r$. The need for a separate vertex attribute for the billboard coordinate could be eliminated by storing it in the w component of the position. As shown in Listing 10.2, the tangent vector $\mathbf{a}$ is calculated in the vertex shader using Equation (10.13), where we have set $\varepsilon = 0.0001$. The final object-space vertex position is then calculated as $\mathbf{p} + b\mathbf{a}$, where the position $\mathbf{p}$ and billboard coordinate b are those provided by the vertex attributes.

10.2.3 Polyboards

A *polyboard* is what we call a group of billboards connected end to end along a piecewise-linear path defined by a series of n points $\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_{n-1}$. The electrical effects shown in Figure 10.3(c) are rendered with polyboards. Each component of a polyboard is similar to a cylindrical billboard because rotation toward the camera takes place along the line segment between two consecutive points. However, the rotation is performed independently at each end, so there is usually some twisting within each component. As shown in Figure 10.7, each point $\mathbf{p}_i$ has an associated tangent direction $\mathbf{t}_i$ and radius r_i, and these values determine where the vertices are placed using a calculations similar to those used previously in this section.

Listing 10.2. This vertex shader function calculates the object-space position of a cylindrical billboard vertex using the position $\boldsymbol{p}$ given by center and the billboard coordinate b given by billboard, both of which are supplied by vertex attributes. The uniform constant cameraPosition holds the object-space camera position $\boldsymbol{c}$.

```
uniform float3 cameraPosition;

float3 CalculateCylindricalBillboardVertexPosition(float3 center, float billboard)
{
    float2 a = float2(center.y - cameraPosition.y, cameraPosition.x - center.x);
    a *= rsqrt(max(dot(a, a), 0.0001));
    return (float3(center.xy + a * billboard, center.z));
}
```

The tangent direction $\mathbf{t}_i$ at the point $\boldsymbol{p}_i$ could be determined in a couple of ways. If the polyboard follows a curve defined by some parametric function, then a valid tangent vector is given by that function's derivative. Otherwise, the tangent can simply be set to the difference of the surrounding points as

$$\mathbf{t}_i = \boldsymbol{p}_{i+1} - \boldsymbol{p}_{i-1}. \tag{10.15}$$

In the case that $\boldsymbol{p}_i$ is one of the polyboard's endpoints, we use the differences

$$\mathbf{t}_0 = \boldsymbol{p}_1 - \boldsymbol{p}_0$$
$$\mathbf{t}_{n-1} = \boldsymbol{p}_{n-1} - \boldsymbol{p}_{n-2}. \tag{10.16}$$
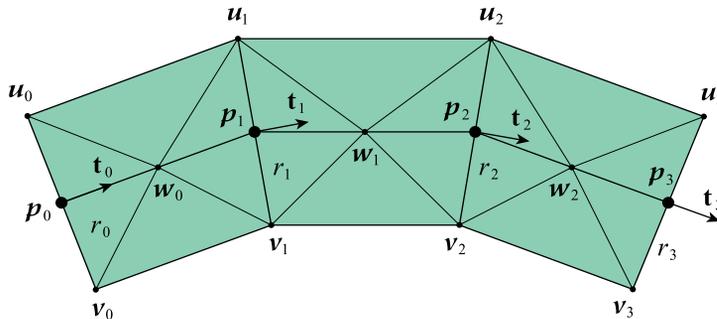
Figure 10.7. A polyboard is defined by a series of n points $\boldsymbol{p}_0, \boldsymbol{p}_1, \ldots, \boldsymbol{p}_{n-1}$ that each have an associated tangent direction $\mathbf{t}_i$ and radius r_i.

For the object-space camera position c, the unit-length tangent vector $\mathbf{a}_i$ perpendicular to both the tangent direction $\mathbf{t}_i$ and the direction to the camera at the point $\mathbf{p}_i$ is given by

$$\mathbf{a}_i = \frac{\mathbf{t}_i \times (c - \mathbf{p}_i)}{\sqrt{\max\left(\left(\mathbf{t}_i \times (c - \mathbf{p}_i)\right)^2, \varepsilon\right)}}, \tag{10.17}$$

where we again use a small constant ε to prevent division by zero. The two vertex positions $\mathbf{u}_i$ and $\mathbf{v}_i$ associated with the point $\mathbf{p}_i$ are then

$$\boxed{\begin{aligned} \mathbf{u}_i &= \mathbf{p}_i - r_i \mathbf{a}_i \\ \mathbf{v}_i &= \mathbf{p}_i + r_i \mathbf{a}_i. \end{aligned}} \tag{10.18}$$

If we were to render each component of a polyboard as a single quad, then the diagonal edge where the two triangles making up the quad meet would tend to create an unsightly fold when there is a lot of twist. The appearance of a polyboard is improved by adding a vertex to the center of each component and rendering four triangles that have a symmetric layout with respect to the line segment connecting adjacent points $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$. These additional vertices are labelled $\mathbf{w}_i$ in Figure 10.7, and they are simply the average positions given by

$$\mathbf{w}_i = \frac{\mathbf{p}_{i+1} + \mathbf{p}_i}{2}. \tag{10.19}$$

Using this configuration, the total number of vertices needed to render a polyboard is $3n - 1$, and the total number of triangles is $4n - 4$.

As with spherical and cylindrical billboards, the vertex attribute arrays for a polyboard include positions, billboard coordinates, texture coordinates, and possibly colors. Polyboards have an additional vertex attribute array containing the tangent directions $\mathbf{t}_i$. Each vertex has a single billboard coordinate, and it can be stored in the w component of either the position or the tangent to reduce the number of vertex attribute arrays by one. The billboard coordinates for each pair of vertices $\mathbf{u}_i$ and $\mathbf{v}_i$ are $-r$ and $+r$, respectively. So that all vertices can be handled in a uniform manner, each vertex $\mathbf{w}_i$ is assigned a billboard coordinate of zero and a tangent direction $\mathbf{p}_{i+1} - \mathbf{p}_i$. Listing 10.3 shows the calculation of the tangent vector $\mathbf{a}$ in a vertex shader using Equation (10.17), where we have set $\varepsilon = 0.0001$, and the calculation of the final object-space vertex position $\mathbf{p} + b\mathbf{a}$, where the position $\mathbf{p}$ and billboard coordinate b are those provided by the vertex attributes.

Listing 10.3. This vertex shader function calculates the object-space position of a polyboard vertex using the position p_i given by `center`, the vector $\mathbf{t}_i$ given by `tangent`, and the billboard coordinate b given by `billboard`, all of which are supplied by vertex attributes. The uniform constant `camera-Position` holds the object-space camera position c.

```
uniform float3 cameraPosition;

float3 CalculatePolyboardVertexPosition(float3 center, float3 tangent, float billboard)
{
    float3 a = cross(tangent, cameraPosition - center);
    a *= rsqrt(max(dot(a, a), 0.0001));
    return (center + a * billboard);
}
```

10.2.4 Trimming

The types of objects that are rendered as billboards often use alpha blending, and their texture maps tend to contain a lot of fully transparent texels, especially near the corners. When these billboards are drawn as a single quad, the pixel shader is executed for every pixel covered by the geometry, but for a large fraction of the billboard area, no change is ultimately made to the frame buffer. This wasted computation can have a significant impact on performance when the billboards are large in size or when many overlapping billboards are rendered with a large amount of overdraw. We can alleviate the problem by trimming the corners off of the quad and rendering a billboard as a polygon having more than four sides. The billboards containing tree impostors shown in Figure 10.3(d) have been trimmed to remove the empty space at their corners.

When we trim one of a billboard's corners, we remove a triangular piece of the original quad, as shown in Figure 10.8. To reduce the number of pixels rendered by as much as possible, our goal is to maximize the area of the triangle that we cut off. This goal is most easily accomplished by calculating the support planes for an array of directions in the quadrant containing the corner (excluding the horizontal and vertical directions) and simply selecting the one that produces the largest trimmed triangle. The *support plane* in the direction $\mathbf{n}$ for a given set of points is the plane with normal vector $\mathbf{n}$ containing at least one of the points in the set and having the property that no other points lie on its positive side. In the case of a billboard, the set of points is composed of all the 2D positions $p(i, j)$ corresponding to texels in the texture map that are not completely transparent. For a specific normal vector $\mathbf{n}$, all we have to do it calculate

$$d = -\max_{\alpha(i,j)\neq 0} (\mathbf{n} \cdot p(i, j)), \qquad (10.20)$$

where the function $\alpha(i, j)$ represents the alpha value of the texel (i, j), and we obtain the support plane $[\mathbf{n} \,|\, d]$. This is illustrated in Figure 10.8.

In the case of a spherical billboard, the function $\boldsymbol{p}(i, j)$ transforms texel coordinates i and j into position coordinates p_x and p_y on the billboard in the ranges $[-r_x, +r_x]$ and $[-r_y, +r_y]$. If we always use integer values of i and j, then the centers of texels are located at the coordinates $(i + \frac{1}{2}, j + \frac{1}{2})$. However, the point inside a single texel that maximizes $\mathbf{n} \cdot \boldsymbol{p}(i, j)$ lies at the corner whose difference with the texel center is closest to the direction $\mathbf{n}$, so we define

$$(i', j') = \left(i + \tfrac{1}{2} + \tfrac{1}{2}\operatorname{sgn}(n_x), \, j + \tfrac{1}{2} + \tfrac{1}{2}\operatorname{sgn}(n_y) \right) \tag{10.21}$$

as the coordinates corresponding to the texel (i, j) for the purposes of calculating the support plane. The values of $\operatorname{sgn}(n_x)$ and $\operatorname{sgn}(n_y)$ are constant over each quadrant, so the offsets added to i and j need to be calculated only once for each corner of the billboard that we trim. The coordinates (i', j') are mapped to billboard positions using the function

$$\boldsymbol{p}(i, j) = \left(\frac{2r_x}{w} i' - r_x, \frac{2r_y}{h} j' - r_y \right), \tag{10.22}$$

where w and h are the width and height of the texture map, in texels. To avoid the appearance of the trimming boundary in filtered texels when a billboard's texture is minified, it's best to use the first or second mipmap when calculating support planes instead of the full-resolution image. This has the additional advantage that far fewer texels need to be considered.
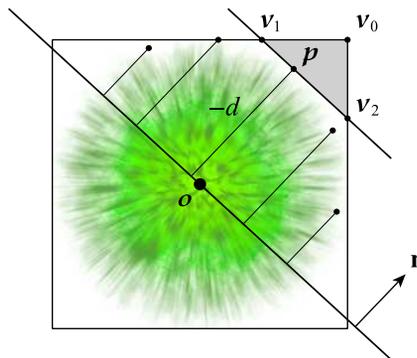
Figure 10.8. For a specific direction $\mathbf{n}$, the support plane $[\mathbf{n} \,|\, d]$ is determined by finding the point $\boldsymbol{p}$ that maximizes the dot product $\mathbf{n} \cdot \boldsymbol{p} = -d$. A small sampling of the points to be tested, corresponding to non-transparent texels, is shown here. The gray triangle lying beyond the support plane can be trimmed away.

After determining the largest value of $\mathbf{n} \cdot \boldsymbol{p}(i, j)$, the vertices of the triangle that can be trimmed away are determined by calculating the points where the original boundary of the billboard intersect the plane $[\mathbf{n} \,|\, d]$. In the case of the upper-right corner being trimmed, shown in Figure 10.8, the vertex $\boldsymbol{v}_1$ on the top edge of the billboard is given by

$$\boldsymbol{v}_1 = \left(-\frac{n_y r_y + d}{n_x}, r_y \right),$$ (10.23)

and the vertex $\boldsymbol{v}_2$ on the right edge is given by

$$\boldsymbol{v}_2 = \left(r_x, -\frac{n_x r_x + d}{n_y} \right).$$ (10.24)

The area of this triangle is $\frac{1}{2}\|(\boldsymbol{v}_1 - \boldsymbol{v}_0) \times (\boldsymbol{v}_2 - \boldsymbol{v}_0)\|$, where $\boldsymbol{v}_0$ is the vertex at the corner of the billboard. As we calculate different support planes for different normal directions $\mathbf{n}$, we keep track of which resulting triangle has the largest area, and that is the triangle that we ultimately trim away. However, we must be careful not to allow any of the triangle's vertices to extend past vertices belonging to the triangles trimmed away from adjacent corners. Also, if the triangle's area is smaller than some threshold for one of the corners, then we don't bother trimming that corner at all.

10.3 The Structure Buffer

Many rendering techniques involving special effects or postprocessing calculations require the camera-space depth of the geometry that has previously been rendered into the frame buffer, and it will often be needed at every pixel location in the viewport. This is the z coordinate in camera space representing the actual distance along the viewing direction and not the nonlinear value generated by the projection matrix and perspective divide that is ultimately written to the depth buffer. Some methods are also able to make use of the per-pixel derivatives of the camera-space depth with respect to the x and y axes, which provide a viewport-space gradient. In this section, we describe how the depth values and their derivatives can be generated for later consumption, and we present a basic application that uses the depth of previously rendered geometry. More advanced applications are described in the sections that follow.

For the camera-space depth information to be available to the pixel shaders that render the main scene, it must first be written to a special frame buffer that we call the *structure buffer*. This step can be combined with an initial rendering pass

that establishes the contents of the depth buffer so that the structure buffer and depth buffer are filled with their final values at the same time. The structure buffer is a render target with four channels using the 16-bit floating-point format, which means that each pixel requires eight bytes of storage. As shown in Figure 10.9, the derivatives $\partial z/\partial x$ and $\partial z/\partial y$ of the camera-space depth z are stored in the x and y channels (red and green). We need more than 16 bits of precision for the value of z itself, however, so it is split between the z and w channels (blue and alpha) as discussed below.

x	y	z	w
$\partial z/\partial x$	$\partial z/\partial y$	z_1	z_2

Figure 10.9. The structure buffer has four channels that can be labelled x, y, z, and w or red, green, blue, and alpha. Each channel holds a 16-bit floating-point value, so a single pixel is eight bytes in size. The camera-space depth z occupies both the z and w channels for high precision. The derivatives $\partial z/\partial x$ and $\partial z/\partial y$ are stored in the x and y channels.

An example that shows the contents of the structure buffer for a typical indoor scene is given in Figure 10.10. The final scene is shown in Figure 10.10(a), and it incorporates several uses of the structure buffer, such as an ambient occlusion technique that affects the lighting. (See Section 10.5.) The camera-space depth for this scene is shown as a grayscale value at each pixel in Figure 10.10(b) after being reconstituted from the information stored in the z and w channels of the structure buffer. The derivatives of the camera-space depth are shown in Figure 10.10(c). Since the change in z coordinate tends to very small compared to the change in x and y coordinates between adjacent pixels, the derivatives have been scaled by a factor of 64 to make them more pronounced. They have also been remapped to the range $[0,1]$ so that a derivative of zero in the x or y direction corresponds to a value of 0.5 in the red or green channel, respectively. A constant value of 0.5 has been assigned to the blue channel in the figure, and this causes any flat surface perpendicular to the viewing direction to appear as a 50% gray color. Pixels for which the depth is increasing over positive changes in the x direction have large red components, but if the depth is decreasing over the positive x direction, then the red component is small. Similarly, pixels for which the depth is increasing over positive changes in the y direction have large green components, but if the depth is decreasing over the positive y direction, then the green component is small.

Although it is sufficient for the derivatives, the 16-bit floating-point format does not provide as much precision as we would like for the camera-space depth.
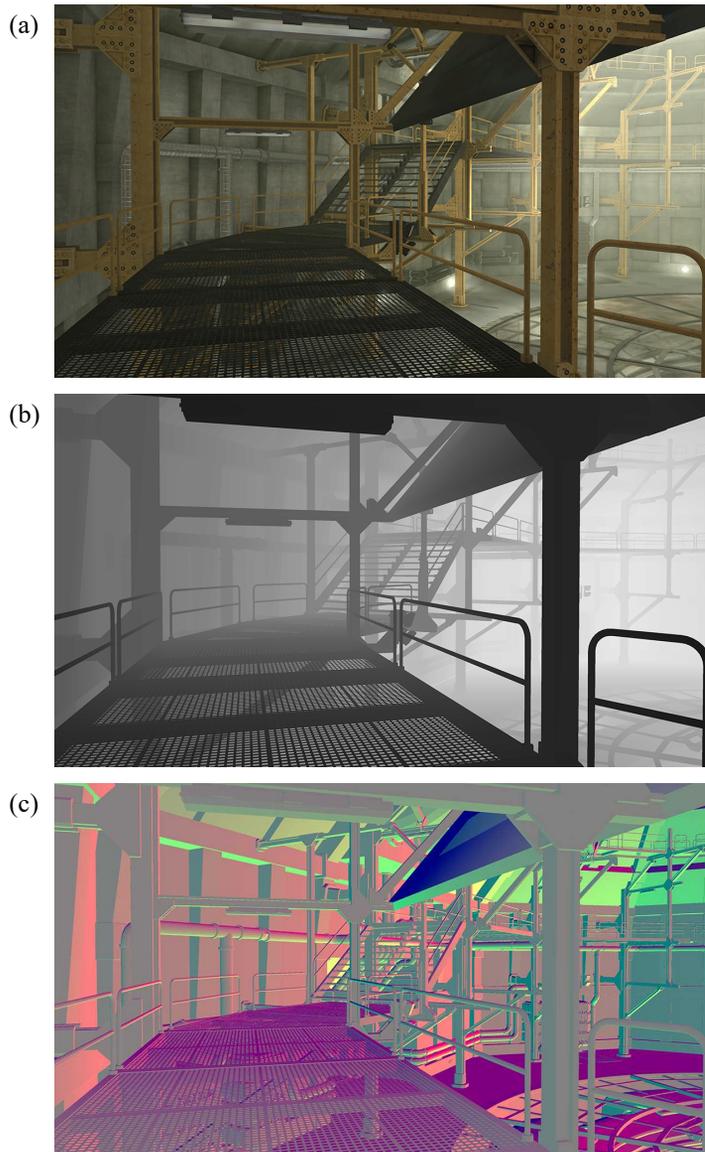
Figure 10.10. (a) Some of the lighting and effects applied when rendering the main scene use information stored in the structure buffer. (b) The z and w channels of the structure buffer are combined to form a high-precision camera-space depth value at every pixel. (c) The x and y channels of the structure buffer contain the viewport-space derivatives of the camera-space depth at every pixel. The red and green color channels show the derivatives after remapping from the range $\left[-\frac{1}{64}, +\frac{1}{64}\right]$ to the range $[0,1]$.

Fortunately, after the x and y channels of the structure buffer are claimed by the derivatives, we still have 32 bits of space available for the depth at each pixel in the z and w channels. We can split a depth z originally provided as a 32-bit floating-point value into two 16-bit floating-point values in such a way that they can later be combined to produce a value retaining nearly the same precision that we started with. This can be accomplished by first creating a depth h having only 10 bits of precision by masking off the 13 least significant bits of the 32-bit representation of z using a logical AND operation with the value 0xFFFFE000. As long as the exponent is in the range $[-15, +15]$, the value of h is preserved exactly when stored as a 16-bit floating-point number. If we now subtract h from the original depth z, then the first one bit after the 10 most significant bits in the mantissa becomes the implicit leading one bit in the difference, and the 10 bits that follow compose the mantissa when it is stored as a 16-bit floating-point number. This guarantees that the value of $z - h$ provides at least 11 more bits of precision, and when we later load the two values and add them together, we will have always retained at least 21 bits of the original 23-bit mantissa.

The pixel shader code shown in Listing 10.4 calculates the four values that are written to the structure buffer. The camera-space depth z is split into the two parts h and $z - h$ by isolating the most significant 10 bits of the mantissa, and the parts are stored in the z and w channels of the result. The viewport-space gradient of z is stored in the x and y channels. The value of z passed to this function would normally be given by the per-pixel w coordinate because the camera-space depth is assigned to it by a perspective projection matrix.

A basic application of the information in the structure buffer eliminates depth testing artifacts where transparent billboards intersect solid geometry. As demonstrated in Figure 10.11(a), large particles such as clouds of mist show unsightly lines where their polygons penetrate the ground geometry and fail the depth test. We can remove this problem by considering the difference Δz between the depth

Listing 10.4. This pixel shader function calculates the four values written to the structure buffer for a camera-space depth value z. The logical AND operation isolates the most significant 10 bits so that the depth can be split into two parts stored in the z and w channels of the result. The derivatives of z are stored in the x and y channels.

```
float4 CalculateStructureOutput(float z)
{
    float h = asfloat(asuint(z) & 0xFFFFE000U);
    return (float4(ddx(z), ddy(z), h, z - h));
}
```

already written to the structure buffer and the depth of the polygon being rendered. When this difference is small, it can be used to increase the transparency of the particles by reducing the alpha value a output by the pixel shader to a new value

$$a' = \text{sat}(s\Delta z)\, a, \tag{10.25}$$

where s is an adjustable scale factor. This causes the parts of each particle near the ground (or any other geometry) to gradually fade to full transparency at the intersection so that no artifacts are visible. This modification has been applied to the same mist particles in Figure 10.11(b), which no longer shows the lines that were previously visible. The pixel shader code shown in Listing 10.5 reads the two channels of depth information from the structure buffer, adds them together to reconstitute the high-precision depth value, calculates the difference Δz, and returns the transparency factor given by Equation (10.25).

Listing 10.5. This pixel shader code samples the structure buffer at the viewport-space pixel coordinates `pixelCoord` and adds the z and w channels together to reconstitute a high-precision depth value. The camera-space depth given by the z parameter is then subtracted to calculate the value of Δz used by Equation (10.25), where the value of s is given by the `scale` parameter.

```
uniform TextureRect    structureBuffer;

float CalculateDepthFade(float2 pixelCoord, float z, float scale)
{
    float2 depth = texture(structureBuffer, pixelCoord).zw;
    float delta = depth.x + depth.y - z;
    return (saturate(scale * delta));
}
```

10.4 Volumetric Effects

The billboards described in Section 10.2 use several tricks to make something flat appear as if it had volume. In this section, we discuss some ways of rendering partially transparent effects having simple geometries that are truly volumetric and thus require no tricks to appear so. We specifically examine two types of volumetric effects called halos and shafts that are demonstrated in Figure 10.12. These volumetric effects are somewhat more expensive to draw than billboards, and we cannot easily apply arbitrary texture maps to them. However, they have the advantage of real volume, and that allows them to be rendered correctly from any camera position, including positions that are inside the volume itself.

Figure 10.11. (a) Mist particles are rendered as billboards, and lines are visible where they intersect the ground geometry because the depth test abruptly begins to fail. (b) Each pixel belonging to a particle has its alpha value adjusted by Equation (10.25) so it becomes more transparent as the difference between its depth and the depth of the ground geometry already stored in the structure buffer decreases. This produces a softening effect that eliminates the line artifacts.

Figure 10.12. Two examples of truly volumetric effects are rendered inside different game worlds. (a) A spherical halo effect illuminates the mist surrounding a light source in the woods. (b) Several cylindrical shaft effects give the appearance of light shining into a dark dungeon.

10.4.1 Halos

A *halo* is a spherical volume that is typically placed around a light source to produce the glow that results from some of the light being scattered toward the camera by the surrounding medium. A halo's geometry is a sphere of radius R centered at the origin in object space, but we actually render a halo by drawing a simple bounding geometry such as the box shown in Figure 10.13. For each pixel covered by that bounding geometry, we trace a ray through the volume to determine how bright the halo should be at that pixel, multiply the result by a color, and additively blend it into the frame buffer. The intensity is zero for any ray that fails to intersect the sphere. Since we want the halo to be rendered when the camera is inside the volume, we turn the bounding geometry inside out so that the triangles on the far side are drawn when ordinary backface culling is enabled. Otherwise, a halo would disappear as soon as the camera crossed into its bounding geometry. Also, because a halo may intersect solid objects that may be nearby, we must render it with the depth test disabled so that any objects in front of the far side of the halo's bounding geometry, but not necessarily in front of the near side, do not prevent closer parts of the volume from being shown. The spherical shape of the halo, not its bounding geometry, should be tested for visibility against the view frustum (or visibility regions extruded from portals) before rendering. As a possible optimization, the near sides of the bounding geometry could be rendered with the depth test enabled if it has been determined, on a frame-by-frame basis that the entire bounding geometry lies beyond the near plane.

The appearance of a halo is largely determined by a density function $\rho(r)$ that controls how much light is scattered as distance from the center increases. It is

usually the case that the density is one at the center of the halo and zero at the radius R, but other functions are possible and produce interesting effects. For a physical light source, we would expect the density to be a linear function of r^{-2}, accounting for the correct attenuation, but the math does not work out well in that case. Instead, we use the density function

$$\rho(r) = 1 - \frac{r^2}{R^2}, \tag{10.26}$$

which, although physically insignificant, produces nice results with a reasonable amount of computation in the pixel shader.

The brightness of the halo at a particular pixel is given by the integral of the density function between the points where the ray $\boldsymbol{p} + t\mathbf{v}$ intersects the sphere of radius R, where the point $\boldsymbol{p}$ is the position on the bounding geometry, and the vector $\mathbf{v}$ is the unnormalized view direction $\boldsymbol{c} - \boldsymbol{p}$ for the object-space camera position $\boldsymbol{c}$. The limiting values of the parameter t are found by solving the equation

$$(\boldsymbol{p} + t\mathbf{v})^2 = R^2, \tag{10.27}$$

which expands to

$$v^2 t^2 + 2(\boldsymbol{p} \cdot \mathbf{v}) t + p^2 - R^2 = 0. \tag{10.28}$$

We make the definition

$$m = \sqrt{\max\left((\boldsymbol{p} \cdot \mathbf{v})^2 - v^2\left(p^2 - R^2\right), 0\right)}, \tag{10.29}$$
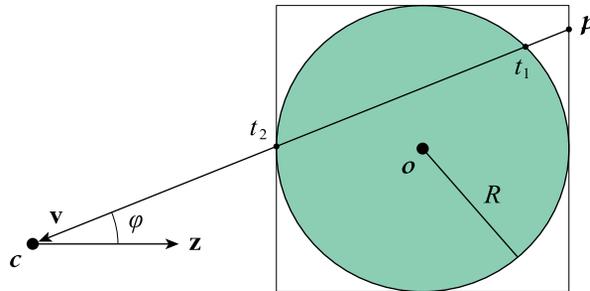
Figure 10.13. A halo of radius R centered at the object-space origin o is rendered by tracing rays from the camera position $\boldsymbol{c}$ to points $\boldsymbol{p}$ on the far side of the halo's bounding geometry, a box in this case. The vector $\mathbf{z}$ is the object-space view direction (z axis) of the camera.

which is the square root of the discriminant of the polynomial in Equation (10.28). The value inside the radical has been clamped to zero so we don't have to explicitly check whether the ray intersects the sphere. In the case of a miss, clamping the discriminant causes both values of t to be equal, and that leads to an integral over no distance, which produces a brightness of zero.

Using the definition of m, the solutions to Equation (10.28) are given by

$$t_{1,2} = \frac{-\boldsymbol{p} \cdot \mathbf{v} \mp m}{v^2},\tag{10.30}$$

where we have written the $\mp$ symbol to indicate that the minus sign applies to t_1, and the plus sign applies to t_2. As shown in Figure 10.13, these signs ensure that t_2 always corresponds to the point of intersection closer to the camera. A value of $t = 0$ corresponds to the point $\boldsymbol{p}$ on the bounding geometry, and a value of $t = 1$ corresponds to the camera position $\boldsymbol{c}$. Values of t greater than one are behind the camera. If the camera is inside the halo, then we only want to integrate along the ray between the camera position and the sphere intersection at t_1, so we will clamp the values of t to a maximum of one. Clamping both solutions handles the case in which the camera lies beyond the sphere but still in front of the far side of the bounding geometry because both values of t will become one.

In the case that the halo penetrates some other solid geometry, the depth of that geometry will be less than the depth of the farthest ray intersection for some pixels. The parts of the halo that are inside the solid geometry do not make any contribution to the its brightness, so we should integrate along the ray only up to the point where it reaches the geometry's depth. For this, we make use of the depth z_0 stored in the structure buffer. This depth corresponds to the perpendicular distance from the camera plane, but to calculate a minimum parameter t_0, we need the distance d along the direction $\mathbf{v}$ of the ray. A little trigonometry lets us calculate this distance using the equation

$$\cos \varphi = \frac{z_0}{d} = -\frac{\mathbf{z} \cdot \mathbf{v}}{\|\mathbf{v}\|},\tag{10.31}$$

where $\mathbf{z}$ is the object-space view direction of the camera given by

$$\mathbf{z} = \mathbf{M}_{\text{object}}^{-1} \mathbf{M}_{\text{camera}[2]},\tag{10.32}$$

and φ is the angle between the vectors $\mathbf{z}$ and $-\mathbf{v}$, as shown in Figure 10.13. The difference between the parameters $t = 0$ at the point $\boldsymbol{p}$ and $t = 1$ at the camera position $\boldsymbol{c}$ corresponds to the distance $\|\mathbf{v}\|$ along the ray, so dividing the distance d by the length $\|\mathbf{v}\|$ and subtracting it from one converts it to the parameter value

$$t_0 = 1 - \frac{d}{\|\mathbf{v}\|} = 1 + \frac{z_0}{\mathbf{z} \cdot \mathbf{v}}. \tag{10.33}$$

The values of t_1 and t_2 given by Equation (10.30) are both clamped to the minimum value specified by t_0. Together with the previous clamp to a maximum value of one, we now have the two limits of integration that represent the path through the halo starting no further away than any penetrating geometry and extending toward the camera without passing the camera position itself.

We are now ready to consider the integral of the density function $\rho(r)$ given by Equation (10.26), where $r = \|\boldsymbol{p} + t\mathbf{v}\|$, over the segment of the ray bounded by the parameters t_1 and t_2. This integral is written as

$$\int_{t_1}^{t_2} \rho(r) \|\mathbf{v}\| \, dt = \|\mathbf{v}\| \int_{t_1}^{t_2} dt - \frac{\|\mathbf{v}\|}{R^2} \int_{t_1}^{t_2} \left(v^2 t^2 + 2(\boldsymbol{p} \cdot \mathbf{v}) t + p^2 \right) dt, \tag{10.34}$$

and evaluating it yields the halo brightness B given by

$$B = \|\mathbf{v}\| \left[\left(1 - \frac{p^2}{R^2} \right)(t_2 - t_1) - \frac{\boldsymbol{p} \cdot \mathbf{v}}{R^2} \left(t_2^2 - t_1^2 \right) - \frac{v^2}{3R^2} \left(t_2^3 - t_1^3 \right) \right]. \tag{10.35}$$

The maximum value that B attains is $4R/3$. (See Exercise 1). We divide by this value to normalize the brightness of the halo so it does not change when the halo is scaled.

Listing 10.6 demonstrates how a halo is rendered by a pixel shader. It reads the structure buffer to determine the depth of any geometry that might penetrate the halo, and uses Equation (10.33) to calculate the corresponding parameter value t_0. The limits of integration t_1 and t_2 are then calculated with Equation (10.30) and clamped to the range $[t_0, 1]$. Finally, the density integral is evaluated with Equation (10.35) and normalized. The brightness of the halo produced by the nonphysical density function tends to fall off too slowly near its center and too quickly near its outer boundary, so we square the resulting value of $3B/4R$ to give it a more realistic appearance.

10.4.2 Shafts

A *shaft* is an extruded volume that is often used to show light from a bright exterior space shining into a relatively dark interior space. Shafts are rendered in much the same way as halos because we find the points at which a ray intersects some surface and integrate a density function through the enclosed volume. In this section, we

Listing 10.6. This pixel shader function renders a halo effect of radius R at the object-space position pobject. The uniform constant R2 holds the value R^2 used by Equation (10.29), the constants recipR2 and recip3R2 hold the values $1/R^2$ and $1/3R^2$ used by Equation (10.35), and the constant normalizer holds the value $3/4R$ used to normalize the density integral. The object-space camera position c and view direction z are given by cameraPosition and cameraView. The pixelCoord parameter specifies the viewport coordinates of the pixel being processed, and it is used to read from the structure buffer.

```
uniform TextureRect    structureBuffer;
uniform float3         cameraPosition, cameraView;
uniform float          R2, recipR2, recip3R2, normalizer;

float CalculateHaloBrightness(float3 pobject, float2 pixelCoord)
{
    float3 vdir = cameraPosition - pobject;
    float v2 = dot(vdir, vdir);
    float p2 = dot(pobject, pobject);
    float pv = -dot(pobject, vdir);
    float m = sqrt(max(pv * pv - v2 * (p2 - R2), 0.0));

    // Read z0 from the structure buffer.
    float2 depth = texture(structureBuffer, pixelCoord).zw;
    float t0 = 1.0 + (depth.x + depth.y) / dot(cameraView, vdir);

    // Calculate clamped limits of integration.
    float t1 = clamp((pv - m) / v2, t0, 1.0);
    float t2 = clamp((pv + m) / v2, t0, 1.0);
    float u1 = t1 * t1;
    float u2 = t2 * t2;

    // Evaluate density integral, normalize, and square.
    float B = ((1.0 - p2 * recipR2) * (t2 - t1) + pv * recipR2 * (u2 - u1)
            - v2 * recip3R2 * (t2 * u2 - t1 * u1)) * normalizer;
    return (B * B * v2);
}
```

describe the rendering process for solitary shafts defined by the extrusion of some simple 2D shape along a height h in the object-space z direction. A different method that renders large numbers of light shafts due to complex shadowing is described in Section 10.6. The specific surfaces that we examine are an elliptical cylinder and a box that can be thought of as the extrusions of an ellipse and a rectangle between the planes $z = 0$ and $z = h$. These surfaces are described by functions of only the x and y coordinates, but it is also possible to create volumes like a cone or pyramid by making them functions of the z coordinate as well.

Inside the volume of a shaft, we use a linear density function $\rho(z)$ that depends only on the z coordinate. It's often convenient to specify that the density has the value ρ_0 at $z = 0$ and the value ρ_1 at $z = h$. Given those constants, we can write

$$\rho(z) = \sigma z + \rho_0, \tag{10.36}$$

where we have made the assignment $\sigma = (\rho_1 - \rho_0)/h$. (Of course, the density is constant throughout the shaft if $\rho_1 = \rho_0$.) As with halos, we determine the brightness of a shaft by integrating this density function along the ray $\boldsymbol{p} + t\boldsymbol{v}$ between the parameters t_1 and t_2 where it intersects the shaft's surface. Here, $\boldsymbol{p}$ and $\boldsymbol{v}$ have the same meaning as they did in Section 10.4.1. Regardless of the overall shape of the shaft, the integral is

$$\int_{t_1}^{t_2} \rho(z)\|\boldsymbol{v}\|\, dt = \|\boldsymbol{v}\| \int_{t_1}^{t_2} (\sigma v_z t + \sigma p_z + \rho_0)\, dt, \tag{10.37}$$

and it evaluates to the shaft brightness B given by

$$B = (t_2 - t_1)\|\boldsymbol{v}\| \left[\frac{\sigma}{2} v_z (t_1 + t_2) + \sigma p_z + \rho_0 \right]. \tag{10.38}$$

The values of t_1 and t_2 are determined by the specific type of geometry that defines the shaft, as discussed below. When these parameters are calculated, it is practical to treat the shaft as if it has infinite extent in the z direction. With this in mind, we need to be careful not to integrate over any part of the ray where the density function $\rho(p_z + tv_z)$ becomes negative in the case that the density is not constant. The parameter value $t_{\lim}$ where the density equals zero is given by

$$t_{\lim} = \frac{-\rho_0/\sigma - p_z}{v_z}. \tag{10.39}$$

This value represents either the minimum limit or maximum limit to the range over which we integrate, depending on the signs of σ and v_z. If they have the same sign, then the density is increasing as we walk along the ray toward the camera. In this case, the parameters t_1 and t_2 must be clamped to be no less than $t_{\lim}$. Otherwise, if σ and v_z have different signs, then the density is decreasing, and the parameters t_1 and t_2 must be clamped to be no greater than $t_{\lim}$.

Listing 10.7 shows how a shaft is rendered by a pixel shader after the intersection parameters t_1 and t_2 have been calculated. These parameters are first clamped to the range $[t_0, 1]$ just as they were for halos, where t_0 is given by Equation (10.33).

They are then clamped to either a minimum or maximum value of t_{lim} based on the sign of σv_z before being plugged into Equation (10.38). (If $\sigma = 0$, meaning the density is constant, then the clamp to the zero-density limit t_{lim} should be omitted.) As with halos, we apply a normalization constant N to ensure that the brightness of a shaft is invariant when it is scaled. This constant would typically be given by

$$N = \frac{1}{D \max\left(\rho_0, \rho_1\right)}, \tag{10.40}$$

where D is some representative diameter of the shaft. Finally, we square the brightness to give the shaft a softer appearance.

The remaining task is to calculate the parameters t_1 and t_2 where the ray $\boldsymbol{p} + t\boldsymbol{v}$ intersects the surface of the shaft, and we demonstrate how this can be done for the

Listing 10.7. This pixel shader function performs the final step of rendering a shaft effect for an object-space position $\boldsymbol{p}$ having z coordinate pz. A preceding step calculates values of t1 and t2 and passes them to this function. The vector vdir is the difference $\boldsymbol{c} - \boldsymbol{p}$, where $\boldsymbol{c}$ is the object-space camera position. The object-space camera view direction $\boldsymbol{z}$ is given by cameraView. The pixelCoord parameter specifies the viewport coordinates of the pixel being processed, and it is used to read from the structure buffer. The uniform constants shaftSigma and shaftRho0 correspond to the values of σ and ρ_0 in Equation (10.36), and shaftTau is the value $\tau = -\rho_0/\sigma$. The value of N given by Equation (10.40) is specified by normalizer.

```
uniform TextureRect     structureBuffer;
uniform float3          cameraView;
uniform float           shaftSigma, shaftRho0, shaftTau, normalizer;

float CalculateShaftBrightness(float pz, float3 vdir, float2 pixelCoord,
                               float t1, float t2)
{
    // Read z0 from the structure buffer, calculate t0, and clamp to [t0,1].
    float2 depth = texture(structureBuffer, pixelCoord).zw;
    float t0 = 1.0 + (depth.x + depth.y) / dot(cameraView, vdir);
    t1 = clamp(t1, t0, 1.0); t2 = clamp(t2, t0, 1.0);

    // Limit to range where density is not negative.
    float tlim = (shaftTau - pz) / vdir.z;
    if (vdir.z * shaftSigma < 0.0) {t1 = min(t1, tlim); t2 = min(t2, tlim);}
    else {t1 = max(t1, tlim); t2 = max(t2, tlim);}

    // Evaluate density integral, normalize, and square.
    float B = (shaftSigma * (pz + vdir.z * ((t1 + t2) * 0.5)) + shaftRho0)
            * (t2 - t1) * normalizer;
    return (B * B * dot(vdir, vdir));
}
```

cylinder and box geometries shown in Figure 10.14. Each one sits on the x-y plane and extends through a height h in the positive z direction. We calculate ray intersections only with the lateral surface of the shaft and not with the top and bottom end caps. Clamping the results to a minimum value of zero will prevent integration over any part of the ray reaching beyond the farther cap from the camera's perspective. We do not bother clamping to any maximum value corresponding to the nearer cap, however, because doing so would make a difference only when the camera is in a position between the shaft effect and the light source that is presumably causing it to appear in the first place. This would happen, for instance, if a light shaft were used inside a building to show light shining in, but the camera is outside the window where the shaft would not be visible. In these kinds of cases, the shaft effect can simply be culled. However, if that is not an option, then the values of t_1 and t_2 can be clamped to the maximum value determined by the intersection of the ray with the plane $z = h$.

We render a cylindrical shaft by drawing a tightly-fitting box with inward-facing triangles around it in the same manner that we drew a box around the spherical geometry of a halo. The cylindrical shaft itself is defined by the equation

$$\frac{x^2}{r_x^2} + \frac{y^2}{r_y^2} = 1, \tag{10.41}$$
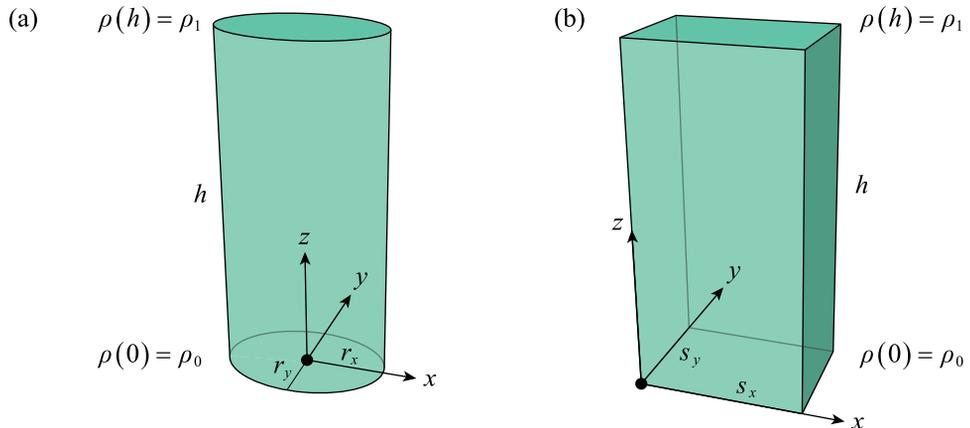
Figure 10.14. Two possible shaft geometries are capped by the planes $z = 0$ and $z = h$. The densities $\rho(z)$ at each end are ρ_0 and ρ_1, respectively. (a) A cylinder shaft is centered on the origin, and it has the radii r_x and r_y along the x and y axes. (b) A box shaft has one corner at the origin, and it extends to the sizes s_x and s_y in the positive x and y directions.

which represents an infinite elliptical cylinder centered on the origin with radii r_x and r_y along the x and y axes. When we multiply both sides by $r_x^2 r_y^2$ and substitute the components of the ray $\boldsymbol{p} + t\boldsymbol{v}$ for x and y, we get

$$r_y^2 \left(p_x + t v_x \right)^2 + r_x^2 \left(p_y + t v_y \right)^2 = r_x^2 r_y^2, \tag{10.42}$$

and this can be rewritten as the quadratic equation

$$\left(r_y^2 v_x^2 + r_x^2 v_y^2 \right) t^2 + 2 \left(r_y^2 p_x v_x + r_x^2 p_y v_y \right) t + r_y^2 p_x^2 + r_x^2 p_y^2 - r_x^2 r_y^2 = 0. \tag{10.43}$$

We define

$$m = \sqrt{\max \left(b^2 - ac, 0 \right)}, \tag{10.44}$$

where $a = r_y^2 v_x^2 + r_x^2 v_y^2$, $b = r_y^2 p_x v_x + r_x^2 p_y v_y$, and $c = r_y^2 p_x^2 + r_x^2 p_y^2 - r_x^2 r_y^2$. We clamp the discriminant to zero, as we did for halos, to handle cases in which the ray misses the cylinder without having to write any special code for it. The parameters t_1 and t_2 where the ray intersects the cylinder are then given by

$$t_{1,2} = \frac{-b \mp m}{a}, \tag{10.45}$$

where the minus sign applies to t_1, and the plus sign applies to t_2. An implementation of this formula is shown in Listing 10.8.

In the case of the box shaft, the inward-facing triangles that we use to render the effect coincide exactly with the shaft itself. The four sides of the lateral surface of a box shaft are defined by the planes

$$\begin{aligned} x = 0, \quad x = s_x \\ y = 0, \quad y = s_y. \end{aligned} \tag{10.46}$$

This encloses a volume that has one corner at the origin and extends to the sizes s_x and s_y in the positive x and y directions. Finding the ray intersections is a simple matter of substituting the components of the ray $\boldsymbol{p} + t\boldsymbol{v}$ for x and y in these equations and solving for t. We need to be careful, however, to correctly associate each plane with the value of either t_1 or t_2 because t_1 must represent the intersection that is farther from the camera. If $v_x > 0$, then the intersection with the plane $x = 0$ is associated with t_1, so we calculate

$$t_1 = -\frac{p_x}{v_x} \quad \text{and} \quad t_2 = \frac{s_x - p_x}{v_x}. \tag{10.47}$$

Otherwise, if $v_x < 0$, then these two values are reversed. A similar determination applies to the y direction, and we calculate two additional values of t_1 and t_2 based on the sign of v_y. The larger value of t_1 and the smaller value of t_2 correspond to the path of the ray lying inside the box volume. Listing 10.9 shows how these values are calculated in a pixel shader.

Listing 10.8. This pixel shader function determines the ray intersection parameters for a cylindrical shaft at the object-space vertex position `pobject` and passes them to the function in Listing 10.7 to calculate the shaft's brightness. The uniform constants `rx2`, `ry2`, and `rx2ry2` hold the values r_x^2, r_y^2, and $r_x^2 r_y^2$ for the cylinder. The object-space camera position c is given by `cameraPosition`. The `pixelCoord` parameter specifies the viewport coordinates of the pixel being processed.

```
uniform float3      cameraPosition;
uniform float       rx2, ry2, rx2ry2;

float CalculateCylinderShaftBrightness(float3 pobject, float2 pixelCoord)
{
    float3 vdir = cameraPosition - pobject;
    float2 v2 = vdir.xy * vdir.xy;
    float2 p2 = pobject.xy * pobject.xy;

    // Calculate quadratic coefficients.
    float a = ry2 * v2.x + rx2 * v2.y;
    float b = -ry2 * pobject.x * vdir.x - rx2 * pobject.y * vdir.y;
    float c = ry2 * p2.x + rx2 * p2.y - rx2ry2;
    float m = sqrt(max(b * b - a * c, 0.0));

    // Calculate limits and integrate.
    float t1 = max((b - m) / a, 0.0);
    float t2 = max((b + m) / a, 0.0);
    return (CalculateShaftBrightness(pobject.z, vdir, pixelCoord, t1, t2));
}
```

Listing 10.9. This pixel shader function determines the ray intersection parameters for a box shaft at the object-space vertex position `pobject` and passes them to the function in Listing 10.7 to calculate the shaft's brightness. The uniform constants `sx` and `sy` hold the sizes of the box. The object-space camera position c is given by `cameraPosition`. The `pixelCoord` parameter specifies the viewport coordinates of the pixel being processed.

```
uniform float3      cameraPosition;
uniform float       sx, sy;

float CalculateBoxShaftBrightness(float3 pobject, float2 pixelCoord)
{
```

```
    float3 vdir = cameraPosition - pobject;
    float t1 = 0.0, t2 = 1.0;

    // Find intersections with planes perpendicular to x axis.
    float a = -pobject.x / vdir.x;
    float b = (sx - pobject.x) / vdir.x;
    if (vdir.x > 0.0) {t1 = max(t1, a); t2 = min(t2, b);}
    else {t1 = max(t1, b); t2 = min(t2, a);}

    // Find intersections with planes perpendicular to y axis.
    a = -pobject.y / vdir.y;
    b = (sy - pobject.y) / vdir.y;
    if (vdir.y > 0.0) {t1 = max(t1, a); t2 = min(t2, b);}
    else {t1 = max(t1, b); t2 = min(t2, a);}

    return (CalculateShaftBrightness(pobject.z, vdir, pixelCoord, t1, t2));
}
```

10.5 Ambient Occlusion

As discussed in Chapter 7, the ambient lighting at any particular point in space is a rough approximation of the illumination coming from every direction due to complex interactions between light and the environment. The ambient light reaching a point on a surface comes from all of the directions covering the hemisphere above the tangent plane at that point. However, the ambient light from some of those directions might be blocked by nearby geometry, and this has the effect of darkening the surface because the total amount of light reaching it is diminished. The fraction of ambient light that is blocked in this way when we consider nearby geometry in every direction over the hemisphere is called *ambient occlusion*.

Determining the correct ambient occlusion at a given point is a very expensive operation that is usually impractical for real-time rendering, especially in dynamic environments where precomputation is not possible. Instead of spending time on a highly accurate simulation, it turns out that we can produce a plausible approximation of ambient occlusion by using the information in the structure buffer to estimate the amount of ambient light reaching the visible geometry at each pixel. This is the basis for a class of techniques generally referred to as *screen-space ambient occlusion (SSAO)*, and they are considerably faster at the cost of accuracy. A wide variety of such techniques exist among a zoo of acronyms, and each one carries out calculations in its own way. In this section, we present a fast and effective method that incorporates elements of a concept called *ambient obscurance* and a technique known as *horizon-based ambient occlusion (HBAO)*.

10.5.1 The Occlusion Buffer

SSAO is applied to a scene by rendering a full-screen pass after the contents of the structure buffer have been established, calculating the fraction H of ambient light blocked for each pixel, and storing the value $1 - \sigma H$ in a single-channel 8-bit *occlusion buffer*, where σ is an adjustable scale factor that controls the overall intensity of the occlusion effect. Despite its name, the occlusion buffer actually stores the fraction of ambient light that reaches the geometry rendered at the pixel location. When the same geometry is later rendered to the color buffer, the fraction of ambient light is fetched from the occlusion buffer at each pixel and used to modify whatever ambient light color has been inserted into the shading calculations.

As shown in Figure 10.15, the results produced by applying SSAO to a scene can be rather striking. In the top row, two scenes containing very little direct lighting are rendered without ambient occlusion. In the middle row, the ambient light level has been reduced by the amount that was determined to be occluded at each pixel. For the indoor scene on the left, this causes interior edges and corners to stand out much more, and it causes the pipes and air ducts to cast an ambient shadow on the walls behind them. For the outdoor scene on the right, ambient occlusion causes the wall to be darkened in areas where light is largely blocked by the vines, and the vines themselves are also affected in such a way that much more geometric detail is visible. In the bottom row of the figure, the unoccluded fraction of ambient light stored in the occlusion buffer is shown for each scene. Ambient occlusion is applied by multiplying the ambient illumination in image (a) by the values in image (c) to produce image (b) in each column.

The fraction H of occluded ambient light is calculated at each pixel by sampling the structure buffer at several surrounding pixels to determine whether any nearby geometry blocks incoming ambient light. The final result written to the occlusion buffer is then based on a weighted average of the occlusion values calculated for all of the samples. In our implementation, we take four samples per pixel at 90-degree increments and at differing radii on a disc parallel to the projection plane. As described below, the sample locations are randomly rotated in every 4×4 group of pixels to effectively produce 64 distinct samples in a small screen area. After the ambient occlusion has been calculated for each pixel, a separate blurring pass smooths the results and eliminates the noise caused by the low sample count.

Let p be the camera-space point lying on the visible surface rendered at the pixel for which we are calculating the ambient occlusion. The depth of p, which we call z_0, is fetched from the z and w channels the structure buffer. Since we are interested in the hemisphere of incoming light directions above the tangent plane
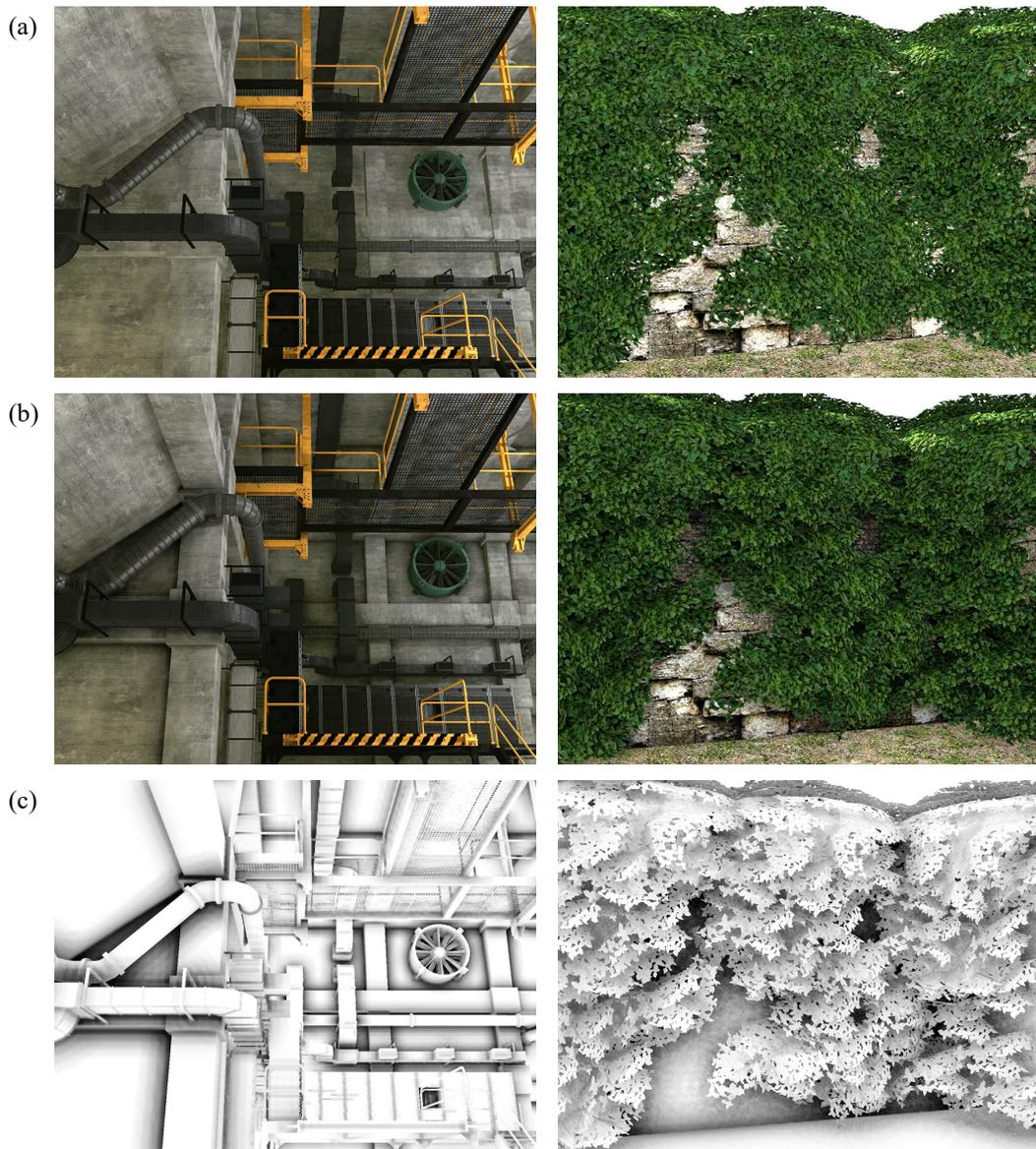
Figure 10.15. (a) Two scenes are rendered without ambient occlusion applied, an interior industrial environment on the left, and an exterior wall with thick vines on the right. (b) Screen-space ambient occlusion is used to scale the per-pixel ambient lighting, producing a better appearance and revealing greater detail. (c) The contents of the corresponding occlusion buffers are shown after the blurring operation has been applied.

at p, we need to calculate the normal vector $\mathbf{n}$. This is easily accomplished by using the derivatives in the structure buffer because the normal vector is given by

$$\mathbf{n} = \left(\frac{\partial z_{\text{camera}}}{\partial x_{\text{camera}}}, \frac{\partial z_{\text{camera}}}{\partial y_{\text{camera}}}, -1 \right). \tag{10.48}$$

We must be careful, however, to recognize that the derivatives in the structure buffer correspond to changes in depth with respect to viewport-space coordinates, not camera-space coordinates. For a viewport of $w \times h$ pixels and a view frustum having an aspect ratio s and a projection plane distance g, the scale factors relating distances between adjacent pixels in viewport space and corresponding distances in camera space at the depth z_0 are

$$\frac{\partial x_{\text{camera}}}{\partial x_{\text{viewport}}} = \frac{2sz_0}{wg} \quad \text{and} \quad \frac{\partial y_{\text{camera}}}{\partial y_{\text{viewport}}} = \frac{2z_0}{hg}. \tag{10.49}$$

Of course, these scale factors are equal because $s = w/h$. When we multiply $\mathbf{n}$ by the first factor, we obtain

$$\mathbf{n} = \left(\frac{\partial z_{\text{camera}}}{\partial x_{\text{viewport}}}, \frac{\partial z_{\text{camera}}}{\partial y_{\text{viewport}}}, -\frac{2s}{wg} z_0 \right), \tag{10.50}$$

which now gives us the normal vector in terms of the exact derivatives and depth value stored in the structure buffer. The scale factor $2s/wg$ is precomputed and passed to the occlusion shader as a constant. We normalize the vector $\mathbf{n}$ to unit length before using it in any calculations.

It is possible to calculate the derivatives needed for the normal vector by taking differences of adjacent depths instead of using derivatives stored in the structure buffer. However, doing so produces errors in the normal vector wherever the derivatives change abruptly, such as edges where two surfaces meet or boundaries where a foreground surface occludes a background surface. This happens because the differences in depths do not correspond to the correct tangent plane of either surface in those locations. Calculating derivatives as geometry is rendered into the structure buffer produces superior results, and Equation (10.50) will not be the only time we use them.

When we sample a neighborhood around a point p, we do so at fixed offsets perpendicular to the z axis in camera space, and this means that the corresponding pixel offsets in the viewport vary with the depth z_0 of p. To determine where we should sample the structure buffer for a camera-space offset Δx_{camera}, we use the inverse of the relationship given by Equation (10.49) to obtain the viewport-space offset

$$\Delta x_{\text{viewport}} = \frac{wg}{2sz_0} \Delta x_{\text{camera}}. \tag{10.51}$$

As z_0 becomes smaller, the viewport-space offsets become larger. This is the main source of performance variation in any particular SSAO technique. Even though the same amount of computation is being done, the larger offsets needed when the camera is closer to the geometry reduces the effectiveness of the GPU texture cache, so more time is spent waiting for data to be fetched from the structure buffer. Fortunately, in cases where a large portion of the viewport contains geometry close to the camera, much less geometry elsewhere tends to be visible, so the increased time needed to render ambient occlusion is often balanced by a decrease in overall scene complexity.

Suppose that we sample the structure buffer at an offset $(\Delta x_{\text{viewport}}, \Delta y_{\text{viewport}})$ from the location of the pixel being rendered by our occlusion shader, and let z be the depth fetched from the structure buffer at that offset. Then define

$$\mathbf{v} = (\Delta x_{\text{camera}}, \Delta y_{\text{camera}}, z - z_0) \tag{10.52}$$

as the camera-space offset vector to a point on some nearby surface that might be occluding the point $\boldsymbol{p}$. Some examples of the vector $\mathbf{v}$ are shown in Figure 10.16. Our goal is to determine, in some reasonable way, how much ambient light should be considered occluded due to the presence of geometry at the point $\boldsymbol{p} + \mathbf{v}$. We will be averaging the results from several different directions, so it's alright to treat the occlusion due to a single sample as a two-dimensional problem in the plane determined by the offset vector $\mathbf{v}$ and the normal vector $\mathbf{n}$.

We assume that the geometry found in the direction of $\mathbf{v}$ occludes ambient light at all angles between $\mathbf{v}$ and the tangent plane at the point $\boldsymbol{p}$ but does not occlude ambient light at any angles between $\mathbf{v}$ and the normal vector $\mathbf{n}$. Let $\alpha_{\mathbf{v}}$ represent the angle between $\mathbf{n}$ and $\mathbf{v}$. Then the fraction $f(\mathbf{v})$ of ambient light occluded due to geometry in the direction of $\mathbf{v}$ is given by

$$f(\mathbf{v}) = \int_{\alpha_{\mathbf{v}}}^{\pi/2} \cos \varphi \, d\varphi = 1 - \sin \alpha_{\mathbf{v}}. \tag{10.53}$$

This is the cosine-weighted integral of the incoming light that is blocked between $\mathbf{v}$ and the tangent plane. It requires no normalization because the total amount of light reaching the surface between $\mathbf{n}$ and $\mathbf{v}$, given by integrating with $\alpha_{\mathbf{v}} = 0$, is simply one. We can calculate the cosine of $\alpha_{\mathbf{v}}$ through the dot product

$$\cos \alpha_{\mathbf{v}} = \mathbf{n} \cdot \hat{\mathbf{v}} = \frac{\mathbf{n} \cdot \mathbf{v}}{\|\mathbf{v}\|}, \tag{10.54}$$

and we can then express $f(\mathbf{v})$ as

$$f(\mathbf{v}) = 1 - \sqrt{1 - (\mathbf{n} \cdot \hat{\mathbf{v}})^2}. \qquad (10.55)$$

Geometry beneath the tangent plane should not occlude any ambient light, so we must ensure that $\alpha_v \leq \frac{\pi}{2}$ by clamping $\mathbf{n} \cdot \hat{\mathbf{v}}$ to zero. In practice, we subtract a small positive constant value τ from $\mathbf{n} \cdot \hat{\mathbf{v}}$ before clamping because it prevents unwanted darkening near the edges between polygons that meet at nearly 180-degree interior angles. Subtracting τ creates an effective tangent plane that makes an angle a little smaller than 90 degrees with the normal vector $\mathbf{n}$, as shown by the red line in Figure 10.16. With these modifications, the revised formula for $f(\mathbf{v})$ is

$$\boxed{f(\mathbf{v}) = 1 - \sqrt{1 - \max(\mathbf{n} \cdot \hat{\mathbf{v}} - \tau, 0)^2}.} \qquad (10.56)$$
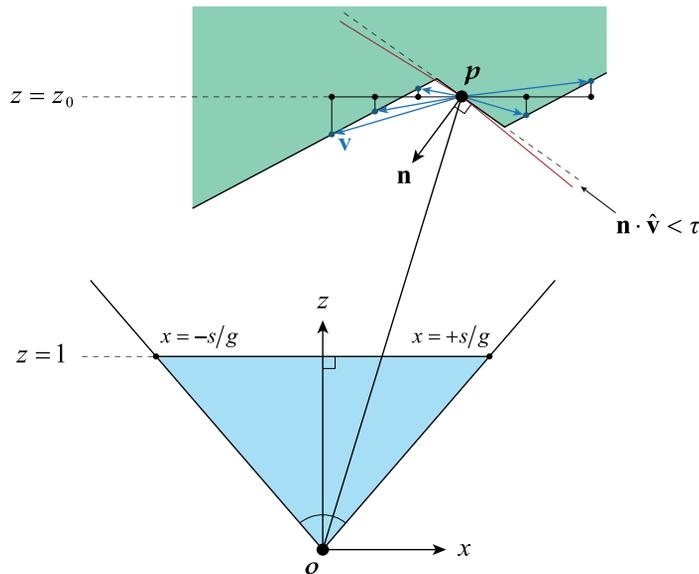
Figure 10.16. The ambient occlusion is calculated at the point p by sampling the structure buffer to obtain the depth at several offsets perpendicular to the camera-space z axis. This produces several vectors $\mathbf{v}$, shown in blue, that correspond to the visible surface location in a neighborhood around p. The three samples to the left of p contribute to the overall ambient occlusion because they are in front of the red line representing the effective tangent plane. The two samples to the right of p make no contribution because they are beneath the tangent plane and thus do not fall in the hemisphere over which ambient light is received.

The incorporation of τ causes a slight decrease in the amount of ambient occlusion that we end up calculating, but the effect is ultimately absorbed by the overall intensity factor σ.

We do not want geometry that is arbitrarily far away to contribute to the ambient occlusion. If it did, then surfaces far in the background would be occluded by objects in the foreground, and the occlusion would be rather strong due to our assumption that the occluding geometry extends all the way to the tangent plane. This would cause dark halos to appear around all foreground objects. To prevent this from happening, we introduce a weighting function $w(\mathbf{v})$ that decreases the amount of occlusion as the magnitude of $\mathbf{v}$ becomes larger. (The presence of this weighting function is what distinguishes techniques using the term *obscurance* from those using the term *occlusion*.) Some SSAO techniques use the magnitude of $\mathbf{v}$ directly in the weighting function, but we will use the dot product $\mathbf{n} \cdot \mathbf{v}$ that will already be available in the shader to derive a weight from the perpendicular distance to the tangent plane. Our weighting function is the simple linear fall-off given by

$$w(\mathbf{v}) = \text{sat}\left(1 - \frac{\mathbf{n} \cdot \mathbf{v}}{d_{\max}}\right), \tag{10.57}$$

where $d_{\max}$ is an adjustable constant corresponding to the distance from the tangent plane at which the weight reaches zero. Note that the vector $\mathbf{v}$ in this formula is not normalized as it is in the formula for $f(\mathbf{v})$.

The final occlusion value $H(\mathbf{v})$ for a single sample is given by the product

$$H(\mathbf{v}) = w(\mathbf{v})f(\mathbf{v}). \tag{10.58}$$

The plot in Figure 10.17 illustrates the amount of occlusion produced by $H(\mathbf{v})$ with respect to components of $\mathbf{v}$ that are parallel and perpendicular to the normal vector $\mathbf{n}$. The angular dependence of $f(\mathbf{v})$ causes the occlusion to fade as the tangential distance from $\mathbf{p}$ increases, and the weighting function $w(\mathbf{v})$ is responsible for fading along the normal direction. The darkest regions near the bottom center of the plot are where the greatest occlusion occurs.

If we take n samples per pixel, then the ambient light factor A written to the occlusion buffer is given by

$$A = 1 - \sigma\left[\frac{\sum_{i=1}^{n} H(\mathbf{v}_i)}{\sum_{i=1}^{n} w(\mathbf{v}_i)}\right], \tag{10.59}$$
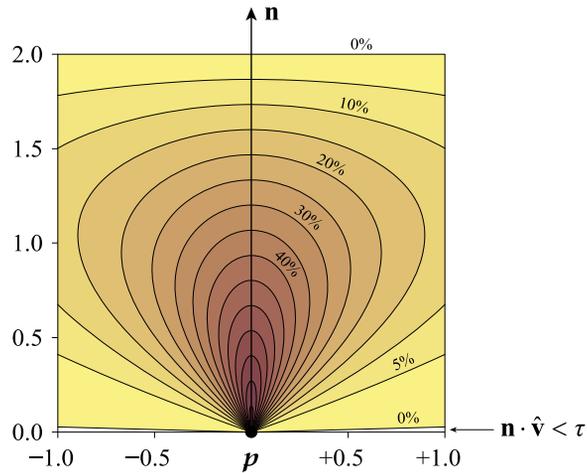
Figure 10.17. This graph shows the ambient occlusion $H(\mathbf{v})$ with respect to the distance along the tangent plane on the horizontal axis and the distance along the normal vector $\mathbf{n}$ on the vertical axis, where we have set $d_{\max} = 2$.

where $\mathbf{v}_i$ is the offset vector for the i-th sample. The constant σ is a tunable scale factor controlling the overall intensity of the ambient occlusion, and it can be set to values greater than one to make the effect more prominent. Equation (10.59) is implemented in the pixel shader code shown in Listing 10.10. This shader fetches four samples from the structure buffer using the offset vectors $\mathbf{v}_1 = \left(\frac{1}{4}r, 0\right)$, $\mathbf{v}_2 = \left(0, \frac{1}{2}r\right)$, $\mathbf{v}_3 = \left(-\frac{3}{4}r, 0\right)$, and $\mathbf{v}_4 = (0, -r)$, where we have chosen a maximum sampling radius of $r = 0.4$.

If we were to use the same four offset vectors for every pixel, then a strong dependence on those directions would show up in the ambient occlusion, and the results would be of low quality. To distribute the offset vectors more isotropically, we apply a rotation of $2\pi k/16$ radians for values of k between 0 and 15 that are randomly assigned within each 4×4 block of pixels. We precompute a small 4×4 texture map having two 16-bit floating-point channels that contain $\cos(2\pi k/16)$ and $\sin(2\pi k/16)$, where all 16 possible values of k are utilized one time each in a random pattern. The rotation for the current pixel is fetched from this texture map by dividing the pixel coordinates by four and setting the wrap mode to repeat. Even though the pixel shader code performs a full 2×2 matrix multiply to rotate the offset vectors, the compiler will recognize that each offset contains a zero in one of its coordinates and optimize the calculation accordingly.

Listing 10.10. This pixel shader function calculates ambient occlusion and returns the ambient light factor A given by Equation (10.59), where we have four samples. The `pixelCoord` parameter specifies the viewport coordinates of the pixel being processed, and it is used to read from the structure buffer and rotation texture. The uniform constant `vectorScale` holds the value $2s/wg$ used in Equation (10.50), and `intensity` holds the overall scale factor σ. The value of τ in Equation (10.56) has been set to $1/32$, and the value of d_{max} in Equation (10.57) has been set to 2.

```glsl
uniform TextureRect    structureBuffer;
uniform Texture2D      rotationTexture;
uniform float          vectorScale, intensity;

float CalculateAmbientOcclusion(float2 pixelCoord)
{
    const float kTangentTau = 0.03125;

    // These are the offset vectors used for the four samples.
    const float dx[4] = {0.1, 0.0, -0.3, 0.0};
    const float dy[4] = {0.0, 0.2, 0.0, -0.4};

    // Sample the structure buffer at the central pixel.
    float4 structure = texture(structureBuffer, pixelCoord);
    float z0 = structure.z + structure.w;

    // Calculate the normal vector.
    float scale = vectorScale * z0;
    float3 normal = normalize(float3(structure.xy, -scale));
    scale = 1.0 / scale;

    // Fetch a cos/sin pair from the 4x4 rotation texture.
    float2 rot = texture(rotationTexture, pixelCoord * 0.25).xy;
    float occlusion = 0.0;
    float weight = 0.0;

    for (int i = 0; i < 4; i++)
    {
        float3    v;

        // Calculate the rotated offset vector for this sample.
        v.x = rot.x * dx[i] - rot.y * dy[i];
        v.y = rot.y * dx[i] + rot.x * dy[i];

        // Fetch the depth from the structure buffer at the offset location.
        float2 depth = texture(structureBuffer, (pixelCoord + v.xy * scale)).zw;
        v.z = depth.x + depth.y - z0;

        // Calculate w(v) and f(v), and accumulate H(v) = w(v)f(v).
        float d = dot(normal, v);
        float w = saturate(1.0 - d * 0.5);
        float c = saturate(d * rsqrt(dot(v, v)) - kTangentTau);
```

```
        occlusion += w - w * sqrt(1.0 - c * c);
        weight += w;
    }

    // Return the ambient light factor.
    return (1.0 - occlusion * intensity / max(weight, 0.0001));
}
```

10.5.2 Depth-Aware Blurring

After the ambient light factor has been rendered with the pixel shader shown in Listing 10.10, it has a noisy appearance due to the low number of per-pixel samples, as shown in Figure 10.18(a). Increasing the number of samples reduces the noise to a degree, but it comes at a high cost in terms of performance, and it doesn't eliminate the noise completely. A much faster and more productive course of action is to blur the contents of the occlusion buffer so that the final appearance of each pixel is influenced by a blend of its own samples and the samples belonging to its neighbors. Because the offset vectors are different for every pixel in a 4×4 block, we can effectively make use of 64 total samples at each pixel by averaging all 16 rotations of the four samples originally taken per pixel. The result is the smooth ambient light factor shown in Figure 10.18(b).

The smoothness of the ambient occlusion can be increased by distributing larger numbers of rotations over larger $n \times n$ blocks and adjusting the coordinates
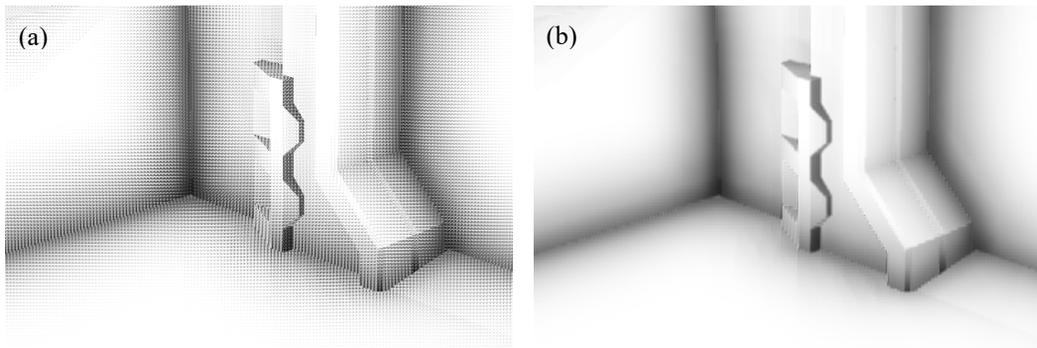
Figure 10.18. (a) The contents of the occlusion buffer have a noisy appearance due to the low number of samples taken per pixel, and a stipple pattern is visible due to the repeating offset vectors every fourth pixel. (b) Every 4×4 block of pixels has been averaged to distribute the samples for all 64 offset vectors to each pixel, and the noise has been eliminated.

used to fetch from the rotation texture accordingly. The blurring stage must then average an $n \times n$ area to produce an image derived from an effective $4n^2$ samples. It is important to realize that for the to blurring work well, the sampling pattern used in every $n \times n$ block has to be the same so that an $n \times n$ blur at any pixel location, aligned to a block boundary or not, ends up collecting samples from the same set of offset vectors, just in different relative positions. Otherwise, if different $n \times n$ blocks have different arrangements of rotation angles, then stipple patterns tend to persist after the blurring operation.

When we blur the occlusion buffer, we want to avoid including values belonging to different surfaces that are separated by a significant distance in depth. If we did include those, then ambient occlusion from one surface could be incorrectly smeared onto an unrelated surface, leading to a poor appearance. In order to limit the blurring operation to the right range of depths, we use the gradient information in the structure buffer to estimate the slope of the surface at the location where we are gathering samples. The absolute values of the derivatives $\partial z / \partial x$ and $\partial z / \partial y$ tell us how much the depth changes for an offset of one pixel in either the x or y direction. Since we are taking samples in many different directions, we simply use the larger derivative as an approximate change in depth per pixel. Multiplying by the maximum pixel offset Δp used by the blur gives us the formula

$$\Delta z = \Delta p \left[\max\left(\left| \frac{\partial z}{\partial x} \right|, \left| \frac{\partial z}{\partial y} \right| \right) + \delta \right] \qquad (10.60)$$

for the maximum difference Δz in depth for which a nearby pixel should be considered part of the same surface. The extra term δ is an adjustable value that we include to account for the fact that the surface may not actually be flat within the distance Δp and could acquire a larger slope away from the center pixel. Calling the depth of the center pixel z_0, only pixels in the occlusion buffer having a depth z such that $|z - z_0| < \Delta z$ are included in the average.

The blur operation requires that we actually have two occlusion buffers with the same single-channel format. The first occlusion buffer holds the original ambient light factors generated by Listing 10.10, and the second occlusion buffer receives the results of the blur operation. Later, when the color buffer is rendered, only the second occlusion buffer is accessed to determine how much ambient light reaches each pixel, and the first occlusion buffer is no longer needed.

The shader code shown in Listing 10.11 implements the depth-aware blur operation. It reads the structure buffer at the center pixel to determine its depth z_0 and the maximum change in depth Δz. It then samples the occlusion buffer and structure buffer four times using pixel offsets of -0.5 and 1.5 in each of the x and y directions. The half-pixel offsets allow the shader to take advantage of the filtering

hardware in order to average four values together. (Of course, this requires that bilinear filtering be enabled for both the occlusion and structure buffers.) In this way, 16 samples can be considered with only four texture fetches, but there is slight loss of quality due to the fact that we can use only one filtered depth z for each group of four samples. Any filtered samples satisfying $|z - z_0| < \Delta z$ are accumulated, and we divide by the number of passing samples at the end to obtain an average. In the case that no samples are accepted, we resort to calculating an average of all 16 samples with no regard for their depths.

Listing 10.11. This pixel shader function performs a depth-aware blur of a 4×4 area of the occlusion buffer. The range of valid depth values for each sample is determined by the gradient calculation, where $\delta = 1/128$.

```
uniform TextureRect    structureBuffer;
uniform TextureRect    occlusionBuffer;

float BlurAmbientOcclusion(float2 pixelCoord)
{
    const float kDepthDelta = 0.0078125;

    // Use depth and gradient to calculate a valid range for the blur samples.
    float4 structure = texture(structureBuffer, pixelCoord);
    float range = (max(abs(structure.x), abs(structure.y)) + kDepthDelta) * 1.5;
    float z0 = structure.z + structure.w;

    float2 sample = float2(0.0, 1.0); float3 occlusion = float3(0.0, 0.0, 0.0);
    for (int j = 0; j < 2; j++)
    {
        float y = float(j * 2) - 0.5;
        for (int i = 0; i < 2; i++)
        {
            float x = float(i * 2) - 0.5;

            // Fetch a filtered sample and accumulate.
            float2 sampleCoord = pixelCoord + float2(x, y);
            sample.x = texture(occlusionBuffer, sampleCoord).x;
            occlusion.z += sample.x;

            // If depth at sample is in range, acculumate the occlusion value.
            float2 depth = texture(structureBuffer, sampleCoord).zw;
            if (abs(depth.x + depth.y - z0) < range) occlusion.xy += sample;
        }
    }

    // Divide the accumulated occlusion value by the number of samples that passed.
    return ((occlusion.y > 0.0) ? occlusion.x / occlusion.y : occlusion.z * 0.25);
}
```

10.6 Atmospheric Shadowing

In the presence of even a slight fog, strong illumination from a light source like the sun is scattered to a noticeable degree by the atmosphere filling the scene. As discussed in Section 8.5, some fraction of the light reaching every tiny volume of air is redirected toward the camera, and this causes the fog to have a uniform brightness. What we did not previously address, however, is what happens when the light is blocked at some locations. Solid objects not only cast shadows on other solid objects but also on the molecules floating around in between. Any volume of air that the light doesn't actually travel through obviously cannot scatter any of that light toward the camera. As a result, scenes containing prominent shadow-casting geometry, like the forest shown in Figure 10.19(a), realistically contain many shafts of lighter and darker air volumes perceived as rays emanating from the light source. The visual effect goes by many names, some of the most common of which are *sun rays*, *god rays*, and *crepuscular rays*.

To accurately render fog that accounts for atmospheric shadowing and produces sun rays, we must be able to estimate what portion of any line of sight starting at the camera position is exposed to the light source or is otherwise in shadow. Fortunately, if the scene has been rendered with the use of a cascaded shadow map, then we already have all of the information we need. However, we need to access a large amount of that information for each ray traced from the camera position in order to calculate a good approximation of the inscattered light. Through the use of some clever engineering, we can achieve a high-quality result while maintaining surprisingly good performance.

10.6.1 The Atmosphere Buffer

For atmospheric shadowing, the basic idea is that we generate a ray for each pixel's location on the projection plane and sample it at many points within some fixed range of distances $[d_{min}, d_{max}]$ from the camera. For each ray, we accumulate the inscattering contributions from samples that are not in shadow and store the sum in a dedicated one-channel *atmosphere buffer*, as shown in Figure 10.19(b). Our technique needs to work with bright rays when looking toward the light source and subtle variances when looking in other directions. Because the values calculated for each ray can cover a wide range of magnitudes, 16-bit floating-point is the most effective storage format. An 8-bit format, even with gamma compression, is not sufficient.

Naturally, taking a greater number of samples per ray produces better results, but doing so comes at the cost of performance. Since the rays that we render do
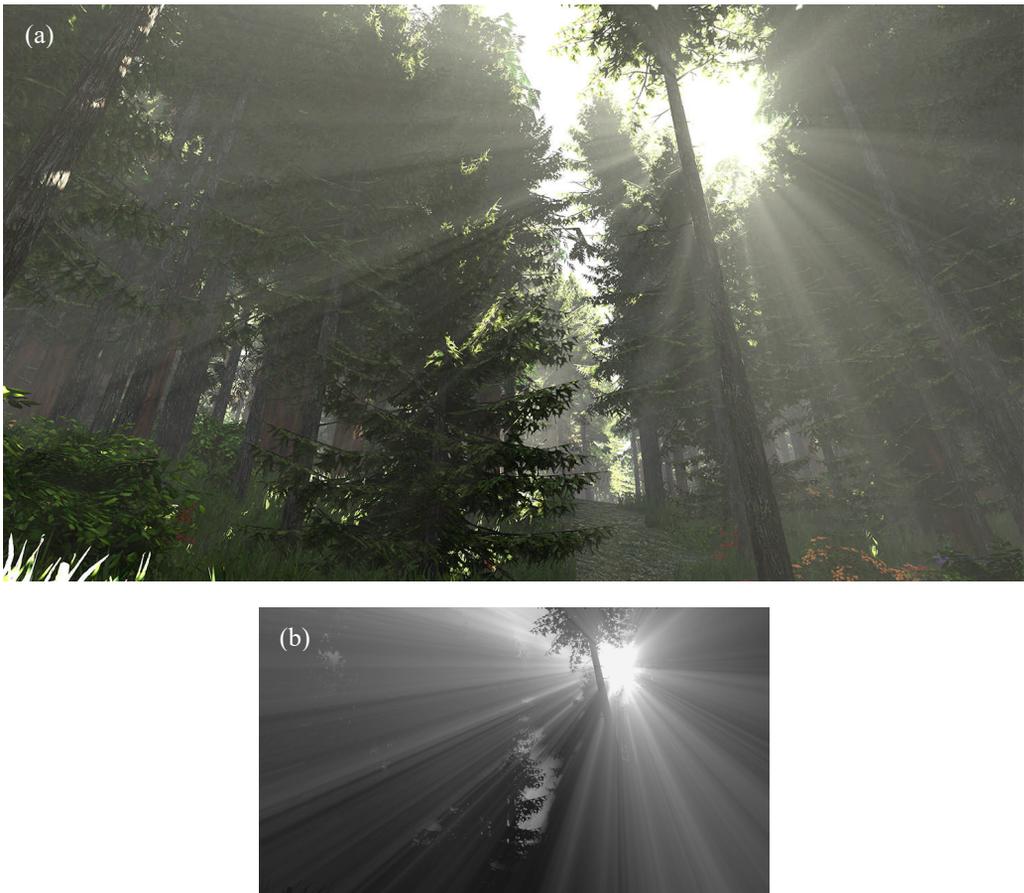
Figure 10.19. (a) Atmospheric shadowing produces accurate sun rays in a forest setting by integrating light visibility along paths through the shadow map. (b) The amount of inscattered light at each pixel, multiplied by three for high visibility in this image, is stored in a half-resolution frame buffer and later added to the main scene during postprocessing.

not need to include sharp details, one easy step we can take to reduce the overall computational cost is to make the atmosphere buffer half the resolution of the main frame buffer so we render only one quarter the number of pixels. This is illustrated by the difference in frame buffer sizes shown in Figure 10.19.

The contents of the atmosphere buffer are generated by rendering a full-screen pass and interpolating points in a plane $\mathbf{f}$ lying at the distance $d_{\min}$ in front of the camera. For a view frustum have projection distance g and aspect ratio s, a vertex

position v specified in normalized device coordinates, as shown in Figure 5.17, is transformed into a camera-space point q in the plane $\mathbf{f}$ by the formula

$$q = d_{\min}\left(\frac{s}{g}v_x, \frac{1}{g}v_y, 1\right). \tag{10.61}$$

These points correspond to the directions of rays emanating from the camera position at the origin. They are calculated by the vertex shader and linearly interpolated across the entire viewport. In the pixel shader, we normalize these directions to a length of $d_{\min}$ by multiplying by the factor $d_{\min}/\|q\|$ so we are working with radial distances from the camera position. This ensures that the sample positions along any particular direction do not move when the camera is rotated.

In the vertex shader, we also transform the ray direction $\mathbf{q}_{xyz}$, as a 3D vector, from camera space to shadow space for cascade 0 to produce a vector $\mathbf{r}$ representing the ray direction relative to the volume of space covered by the shadow map. The vector $\mathbf{r}$ is given by

$$\mathbf{r} = \mathbf{M}_{\text{shadow}}^{0}\mathbf{M}_{\text{camera}}\mathbf{q}_{xyz}, \tag{10.62}$$

where $\mathbf{M}_{\text{camera}}$ is the matrix that transforms from camera space to world space, and $\mathbf{M}_{\text{shadow}}^{0}$ is the matrix given by Equation (8.81) that transforms from world space into shadow space for cascade 0. The vector $\mathbf{r}$ is also interpolated over the entire viewport and multiplied by the factor $d_{\min}/\|q\|$ in the pixel shader to obtain the offset from the camera position at which sampling begins. The shadow-space camera position c for cascade 0 is a constant given by

$$c = \mathbf{M}_{\text{shadow}}^{0}\mathbf{M}_{\text{camera}[3]}. \tag{10.63}$$

Points p_1 and p_2 in shadow space representing the minimum and maximum distances along the ray at which we sample the shadow map are finally given by

$$p_1 = c + \frac{d_{\min}}{\|q\|}\mathbf{r} \quad \text{and} \quad p_2 = c + \left(\frac{d_{\max}}{d_{\min}}\right)\frac{d_{\min}}{\|q\|}\mathbf{r}. \tag{10.64}$$

As we take steps along a ray, we must stop sampling once the camera-space depth exceeds the depth of the scene geometry in the ray's direction because the ray has penetrated an opaque surface. The maximum allowable depth $z_{\max}$ of any sample is provided by the depth stored in the structure buffer. Since the dimensions of the atmosphere buffer are half those of the structure buffer, we must be careful to sample the structure buffer at double the viewport coordinates of the pixel being

rendered. So that we are able to determine the depth of any particular sample, we first calculate the depths z_1 and z_2 corresponding to the points $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ where sampling begins and ends. These depths are equal to the z coordinates of the direction $\mathbf{q}_{xyz}$ after it has been scaled to the lengths d_{min} and d_{max}. Since $q_z = d_{min}$, the formulas for z_1 and z_2 can be written as

$$z_1 = \frac{d_{min}^2}{\|\boldsymbol{q}\|} \quad \text{and} \quad z_2 = \frac{d_{max}}{d_{min}} z_1. \tag{10.65}$$

The depth at each sample position is also used to determine which shadow cascade provides visibility information at that position. We read from cascade 0 until the sample depth exceeds the cascade's maximum depth. We then proceed to the cascade 1 and read from it until we've exceeded its maximum depth, and so on. Since no visibility information exists beyond the final cascade, the value of d_{max} should never be greater than that cascade's maximum depth. When reading from cascade $k > 0$, we multiply the sample position by the matrix $\mathbf{C}_k$ given by Equation (8.83) that scales and offsets the shadow-space coordinates in cascade 0 to match the bounding box of cascade k.

The most straightforward way to take samples along a ray would be to divide the line segment between $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ into n equally sized steps. The sample positions would be given by

$$\boldsymbol{p}(t) = (1-t)\,\boldsymbol{p}_1 + t\boldsymbol{p}_2, \tag{10.66}$$

where t ranges from zero to one and increments by $\Delta t = 1/n$ at each step. Including samples at both $t = 0$ and $t = 1$, this produces $n+1$ equally weighted samples that each represent a length of $(d_{max} - d_{min})\,\Delta t$ along the ray. However, because shadow map cascades farther from the camera store information about the visibility of the light source at a lower density than cascades closer to the camera, taking equally sized steps tends to cause oversampling in the higher numbered cascades and may fail to take advantage of the greater detail available closer to the camera. We would like to take more samples closer to the camera where shadow map texels cover smaller volumes of space and fewer samples farther away where shadow map texels are more spread out. This can be accomplished by remapping the parameter t to another parameter u in such a way that samples are redistributed over the range $[0,1]$ so that more of them are closer to the point $\boldsymbol{p}_1$. There are many ways we could do this, but a simple choice is to set $u(t) = at^2 + bt$. By requiring that $u(1) = 1$ and that du/dt is equal to a specific slope m at $t = 0$, we have

$$u(t) = (1-m)\,t^2 + mt. \tag{10.67}$$

We still begin sampling with $t = 0$, and we still increment t by $1/n$ at each step, but now the sample positions are given by

$$\boldsymbol{p}(u) = (1 - u(t))\, \boldsymbol{p}_1 + u(t)\, \boldsymbol{p}_2. \tag{10.68}$$

The sample positions produced by Equation (10.68) are distributed like those shown in Figure 10.20, where we have set $m = 0.25$. Using the same parameter u, the camera-space depth of a sample is given by $z(u) = (1 - u)\, z_1 + u z_2$, and this is used both to determine when we have hit solid geometry and to select which shadow map cascade is sampled. Even though the cascade index is based on the depth $z(u)$, the sample positions are based on radial distance from the camera so that the distribution of samples along any particular world-space direction does not change as the camera rotates in place. Otherwise, if sample distributions varied with the direction of a ray, the calculated brightness of the inscattered light could change with camera orientation, possibly leading to flickering artifacts.
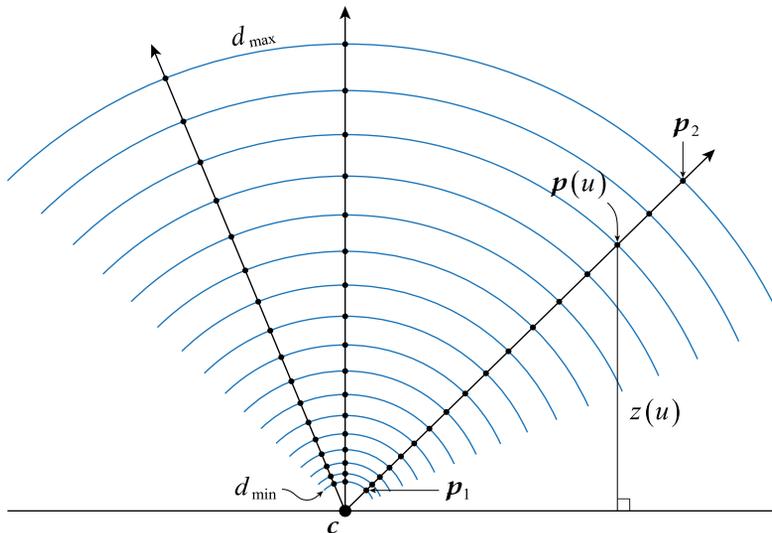
Figure 10.20. Along any ray starting at the camera position $\boldsymbol{c}$ in shadow space, sample positions are distributed nonlinearly between the radial camera-space distances $d_{\min}$ and $d_{\max}$. The parameter u is given by Equation (10.67), where $m = 0.25$ in this example. The shadow-space sample position is given by $(1 - u)\, \boldsymbol{p}_1 + u \boldsymbol{p}_2$, and the camera-space depth $z(u)$ is given by $(1 - u)\, z_1 + u z_2$.

Samples having a nonlinear distribution cannot be weighted equally because they represent intervals of varying length along each ray. The portion of the ray corresponding to a sample taken at a specific value of t can be approximated by evaluating the derivative

$$\frac{du}{dt} = 2(1-m)t + m \qquad (10.69)$$

and multiplying it by the length $(d_{max} - d_{min})\Delta t$ of a fixed-size step. For the i-th sample along the ray, we have $t = i/n$, and the length ε_i represented by the sample is given by

$$\varepsilon_i = (d_{max} - d_{min})\Delta t \left.\frac{du}{dt}\right|_{t=i/n} = \frac{d_{max} - d_{min}}{n}\left[\frac{2(1-m)}{n}i + m\right]. \qquad (10.70)$$

The sum of all the per-sample lengths, where sample $i = 0$ corresponds to $t = 0$ and sample $i = n$ corresponds to $t = 1$, is then

$$\sum_{i=0}^{n}\varepsilon_i = \frac{d_{max} - d_{min}}{n}\sum_{i=0}^{n}\left(\frac{2(1-m)}{n}i + m\right) = \frac{n+1}{n}(d_{max} - d_{min}), \qquad (10.71)$$

which is independent of the starting slope m. If we assign the weight w_i given by

$$w_i = \frac{2(1-m)}{n}i + m \qquad (10.72)$$

to each sample, then the sum of all weighted sample values along a ray is normalized to the full length of the ray with the constant factor

$$\frac{d_{max} - d_{min}}{n+1}. \qquad (10.73)$$

As we step along a ray, we start with the weight $w_0 = m$ for the first sample and simply add the constant $\Delta w = 2(1-m)/n$ at each step to obtain the weights for successive samples.

The final brightness B of the inscattered light along a ray is given by the sum

$$B = \lambda\frac{d_{max} - d_{min}}{n+1}\sum_{i=0}^{n}w_i s_i, \qquad (10.74)$$

where s_i is the light visibility value read from the shadow map at the position of sample i. Due to filtering applied when the shadow map is read, each s_i can have a

value anywhere between zero and one. For any samples having positions beyond a solid surface, $s_i = 0$, but the normalization factor remains the same. The constant λ is a density value, a configurable property of the atmosphere controlling how much light is scattered toward the camera per unit distance.

10.6.2 Sample Randomization

When Equation (10.74) is used to render atmospheric shadowing, the result is the unappealing pattern shown in Figure 10.21(a). Here, a maximum of 65 samples have been read from the shadow map over a distance of 40 meters along each ray, but this rather large number of samples is still inadequate for generating an image of acceptable quality. To overcome this problem without increasing the number of samples, we reach into our bag of tricks and once again pull out random noise. By adding a different random value in the range $[0, 1/n]$ to the parameter t for each ray, we produce radial displacements that fill in the gaps between the concentric spheres on which all samples were previously positioned. The value of t for the i-th sample is then given by

$$t = (i + \delta)/n, \tag{10.75}$$

where $\delta \in [0,1]$ is fetched from a noise texture. In a manner similar to how random rotations diversified the sample positions over a small neighborhood of pixels for ambient occlusion in Section 10.5, random displacements along each ray greatly increase the spatial coverage of the samples within a small neighborhood for atmospheric shadowing. Adding noise produces the much smoother rays shown in Figure 10.21(b), resulting in a obviously superior image.

The fact that noise was used to generate the image in Figure 10.21(b) is easily detected upon close inspection. In our ambient occlusion method, the appearance of noise was eliminated by blurring neighborhoods with the same size as the noise texture. Unfortunately, we cannot employ the same strategy here because the light shafts are shining through open space and are not associated with a solid surface having a specific depth at every pixel. Using a similar blur operation, there would be no reliable way to prevent light shafts in the background from bleeding into solid objects obscuring them in the foreground. However, we can hide the noise to a large degree by shifting the alignment of the noise texture within the viewport by a random offset every frame. At typical intensity levels, this makes the noise almost completely unnoticeable, especially when the camera is moving. It is also still possible to apply a small amount of directional blurring when the light shafts are added to the final image in postprocessing, as discussed below.
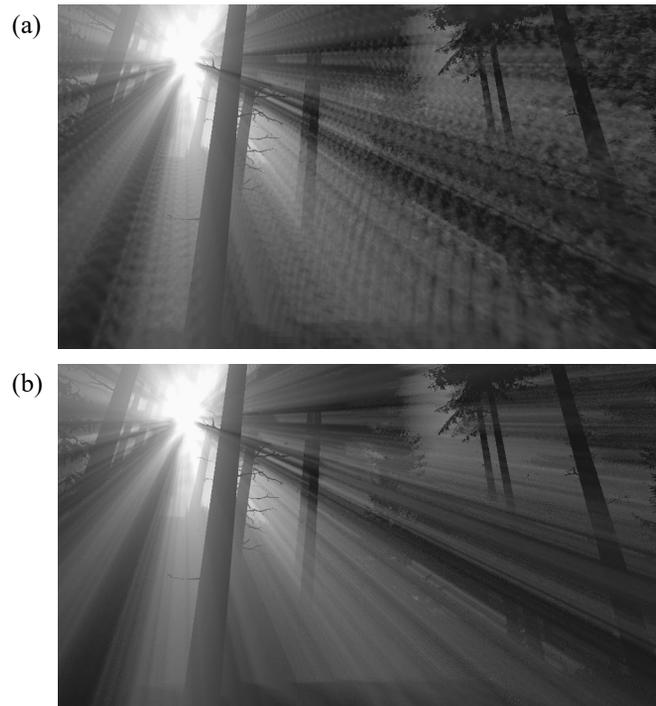
Figure 10.21. Atmospheric shadowing is rendered using up to 65 samples per ray over a distance of 40 meters. (a) The i-th sample along every ray is placed at the same distance from the camera according to the parameter value $t = i/n$. (b) A random constant $\delta \in [0,1]$ is fetched from a 32×32 noise texture for each ray and added to all parameter values for the ray so that $t = (i + \delta)/n$.

10.6.3 Anisotropic Scattering

Light from a directional source like the sun is not scattered isotropically by the atmosphere. More of the light tends to be deflected at small angles without changing its direction by very much, and less of the light tends to be deflected at large angles for which the light is redirected off to the side or back toward its source. For this reason, sun rays can be prominent when the camera is looking mainly toward the light source but can be significantly less apparent when looking away. We can account for this anisotropy by multiplying the final brightness B calculated for a ray by a function of the angle between the direction of the ray and the direction to the light.

A physically-based model for anisotropic scattering is implemented by a function $\Phi_g(\alpha)$ called the *Henyey-Greenstein phase function*, defined as

$$\Phi_g(\alpha) = \frac{1-g^2}{4\pi\left(1+g^2-2g\cos\alpha\right)^{3/2}}. \tag{10.76}$$

The angle α represents the angle between the direction of the incoming light and the direction to the camera where the scattered outgoing light is observed. When this angle is less than 90 degrees, the light is said to be *forward scattered* because it is still travelling away from its source. When this angle is greater than 90 degrees, the light is said to be *backscattered*.

The cosine of α is conveniently calculated with the dot product

$$\cos\alpha = \mathbf{l} \cdot \frac{\mathbf{q}}{\|\mathbf{q}\|}, \tag{10.77}$$

where $\mathbf{l}$ is the unit-length camera-space direction to the light, and $\mathbf{q}$ is the camera-space ray direction given by Equation (10.61). The value of g is usually chosen to be in the range $[0,1)$, and it controls the degree of anisotropy. If $g = 0$, then light is scattered equally in all directions. If $g > 0$, then more light tends to be scattered in directions closer to that of the incoming light, generally continuing forward, and this happens to a greater degree as g gets larger. Polar plots illustrating the distribution of scattered rays as a function of the angle α for several values of g are shown in Figure 10.22(a).

As the value of g goes up, the largest values produced by $\Phi_g(\alpha)$, in directions near $\alpha = 0$, increase considerably. The maximum intensity produced for any value of g is given by

$$\Phi_g(0) = \frac{1+g}{4\pi\left(1-g\right)^2}. \tag{10.78}$$

The value of g would typically be adjusted to achieve some preferred level of anisotropy, but making these adjustments has a large effect on the apparent brightness of the sun rays, which is undesirable. We can turn the level of anisotropy and the overall brightness of the effect into independent parameters by normalizing $\Phi_g(\alpha)$ so that the maximum intensity is always one. This is done by dividing $\Phi_g(\alpha)$ by $\Phi_g(0)$ to create a new function $\hat{\Phi}_g(\alpha)$ given by

$$\boxed{\hat{\Phi}_g(\alpha) = \frac{\Phi_g(\alpha)}{\Phi_g(0)} = \left(\frac{1-g}{\sqrt{1+g^2-2g\cos\alpha}}\right)^3.} \tag{10.79}$$
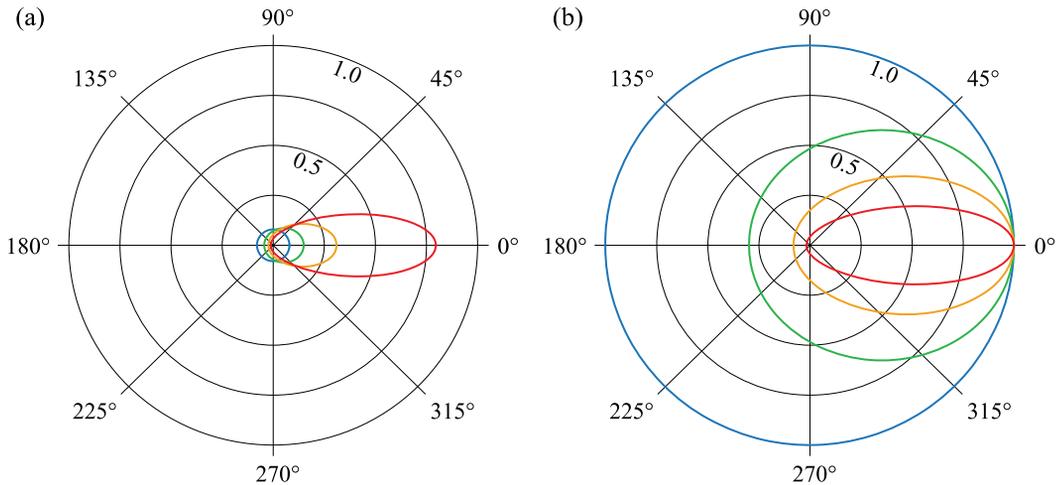
Figure 10.22. (a) The Henyey-Greenstein phase function $\Phi_g(\alpha)$ is plotted with the polar angle α for anisotropy values $g = 0$ (blue), $g = 0.2$ (green), $g = 0.4$ (orange), and $g = 0.6$ (red). Higher values of g result in greater maximum intensity at $\alpha = 0°$. (b) The normalized function $\hat{\Phi}_g(\alpha)$ is plotted for the same four values of g, and the maximum intensity is always one.

Figure 10.22(b) illustrates the intensity produced by this function at all angles for the same values of g shown in Figure 10.22(a).

Instead of specifying the value of g directly, we can calculate the value of g that produces a chosen minimum intensity R for the function $\hat{\Phi}_g(\alpha)$. This minimum intensity occurs at the angle $\alpha = 180°$, where light is backscattered directly toward its source. When we solve the equation $R = \hat{\Phi}_g(180°)$ for g, we get

$$g = \frac{1 - \sqrt[3]{R}}{1 + \sqrt[3]{R}}. \tag{10.80}$$

Since the maximum forward scattered intensity of $\hat{\Phi}_g(\alpha)$ is always one, the value of R can be thought of as the ratio of backscattered light to forward scattered light. The value of R can be greater than one, in which case $g < 0$. This has the effect of making the backscattered light brighter than the forward scattered light.

10.6.4 Implementation

As mentioned earlier, the intensity of the inscattered light along each ray is stored in a half-resolution atmosphere buffer. This buffer can be filled at any time after the

cascaded shadow map for the light source has been generated. The contents of the atmosphere buffer are later combined with the main color buffer during the post-processing stage.

To fill the atmosphere buffer, we render a full-screen pass using vertex positions specified directly in normalized device coordinates. These positions require no transformation in the vertex shader and are passed through unchanged. As demonstrated in Listing 10.12, the vertex shader calculates the camera-space point q given by Equation (10.61) and the shadow-space ray direction $\mathbf{r}$ given by Equation (10.62). The uniform constant `frustumParams` provides the values derived from d_{min}, $1/g$, and s/g needed to calculate the ray direction in camera space. The uniform constant `shadowMatrix` holds the upper-left 3×3 portion of the matrix $\mathbf{M}^0_{shadow} \mathbf{M}_{camera}$ that transforms the ray direction into shadow space as an array of three vectors.

The camera-space point q and shadow-space ray direction $\mathbf{r}$ calculated by the vertex shader are interpolated over the entire screen and fed into the pixel shader shown in Listing 10.13. Only the x and y coordinates of q are interpolated because it is always the case that $q_z = d_{min}$. This pixel shader calculates the brightness of the inscattered light using Equation (10.74) and multiplies it by the anisotropic intensity level given by Equation (10.79). The result is stored in the atmosphere buffer.

The first thing the pixel shader does is read the structure buffer to determine the maximum depth at which the ray will be sampled. Because the structure buffer is twice the size of the atmosphere buffer, it must be accessed at coordinates that are double those of the pixel being rendered. As samples are taken along the ray in each shadow map cascade, the limiting depth is always the smaller of the depth fetched from the structure buffer and the farthest depth covered by the cascade.

The shadow-space endpoints p_1 and p_2 are calculated using Equation (10.64), where the camera position c is supplied by the uniform constant `shadowCameraPosition`, and the ratio d_{max}/d_{min} is given by `atmosphereDepthRatio`. The depths z_1 and z_2 at the endpoints are calculated using Equation (10.65).

The normalized Henyey-Greenstein function is evaluated by calculating the dot product $(\mathbf{l} \cdot q)/\|q\|$ and plugging it into Equation (10.79) as $\cos \alpha$. The camera-space direction to the light $\mathbf{l}$ is supplied by the uniform constant `cameraLightDirection`, and various constants derived from the anisotropy parameter g are given by `anisotropyConst`. The resulting value of $\hat{\Phi}_g$ is multiplied by the constant $\lambda(d_{max} - d_{min})/(n+1)$ appearing in Equation (10.74), which is passed in through `atmosphereBrightness`, to produce an overall intensity level.

The final preparatory step before ray sampling begins consists of fetching the ray parameter offset δ from a 32×32 noise texture containing a single channel of random values and having a wrap mode set to repeat. The coordinates at which the

Listing 10.12. This vertex shader function calculates the camera-space point q and returns its x and y coordinates in the `cameraRay` parameter. The corresponding shadow-space ray direction $\mathbf{r}$, transformed by `shadowMatrix` constant, is returned in the `shadowRay` parameter. The input parameter `position` is the vertex position v in normalized device coordinates.

```
uniform float3    frustumParams;      // (dmin * s/g, dmin * 1/g, dmin)
uniform float3    shadowMatrix[3];

void CalculateAtmosphericShadowingRays(float2 position,
                                       out float2 cameraRay, out float3 shadowRay)
{
    // Calculate point on camera-space plane z = dmin.
    float3 q = float3(position.xy * frustumParams.xy, frustumParams.z);
    cameraRay = q.xy;

    // Transform camera-space ray direction into shadow space for cascade 0.
    shadowRay.x = dot(q, shadowMatrix[0]);
    shadowRay.y = dot(q, shadowMatrix[1]);
    shadowRay.z = dot(q, shadowMatrix[2]);
}
```

offset δ is fetched are determined by dividing the atmosphere buffer pixel coordinates by 32 and adding a random shift between 0 and 31 texels in both the x and y directions. A different shift should be specified for each frame in the two-component `noiseShift` constant so that the ray sampling pattern changes rapidly.

The first samples are taken along the ray in shadow map cascade 0. The sampling continues in a loop until the depth of a sample exceeds either the depth that was fetched from the structure buffer or the farthest depth covered by the first cascade. The farthest depths covered by all four cascades are passed to the shader in the `maxCascadeDepth` array. At each step, the parameter t along the ray is incremented by the fixed value of $1/n$, but sample positions are based on the parameter $u(t)$ given by Equation (10.67). The sample weight is always incremented by the constant $2(1-m)/n$ appearing in Equation (10.72).

After the sample position has exited the first cascade, samples are taken in the remaining three cascades inside a separate loop. If the depth fetched from the structure buffer has been exceeded, then the shader code quickly breaks out of these loops. Samples in higher-numbered cascades are handled in a manner similar to how they were handled in the first cascade. However, sample positions are calculated in the space of the first cascade, so they need to be transformed by the matrix $\mathbf{C}_k$ into the cascade currently being sampled. For cascade k, the scale is specified by the uniform constant `cascadeScale[k−1]`, and the translation is specified by `cascadeOffset[k−1]`.

Listing 10.13. This pixel shader function calculates atmospheric shadowing and returns the intensity of inscattered light sampled for a single ray. The `pixelCoord` parameter specifies the viewport coordinates of the pixel being processed, and it is used to read from the structure buffer and noise texture. The `cameraRay` and `shadowRay` parameters are the interpolated outputs of the vertex shader function shown in Listing 10.12. The number of steps n has been set to 64, and the initial slope m of $u(t)$ has been set to 0.25.

```
uniform TextureRect        structureBuffer;
uniform Texture2DArrayShadow shadowTexture;
uniform Texture2D          noiseTexture;
uniform float              minAtmosphereDepth;    // dmin
uniform float              atmosphereDepthRatio;  // dmax / dmin
uniform float              atmosphereBrightness;  // lambda * (dmax - dmin) / (n + 1)
uniform float              maxCascadeDepth[4];
uniform float3             cascadeScale[3];
uniform float3             cascadeOffset[3];
uniform float3             shadowCameraPosition;
uniform float3             cameraLightDirection;
uniform float3             anisotropyConst;       // (1 - g, 1 + g * g, 2 * g)
uniform float2             noiseShift;

float CalculateAtmosShadowing(float2 pixelCoord, float2 cameraRay, float3 shadowRay)
{
    float4     sampleCoord;

    const float m = 0.25;                    // Slope at first sample.
    const float dt = 1.0 / 64.0;             // Delta t = 1 / n for n samples.
    const float dw = 2.0 * (1.0 - m) * dt;   // Change in weight per step.

    // Fetch depth of solid surface from structure buffer at pixel location.
    float2 strc = texture(structureBuffer, pixelCoord * 2.0).zw;
    float depth = strc.x + strc.y;

    // Calculate scale making camera ray have length of dmin.
    float dmin = minAtmosphereDepth;
    float invlength = rsqrt(dot(cameraRay, cameraRay) + dmin * dmin);
    float scale = dmin * invlength;

    // Calculate begin and end points in shadow space.
    float3 p1 = shadowRay * scale;
    float3 p2 = p1 * atmosphereDepthRatio + shadowCameraPosition;
    p1 += shadowCameraPosition;

    // Calculate begin and end depths in camera space.
    float z1 = dmin * scale;
    float z2 = z1 * atmosphereDepthRatio;

    // Calculate Henyey-Greenstein function and multiply by density.
    float h = anisotropyConst.x * rsqrt(anisotropyConst.y - anisotropyConst.z *
            dot(float3(cameraRay, dmin), cameraLightDirection) * invlength);
```

```
    float intensity = h * h * h * atmosphereBrightness;

    // Fetch random offset from 32x32 noise texture.
    float t = texture(noiseTexture, pixelCoord * 0.03125 + noiseShift).x * dt;
    float tmax = t + 1.0;
    float atm = 0.0;        // Atmsophere accumulator.
    float weight = m;       // First weight is always m.

    // Start with cascade 0.
    sampleCoord.z = 0.0;
    float zmax = min(depth, maxCascadeDepth[0]);

    for (; t <= tmax; t += dt)
    {
        float u = (t * (1.0 - m) + m) * t;
        float z = lerp(z1, z2, u);
        if (z > zmax) break;

        sampleCoord.xyw = lerp(p1, p2, u);                      // Calculate position.
        atm += texture(shadowTexture, sampleCoord) * weight;    // Accumulate sample.
        weight += dw;                                           // Increase weight.
    }

    for (int cascade = 1; cascade < 4; cascade++)
    {
        // Handle remaining cascades.
        sampleCoord.z = float(cascade);
        zmax = min(depth, maxCascadeDepth[cascade]);

        for (; t <= tmax; t += dt)
        {
            float u = (t * (1.0 - m) + m) * t;
            float z = lerp(z1, z2, u);
            if (z > zmax) break;

            // Calculate position and transform into current cascade space.
            sampleCoord.xyw = lerp(p1, p2, u) *
                    cascadeScale[cascade - 1].xyz + cascadeOffset[cascade - 1].xyz;

            atm += texture(shadowTexture, sampleCoord) * weight;
            weight += dw;
        }
    }

    return (atm * intensity);
}
```

For each pixel in the postprocessing stage, we can simply read the intensity stored in the atmosphere buffer, multiply it by the light's color, and add it to the color of the main scene. We cannot perform a depth-aware blur to completely eliminate the random noise inherent in our technique because the atmospheric shadowing calculated for each ray is derived from information at many different depths in open space. Fortunately, the noise is not very prominent at typical intensities. Still, there is one small action we can take to make the noise a little less noticeable, and that is to use a directional median filter. This filter lets us remove some noise from the image while preserving hard edges in the places where sun rays go behind foreground geometry.

We can calculate the projected position of the light source in the viewport by applying the projection matrix $\mathbf{P}$, perspective divide, and viewport transformation to the camera-space light direction $\mathbf{l}$ that we are already using in the anisotropy calculation. The resulting coordinates of the light source are given by

$$x_{\text{light}} = \left(\frac{P_{00}l_x + P_{01}l_y + P_{02}l_z}{2l_z} + \frac{1}{2} \right) w$$

$$y_{\text{light}} = \left(\frac{P_{10}l_x + P_{11}l_y + P_{12}l_z}{2l_z} + \frac{1}{2} \right) h, \qquad (10.81)$$

where w and h are the width and height of the atmosphere buffer (half the size of the main buffer), and we have assumed that the fourth row of the projection matrix is $[0 \ \ 0 \ \ 1 \ \ 0]$. Subtracting the pixel's location $(x_{\text{pixel}}, y_{\text{pixel}})$ from these coordinates gives us a direction $\mathbf{d}$ in screen space along which we can take a few samples from the atmosphere buffer and calculate their median. It is possible for the direction to light to be perpendicular to the viewing direction, in which case $l_z = 0$. To avoid division by zero, we multiply through by $2l_z$, and use the formulas

$$d_x = 2l_z \left(x_{\text{light}} - x_{\text{pixel}} \right) = \left(P_{00}l_x + P_{01}l_y + P_{02}l_z + l_z \right) w - 2l_z x_{\text{pixel}}$$

$$d_y = 2l_z \left(y_{\text{light}} - y_{\text{pixel}} \right) = \left(P_{10}l_x + P_{11}l_y + P_{12}l_z + l_z \right) h - 2l_z y_{\text{pixel}}. \qquad (10.82)$$

This scales the direction and may reverse its direction, but we will normalize its length and take samples symmetrically around the pixel location.

Some GPUs have a native instruction that can calculate the median value $\text{med}(a, b, c)$ of a set of three inputs a, b, and c. In the absence of that feature, the median can be computed with standard min and max functions using the formula

$$\text{med}(a, b, c) = \min \left(\max \left(\min(a, b), c \right), \max(a, b) \right). \qquad (10.83)$$

This formula is implemented in Listing 10.14 to find the median of three samples

read from the atmosphere buffer. One sample is taken at the current pixel location, and two more samples are taken along the direction to the screen-space light position, one on either side of the current pixel location.

Listing 10.14. This pixel shader function uses Equation (10.83) to calculate the median of three samples taken from the atmosphere buffer along the screen-space direction to the light source. The `pixelCoord` parameter specifies the viewport coordinates of the pixel being processed in the main frame buffer, and it is divided by two to give the corresponding location in the half-size atmosphere buffer. The uniform constant `lightPosition` holds the value $2l_z(x_{\text{light}}, y_{\text{light}}, 1)$, where the screen-space coordinates of the light source are given by Equation (10.81).

```
uniform TextureRect    atmosphereBuffer;
uniform float3         lightPosition;

float GetAtmosphereIntensity(float2 pixelCoord)
{
    float2 center = pixelCoord * 0.5;
    float2 direction = normalize(lightPosition.xy – center * lightPosition.z);
    float a = texture(atmosphereBuffer, center).x;
    float b = texture(atmosphereBuffer, center + direction).x;
    float c = texture(atmosphereBuffer, center – direction).x;
    return (min(max(min(a, b), c), max(a, b)));
}
```

10.7 Motion Blur

Every sensor that is capable of observing an image, whether it be a camera or the human eye, requires a small interval of exposure time. Since no image can be created instantaneously out of the incoming light, any objects in motion travel a short distance through the image over the exposure interval. The causes a phenomenon called *motion blur* that is present in photographs, videos, and even the retina. The brain expects fast-moving objects to be blurred on the retina, and it is able to compensate for the effect. A rendered scene can be made to look more realistic by adding motion blur because it better approximates what we see in the real world.

The most straightforward method for simulating motion blur is to render many subframes evenly distributed in time over a fixed exposure interval and taking the average of the results. This method can be made arbitrarily accurate by rendering greater numbers of subframes, but it is also extremely expensive and thus impractical for real-time applications. A group of considerably faster but much less accurate methods create the illusion of motion blur in the postprocessing stage for a

single rendered frame. The method we discuss here makes use of a *velocity buffer* to control the radius and direction of a one-dimensional per-pixel blurring operation. This produces an adequate motion blur effect as long as measures are taken to avoid interference between foreground and background objects where appropriate.

10.7.1 The Velocity Buffer

Our motion blur technique requires that we calculate a velocity vector in viewport space at every pixel. Each particular velocity represents the motion of a point on the surface of whatever object happens to be rendered at the corresponding pixel location. The following three sources of motion can each make a contribution to the velocity, and we must consider the cumulative effect of all three when they are present.

■ The motion of the camera generally affects every object in the scene. The main exception is an object that is attached to the viewer in some way, such as the player's weapon in a first-person shooter. The camera's contribution to motion is determined by considering the world-space to camera-space transformation matrices for both the current frame and preceding frame. If the projection matrix is not constant, perhaps due to a zoom effect, then its difference must also be taken into account.

■ Each visible object could be moving as a whole, and its motion could include both a linear translation, a rotation about some axis, and maybe even a change in scale. The world-space velocity of a point on the surface of the object is determined by considering the differing effects of the object-space to world-space transformation matrices for the current frame and preceding frame.

■ The vertices of a triangle mesh might have independent motion because their object-space positions are recalculated every frame. This is very common for skinned character meshes and physically simulated objects like cloth or rope. In this case, vertex positions for the current frame and preceding frame must both be taken into account in addition to the change in the object-to-world transformation. For physically simulated objects, the vertex position for the preceding frame could be calculated using the position for the current frame and the vertex's velocity, which would be known to the simulation.

The velocity buffer is a render target having two channels in which the x and y components of the per-pixel velocity vectors are stored. The velocities are rescaled and clamped so that a magnitude of one corresponds to the largest allowed blur radius, and this means that the component values in the velocity buffer always fall in the range $[-1, +1]$. Sufficient precision for the motion blur effect is provided by

using 8-bit channels and remapping values to the range $[0,1]$. A single pixel in the velocity buffer thus requires only two bytes of storage.

A scene with motion blur applied and its associated velocity buffer are shown in Figure 10.23. In this example, the camera is moving backward away from some stone objects that are falling to the ground. A vector field has been overlaid on the velocity buffer to show the magnitude and direction of the velocity at regularly spaced locations in the viewport. Values of 0.5 in the red and green channels correspond to a zero velocity in the x and y directions, respectively. Values smaller than 0.5 correspond to negative velocities, and values larger than 0.5 correspond to positive velocities. Since there is no blue channel in the velocity buffer, it has been assigned a constant value of 0.5 in the image, and this causes pixels having no motion associated with them to appear as a 50% gray color. Pixels having rightward motion have large red components, but red is mostly absent from pixels having leftward motion. Similarly, pixels having downward motion have large green components, but green is mostly absent from pixels having upward motion.

There are two regions in the image where the velocity is zero and thus appear as the gray color. First, the sky at the top of the image is rendered infinitely far away, and it has no viewport-space motion because the camera is moving directly backward. If the camera had been turning in any way, then the sky would be moving through the viewport, and this case is discussed below. Second, the weapon in the lower-right portion of the image is indirectly attached to the camera position and therefore has no relative motion except for a part inside that can be seen rotating as a subnode in the weapon's transform hierarchy. The surrounding geometry in the nonstationary regions of the image tends to be moving toward the center of the viewport due to the camera's backward motion. The contrasting velocities of the falling objects are clearly visible and they include some rotation in addition to their generally downward motion, especially in the piece of stone near the bottom center that has already hit the ground and started tumbling. The demonstrates how the viewport-space velocity can vary significantly over the surface of a single rigid object.

The contents of the velocity buffer are generated by rendering all of the visible geometry in a separate pass using special vertex and pixel shaders. These can be incorporated into the same shaders that write data to the structure buffer by outputting results to multiple render targets. As discussed below, high-quality motion blur will require the depth and gradient information in the structure buffer, so we will need the structure buffer even if it's not being used for anything else. Combining the calculations for both buffers into a single pass simplifies the code and leads to significantly better performance.
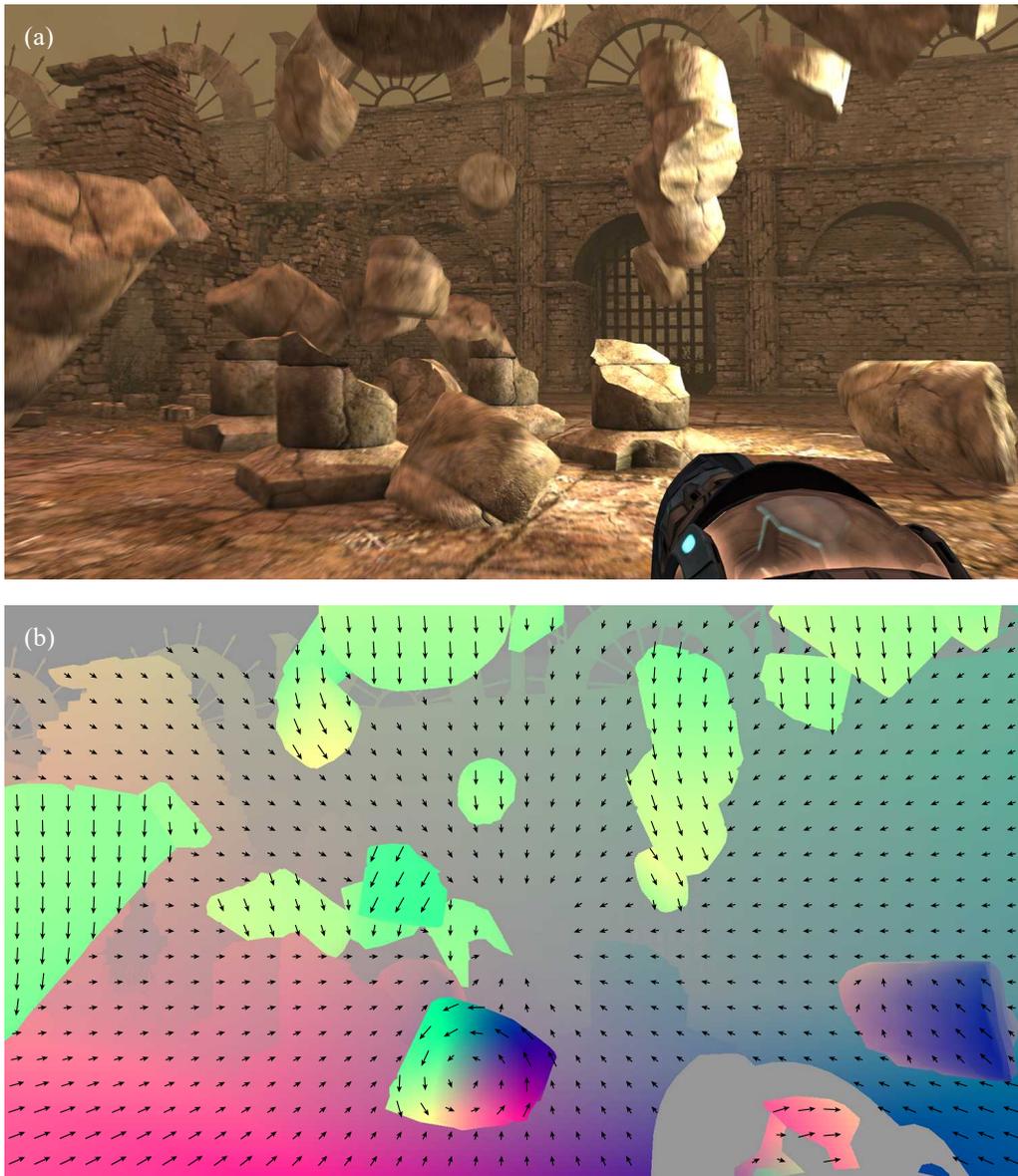
Figure 10.23. (a) Motion blur is rendered as a postprocessing effect for some crumbling ruins. The stone pieces are blurred primarily due to their own motion, and the ground is blurred due to the backward motion of the camera. (b) The velocity buffer contains the 2D viewport-space velocity at every pixel, and the corresponding vector field has been overlaid. The color channels show velocity components after remapping from the range $[-1, +1]$ to the range $[0, 1]$.

To determine the values that are written to the velocity buffer, we calculate the position of each vertex in viewport space for the current frame and the preceding frame. Dividing the difference between the two positions by the time between the two frames produces the velocity of the vertex. An object-space vertex position $\mathbf{p}_{object}$ is transformed into a 4D homogeneous viewport-space position $\mathbf{p}_{viewport}$ by the equation

$$\mathbf{p}_{viewport} = \mathbf{M}_{viewport}\mathbf{M}_{projection}\mathbf{M}_{camera}^{-1}\mathbf{M}_{object}\,\mathbf{p}_{object}. \qquad (10.84)$$

Here, $\mathbf{M}_{object}$ is the matrix that transforms from object space to world space for the object being rendered, $\mathbf{M}_{camera}$ is the matrix that transforms from camera space to world space, $\mathbf{M}_{projection}$ is the projection matrix, and $\mathbf{M}_{viewport}$ is the viewport transformation given by

$$\mathbf{M}_{viewport} = \begin{bmatrix} w/2 & 0 & 0 & w/2 \\ 0 & h/2 & 0 & h/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad (10.85)$$

which is the matrix equivalent of Equation (5.34) for a viewport having dimensions $w \times h$ pixels and a depth range of $[0,1]$. Since the translation in the fourth column of $\mathbf{M}_{viewport}$ cancels out when viewport-space positions are subtracted, we can ignore it and consider the viewport transformation to be nothing more than a scale up to the size of the render target. We calculate $\mathbf{p}_{viewport}$ for two consecutive frames in the vertex shader and interpolate the results across each triangle. We do not need the z coordinates, so only the x, y, and w coordinates are calculated and then output as two 3D vectors. The perspective divide by the w coordinate of $\mathbf{p}_{viewport}$ must be deferred to the pixel shader because it introduces a nonlinearity that would not be interpolated properly.

Let $\mathbf{M}_{motion}$ be the product of the four matrices in Equation (10.84). When an object is rendered into the velocity buffer, $\mathbf{M}_{motion}$ is precalculated for the current frame and the preceding frame, and the results are sent to the vertex shader as uniform parameters. The viewport transformation does not change, so the same matrix $\mathbf{M}_{viewport}$ is used in both products. The projection matrix normally does not change either, so the same matrix $\mathbf{M}_{projection}$ is also used in both products. The matrices $\mathbf{M}_{camera}$ and $\mathbf{M}_{object}$, however, can change frequently and have different values for the current and preceding frames. The matrices for the preceding frame need to be saved and stored separately whenever node transforms are updated. If no change is made to a node's transform during a particular frame, then its current transformation matrix needs to be copied into the matrix for the preceding frame so that

its velocity does not persist. This copy also needs to be made in cases where a completely new transform is assigned to a node to prevent a very high erroneous velocity from being calculated. For example, if a player is teleported to a new location, then the camera's new transform needs to be copied into its old transform or else the maximum blur will likely be applied to the entire viewport during the next frame in the direction between the previous and current camera positions.

The vertex shader code in Listing 10.15 transforms the object-space vertex position $\boldsymbol{p}_{\text{object}}$ into viewport space with the matrices $\mathbf{M}_{\text{motion}}$ corresponding to the current frame and preceding frame. Since the z coordinates of the results will not be needed, the third rows of the two matrices are also not needed. The first, second, and fourth rows of $\mathbf{M}_{\text{motion}}$ for the current frame and preceding frame are passed in through the uniform arrays newMotionMatrix and oldMotionMatrix, respectively. The transformed x, y, and w coordinates of $\mathbf{p}_{\text{viewport}}$ for the current frame and preceding frame are returned in the 3D vectors pnew and pold. These values should be output by the vertex shader to be interpolated over a triangle.

In the pixel shader, we receive the two interpolated viewport-space positions $\mathbf{p}_{\text{new}} = (x_{\text{new}}, y_{\text{new}}, w_{\text{new}})$ and $\mathbf{p}_{\text{old}} = (x_{\text{old}}, y_{\text{old}}, w_{\text{old}})$ as homogeneous vectors without their z coordinates. We perform the perspective divide by the w coordinate for each position and subtract the results to obtain the displacement vector

$$\mathbf{d} = \frac{(x_{\text{new}}, y_{\text{new}})}{w_{\text{new}}} - \frac{(x_{\text{old}}, y_{\text{old}})}{w_{\text{old}}}. \tag{10.86}$$

Dividing the displacement by the actual time Δt between the current and preceding frames gives us the velocity vector $\mathbf{v} = \mathbf{d}/\Delta t$ in viewport space. This velocity is unbounded, but we need to write values in the range $[-1, +1]$ to the velocity buffer, so we will scale it by several factors. The final values that we output are technically distances that correspond to blur radii, but we can consider them to be scaled velocities that are multiplied by one unit of time.

To produce a uniform appearance and smooth out irregularities in the time between consecutive frames, we choose a constant time t_0 by which the velocity $\mathbf{v}$ is always multiplied to obtain the blur vector $\mathbf{r}$. This normalization time would usually be something like 1/60th of a second. We also need to apply a scale so that a magnitude of one in the velocity buffer corresponds to the maximum blur radius r_{max}. The blur vector is then given by

$$\mathbf{r} = \frac{t_0}{r_{\text{max}}} \mathbf{v}. \tag{10.87}$$

Listing 10.15. This vertex shader function transforms the object-space vertex position `pobject` into viewport space with the matrices stored in the uniform constants `newMotionMatrix` and `oldMotionMatrix` corresponding to the current frame and preceding frame, respectively. Each matrix is represented by three 4D vectors holding the first, second, and fourth row of the matrix $\mathbf{M}_{\text{motion}}$. The x, y, and w coordinates of each transformed vertex position are returned in the 3D vectors `pnew` and `pold` to be interpolated. The z coordinates will not be needed by the pixel shader.

```
uniform float4 newMotionMatrix[3];
uniform float4 oldMotionMatrix[3];

void TransformMotionBlurPositions(float3 pobject, out float3 pnew, out float3 pold)
{
    pnew.x = dot(newMotionMatrix[0].xyz, pobject) + newMotionMatrix[0].w;
    pnew.y = dot(newMotionMatrix[1].xyz, pobject) + newMotionMatrix[1].w;
    pnew.z = dot(newMotionMatrix[2].xyz, pobject) + newMotionMatrix[2].w;

    pold.x = dot(oldMotionMatrix[0].xyz, pobject) + oldMotionMatrix[0].w;
    pold.y = dot(oldMotionMatrix[1].xyz, pobject) + oldMotionMatrix[1].w;
    pold.z = dot(oldMotionMatrix[2].xyz, pobject) + oldMotionMatrix[2].w;
}
```

This can further be scaled by an adjustable parameter k in the range $[0,1]$ that controls the overall intensity of the motion blur effect. The fully scaled displacement is then $m\mathbf{d}$, where the factor m is given by

$$m = \frac{kt_0}{r_{\text{max}}\Delta t}. \tag{10.88}$$

The value of Δt can vary from one frame to the next if a fixed frame rate is not maintained, but the values of k, t_0, and r_{max} are global constants. Only one step remains, and that is to clamp the scaled displacement to a maximum magnitude of one. The final value $\mathbf{d}'$ written to the velocity buffer is

$$\mathbf{d}' = \frac{m\mathbf{d}}{\max\left(\|m\mathbf{d}\|,1\right)}. \tag{10.89}$$

The pixel shader code shown in Listing 10.16 calculates the value of $\mathbf{d}'$ using the interpolated values of $\mathbf{p}_{\text{new}}$ and $\mathbf{p}_{\text{old}}$ generated by the vertex shader code in Listing 10.15. The value of m is calculated once per frame and sent to the pixel shader as the uniform constant `velocityScale`. The components of $\mathbf{d}'$ are finally remapped from the range $[-1,+1]$ to the range $[0,1]$ so they can be written to the unsigned 8-bit channels of the velocity buffer.

Listing 10.16. This pixel shader function calculates the scaled displacement $\mathbf{d}'$ using the interpolated vertex positions pnew and pold generated by the vertex shader function in Listing 10.15. The uniform constant velocityScale holds the value of m in Equation (10.89).

```
uniform float velocityScale;

float2 CalculateMotionBlurVelocity(float3 pnew, float3 pold)
{
    float2 velocity = (pnew.xy / pnew.z - pold.xy / pold.z) * velocityScale;
    return (velocity / max(length(velocity), 1.0) * 0.5 + 0.5);
}
```

There may be regions in the viewport where no geometry is drawn and the sky shows through. In the case that a skybox is rendered at infinity, we could fill the parts of the velocity buffer with the correct values where the sky is visible by rendering the skybox's faces. However, we would need to be careful to modify Listing 10.15 so that the translation components of the matrices are not added because the skybox's vertices have implicit w coordinates of zero. An alternative that is not exactly correct everywhere but still provides a good approximation is to calculate a single representative background velocity and use it as the value to which the velocity buffer is initially cleared.

Consider the point at infinity in the camera's view direction for the preceding frame. In camera space, this point is simply the homogeneous vector $(0, 0, 1, 0)$, and in clip space, it transforms into the third column of the projection matrix, which we denote by $\mathbf{M}_{\text{projection}[2]}$. After dividing by the w coordinate and scaling to the size of the render target, we have a location for the point in viewport space. If we calculate the location in viewport space for the same point at infinity, corresponding to the preceding frame's view direction, but now using the current frame's camera transform, then we can see how far the point has moved, and this gives us a background velocity.

Let $\mathbf{L}_{\text{camera}}$ be the camera-space to world-space transform for the preceding frame. The point at infinity in the camera's view direction is given by the third column of this matrix, which we denote by $\mathbf{L}_{\text{camera}[2]}$. We transform this point into homogeneous vector $\mathbf{q}$ in clip space using the formula

$$\boxed{\mathbf{q} = \mathbf{M}_{\text{projection}}\mathbf{M}_{\text{camera}}^{-1}\mathbf{L}_{\text{camera}[2]},} \tag{10.90}$$

where $\mathbf{M}_{\text{camera}}$ is the camera-space to world-space transform for the current frame. By making the assignment $\mathbf{p} = \mathbf{M}_{\text{projection}[2]}$, the displacement vector $\mathbf{d}$ in viewport space can then be expressed as

$$\mathbf{d} = \mathbf{M}_{\text{viewport}} \left[\frac{(q_x, q_y)}{q_w} - \frac{(p_x, p_y)}{p_w} \right]. \tag{10.91}$$

For any ordinary perspective projection, $p_w = 1$, and for any projection that is not off center, $p_x = p_y = 0$. Thus, it is usually the case that $\mathbf{d}$ is simply given by

$$\mathbf{d} = \left(\frac{wq_x}{2q_w}, \frac{hq_y}{2q_w} \right) \tag{10.92}$$

for a $(w \times h)$-pixel viewport. This background displacement must finally be subjected to the same scale and clamping operations performed by Equation (10.89). After remapping the resulting value $\mathbf{d}'$ from the range $[-1, +1]$ to the range $[0, 1]$, it can be specified as the clear color for the velocity buffer's red and green channels.

10.7.2 Image Postprocessing

In the postprocessing stage at the end of the rendering process for each frame, the velocity buffer is used to perform a directional blurring operation. At every pixel, we read many samples from the color buffer along the direction given by the velocity buffer and average them together. We assume that the original color image represents the state of the scene in the middle of the time interval between consecutive frames, and we therefore read samples symmetrically around each pixel. Otherwise, leading and trailing edges on a moving object would not have the same appearance because they would sample the background differently.

The pixel shader code shown in Listing 10.17 reads eight samples from the color buffer in addition to the center pixel being processed. The velocity buffer is read one time at the center pixel, and it is expanded from the range $[0, 1]$ to the range $[-1, +1]$. Four samples are read from the color buffer in the positive velocity direction, and four more samples are read at the same relative locations in the negative velocity direction. The step size between samples is specified as a uniform constant equal to the maximum blur radius r_{max} divided by the number of samples on one side of the center pixel. The images shown in this section use $r_{\text{max}} = 7.0$ with four samples read forward and backward at a step size of 1.75. The step size is multiplied by the velocity vector, so it represents a distance between samples in pixel units when the velocity has a magnitude of one, but represents shorter distances otherwise.

The blur operation shown in Listing 10.17 is simple, and it produces satisfactory results in many situations. However, there are cases in which combining samples with no regard for what surfaces they come from creates undesirable artifacts.

Listing 10.17. This pixel shader function applies simple motion blur in the postprocessing stage. The `pixelCoord` parameter specifies the viewport coordinates of the pixel being processed, and it is used to read from the color and velocity buffers. The uniform constant `vstep` holds the maximum blur radius r_{max} divided by the number of loop iterations, four in this example, which is the number of samples read from the color buffer on each side of the center pixel along the direction of the velocity vector. The samples are accumulated in the `color` variable, and their average is returned by dividing by nine in this case.

```
uniform TextureRect    colorBuffer;
uniform TextureRect    velocityBuffer;
uniform float          vstep;

float3 ApplySimpleMotionBlur(float2 pixelCoord)
{
    // Read color buffer and velocity buffer at center pixel.
    float3 color = texture(colorBuffer, pixelCoord).xyz;
    float2 velocity = texture(velocityBuffer, pixelCoord).xy * 2.0 - 1.0;

    // Add 8 more samples along velocity direction.
    for (int i = 1; i <= 4; i++)
    {
        float dp = float(i) * vstep;
        color += texture(colorTexture, pixelCoord + velocity * dp).xyz;
        color += texture(colorTexture, pixelCoord - velocity * dp).xyz;
    }

    // Return average of all samples.
    return (color * 0.1111111);
}
```

The main example is the case in which a stationary object is rendered in front of a moving background, as shown in Figure 10.24(a). Here, the player's weapon is connected to the camera, so its velocity is zero in viewport space, but the background has a nonzero velocity due to the camera's motion as the player runs through the environment. A fuzzy halo is visible around the edges of the weapon due to samples being read from the foreground object when the motion blur effect is applied to nearby pixels belonging to the background. This artifact can be eliminated by detecting the difference between the two surfaces in the pixel shader and rejecting samples that do not belong to the same surface as the pixel being processed. The result is the image shown in Figure 10.24(b), where the halo is no longer present, and the stationary foreground object has a nice clean boundary.

We detect surface discontinuities by using the depth and gradient information available in the structure buffer. If we consider the surface at a particular pixel to be a flat plane passing through the depth z_0 and having a 2D slope given by the

gradient ∇z, then we can determine the change in depth within the maximum blur radius $r_{\max}$ along the direction of the velocity vector $\mathbf{v}_0$ by calculating a directional derivative. This leads us to the formula

$$z_{\min} = z_0 - r_{\max} \left(|\nabla z \cdot \mathbf{v}_0| + \delta \right) \qquad (10.93)$$

specifying the minimum depth $z_{\min}$ that any sample must have to be considered part of the same surface as the pixel being processed. The extra term δ is an adjustable value that we include to account for the fact that the surface may not actually
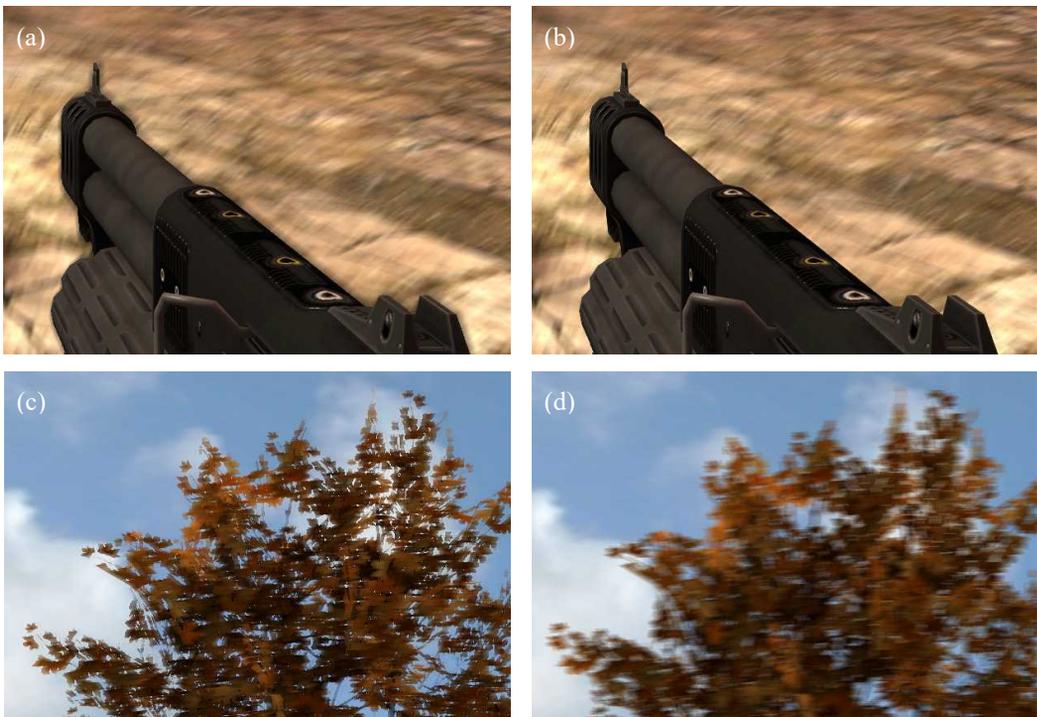
Figure 10.24. (a) When no distinction is made between a moving background and a stationary foreground, motion blur applied to background pixels near the boundary incorporates samples read from the foreground object, and this creates a fuzzy halo artifact. (b) Information in the structure buffer is used to detect surface discontinuities and reject samples that do not belong to the same surface as the pixel being processed, eliminating the artifact. (c) Rejecting samples based on large differences in depth can surfaces moving through the viewport at the same rate due to camera rotation not to be blurred properly. (d) Correct blurring is restored by overriding depth-based rejection when sample velocities are similar.

be flat within the blur radius and could acquire a larger slope away from the center pixel. When we read samples from the color buffer in the motion blur shader, we also read from the structure buffer to determine the depths at the same locations. If a sample does not have a depth greater than z_{min}, then we consider it to be part of a distinct foreground object, and we reject that sample.

Comparing depths effectively eliminates halo artifacts around stationary foreground objects, as shown in Figure 10.24(b), but it introduces a new type of artifact at the same time. In Figure 10.24(c), the camera is rotating about the vertical axis, and both the sky and a tree are moving through the viewport at the same velocity. However, because the sky's depth is much greater than the tree's depth (possibly infinitely greater), the motion blur shader does not accept samples from the tree when it processes pixels belonging to the sky. This causes the edges of the leaves to appear much less blurred than they should be in the direction of motion. To restore the correct blurring that would have been produced by the simple shader code in Listing 10.17, we do not reject samples from foreground objects when the velocity $\mathbf{v}$ at the sample location is similar to the velocity $\mathbf{v}_0$ at the center pixel. We implement this condition by making the comparison

$$(\mathbf{v} - \mathbf{v}_0)^2 < \sigma, \tag{10.94}$$

where σ is another adjustable value representing the maximum difference in velocities allowed in order for a foreground sample to be accepted. The results produced by the application of Equation (10.94) are demonstrated in Figure 10.24(d).

The pixel shader code in Listing 10.18 performs motion blur using the same sample locations as the simpler code in Listing 10.17, but it rejects samples having depth z and velocity $\mathbf{v}$ for which neither Equation (10.93) nor Equation (10.94) is satisfied, where the adjustable values have been set to $\delta = 1/128$ and $\sigma = 1/16$. In this more complex shader, values are read from the color buffer, structure buffer, and velocity buffer at all sample locations. As color samples are accumulated in the first three components of the `color` variable, the w component keeps a count of the number of samples that have been accepted, and we divide by this value at the end to calculate the correct average.

The added complexity of the pixel shader in Listing 10.18 makes it significantly more expensive than the unconditional blurring operation performed by the pixel shader in Listing 10.17. To save time in the postprocessing stage, we can limit the use of the more complex motion blur code to the areas of the viewport where the artifacts it prevents are likely to occur. As shown in Figure 10.25, one way this can be done is to partition the viewport into a grid for which each cell is processed entirely by either the simple shader or the complex shader. A separate

triangle mesh can be constructed to include all of the cells of each type so that postprocessing the full viewport requires only two draw calls.

To determine which cells should be processed by the more complex motion blur shader, we can consider the areas inside the viewport covered by the visible objects that are likely to have velocities much different than the background. In Figure 10.25, two such objects are visible, a weapon connected to the camera and a moving object in the environment, and the cells they intersect are highlighted in red. The group of cells covered by any one object can be quickly derived from a rectangle bounding the object on the projection plane. The method described for a point light in Section 8.2.1 can calculate this rectangle for a bounding sphere, but

Listing 10.18. This pixel shader function applies motion blur in the postprocessing stage using the same basic technique as in Listing 10.17, but it requires that each sample satisfy either Equation (10.93) or Equation (10.94), where $\delta = 1/128$ and $\sigma = 1/16$. The uniform constant $rmax$ holds the maximum blur radius r_{max}, and $vstep$ holds this radius divided by the number of loop iterations. The accepted samples are accumulated in the first three components of the $color$ variable, and a count of the accepted samples is kept in the w component.

```
uniform TextureRect    colorBuffer;
uniform TextureRect    structureBuffer;
uniform TextureRect    velocityBuffer;
uniform float          rmax, vstep;

float3 ApplyComplexMotionBlur(float2 pixelCoord)
{
    const float kDepthDelta = 0.0078125;
    const float kVeloSigma = 0.0625;

    // Read color buffer, structure buffer, and velocity buffer at center pixel.
    float4 color = float4(texture(colorTexture, pixelCoord).xyz, 1.0);
    float4 structure = texture(structureTexture, pixelCoord);
    float2 velocity = texture(velocityTexture, pixelCoord).xy * 2.0 - 1.0;

    // Use gradient to calculate minimum depth for other samples.
    float zmin = structure.z + structure.w
                - (abs(dot(velocity, structure.xy)) + kDepthDelta) * rmax;

    float4 sample = float4(0.0, 0.0, 0.0, 1.0);
    for (int i = 1; i <= 4; i++)
    {
        float dp = float(i) * vstep;

        // Read all three buffers at sample location, and subtract center velocity.
        float2 sampleCoord = pixelCoord + velocity * dp;
        sample.xyz = texture(colorTexture, sampleCoord).xyz;
```

```
        float2 depth = texture(structureTexture, sampleCoord).zw;
        float2 dv = texture(velocityTexture, sampleCoord).xy * 2.0 – 1.0 – velocity;

        // Add sample only if depth greater than minimum or velocities similar enough.
        if ((depth.x + depth.y > zmin) || (dot(dv, dv) < kVeloSigma)) color += sample;

        // Repeat in opposite direction.
        sampleCoord = pixelCoord – velocity * dp;
        sample.xyz = texture(colorTexture, sampleCoord).xyz;
        depth = texture(structureTexture, sampleCoord).zw;

        dv = texture(velocityTexture, sampleCoord).xy * 2.0 – 1.0 – velocity;
        if ((depth.x + depth.y > zmin) || (dot(dv, dv) < kVeloSigma)) color += sample;
    }

    // Divide color sum by number of passing samples.
    return (color.xyz / color.w);
}
```


Figure 10.25. The viewport is covered by a 16×9 grid, and motion blur is applied by the same shader for every pixel in each cell. The cells highlighted in red use the complex motion blur shader in Listing 10.18, and all other cells use the simple motion blur shader in Listing 10.17. The red cells are determined by considering objects that are likely to have velocities much different than the background and calculating a bounding rectangle for them in the viewport.

this tends to produce very conservative results for most objects. A much tighter rectangle can be calculated by transforming the eight vertices of a bounding box into clip space and finding the minimum and maximum x and y coordinates. However, care must be taken to ignore vertices in clip space that have negative z coordinates and instead consider points where any of the bounding box's edges cross the near plane. Edges could also be clipped against the side planes of the canonical view volume to reduce the area calculated for objects close to the camera.

The code in Listing 10.19 calculates a rectangle in device space that encloses the projected vertices of a bounding box specified in object space. After transforming the vertices into clip space, it loops over the 12 edges of the bounding box and clips each one against the near plane if necessary. The perspective divide is applied, and the minimum and maximum values of the x and y coordinates of the resulting endpoints are kept at the end of each loop. The rectangle that the code returns still needs to be transformed into viewport space, clamped to the dimensions of the viewport, and expanded to integral cell boundaries.

Listing 10.19. This function takes an object-space bounding box with minimum coordinates `bmin` and maximum coordinates `bmax`, transforms it into clip space using the model-view-projection matrix `mvp`, and calculates the box's device-space bounding rectangle. If the box is not completely clipped away by the near plane, then the minimum and maximum extents of the rectangle are stored in `vmin` and `vmax`, and the function returns `true`. Otherwise, the function returns `false`.

```cpp
bool CalculateDeviceSpaceExtents(const Point3D& bmin, const Point3D& bmax,
                                 const Matrix4D& mvp, Point2D *vmin, Point2D *vmax)
{
    static const int8 edgeVertexIndex[24] =
        {0, 1, 1, 2, 2, 3, 3, 0, 4, 5, 5, 6, 6, 7, 7, 4, 0, 4, 1, 5, 2, 6, 3, 7};

    // Transform bounding box vertices into clip space.
    Vector4D vertex[8];
    vertex[0] = mvp * Point3D(bmin.x, bmin.y, bmin.z);
    vertex[1] = mvp * Point3D(bmax.x, bmin.y, bmin.z);
    vertex[2] = mvp * Point3D(bmax.x, bmax.y, bmin.z);
    vertex[3] = mvp * Point3D(bmin.x, bmax.y, bmin.z);
    vertex[4] = mvp * Point3D(bmin.x, bmin.y, bmax.z);
    vertex[5] = mvp * Point3D(bmax.x, bmin.y, bmax.z);
    vertex[6] = mvp * Point3D(bmax.x, bmax.y, bmax.z);
    vertex[7] = mvp * Point3D(bmin.x, bmax.y, bmax.z);

    // Initialize device-space rectangle.
    float xmin = FLT_MAX, xmax = -FLT_MAX;
    float ymin = FLT_MAX, ymax = -FLT_MAX;

    const int8 *edge = edgeVertexIndex;
```

```
    for (int32 i = 0; i < 12; i++, edge += 2)
    {
        Vector4D p1 = vertex[edge[0]], p2 = vertex[edge[1]];

        // Clip edge against near plane.
        if (p1.z < 0.0F)
        {
            if (p2.z < 0.0F) continue;    // Edge completely clipped away.
            Vector4D dp = p1 - p2;
            p1 -= dp * (p1.z / dp.z);
        }
        else if (p2.z < 0.0F)
        {
            Vector4D dp = p2 - p1;
            p2 -= dp * (p2.z / dp.z);
        }

        // Perform perspective divide.
        float f1 = 1.0F / p1.w;
        float f2 = 1.0F / p2.w;
        p1.x *= f1; p1.y *= f1;
        p2.x *= f2; p2.y *= f2;

        // Update device-space rectangle.
        xmin = fmin(xmin, fmin(p1.x, p2.x)); xmax = fmax(xmax, fmax(p1.x, p2.x));
        ymin = fmin(ymin, fmin(p1.y, p2.y)); ymax = fmax(ymax, fmax(p1.y, p2.y));
    }

    if (xmin < xmax)    // At least one edge was visible.
    {
        vmin->Set(xmin, ymin); vmax->Set(xmax, ymax);
        return (true);
    }

    return (false);
}
```

10.8 Isosurface Extraction

A *scalar field* is the name given to a region of space inside which a meaningful scalar quantity can be defined at every point. There are many situations in which a physical structure of some kind is represented by a three-dimensional scalar field, often because it allows for the practical simulation of a dynamic system or the easier construction of solid geometry. A couple examples of scalar fields are the volumetric results of rendering a system of metaball particles, as shown in Figure 10.26, and the implicit definition of an arbitrarily-shaped terrain surface, as

shown in Figure 10.27. It is usually the case, as is true for both of these examples, that the scalar quantities composing a field represent the signed perpendicular distance to some surface of interest, and thus each of these fields is described by the more specific term *signed distance field*.
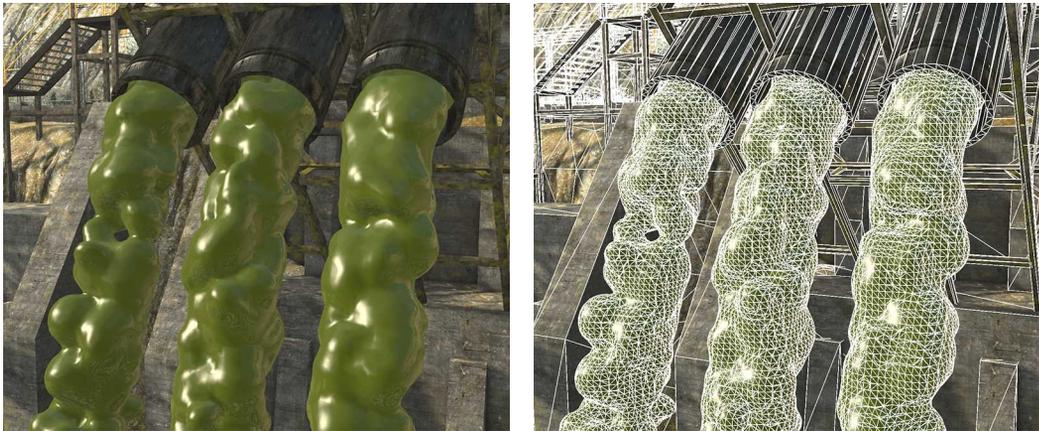
Figure 10.26. A particle system composed of metaballs is used to render the sludge flowing out of the three pipes. The grid visible in the wireframe reveals the underlying 3D scalar field from which the surface of the fluid is extracted.
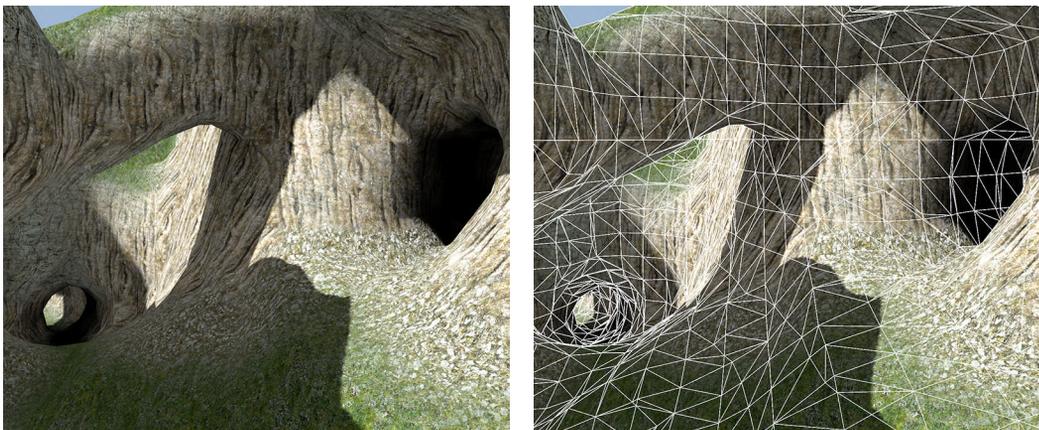
Figure 10.27. Terrain containing caves and overhangs is generated from an underlying 3D scalar field. Such structures would not be possible with a simple height field.

The data composing a signed distance field consists of an $n \times m \times h$ array of *voxels* (a shortened form of *volume elements*) placed on a regular three-dimensional grid measuring n units in the x direction, m units in the y direction, and h units in the z direction. To minimize storage space, it is common for the scalar distances to be represented by 8-bit signed integers, thus requiring a total of nmh bytes of memory in uncompressed form, but higher precision integers or floating-point numbers could be used. Each $2 \times 2 \times 2$ subset of the array delineates a cubic volume of space called a *cell*, and there are thus $(n-1)(m-1)(h-1)$ individual cells in the entire field.

Typically, positive voxel values correspond to distances on the outside of the surface in empty space, and negative values correspond to distances on the inside of the surface in solid space. The opposite convention could also be used, but this choice has the benefit that gradients in the scalar field, which can be used to calculate normal vectors, point outward from the surface.

In order to render the surface defined by a signed distance field, we must be able to transform the scalar data into vertices and triangles that can be consumed by the graphics hardware through a process called *isosurface extraction*. The *isosurface* is the two-dimensional surface where a continuous trilinear interpolation of the scalar field attains a given constant value called the *isovalue*, which is simply zero when the scalar values represent signed distances. Pieces of the isosurface are constructed by searching the scalar field for cells that contain the isovalue and generating triangles where the isosurface intersects those cells. The details of this process are described throughout this section.

10.8.1 Marching Cubes

A well-known method for constructing a triangle mesh whose vertices coincide with the zero-valued isosurface is the *marching cubes* algorithm. This algorithm extracts the isosurface by examining every cell in a scalar field in isolation. For each cell, it determines what part of the isosurface passes through the cell, if any, and generates a triangulation inside the cell's boundary based on the voxel values at the eight corners of the cell. The results from all cells are combined to form the complete isosurface.

When processing a single cell, the marching cubes algorithm first classifies each of the eight voxel values at the corners of the cell as being either in empty space or in solid space, corresponding to the sign of the value being positive or negative, respectively. A choice must be made globally as to whether a value of exactly zero is considered to be in empty space or in solid space, and a consistent classification either way is acceptable. It is convenient to group zero with positive

values in empty space so that integer sign bits can be used to make classifications. Vertices belonging to the cell's internal triangulation lie on the edges of the cell for which one endpoint has been classified as lying *outside*, in empty space, and the other endpoint has been classified as lying *inside*, in solid space. However, we do not explicitly check each of the 12 edges for this condition. Instead, the marching cubes algorithm exploits symmetry to reduce the problem space to a relatively small number of unique configurations that are then processed in a uniform manner with the aid of lookup tables.

When the eight corners of a cell are classified into a binary inside or outside state, it gives rise to precisely $2^8 = 256$ possible distinct cases to consider. Many of these cases are equivalent to another case except that the cell has simply been rotated into a different orientation. For example, there are eight cases in which exactly one corner of a cell is classified as inside, and the other seven are classified as outside. These cases can all be rotated about some axis through the cell's center so that the inside corner lies at the position having the minimum x, y, and z coordinates. All eight cases end up producing a single triangle and can be handled in the same way, so they are grouped into a single *equivalence class*. The triangle is generated by placing vertices on the three edges containing the inside corner and connecting them in the order that causes the normal vector to point into empty space. The actual positions of the vertices on the edges is determined by finding the position at which a linear interpolation of the voxel values is zero.

We can consider another group of eight cases having the *inverse* configuration such that exactly one corner of a cell is classified as outside and the other seven are classified as inside. Like before, these cases are related by the ability to rotate cells for any one of them so that they coincide with cells for any other case in the group, so they form another equivalence class. In this instance, a single triangle is again produced for all cases in the equivalence class, but it has the opposite winding order compared to each case's inverse so that the normal vector still points outward from the surface. Although not possible for all equivalence classes and their inverses, as discussed below, we can combine the eight cases having one corner inside and the eight cases having seven corners inside to form a single equivalence class containing 16 cases.

When the rotational symmetries of all 256 possible cell configurations are considered along with the potential for combining some cases with their inverses, the result is the set of 18 equivalence classes shown in Figure 10.28. These equivalence classes constitute one of several variants of the marching cubes algorithm that each possess different degrees of complexity and robustness. We choose this particular version because it is the simplest one that produces watertight triangle meshes. Note that for the example cases shown in the figure, vertices are always placed at
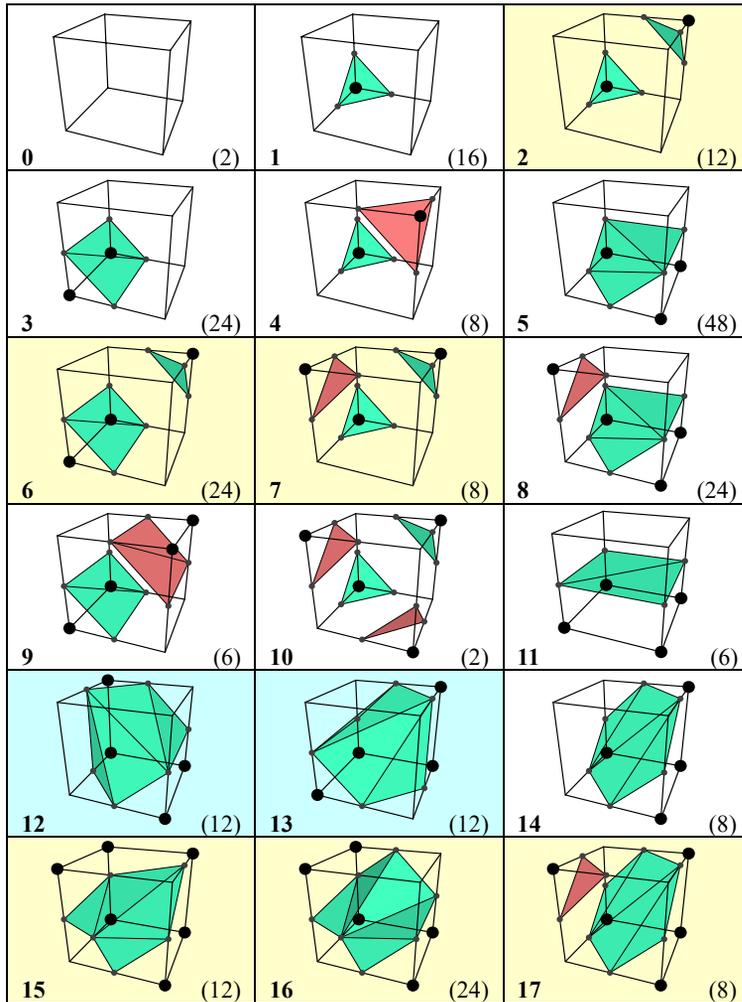
Figure 10.28. These are the 18 equivalence classes arising in the marching cubes algorithm with preferred polarity. The bold number in the lower-left corner of each cell is the class index, and the number in the lower-right corner is the number of cases belonging to the equivalence class out of the 256 total cases. A black dot indicates a corner that is inside the solid volume, and corners without a dot are outside. Green triangles are front-facing, and red triangles are back-facing. Classes 12 and 13 are reflections of each other and are the only classes without mirror symmetry. Classes 15, 16, and 17 are the inverses of classes 2, 6, and 7, respectively, which each have ambiguous faces and cannot be rotated onto their inverses.

the exact centers of the edges on which they lie, but they could be placed anywhere on the same edges depending on what the actual voxel values are at each edge's endpoints.

In general, there are 24 ways to rotate a cube onto itself, and the size of each equivalence class, without inverses, must divide 24. If inverses are included, then there are 48 ways in which a cell configuration can be considered equivalent to another through a rotation that is possibly combined with an inversion of the corner states. There is only one class for which all 48 such transformations produce distinct cases, and that is class number 5 in Figure 10.28. The other classes all have smaller sizes that divide 48.

Another operation that could be applied when partitioning the 256 cases into equivalence classes is reflection. This would cause two cases that are mirror images of each other to be considered equivalent. However, for most of the classes in Figure 10.28, every case produced by reflection through a plane can also be produced by a rotation due to the fact that the corner states possess a mirror symmetry. The only exception is the pair of classes numbered 12 and 13, which are reflections of each other but do not independently possess any mirror symmetry and thus cannot be rotated to coincide. By including reflection in the equivalence relation, these two classes could be collapsed into one, but we keep them as distinct classes to follow established convention.

For a particular cell, bits representing whether each corner is inside or outside the solid volume are concatenated to form a single 8-bit case index, and this index is used to fetch an equivalence class index from a 256-entry lookup table. Once the index of the equivalence class has been selected, it is used to fetch class-specific information from an 18-entry table. This information includes the vertex count and triangulation data that is identical for all cases belonging to the equivalence class. A separate 256-entry table identifies the edges on which the vertices lie for each case and includes information about how vertices can be shared with neighboring cells. The details are provided in Section 10.8.3 below.

10.8.2 Preferred Polarity

The original marching cubes algorithm recognized only the first 15 equivalence classes shown in Figure 10.28, and every class included its inverses. This led to a defect causing holes to appear in the meshes that were generated because the triangulations of two adjacent cells did not always match along the boundary between them. The problem becomes evident when we consider a single face for which two diagonally opposing corners are inside the solid volume and the other two corners are outside, as shown in Figure 10.29. Vertices are placed on all four edges of the
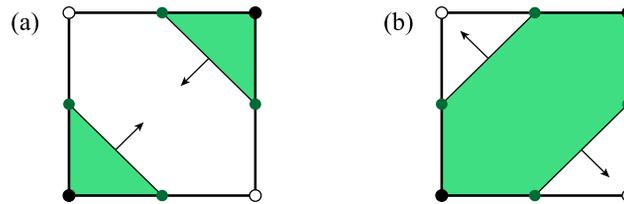
Figure 10.29. There are two ways to connect vertices on an ambiguous cell face for which diagonally opposing corners have the same inside or outside state. Corners with solid dots are classified as inside, and corners with open dots are classified as outside. The green regions correspond to the parts of the faces that are inside solid space, and the arrows show the outward surface normal directions along the edges.

face since a transition from solid space to empty space takes place along each edge. Such a face is called an *ambiguous face* because there are two ways to connect these four vertices with edges so that the inside corners lie in the interior of the mesh and the outside corners lie in the exterior. In Figure 10.29(a), triangle edges connect vertices lying on face edges that share a corner having the inside state, and two separated components of solid space are created on the face. In Figure 10.29(b), triangle edges connect vertices lying on face edges that share a corner having the outside state, and a single component of solid space is created.

Equivalence class number 2 possesses a single ambiguous face, and that face has the edge configuration shown in Figure 10.29(a) for the 12 class members having two corners in solid space. If the 12 inverse cases having six corners in solid space were to be included in the same equivalence class, then the ambiguous faces for those cases would have the opposite edge configuration shown in Figure 10.29(b) because the two triangles generated for the cell would be inverted. This would create an unwanted hole in the mesh whenever the two different edge configurations occurred on the boundary between adjacent cells, as shown in Figure 10.30(a). In order to avoid this problem, we can explicitly forbid the edge configuration shown in Figure 10.29(b) and require that all ambiguous faces use the edge configuration shown in Figure 10.29(a). This solution is called *preferred polarity*, and it can be implemented with full consistency by excluding inverse cases from the three classes numbered 2, 6, and 7. Instead, these inverse cases constitute the separate classes numbered 15, 16, and 17, respectively. After this modification, the left cell in Figure 10.30(b) no longer belongs to class 2, but is now a member of class 15. It consequently has a different triangulation that no longer causes a hole to form.

When preferred polarity is enforced, every nontrivial edge configuration on the face of a cell must match one of the four shown in Figure 10.31 after a possible

rotation. (A fifth configuration is the trivial face having no edges because all four corners have the same state.) The full set of 18 equivalence classes shown in Figure 10.28 satisfy the preferred polarity rule, and only those highlighted in yellow (classes 2, 6, 7, 15, 16, and 17) cannot contain inverses. For the classes numbered 8–14, the inverse of any case is identical to a rotation of the case, so the triangles never need to be inverted, and no potential problem of mismatched edges exists. This is true even for classes 8, 9, and 10 possessing ambiguous faces. The remaining classes (0, 1, 3, 4, and 5) do not have any ambiguous faces and can thus safely include inverse cases because inverting their triangulations never creates an illegal edge configuration.
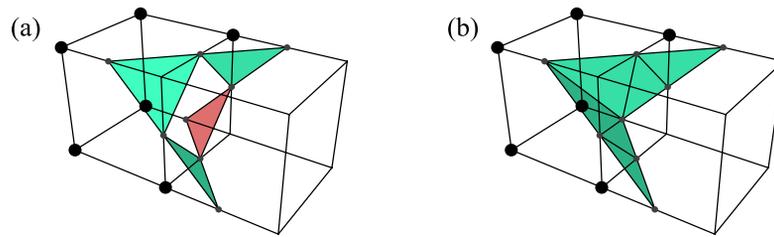
Figure 10.30. Two cells share an ambiguous face. Green polygons are front-facing, and red polygons are back-facing. (a) If all equivalence classes include inverse cases, then both cells belong to class 2, and the cell on the left must be inverted. The edges of the resulting cell triangulations do not match along the shared face, and a large rectangular hole appears. (b) If inverse cases are excluded from equivalence class 2, then the cell on the left is a member equivalence class 15. It gets a different triangulation that satisfies the preferred polarity rule and eliminates the hole in the mesh.
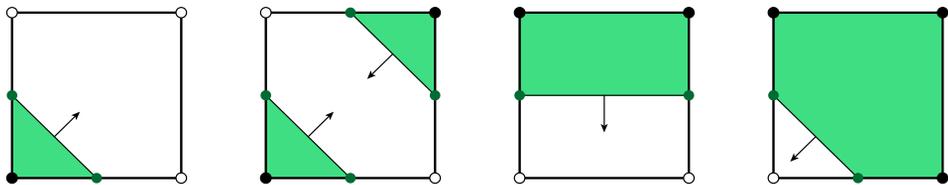
Figure 10.31. These configurations and their rotations represent the full set of nontrivial edge configurations allowed on a cell face. The configuration shown in Figure 10.29(b) is explicitly disallowed. Corners with solid dots are classified as inside, and corners with open dots are classified as outside. The green regions correspond to the parts of the faces that are inside solid space, and the arrows show the outward surface normal directions along the edges.

10.8.3 Implementation

There are many ways in which the marching cubes algorithm could be implemented. In the case that the scalar field data is continually changing, as it typically would be for a metaball particle system, it may be practical to generate triangles for each cell on the GPU in a geometry shader. This makes effective use of the available parallelism because each cell can be processed independently, but it ends up calculating the position, normal, and texture coordinates for most vertices multiple times due to the lack of data sharing between adjacent cells. In cases for which the scalar field is less dynamic, as it is for topographically unrestricted terrain, it makes sense to share vertex data among neighboring cells as often as possible to minimize overall storage requirements and take advantage of post-transform vertex caching.

The marching cubes implementation discussed in this section is designed to run on the CPU and generate a minimal set of vertices by reusing data from previously processed cells. The implementation runs serially over the scalar field data that is provided as input, but coarse parallelism appropriate for relatively low CPU core counts compared to the GPU can be achieved by dividing the data into large pieces that can be processed independently. For metaballs, it's common to find disjoint islands of solid values that can be separated by planes lying in empty space. For terrain, a large landscape is often partitioned into smaller cube-shaped blocks that can be individually culled for visibility. In this latter case, there is some vertex duplication on the boundaries between blocks, but it can usually be justified for the added flexibility that having blocks provides.

Our implementation takes as input an array of $n \times m \times h$ voxels having values stored as 8-bit signed integers in the range $[-127, 127]$. (We exclude the value -128 from our data for symmetry.) As mentioned earlier, any $2 \times 2 \times 2$ subset of voxels is called a *cell*. The coordinates (i, j, k) of a cell correspond to the coordinates of the voxel at the cell's minimal corner is the x, y, and z directions. Each set of $n-1$ cells spanning the input data in the x direction with fixed coordinates $y = j$ and $z = k$ is called a *row*, and each set of $(n-1) \times (m-1)$ cells spanning the input data in the x and y directions with a fixed coordinate $z = k$ is called a *deck*. We consider the x and y directions to be horizontal in the space of the scalar field, and as such, decks can be said to be stacked vertically.

The corners and edges of each cell are numbered as shown in Figure 10.32. This numbering is not arbitrary, but is purposely designed to allow the indices of coincident corners and edges belonging to adjacent cells to be easily calculated. When moving in the x direction from one cell to the previous or next cell in the same row, one is added or subtracted from any corner index. In the y and z direc-
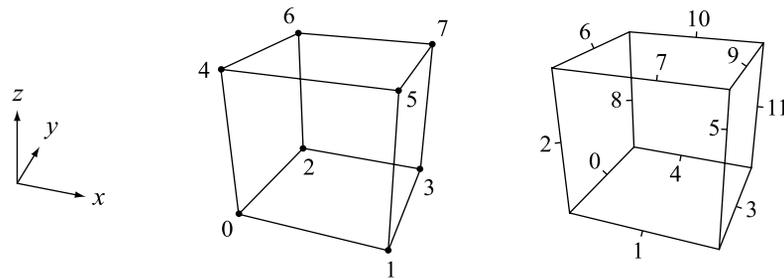
Figure 10.32. The eight corner voxels and twelves edges of a cell are numbered as shown. Incrementing the *x*, *y*, and *z* voxel coordinates by one always adds 1, 2, and 4 to the corner index, respectively. Incrementing the *x* coordinate adds 3 to all edge indices. Incrementing the *y* coordinate adds 3 to the index for horizontal edges and 6 to the index for vertical edges. Incrementing the *z* coordinate adds 6 for edge indices.

tions, corner indices differ by two and four, respectively. For all edges shared by cells that are adjacent in the *x* direction, the indices differ by three, and the same is true for *horizontal* edges shared by cells that are adjacent in the *y* direction. For all edges shared by cells that are adjacent in the *z* direction, the indices differ by six, and the same is true for *vertical* edges shared by cells that are adjacent in the *y* direction.

When a particular cell is processed, the voxels at each of the eight corners of the cell are obtained from the scalar field data, as demonstrated in Listing 10.20. The sign bits of these eight values are extracted and concatenated in the order of corner numbering shown in Figure 10.32 to form a new 8-bit quantity that we call the *case index*. Each zero bit corresponds to a corner outside the volume, in empty space, and each one bit corresponds to a corner inside the volume, in solid space. (This implies that a voxel having a value of exactly zero is considered to be outside.) If the case index is 0 or 255, then all of the corners have the same state and no further processing occurs because the cell trivially contains no part of the isosurface. Otherwise, at least one triangle is generated, and we use lookup tables to determine where the vertices should be placed. Vertices never occur in the interior of a cell, but only on the edges joining corners that have opposite states, and as few as three or as many as all 12 edges may participate.

For each nontrivial cell, the equivalence class index is determined by using the case index to look it up in a 256-entry table. All cases belonging to an equivalence class have the same numbers of vertices and triangles, and the way in which the vertices are connected to form triangles is the same. This information is found in a separate table containing a data structure for each equivalence class. Even though

there are 18 equivalence classes shown in Figure 10.28, a couple pairs of them have identical triangulations, so we have only 16 distinct sets of triangulation data after those pairs are collapsed into one set each. Equivalence classes 2 and 4 both generate two independent triangles using six vertices, and equivalence classes 3 and 11 both generate a single quad using four vertices. The structure of these tables is shown in Listing 10.21.

The set of edges on which vertices are ultimately placed are unique to each of the 256 possible cases, and information about these locations is stored in a third lookup table, also shown in Listing 10.21, accessed with the original case index. In our implementation, each entry of this table contains an array of 16-bit codes having the layout shown in Figure 10.33. Each of these vertex codes contains a

Listing 10.20. The `LoadCell()` function loads the voxel values at the eight corners of a cell having coordinates given by `i`, `j`, and `k` parameters and stores them in the array of eight distances pointed to by the `distance` parameter. The sign bits of the eight values are concatenated to form the case index for the cell, which is then returned by the function. The `field` parameter points to the beginning of the entire scalar field, and the `n` and `m` parameters give the dimensions of the scalar field in the x and y directions.

```
typedef int8 Voxel;

// The GetVoxel() function loads a single value from the scalar field at coords (i,j,k).
inline Voxel GetVoxel(const Voxel *field, int n, int m, int i, int j, int k)
{
    return (field[(k * m + j) * n + i]);
}

uint32 LoadCell(const Voxel *field, int n, int m, int i, int j, int k, Voxel *distance)
{
    distance[0] = GetVoxel(field, n, m, i, j, k);
    distance[1] = GetVoxel(field, n, m, i + 1, j, k);
    distance[2] = GetVoxel(field, n, m, i, j + 1, k);
    distance[3] = GetVoxel(field, n, m, i + 1, j + 1, k);
    distance[4] = GetVoxel(field, n, m, i, j, k + 1);
    distance[5] = GetVoxel(field, n, m, i + 1, j, k + 1);
    distance[6] = GetVoxel(field, n, m, i, j + 1, k + 1);
    distance[7] = GetVoxel(field, n, m, i + 1, j + 1, k + 1);

    // Concatenate sign bits of the voxel values to form the case index for the cell.
    return (((distance[0] >> 7) & 0x01) | ((distance[1] >> 6) & 0x02)
          | ((distance[2] >> 5) & 0x04) | ((distance[3] >> 4) & 0x08)
          | ((distance[4] >> 3) & 0x10) | ((distance[5] >> 2) & 0x20)
          | ((distance[6] >> 1) & 0x40) |  (distance[7] & 0x80));
}
```

4-bit value e_1 identifying the edge on which the vertex lies, using the numbering shown in Figure 10.32, and two 3-bit fields c_0 and c_1 that identify the corners coinciding with the endpoints of that edge. The corner numbers are the only values needed to determine where on the edge the vertex will be placed. The edge number and information contained in the remaining six bits are used to determine how vertices belonging to previously processed cells can be reused.

For each edge on which a vertex is to be generated, we calculate a fixed-point interpolation parameter t using the 8-bit signed distances d_0 and d_1 given by the voxels at the corners c_0 and c_1. The value of t is given by

$$t = \frac{256 \cdot d_1}{d_1 - d_0}, \tag{10.95}$$

where we multiply by 256 before performing the division to obtain an 8-bit fixed-point fraction. In the case that a large scalar field is divided into blocks for which independent triangle meshes are generated, it is important that the parameter t is calculated using a consistent ordering of the endpoints to avoid the appearance of

Listing 10.21. This code shows the structure of the three lookup tables used in our marching cubes implementation. Due to its size, the data in the tables has been omitted, but it is included in the source code available on the website.

```
const uint8 equivClassTable[256] =
{
    // Equivalence class index for each of the 256 possible cases.
};

struct ClassData
{
    uint8     geometryCounts;     // Vertex and triangle counts.
    uint8     vertexIndex[15];    // List of 3-15 vertex indices.
};

const ClassData classGeometryTable[16] =
{
    // Triangulation data for each equivalence class.
};

const uint16 vertexCodeTable[256][12] =
{
    // List of 3-12 vertex codes for each of the 256 possible cases.
    // The meanings of the bit fields are shown in Figure 10.33.
};
```

15 12	11 8	7 6	5 3	2 0
Δ	e_1	e_0	c_1	c_0

15	14 13	12
Δz	Δy	Δx
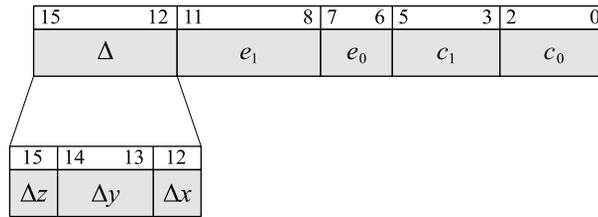
Figure 10.33. Information about each vertex is packed into a 16-bit code having this layout. Using the numbering shown in Figure 10.32, the 4-bit value of e_1 identifies the edge index where the vertex lies, and the 3-bit values of c_0 and c_1 identify the corner indices for that edge. The 2-bit value of e_0 is the reduced edge index given by $e_1 \bmod 3 + 1$. The highest four bits contain a delta code that maps the edge to a coincident edge in a preceding cell, and it is divided in x, y, and z components utilizing one bit, two bits, and one bit, respectively.

seams along block boundaries. For this reason, the value of c_0 in the edge code is always the lower-numbered corner, and the value of c_1 is always the higher-numbered corner.

Using the value of the parameter t, we calculate the position of a vertex using the integer coordinates of the endpoints. The position is later scaled by the physical size of a cell to transform it into a floating-point position in some object-space coordinate system. Let p_0 and p_1 be the integer coordinates of the voxels at the corners c_0 and c_1. (The coordinates (i, j, k) correspond to corner number 0, and the coordinates $(i+1, j+1, k+1)$ correspond to corner number 7.) The fixed-point vertex position v, with eight bits of fraction, is then given by

$$v = t p_0 + (256 - t) p_1. \tag{10.96}$$

If the fraction bits of t are all zero, then v is set to exactly p_0 in the case that $t = 256$ or exactly p_1 in the case that $t = 0$. Otherwise, there are 255 possible interpolated positions at which v can be placed along the interior of the edge.

It is often the case that a vertex needed by the triangulation for one cell has already been generated for the same corner or edge shared by a neighboring cell. This cell could be the preceding cell in the same row, one of two adjacent cells in the preceding row, or one of four adjacent cells in the preceding deck, as shown in Figure 10.34. We handle the cases of a vertex falling exactly at a corner position and a vertex falling in the interior of an edge separately because a vertex at a corner position cannot be uniquely assigned to just one edge.

When a vertex is placed at a corner position, which happens whenever the voxel value at the corner is exactly zero, a 3-bit delta code can be created by simply
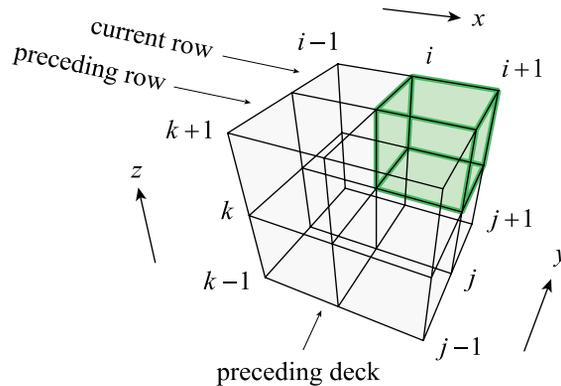
Figure 10.34. The cell currently being processed at coordinates (i, j, k), highlighted in green, may reuse vertices previously generated for the preceding cell in the current row, the two adjacent cells in the preceding row, or the four adjacent cells in the preceding deck.

inverting the bits of the corner number. The delta code tells us which direction to travel in order to find the correct preceding cell by instructing us to subtract one from the x, y, and z coordinates of the cell whenever a one is found in bit positions 0, 1, and 2, respectively. For example, the delta code for corner number 2 is the binary value 101, and this tells us to find the cell whose x and z coordinates are each one less than the coordinates for the current cell, but whose y coordinate is the same. We must be careful not to allow coordinates to become negative, however, and to ensure that never happens, we maintain a 3-bit delta mask whose bits indicate whether it is alright to subtract one in the x, y, and z directions. The delta code generated by inverting a corner number is logically ANDed with the delta mask before it is used to find a preceding cell. If the result is zero, then a new vertex is created for the current cell. Otherwise, a vertex can be reused from a preceding cell, and the index of the corner where that vertex lies is given by adding the masked delta code to the original corner number. If the delta mask is the binary value 111, then the corner index of the reused vertex is always 7 because the original corner number is added to its 3-bit inverse. If the delta mask is any other nonzero value, then the reused vertex can correspond to any corner index except 0.

Continuing the example of corner number 2, suppose that the delta mask is the binary value 110 because we are processing the first cell in a row, meaning that we cannot subtract one from the cell's x coordinate. The masked delta code for this corner is 100 because we logically AND the inverted corner number 101 with the delta mask 110, and this delta code tells us to reuse a vertex from the cell with the same x and y coordinates in the preceding deck. The corner number in the preceding

cell is given by adding the masked delta code 100 to the original corner number 010 to obtain the binary value 110, which is corner number 6. This calculation is performed by the `ReuseCornerVertex()` function in Listing 10.22.

When a vertex is placed in the interior of an edge, a delta code cannot be calculated as easily as it could be for the corners, so we instead include the necessary information in the 16-bit vertex code shown in Figure 10.33. For edges, the delta codes are four bits in size, with an additional bit accounting for the fact that coincident edge numbers can differ by either 3 or 6 when moving from cell to cell in the y direction. Figure 10.35 shows the hexadecimal vertex codes corresponding to each of the 12 edges of a cell, where the highest nibble of each code contains the 4-bit delta code.
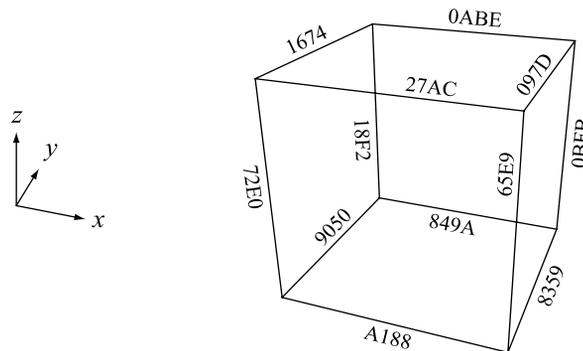
Figure 10.35. These are the 16-bit hexadecimal vertex codes corresponding to each of the 12 edges of a cell. The meanings of the various bit fields making up these codes are shown in Figure 10.33.

The Δx bit of the delta code indicates whether to subtract one from the x coordinate of the cell currently being processed. When its value is one, it also indicates that 3 must be added to the edge number to find the coincident edge in the preceding cell. The two Δy bits can have the binary values 00, 01, or 11, and a nonzero value indicates that one should be subtracted from the y coordinate of the cell. Values of 01 and 11 indicate that 3 and 6 must be added to the edge number, respectively. Finally, the Δz bit indicates whether to subtract one from the z coordinate, and if so, that 6 must be added to the edge number. Any two of these fields can be simultaneously nonzero, but not all three. This calculation is performed by the `ReuseEdgeVertex()` function in Listing 10.22.

A mask similar to the one used for restricting the delta codes for the corners is also used to restrict delta codes for the edges so that cell coordinates never become

negative. The only difference is that the mask for the edges is four bits instead of three, and the bit corresponding to the y direction is duplicated to cover both bits in the delta code. When a vertex needs to be placed in the interior of an edge, we extract the edge number from the e_1 field and the delta code from the Δ field of the vertex code. The delta code is logically ANDed with the 4-bit delta mask, and if the result is nonzero, then the masked delta code is used to determine what preceding cell contains the correct reusable vertex and on which edge it lies in that cell. It is possible for a vertex lying on any edge numbered 3 or greater to be reused.

Listing 10.22. Vertex indices corresponding to corners numbered 1 to 7 and edges numbered 3 to 11 are stored in `CellStorage` structures for every cell in the current deck and preceding deck. The `ReuseCornerVertex()` function determines where the vertex index for a corner belonging to a preceding cell is stored and returns its address. (Corner vertices are not guaranteed to have been generated at the time the preceding cell was processed.) The `ReuseEdgeVertex()` function determines where the vertex index for an edge belonging to a preceding cell is stored and returns its value. (Edge vertices are guaranteed to exist in preceding cells.) For both functions, the `deltaCode` parameter has already had the delta mask applied to it.

```
struct CellStorage
{
    uint16    corner[7];
    uint16    edge[9];
};

inline uint16 *ReuseCornerVertex(int32 n, int32 i, int32 j, int32 k,
            CellStorage *const (& deckStorage)[2], uint16 cornerIndex, uint16 deltaCode)
{
    // The corner index in the preceding cell is the sum of the original
    // corner index and the masked delta code.
    cornerIndex += deltaCode;

    // The three bits of the delta code indicate whether one should
    // be subtracted from the cell coords in the x, y, and z directions.
    int32 dx = deltaCode & 1;
    int32 dy = (deltaCode >> 1) & 1;
    int32 dz = deltaCode >> 2;

    // deckStorage[0] points to the current deck, and
    // deckStorage[1] points to the preceding deck
    CellStorage *deck = deckStorage[dz];

    // Return the address of the vertex index in the preceding cell.
    // The new corner index can never be zero.
    return (&deck[(j - dy) * n + (i - dx)].corner[cornerIndex - 1]);
}
```

```
inline uint16 ReuseEdgeVertex(int32 n, int32 i, int32 j, int32 k,
        const CellStorage *const (& deckStorage)[2], uint16 edgeIndex, uint16 deltaCode)
{
    // Edge index in preceding cell differs from original edge index by 3 for each of
    // the lowest three bits in the masked delta code, and by 6 for the highest bit.
    edgeIndex += ((deltaCode & 1) + ((deltaCode >> 1) & 1) + ((deltaCode >> 2) & 3)) * 3;

    // Bits 0, 1, and 3 of the delta code indicate whether one should
    // be subtracted from the cell coords in the x, y, and z directions.
    int32 dx = deltaCode & 1, dy = (deltaCode >> 1) & 1, dz = deltaCode >> 3;

    // deckStorage[0] points to the current deck, and
    // deckStorage[1] points to the preceding deck
    const CellStorage *deck = deckStorage[dz];

    // Return the vertex index stored in the preceding cell.
    // The new edge index can never be less than 3.
    return (deck[(j - dy) * n + (i - dx)].edge[edgeIndex - 3]);
}
```

As we process each cell and generate its triangles, we need to remember vertex indices for up to seven corners (all except corner 0) and for up to nine edges (all except edges 0, 1, and 2) so that they can be reused by cells processed at a later time. It is never the case that all 16 vertex indices are required at the same time, but it's convenient to have fixed storage locations for all of them, and we accept the additional space usage. Because we never look back further than one deck, we only need to store two decks of $n \times m$ cells worth of history, and we can ping-pong back and forth between which one corresponds to the current deck and which one corresponds to the preceding deck as the z coordinate is incremented.

The ProcessCell() function shown in Listing 10.23 loads the corner data for a single cell, accesses the lookup tables to determine how the cell should be triangulated, and reused vertices generated for preceding cells whenever possible. This function is called repeatedly by the ExtractIsosurface() function shown in Listing 10.24 to generate the complete triangle mesh for an entire scalar field.

In some situations, it is known ahead of time that the values of the scalar field are positive everywhere on the field's boundary. This is typically true for metaball particle systems because if it were not true, then the particle surfaces would not be closed. In these cases, vertices are never generated on the field's boundary, and we never have to worry about whether a preceding cell's coordinates might be negative when we want to reuse vertices. This allows for a simplified implementation in which we may dispense with the delta mask and store fewer reusable vertex indices with each cell. When a corner vertex is reused, it always comes from corner

number 7 in a preceding cell, and when an edge vertex is reused, it always comes from edge number 9, 10, or 11. Thus, we need to remember vertex indices for only four locations in each cell, and we number those locations as shown in Figure 10.36. The location of an on-edge reusable vertex in a preceding cell is given by the reduced edge number e_0 shown in Figure 10.33. Even though this number is related to the original edge number e_1 by the equation $e_0 = e_1 \bmod 3 + 1$, we store it in two bits of the vertex code to avoid having to perform this calculation.
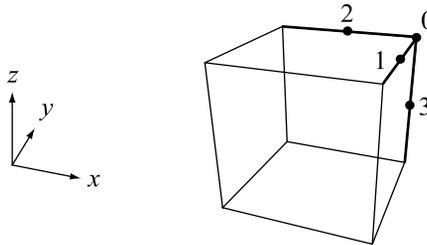
Figure 10.36. When the entire boundary of the scalar field is known to have positive values, meaning that empty space surrounds the isosurface, new vertices are created only at the maximal corner and three maximal edges of any cell, and these locations are numbered 0 through 3 as shown. Vertices generated at other locations can always reuse a vertex previously generated at one of the maximal locations in a preceding cell.

It is possible to generate triangles having zero area when one or more of the voxel values at the corners of a cell is zero. For example, when we triangulate a cell for which one voxel value is zero and the seven remaining voxel values are negative, then we generate the single triangle required by equivalence class 1 in Figure 10.28. However, all three vertices lie exactly at the corner corresponding to the zero voxel value. Such triangles can be eliminated after a simple area calculation, using the fixed-point vertex positions given by Equation (10.96), indicates that they are degenerate.

Listing 10.23. The `ProcessCell()` function generates the vertices and triangles for the cell at coordinates (i, j, k). The `deckStorage` parameter holds two pointers to the vertex reuse storage for the current deck and the preceding deck. The total vertex and triangle counts for the entire mesh are passed in through the `meshVertexCount` and `meshTriangleCount` parameters, and their values are updated when the function returns.

```
void ProcessCell(const Voxel *field, int n, int m, int i, int j, int k,
                 CellStorage *const (& deckStorage)[2], uint32 deltaMask,
                 int& meshVertexCount, int& meshTriangleCount,
                 Integer3D *meshVertexArray, Triangle *meshTriangleArray)
{
    Voxel     distance[8];

    // Get storage for current cell and set vertex indices at corners
    // to invalid values so those not generated here won't get reused.
    CellStorage *cellStorage = &deckStorage[0][j * n + i];
    for (int32 a = 0; a < 7; a++) cellStorage->corner[a] = 0xFFFF;

    // Call LoadCell() to populate the distance array and get case index.
    uint32 caseIndex = LoadCell(field, n, m, i, j, k, distance);

    // Look up the equivalence class index and use it to look up
    // geometric data for this cell. No geometry if case is 0 or 255.
    int32 equivClass = equivClassTable[caseIndex];
    const ClassData *classData = &classGeometryTable[equivClass];
    uint32 geometryCounts = classData->geometryCounts;

    if (geometryCounts != 0)
    {
        uint16    cellVertexIndex[12];

        int32 vertexCount = geometryCounts >> 4;
        int32 triangleCount = geometryCounts & 0x0F;

        // Look up vertex codes using original case index.
        const uint16 *vertexCode = vertexCodeTable[caseIndex];

        // Duplicate middle bit of delta mask to construct 4-bit mask used for edges.
        uint16 edgeDeltaMask = ((deltaMask << 1) & 0x0C) | (deltaMask & 0x03);

        for (int32 a = 0; a < vertexCount; a++)
        {
            uint16        vertexIndex;
            uint8         corner[2];
            Integer3D     position[2];

            // Extract corner numbers from low 6 bits of vertex code.
            uint16 vcode = vertexCodeTable[a];
            corner[0] = vcode & 0x07;
            corner[1] = (vcode >> 3) & 0x07;
```

```cpp
            // Construct integer coordinates of edge's endpoints.
            position[0].x = i + (corner[0] & 1);
            position[0].y = j + ((corner[0] >> 1) & 1);
            position[0].z = k + ((corner[0] >> 2) & 1);
            position[1].x = i + (corner[1] & 1);
            position[1].y = j + ((corner[1] >> 1) & 1);
            position[1].z = k + ((corner[1] >> 2) & 1);

            // Calculate interpolation parameter with Equation (10.95).
            int32 d0 = distance[corner[0]], d1 = distance[corner[1]];
            int32 t = (d1 << 8) / (d1 - d0);

            if ((t & 0x00FF) != 0)
            {
                // Vertex falls in the interior of an edge.
                // Extract edge index and delta code from vertex code.
                uint16 edgeIndex = (vcode >> 8) & 0x0F;
                uint16 deltaCode = (vcode >> 12) & edgeDeltaMask;

                if (deltaCode != 0)
                {
                    // Reuse vertex from edge in preceding cell.
                    vertexIndex = ReuseEdgeVertex(n, i, j, k, deckStorage,
                                        edgeIndex, deltaCode);
                }
                else
                {
                    // Generate a new vertex with Equation (10.96).
                    vertexIndex = meshVertexCount++;
                    Integer3D *vertex = &meshVertexArray[vertexIndex];
                    *vertex = position[0] * t + position[1] * (0x0100 - t);

                    if (edgeIndex >= 3)
                    {
                        // Store vertex index for potential reuse later.
                        cellStorage->edge[edgeIndex - 3] = vertexIndex;
                    }
                }

                cellVertexIndex[a] = vertexIndex;
            }
            else
            {
                // Vertex falls exactly at the first corner of the cell if
                // t == 0, and at the second corner if t == 0x0100.
                uint8 c = (t == 0), cornerIndex = corner[c];

                // Corner vertex in preceding cell may not have been
                // generated, so we get address and look for valid index.
                uint16 *indexAddress = nullptr;
                uint16 deltaCode = (cornerIndex ^ 7) & deltaMask;
```

```
                if (deltaCode != 0)
                {
                    // Reuse vertex from corner in preceding cell.
                    indexAddress = ReuseCornerVertex(n, i, j, k,
                                          deckStorage, cornerIndex, deltaCode);
                }
                else if (cornerIndex != 0)
                {
                    // Vertex will be stored for potential reuse later.
                    indexAddress = &cellStorage->corner[cornerIndex - 1];
                }

                vertexIndex = (indexAddress) ? *indexAddress : 0xFFFF;
                if (vertexIndex == 0xFFFF)
                {
                    // Vertex was not previously generated.
                    vertexIndex = meshVertexCount++;
                    if (indexAddress) *indexAddress = vertexIndex;

                    // Shift corner position to add 8 bits of fraction.
                    meshVertexArray[vertexIndex] = position[c] << 8;
                }

                cellVertexIndex[a] = vertexIndex;
            }
        }

        // Generate triangles for this cell using table data.
        const uint8 *classVertexIndex = classData->vertexIndex;
        Triangle *meshTriangle = &meshTriangleArray[meshTriangleCount];
        meshTriangleCount += triangleCount;

        for (int32 a = 0; a < triangleCount; a++)
        {
            meshTriangle[a].index[0] = cellVertexIndex[classVertexIndex[0]];
            meshTriangle[a].index[1] = cellVertexIndex[classVertexIndex[1]];
            meshTriangle[a].index[2] = cellVertexIndex[classVertexIndex[2]];
            classVertexIndex += 3;
        }
    }
}
```

Listing 10.24. The `ExtractIsosurface()` function generates the complete triangle mesh for an isosurface by visiting every cell in the voxel data specified by the `field` parameter having dimensions given by the n, m, and h parameters. The final numbers of vertices and triangles are returned in the `meshVertexCount` and `meshTriangleCount` parameters. The vertices are stored as fixed-point integers in the buffer specified by the `meshVertexArray` parameter, and the triangles are stored in the buffer specified by the `meshTriangleArray` parameter. These buffers must be allocated by the caller and must be large enough to hold the maximum possible numbers of entries.

```cpp
void ExtractIsosurface(const Voxel *field, int32 n, int32 m, int32 h,
        int32 *meshVertexCount, int32 *meshTriangleCount,
        Integer3D *meshVertexArray, Triangle *meshTriangleArray)
{
    CellStorage      *deckStorage[2];

    // Allocate storage for two decks of history.
    CellStorage *precedingCellStorage = new CellStorage[n * m * 2];

    int32 vertexCount = 0, triangleCount = 0;
    uint16 deltaMask = 0;

    for (int32 k = 0; k < h; k++)
    {
        // Ping-pong between history decks.
        deckStorage[0] = &precedingCellStorage[n * m * (k & 1)];

        for (int32 j = 0; j < m; j++)
        {
            for (int32 i = 0; i < n; i++)
            {
                ProcessCell(field, n, m, i, j, k, deckStorage, deltaMask,
                        vertexCount, triangleCount, meshVertexArray, meshTriangleArray);

                deltaMask |= 1;                     // Allow reuse in x direction.
            }

            deltaMask = (deltaMask | 2) & 6;    // Allow reuse in y direction, but not x.
        }

        deckStorage[1] = deckStorage[0];        // Current deck becomes preceding deck.
        deltaMask = 4;                          // Allow reuse only in z direction.
    }

    delete[] precedingCellStorage;

    *meshVertexCount = vertexCount;
    *meshTriangleCount = triangleCount;
}
```

Exercises for Chapter 10

1. Prove that the maximum brightness B produced by Equation (10.35) for a halo of radius R is $4R/3$.

2. For the sample weights w_i given by Equation (10.72), prove that

$$\sum_{i=0}^{n} w_i = n+1.$$

3. Consider the integral of the Henyey-Greenstein phase function given by Equation (10.76) over all directions in the sphere:

$$\int_{0}^{2\pi} \int_{0}^{\pi} \Phi_g(\alpha) \sin \alpha \, d\alpha \, d\varphi.$$

   Show that this integral is one for all values of g.

4. Prove that the median formula given by Equation (10.83) produces correct results for all values of a, b, and c.

5. Identify symmetry planes for every marching cubes equivalence class except class numbers 0, 12, and 13. For each class, identify two corners that give the plane when combined with cube center.

6. In the marching cubes algorithm, there are five unique ways to triangulate the vertices for class number 5, shown in Figure 10.28. Draw all five possibilities.

7. Formulate a two-dimensional analog of the marching cubes algorithm called *marching squares*. Draw all cases in which the four corners of a square can be classified as either in empty space or in solid space. Determine the equivalence classes (with inverses included), and give their sizes.

Index