CHAPTER FIVE

05 COLOR AND RADIOMETRY

In order to precisely describe how light is represented and sampled to compute images, we must first establish some background in *radiometry*—the study of the propagation of electromagnetic radiation in an environment. Of particular interest in rendering are the wavelengths (λ) of electromagnetic radiation between approximately 370 nm and 730 nm, which account for light visible to humans. The lower wavelengths ($\lambda \approx 400$ nm) are the bluish colors, the middle wavelengths ($\lambda \approx 550$ nm) are the greens, and the upper wavelengths ($\lambda \approx 650$ nm) are the reds.

In this chapter, we will introduce four key quantities that describe electromagnetic radiation: flux, intensity, irradiance, and radiance. These radiometric quantities are each described by their *spectral power distribution* (SPD)—a distribution function of wavelength that describes the amount of light at each wavelength. The `Spectrum` class, which is defined in Section 5.1, is used to represent SPDs in `pbrt`.

5.1 SPECTRAL REPRESENTATION

The SPDs of real-world objects can be quite complicated; Figure 5.1 shows graphs of the spectral distribution of emission from a fluorescent light and the spectral distribution of the reflectance of lemon skin. A renderer doing computations with SPDs needs a compact, efficient, and accurate way to represent functions like these. In practice, some trade-off needs to be made between these qualities.

A general framework for investigating these issues can be developed based on the problem of finding good *basis functions* to represent SPDs. The idea behind basis functions is to map the infinite-dimensional space of possible SPD functions to a low-dimensional space of coefficients $c_i \in \mathbb{R}$. For example, a trivial basis function is the constant function $B(\lambda) = 1$. An arbitrary SPD would be represented in this basis by a single coefficient c equal to its average value, so that its approximation would be $cB(\lambda) = c$. This is obviously

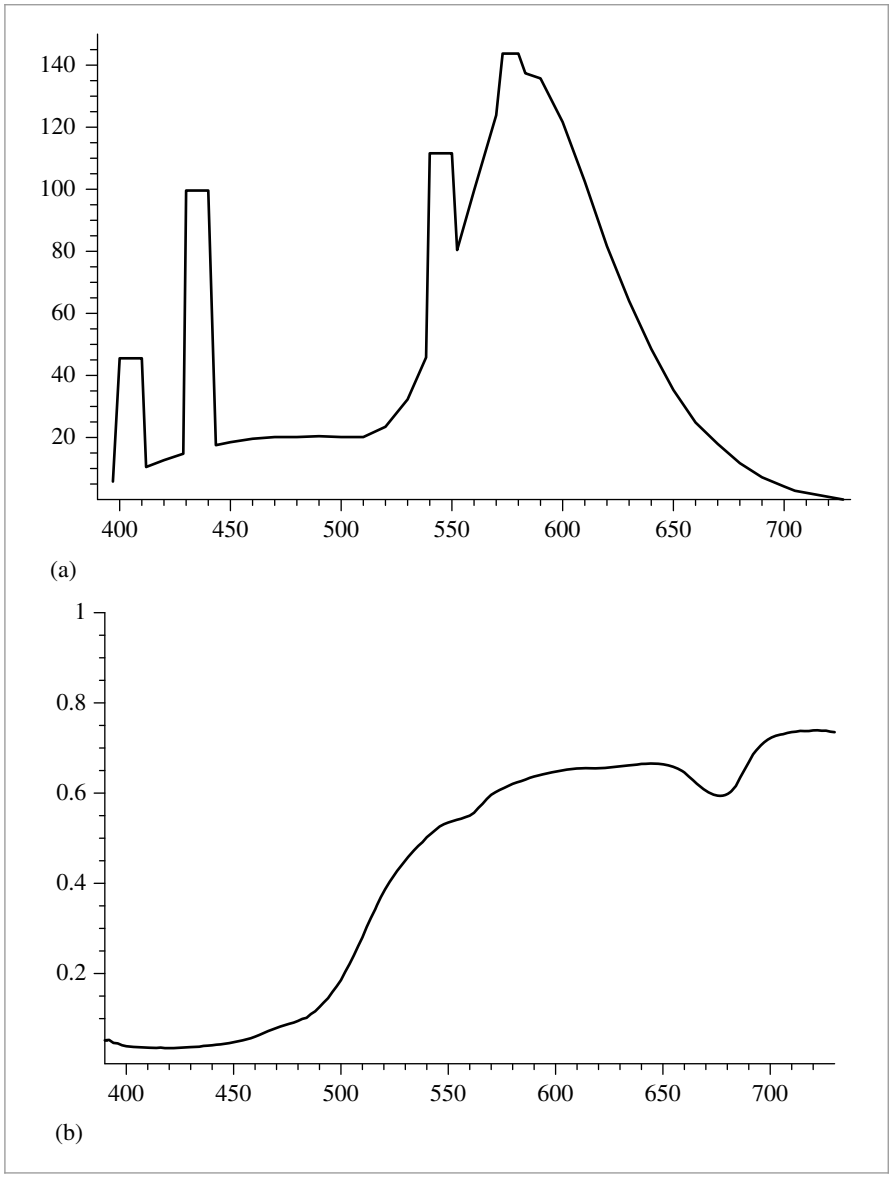

Figure 5.1: (a) Spectral power distributions of a fluorescent light and (b) the reflectance of lemon skin. Wavelengths around 400 nm are bluish colors, greens and yellows are in the middle range of wavelengths, and reds have wavelengths around 700 nm. The fluorescent light's SPD is even spikier than shown here, where the SPDs have been binned into 10 nm ranges; it actually emits much of its illumination at single discrete frequencies.

a poor approximation, since most SPDs are much more complex than this single-basis function is capable of representing accurately.

Many different basis functions have been investigated for spectral representation in computer graphics; the "Further Reading" section cites a number of papers and further resources on this topic. Different sets of basis functions can offset substantially different trade-offs in the complexity of the key operations like converting an arbitrary SPD into a set of coefficients (projecting it into the basis), computing the coefficients for the SPD given by the product of two SPDs expressed in the basis, and so on. In this chapter, we'll introduce two representations that can be used for spectra in `pbrt`: `RGBSpectrum`, which follows the typical computer graphics practice of representing SPDs with coefficients representing red, green, and blue colors, and `SampledSpectrum`, which represents the spectrum as a set of point samples over a range of wavelengths.

5.1.1 THE `Spectrum` TYPE

Throughout `pbrt`, we have been careful to implement all computations involving SPDs in terms of the `Spectrum` type, expecting a particular set of built-in operators (addition, multiplication, etc.). The `Spectrum` type hides the details of the particular spectral representation used, so that changing this detail of the system only requires changing the `Spectrum` implementation; other code can remain unchanged. The implementations of the `Spectrum` type are in the files `core/spectrum.h` and `core/spectrum.cpp`.

The selection of which spectrum representation is used for the `Spectrum` type in `pbrt` is done with a `typedef` in the file `core/pbrt.h`. By default, `pbrt` uses the more efficient but less accurate RGB representation.

⟨*Global Forward Declarations*⟩ ≡
```
typedef RGBSpectrum Spectrum;
// typedef SampledSpectrum Spectrum;
```

We have not written the system such that the selection of which `Spectrum` implementation to use could be resolved at run time; if the implementation is changed, the entire system must be recompiled. One advantage to this design is that many of the various `Spectrum` methods can be implemented as short functions that can be inlined by the compiler, rather than being left as stand-alone functions that have to be invoked through the slow virtual method call mechanism. Inlining frequently used short functions like these can give a substantial improvement in performance. A second advantage is that structures in the system that hold instances of the `Spectrum` type can hold them directly, rather than needing to allocate them dynamically based on the spectral representation chosen at run time.

5.1.2 `CoefficientSpectrum` IMPLEMENTATION

`CoefficientSpectrum` 264
`RGBSpectrum` 279
`SampledSpectrum` 266
`Spectrum` 263

Both of the representations implemented in this chapter are based on storing a fixed number of samples of the SPD. Therefore, we'll start by defining the `CoefficientSpectrum` template class, which represents a spectrum as by a particular number of samples given by the `nSamples` template parameter. Both `RGBSpectrum` and `SampledSpectrum` are built on top of it.

⟨*Spectrum Declarations*⟩ ≡
```
template <int nSamples> class CoefficientSpectrum {
public:
    ⟨CoefficientSpectrum Public Methods 264⟩
protected:
    ⟨CoefficientSpectrum Protected Data 264⟩
};
```

One `CoefficientSpectrum` constructor is provided; it initializes a spectrum with a constant value across all wavelengths.

⟨*CoefficientSpectrum Public Methods*⟩ ≡ 264
```
CoefficientSpectrum(float v = 0.f) {
    for (int i = 0; i < nSamples; ++i)
        c[i] = v;
}
```

⟨*CoefficientSpectrum Protected Data*⟩ ≡ 264
```
float c[nSamples];
```

A variety of arithmetic operations on `Spectrum` objects are needed; the implementations in `CoefficientSpectrum` are all straightforward. First are operations to add pairs of spectral distributions. For the sampled representation, it's easy to show that each sample value for the sum of two SPDs is equal to the sum of the corresponding sample values.

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
CoefficientSpectrum &operator+=(const CoefficientSpectrum &s2) {
    for (int i = 0; i < nSamples; ++i)
        c[i] += s2.c[i];
    return *this;
}
```

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
CoefficientSpectrum operator+(const CoefficientSpectrum &s2) const {
    CoefficientSpectrum ret = *this;
    for (int i = 0; i < nSamples; ++i)
        ret.c[i] += s2.c[i];
    return ret;
}
```

Similarly, subtraction, multiplication, division, and unary negation are defined component-wise. These methods are very similar to the ones already shown, so we won't include them here. `pbrt` also provides equality and inequality tests, also not included here.

It is often useful to know if a spectrum represents an SPD with value zero everywhere. If, for example, a surface has zero reflectance, the light transport routines can avoid the computational cost of casting reflection rays that have contributions that would eventually be multiplied by zeros and thus do not need to be traced.

CoefficientSpectrum 264
CoefficientSpectrum::c 264
Spectrum 263

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
bool IsBlack() const {
    for (int i = 0; i < nSamples; ++i)
        if (c[i] != 0.) return false;
    return true;
}
```

The `Spectrum` implementation (and thus the `CoefficientSpectrum` implementation) must also provide implementations of a number of slightly more esoteric methods, including ones that take the square root of a spectrum or raise the components of a `Spectrum` to a given power. These are needed for some of the computations performed by the `Fresnel` classes in Chapter 8. The implementation of `Sqrt()` takes the square root of each component to give the square root of the SPD. The implementations of `Pow()` and `Exp()` are analoguous and won't be included here.

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
friend CoefficientSpectrum Sqrt(const CoefficientSpectrum &s) {
    CoefficientSpectrum ret;
    for (int i = 0; i < nSamples; ++i)
        ret.c[i] = sqrtf(s.c[i]);
    return ret;
}
```

It's frequently useful to be able to linearly interpolate between two SPDs with a parameter t.

⟨*Spectrum Inline Functions*⟩ ≡
```
inline Spectrum Lerp(float t, const Spectrum &s1, const Spectrum &s2) {
    return (1.f - t) * s1 + t * s2;
}
```

Some portions of the image processing pipeline will want to clamp a spectrum to ensure that the function it represents is within some allowable range.

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
CoefficientSpectrum Clamp(float low = 0, float high = INFINITY) const {
    CoefficientSpectrum ret;
    for (int i = 0; i < nSamples; ++i)
        ret.c[i] = ::Clamp(c[i], low, high);
    return ret;
}
```

Finally, we provide a debugging routine to check if any of the sample values of the SPD is the not-a-number floating-point value (NaN). This situation can happen due to an accidental division by zero; `Assert()`s throughout the system use this method to catch this case close to where it happens.

Assert() 1005
Clamp() 1000
CoefficientSpectrum 264
CoefficientSpectrum::c 264
Fresnel 436
INFINITY 1002
Spectrum 263

⟨*CoefficientSpectrum Public Methods*⟩ +≡ 264
```
bool HasNaNs() const {
    for (int i = 0; i < nSamples; ++i)
        if (isnan(c[i])) return true;
    return false;
}
```

5.2 THE SampledSpectrum CLASS

`SampledSpectrum` uses the `CoefficientSpectrum` infrastructure to represent a SPD with uniformly spaced samples between a starting and an ending wavelength. The wavelength range covers from 400 nm to 700 nm—the range of the visual spectrum where the human visual system is most sensitive. The number of samples, 30, is generally more than enough to accurately represent complex SPDs for rendering. Thus, the first sample represents the wavelength range [400, 410), the second represents [410, 420), and so forth. These values can easily be changed here as needed.

⟨*Spectrum Utility Declarations*⟩ ≡
```
static const int sampledLambdaStart = 400;
static const int sampledLambdaEnd = 700;
static const int nSpectralSamples = 30;
```

⟨*Spectrum Declarations*⟩ +≡
```
class SampledSpectrum : public CoefficientSpectrum<nSpectralSamples> {
public:
    ⟨SampledSpectrum Public Methods 266⟩
private:
    ⟨SampledSpectrum Private Data 271⟩
};
```

By inheriting from the `CoefficientSpectrum` class, `SampledSpectrum` automatically has all of the basic spectrum arithmetic operations defined earlier. The main methods left to define for it are those that initialize it from spectral data and convert the SPD it represents to other spectral representations (such as RGB). The constructor for initializing it with a constant SPD is straightforward.

⟨*SampledSpectrum Public Methods*⟩ ≡ 266
```
SampledSpectrum(float v = 0.f) {
    for (int i = 0; i < nSpectralSamples; ++i) c[i] = v;
}
```

We will often be provided spectral data as a set of (λ_i, v_i) samples, where the ith sample has some value v_i at wavelength λ_i. In general, the samples may have irregular spacing and there may be more or fewer of them than the `SampledSpectrum` stores. (See the directory `scenes/spds` in the `pbrt` distribution for a variety of SPDs for use with `pbrt`, many of them with irregular sample spacing. The file `docs/fileformat.pdf` describes how to use measured spectral data in `pbrt`'s scene description files.)

CoefficientSpectrum 264
CoefficientSpectrum::c 264
nSpectralSamples 266
SampledSpectrum 266

The FromSampled() method takes arrays of SPD sample values v at given wavelengths lambda and uses them to define a piecewise linear function to represent the SPD. For each SPD sample in the SampledSpectrum, it uses the AverageSpectrumSamples() utility function, defined below, to compute the average of the piecewise linear function over the range of wavelengths that each SPD sample is responsible for.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
static SampledSpectrum FromSampled(const float *lambda,
                                   const float *v, int n) {
    ⟨Sort samples if unordered, use sorted for returned spectrum 267⟩
    SampledSpectrum r;
    for (int i = 0; i < nSpectralSamples; ++i) {
        ⟨Compute average value of given SPD over i th sample's range 267⟩
    }
    return r;
}
```

The AverageSpectrumSamples() function requires that the (λ_i, v_i) values be sorted by wavelength. The SpectrumSamplesSorted() function checks that they are; if not, SortSpectrumSamples() sorts them. Note that we allocate new storage for the sorted samples and do not modify the values passed in by the caller; in general, doing so would likely be unexpected behavior for a user of this function (who shouldn't need to worry about these requirements of its specific implementation). We won't include the implementations of either of these two functions here, as they are straightforward.

⟨*Sort samples if unordered, use sorted for returned spectrum*⟩ ≡ 267, 280
```
if (!SpectrumSamplesSorted(lambda, v, n)) {
    vector<float> slambda(&lambda[0], &lambda[n]);
    vector<float> sv(&v[0], &v[n]);
    SortSpectrumSamples(&slambda[0], &sv[0], n);
    return FromSampled(&slambda[0], &sv[0], n);
}
```

In order to compute the value for the ith spectral sample, we compute the range of wavelengths that it's responsible for—lambda0 to lambda1—and use the AverageSpectrumSamples() function to compute the average value of the given piecewise linear SPD over that range. This is a one-dimensional instance of sampling and reconstruction, a topic that will be discussed in more detail in Chapter 7.

⟨*Compute average value of given SPD over i th sample's range*⟩ ≡ 267
```
float lambda0 = Lerp(float(i) / float(nSpectralSamples),
                     sampledLambdaStart, sampledLambdaEnd);
float lambda1 = Lerp(float(i+1) / float(nSpectralSamples),
                     sampledLambdaStart, sampledLambdaEnd);
r.c[i] = AverageSpectrumSamples(lambda, v, n, lambda0, lambda1);
```

Figure 5.2 shows the basic approach taken by AverageSpectrumSamples(): it iterates over each of the linear segments between samples that are partially or fully within the range of wavelengths, lambdaStart to lambdaEnd. For each such segment, it computes the average value over its range, scales the average by the wavelength range the segment covers, and

AverageSpectrumSamples() 268
CoefficientSpectrum::c 264
Lerp() 1000
nSpectralSamples 266
sampledLambdaEnd 266
sampledLambdaStart 266
SampledSpectrum 266
SampledSpectrum::
FromSampled() 267
SortSpectrumSamples() 267
SpectrumSamplesSorted() 267

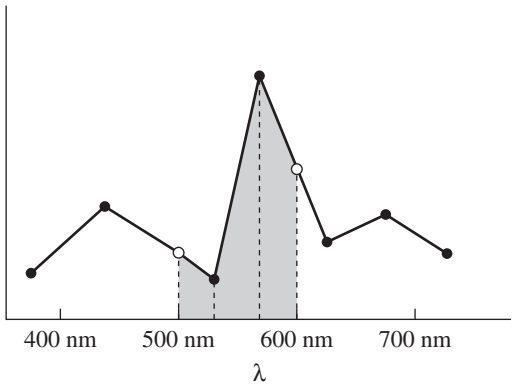

Figure 5.2: When resampling an irregularly defined SPD, we need to compute the average value of the piecewise linear function defined by the SPD samples. Here, we want to average the value from 500 nm to 600 nm—the shaded region under the plot. The `FromSampled()` function does this by computing the areas of each of the regions denoted by dashed lines in this figure.

accumulates a sum of these values. The final average value is this sum divided by the total wavelength range.

⟨*Spectrum Method Definitions*⟩ ≡
```
float AverageSpectrumSamples(const float *lambda, const float *vals,
        int n, float lambdaStart, float lambdaEnd) {
    ⟨Handle cases with out-of-bounds range or single sample only 268⟩
    float sum = 0.f;
    ⟨Add contributions of constant segments before/after samples 269⟩
    ⟨Advance to first relevant wavelength segment 269⟩
    ⟨Loop over wavelength sample segments and add contributions 269⟩
    return sum / (lambdaEnd - lambdaStart);
}
```

The function starts by checking for and handling the edge cases where the range of wavelengths to average over is outside the range of provided wavelengths or the case where there is only a single sample, in which case the average value is trivially computed. We assume that the SPD has a constant value (the values at the two respective end points) outside of the provided sample range; if this isn't a reasonable assumption for a particular set of data, the data provided should have explicit values of (for example) zero at the end points.

⟨*Handle cases with out-of-bounds range or single sample only*⟩ ≡ 268
```
if (lambdaEnd   <= lambda[0])   return vals[0];
if (lambdaStart >= lambda[n-1]) return vals[n-1];
if (n == 1) return vals[0];
```

Having handled these cases, the next step is to check to see if part of the range to average over goes beyond the first and/or last sample value. If so, we accumulate the contribution of the constant segment(s), scaled by the out-of-bounds wavelength range.

⟨*Add contributions of constant segments before/after samples*⟩ ≡ 268
```
if (lambdaStart < lambda[0])
    sum += vals[0] * (lambda[0] - lambdaStart);
if (lambdaEnd > lambda[n-1])
    sum += vals[n-1] * (lambdaEnd - lambda[n-1]);
```

And now we advance to the first index `i` where the starting wavelength of the interpolation range overlaps the segment from λ_i to λ_{i+1}. A more efficient implementation would use a binary search rather than a linear search here.[1] However, this code is currently only called at scene initialization time, so the lack of these optimizations doesn't currently impact rendering performance.

⟨*Advance to first relevant wavelength segment*⟩ ≡ 268
```
int i = 0;
while (lambdaStart > lambda[i+1]) ++i;
```

The loop below iterates over each of the linear segments that the averaging range overlaps. For each one, the `SEG_AVG()` macro computes the average value over the wavelength range `segStart` to `segEnd` by averaging the values of the function at those two points. The values in turn are computed by `INTERP()`, which linearly interpolates between the two end points at the given wavelength.

The `min()` and `max()` calls below compute the wavelength range to average over within the segment; note that they naturally handle the cases where `lambdaStart`, `lambdaEnd`, or both of them are within the current segment.

⟨*Loop over wavelength sample segments and add contributions*⟩ ≡ 268
```
#define INTERP(w, i) \
    Lerp(((w) - lambda[i]) / (lambda[(i)+1] - lambda[i]), \
         vals[i], vals[(i)+1])
#define SEG_AVG(wl0, wl1, i) (0.5f * (INTERP(wl0, i) + INTERP(wl1, i)))
for (; i+1 < n && lambdaEnd >= lambda[i]; ++i) {
    float segStart = max(lambdaStart, lambda[i]);
    float segEnd =   min(lambdaEnd,   lambda[i+1]);
    sum += SEG_AVG(segStart, segEnd, i) * (segEnd - segStart);
}
#undef INTERP
#undef SEG_AVG
```

1 An even more efficient implementation would take advantage of the fact that the calling code will generally ask for interpolated values over a series of adjacent wavelength ranges, and possibly take all of the ranges in a single call. It could then incrementally find the starting index for the next interpolation starting from the end of the previous one.

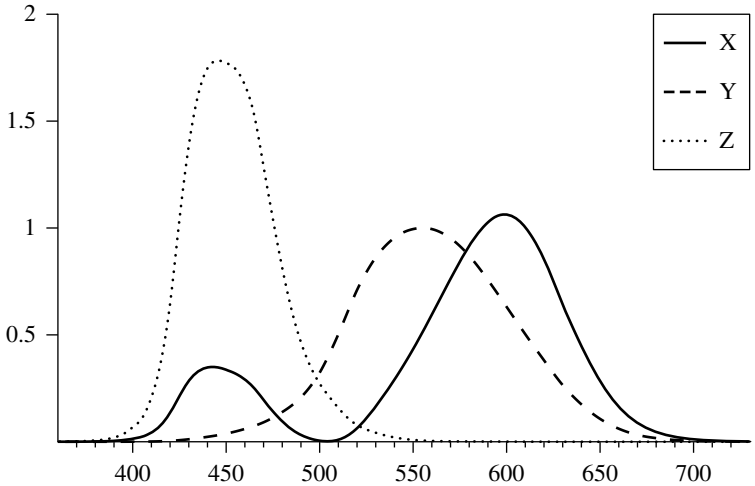

Figure 5.3: Computing the XYZ Values for an Arbitrary SPD. The SPD is convolved with each of the three matching curves to compute the values x_λ, y_λ, and z_λ, using Equation (5.1).

5.2.1 XYZ COLOR

A remarkable property of the human visual system makes it possible to represent colors for human perception with just three floating-point numbers. The *tristimulus theory* of color perception says that all visible SPDs can be accurately represented for human observers with three values, x_λ, y_λ, and z_λ. Given an SPD $S(\lambda)$, these values are computed by integrating its product with the *spectral matching curves* $X(\lambda)$, $Y(\lambda)$, and $Z(\lambda)$:

$$\begin{aligned} x_\lambda &= \frac{1}{\int Y(\lambda)\mathrm{d}\lambda} \int_\lambda S(\lambda)\, X(\lambda)\, \mathrm{d}\lambda \\ y_\lambda &= \frac{1}{\int Y(\lambda)\mathrm{d}\lambda} \int_\lambda S(\lambda)\, Y(\lambda)\, \mathrm{d}\lambda \\ z_\lambda &= \frac{1}{\int Y(\lambda)\mathrm{d}\lambda} \int_\lambda S(\lambda)\, Z(\lambda)\, \mathrm{d}\lambda. \end{aligned} \tag{5.1}$$

These curves were determined by the Commission Internationale de l'Éclairage (CIE) standards body after a series of experiments with human test subjects and are graphed in Figure 5.3. It is believed that these matching curves are generally similar to the responses of the three types of color-sensitive cones in the human retina. Remarkably, SPDs with substantially different distributions may have very similar x_λ, y_λ, and z_λ values. To the human observer, such SPDs actually appear the same visually. Pairs of such spectra are called *metamers*.

This brings us to a subtle point about representations of spectral power distributions. Most color spaces attempt to model colors that are visible to humans, and therefore use

only three coefficients, exploiting the tristimulus theory of color perception. Although XYZ works well to represent a given SPD to be displayed for a human observer, it is *not* a particularly good set of basis functions for spectral computation. For example, although XYZ values would work well to describe the perceived color of lemon skin or a fluorescent light individually (recall Figure 5.1), the product of their respective XYZ values is likely to give a noticeably different XYZ color than the XYZ value computed by multiplying more accurate representations of their SPDs and *then* computing the XYZ value.

pbrt provides the values of the standard $X(\lambda)$, $Y(\lambda)$, and $Z(\lambda)$ response curves sampled at 1 nm increments from 360 nm to 830 nm. The wavelengths of the ith sample in the arrays below are given by the ith element of CIE_lambda; having the wavelengths of the samples explicitly represented in this way makes it easy to pass the XYZ samples into functions like AverageSpectrumSamples() that take such an array of wavelengths as a parameter.

⟨*Spectral Data Declarations*⟩ ≡
```
static const int nCIESamples = 471;
extern const float CIE_X[nCIESamples];
extern const float CIE_Y[nCIESamples];
extern const float CIE_Z[nCIESamples];
extern const float CIE_lambda[nCIESamples];
```

SampledSpectrum uses these samples to compute the XYZ matching curves in its spectral representation (i.e., themselves as SampledSpectrums). The yint value stores the integral of the Y matching curve over the wavelength range; this value is needed for some conversions between full spectra and x_λ, y_λ, z_λ coefficients.

⟨*SampledSpectrum Private Data*⟩ ≡ 266
```
static SampledSpectrum X, Y, Z;
static float yint;
```

The SampledSpectrum XYZ matching curves are computed in the SampledSpectrum::Init() method, which is called at system startup time by the pbrtInit() function defined in Section B.2.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
static void Init() {
    ⟨Compute XYZ matching functions for SampledSpectrum 272⟩
    ⟨Compute RGB to spectrum functions for SampledSpectrum⟩
}
```

AverageSpectrumSamples() 268
nCIESamples 271
pbrtInit() 1051
SampledSpectrum 266

Given the wavelength range and number of samples for SampledSpectrum, computing the values of the matching functions for each sample is just a matter of computing the sample's wavelength range and using the AverageSpectrumSamples() routine.

⟨*Compute XYZ matching functions for* SampledSpectrum⟩ ≡ 271
```
for (int i = 0; i < nSpectralSamples; ++i) {
    float wl0 = Lerp(float(i) / float(nSpectralSamples),
                     sampledLambdaStart, sampledLambdaEnd);
    float wl1 = Lerp(float(i+1) / float(nSpectralSamples),
                     sampledLambdaStart, sampledLambdaEnd);
    X.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_X, nCIESamples,
                                    wl0, wl1);
    Y.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_Y, nCIESamples,
                                    wl0, wl1);
    Z.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_Z, nCIESamples,
                                    wl0, wl1);
    yint += Y.c[i];
}
```

All Spectrum implementations in pbrt must provide a method that converts their SPD to $(x_\lambda, y_\lambda, z_\lambda)$ coefficients. This method is called, for example, in the process of updating pixels in the image. When a Spectrum is provided to the ImageFilm, the ImageFilm converts the SPD into XYZ coefficients as a first step in the process of finally turning them into RGB values used for storage and/or display. The implementation for SampledSpectrum computes the integrals from Equation (5.1) with a Riemann sum:

$$x_\lambda \approx \frac{1}{\int Y(\lambda)\mathrm{d}\lambda} \sum_i X_i c_i,$$

and so forth.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
void ToXYZ(float xyz[3]) const {
    xyz[0] = xyz[1] = xyz[2] = 0.f;
    for (int i = 0; i < nSpectralSamples; ++i) {
        xyz[0] += X.c[i] * c[i];
        xyz[1] += Y.c[i] * c[i];
        xyz[2] += Z.c[i] * c[i];
    }
    xyz[0] /= yint;
    xyz[1] /= yint;
    xyz[2] /= yint;
}
```

The y coordinate of XYZ color is closely related to *luminance*, which measures the perceived brightness of a color. Luminance is described in more detail in Section 5.4.3. We provide a method to compute y alone in a separate method as often only the luminance of a spectrum is desired. (For example, some of the light transport algorithms in Chapters 15–17 use luminance as a measure of relative importance of light-carrying paths through the scene.)

AverageSpectrumSamples() 268
CIE_lambda 271
CIE_X 271
CIE_Y 271
CIE_Z 271
ImageFilm 404
Lerp() 1000
nCIESamples 271
nSpectralSamples 266
sampledLambdaEnd 266
sampledLambdaStart 266
Spectrum 263

⟨*SampledSpectrum Public Methods*⟩ +≡ 266

```
float y() const {
    float yy = 0.f;
    for (int i = 0; i < nSpectralSamples; ++i)
        yy += Y.c[i] * c[i];
    return yy / yint;
}
```

5.2.2 RGB COLOR

When we display an RGB color on a display, the spectrum that is actually displayed is basically determined by the weighted sum of three spectral response curves, one for each of red, green, and blue, as emitted by the display's phosphors, LED or LCD elements, or plasma cells.[2] Figure 5.4 plots the red, green, and blue distributions emitted by a LED display and a LCD display; note that they are remarkably different. Figure 5.5 in turn shows the SPDs that result from displaying the RGB color (0.6, 0.3, 0.2) on those displays. Not surprisingly, the resulting SPDs are quite different as well. This example illustrates that using RGB values provided by the user to describe a particular color is actually only meaningful given knowledge of the characteristics of the display they were using when they selected the RGB values.

Given an $(x_\lambda, y_\lambda, z_\lambda)$ representation of an SPD, we can convert it to corresponding RGB coefficients, given the choice of a particular set of SPDs that define red, green, and blue for a display of interest. The conversion routines implemented in `pbrt` are based on a standard set of these RGB spectra that has been defined for high-definition television. Given the spectral response curves $R(\lambda)$, $G(\lambda)$, and $B(\lambda)$, for a particular display, RGB coefficients can be computed by integrating the reponse curves with the SPD:

$$\begin{aligned} r &= \int R(\lambda)S(\lambda)\mathrm{d}\lambda = \int R(\lambda)(x_\lambda X(\lambda) + y_\lambda Y(\lambda) + z_\lambda Z(\lambda))\,\mathrm{d}\lambda \\ &= x_\lambda \int R(\lambda)X(\lambda)\,\mathrm{d}\lambda + y_\lambda \int R(\lambda)Y(\lambda)\,\mathrm{d}\lambda + z_\lambda \int R(\lambda)Z(\lambda)\,\mathrm{d}\lambda. \end{aligned}$$

The integrals of the products of $R(\lambda)X(\lambda)$ and so forth can be precomputed for given response curves, making it possible to express the full conversion as a matrix:

$$\begin{bmatrix} r \\ g \\ b \end{bmatrix} = \begin{pmatrix} \int R(\lambda)X(\lambda)\mathrm{d}\lambda & \int R(\lambda)Y(\lambda)\mathrm{d}\lambda & \int R(\lambda)Z(\lambda)\mathrm{d}\lambda \\ \int G(\lambda)X(\lambda)\mathrm{d}\lambda & \int G(\lambda)Y(\lambda)\mathrm{d}\lambda & \int G(\lambda)Z(\lambda)\mathrm{d}\lambda \\ \int B(\lambda)X(\lambda)\mathrm{d}\lambda & \int B(\lambda)Y(\lambda)\mathrm{d}\lambda & \int B(\lambda)Z(\lambda)\mathrm{d}\lambda \end{pmatrix} \begin{bmatrix} x_\lambda \\ y_\lambda \\ z_\lambda \end{bmatrix}.$$

⟨*Spectrum Utility Declarations*⟩ +≡

```
inline void XYZToRGB(const float xyz[3], float rgb[3]) {
    rgb[0] =  3.240479f*xyz[0] - 1.537150f*xyz[1] - 0.498535f*xyz[2];
    rgb[1] = -0.969256f*xyz[0] + 1.875991f*xyz[1] + 0.041556f*xyz[2];
    rgb[2] =  0.055648f*xyz[0] - 0.204043f*xyz[1] + 1.057311f*xyz[2];
}
```

CoefficientSpectrum::c 264
nSpectralSamples 266
SampledSpectrum::Y 271

2 This model is admittedly a simplification in that it neglects any additional processing the display does; in particular, many displays perform nonlinear remappings of the displayed values.

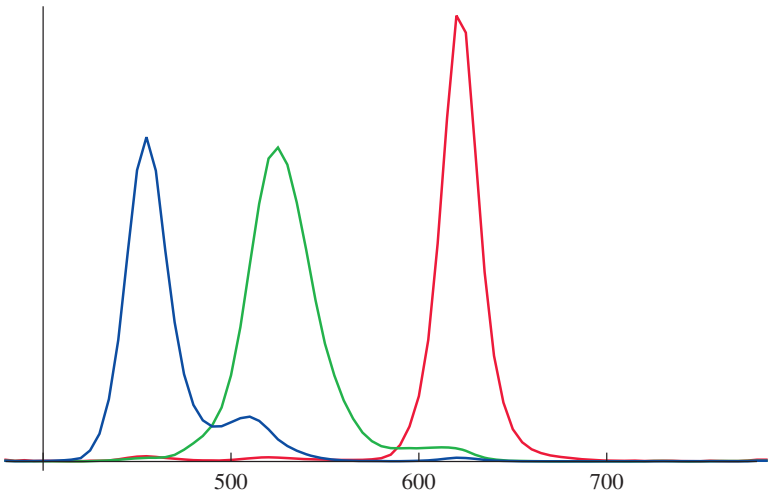

Figure 5.4: Red, Green, and Blue Emission Curves for an LCD Display and a LED Display. The top plot shows the curves for an LCD display and the bottom shows them for an LED. These two displays have quite different emission profiles. *(Data courtesy of X-Rite, Inc.)*

The inverse of this matrix gives coefficients to convert given RGB values, expressed with respect to a particular set of RGB response curves, to $(x_\lambda, y_\lambda, z_\lambda)$ coefficients.

⟨*Spectrum Utility Declarations*⟩ +≡

```
inline void RGBToXYZ(const float rgb[3], float xyz[3]) {
    xyz[0] = 0.412453f*rgb[0] + 0.357580f*rgb[1] + 0.180423f*rgb[2];
    xyz[1] = 0.212671f*rgb[0] + 0.715160f*rgb[1] + 0.072169f*rgb[2];
    xyz[2] = 0.019334f*rgb[0] + 0.119193f*rgb[1] + 0.950227f*rgb[2];
}
```

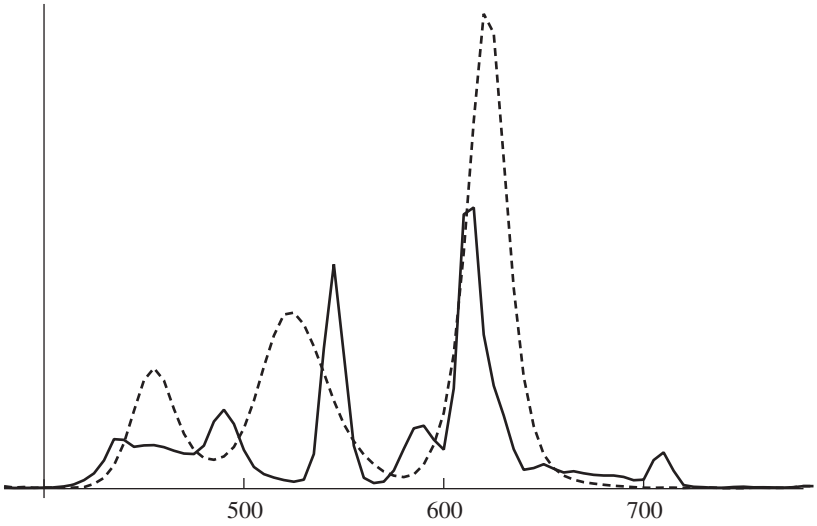

Figure 5.5: SPDs from Displaying the RGB Color (0.6, 0.3, 0.2) **on LED and LCD Displays.** The resulting emitted SPDs are remarkably different, even given the same RGB values, due to the different emission curves illustrated in Figure 5.4.

Given these functions, a `SampledSpectrum` can convert to RGB coefficients by first converting to XYZ and then using the `XYZToRGB()` utility function.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
void ToRGB(float rgb[3]) const {
    float xyz[3];
    ToXYZ(xyz);
    XYZToRGB(xyz, rgb);
}
```

An `RGBSpectrum` can also be created easily, using the `ToRGB()` method.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
RGBSpectrum ToRGBSpectrum() const;
```

Going the other way and converting from RGB or XYZ values to a SPD isn't as easy: the problem is highly under-constrained. Recall that an infinite number of different SPDs have the same $(x_\lambda, y_\lambda, z_\lambda)$ (and thus, RGB) coefficients. Thus, given an RGB or $(x_\lambda, y_\lambda, z_\lambda)$ value, there are an infinite number of possible SPDs that could be chosen for it. There are a number of desirable criteria that we'd like a conversion function to have:

RGBSpectrum 279
SampledSpectrum 266
SampledSpectrum::ToXYZ() 272
XYZToRGB() 273

- If all of the RGB coefficients have the same value, the resulting SPD should be constant.
- In general, it's desirable that the computed SPD be smooth. Most real-world objects have relatively smooth spectra. (The main source of spiky spectra is light sources,

especially fluorescents. Fortunately, actual spectral data is more commonly available for illuminants than it is for reflectances.)

The smoothness goal is one of the reasons why constructing a SPD as a weighted sum of a display's $R(\lambda)$, $G(\lambda)$, and $B(\lambda)$ SPDs is not a good solution: as shown in Figure 5.4, those functions are generally irregular and spiky, and a weighted sum of them will thus not be a very smooth SPD. Although the result will be a metamer of the given RGB values, it's likely not an accurate representation of the SPD of the actual object.

Here we implement a method for converting RGBs to SPDs suggested by Smits (1999) that tries to achieve the goals above. This approach is based on the observation that a good start is to compute individual SPDs for red, green, and blue that are smooth and such that computing the weighted sum of them with the given RGB coefficients and then converting back to RGB give a result that is close to the original RGB coefficients. He found such spectra through a numerical optimization procedure.

Smits observed that two additional improvements could be made to this basic approach. First, rather than representing constant spectra by the sums of the computed red, green, and blue SPDs, the sum of which isn't exactly constant, it's better to represent constant spectra with constant SPDs. Second, mixtures of colors like yellow (a mixture of red and green) that are a mixture of two of the primaries are better represented by their own precomputed smooth SPDs rather than the sum of SPDs for the two corresponding primaries.

The following arrays store SPDs that meet these criteria, with their samples' wavelengths in `RGB2SpectLambda[]` (these data were generated by Karl vom Berge).

⟨*Spectral Data Declarations*⟩ +≡
```
static const int nRGB2SpectSamples = 32;
extern const float RGB2SpectLambda[nRGB2SpectSamples];
extern const float RGBRefl2SpectWhite[nRGB2SpectSamples];
extern const float RGBRefl2SpectCyan[nRGB2SpectSamples];
extern const float RGBRefl2SpectMagenta[nRGB2SpectSamples];
extern const float RGBRefl2SpectYellow[nRGB2SpectSamples];
extern const float RGBRefl2SpectRed[nRGB2SpectSamples];
extern const float RGBRefl2SpectGreen[nRGB2SpectSamples];
extern const float RGBRefl2SpectBlue[nRGB2SpectSamples];
```

If a given RGB color describes illumination from a light source, better results are achieved if the conversion tables are computed using the spectral power distribution of a representative illumination source to define "white" rather than using a constant spectrum as they are for the tables above that are used for reflectances. The `RGBIllum2Spect` arrays use the D65 spectral power distribution, which has been standardized by the CIE to represent midday sunlight.

⟨*Spectral Data Declarations*⟩ +≡
```
extern const float RGBIllum2SpectWhite[nRGB2SpectSamples];
extern const float RGBIllum2SpectCyan[nRGB2SpectSamples];
extern const float RGBIllum2SpectMagenta[nRGB2SpectSamples];
```

nRGB2SpectSamples 276

```
extern const float RGBIllum2SpectYellow[nRGB2SpectSamples];
extern const float RGBIllum2SpectRed[nRGB2SpectSamples];
extern const float RGBIllum2SpectGreen[nRGB2SpectSamples];
extern const float RGBIllum2SpectBlue[nRGB2SpectSamples];
```

The fragment ⟨*Compute RGB to spectrum functions for* SampledSpectrum⟩, which is called from SampledSpectrum::Init(), isn't included here; it initializes the following SampledSpectrum values by resampling the RGBRefl2Spect and RGBIllum2Spect distributions using the AverageSpectrumSamples() function.

⟨*SampledSpectrum Private Data*⟩ +≡ 266

```
static SampledSpectrum rgbRefl2SpectWhite, rgbRefl2SpectCyan;
static SampledSpectrum rgbRefl2SpectMagenta, rgbRefl2SpectYellow;
static SampledSpectrum rgbRefl2SpectRed, rgbRefl2SpectGreen;
static SampledSpectrum rgbRefl2SpectBlue;
```

⟨*SampledSpectrum Private Data*⟩ +≡ 266

```
static SampledSpectrum rgbIllum2SpectWhite, rgbIllum2SpectCyan;
static SampledSpectrum rgbIllum2SpectMagenta, rgbIllum2SpectYellow;
static SampledSpectrum rgbIllum2SpectRed, rgbIllum2SpectGreen;
static SampledSpectrum rgbIllum2SpectBlue;
```

The SampledSpectrum::FromRGB() method converts from the given RGB values to a full SPD. In addition to the RGB values, it takes an enumerant value that denotes whether the RGB value represents surface reflectance or an illuminant; the corresponding rgbIllum2Spect or rgbRefl2Spect values are used for the conversion.

⟨*Spectrum Utility Declarations*⟩ +≡

```
enum SpectrumType { SPECTRUM_REFLECTANCE, SPECTRUM_ILLUMINANT };
```

⟨*Spectrum Method Definitions*⟩ +≡

```
SampledSpectrum SampledSpectrum::FromRGB(const float rgb[3],
                                         SpectrumType type) {
    SampledSpectrum r;
    if (type == SPECTRUM_REFLECTANCE) {
        ⟨Convert reflectance spectrum to RGB 278⟩
    }
    else {
        ⟨Convert illuminant spectrum to RGB⟩
    }
    return r.Clamp();
}
```

AverageSpectrumSamples() 268
SampledSpectrum 266
SampledSpectrum::Init() 271
Spectrum::Clamp() 265
SpectrumType 277
SPECTRUM_REFLECTANCE 277

Here we'll show the conversion process for reflectances. The computation for illuminants is the same, just using the different conversion values. First, the implementation determines whether the red, green, or blue channel is the smallest.

⟨*Convert reflectance spectrum to RGB*⟩ ≡ 277
```
if (rgb[0] <= rgb[1] && rgb[0] <= rgb[2]) {
    ⟨Compute reflectance SampledSpectrum with rgb[0] as minimum 278⟩
}
else if (rgb[1] <= rgb[0] && rgb[1] <= rgb[2]) {
    ⟨Compute reflectance SampledSpectrum with rgb[1] as minimum⟩
}
else {
    ⟨Compute reflectance SampledSpectrum with rgb[2] as minimum⟩
}
```

Here is the code for the case of a red component being the smallest. (The cases for green and blue are analogous and not included in the book here.) If red is the smallest, we know that green and blue have greater values than red. As such, we can start to convert the final SPD to return by assigning to it the value of the red component times the white spectrum in `rgbRefl2SpectWhite`. Having done this, the remaining RGB value left to process is $(0, g - r, b - r)$. The code in turn determines which of the remaining two components is the smallest. This value, times the cyan (green and blue) spectrum, is added to the result and we're left with either $(0, g - b, 0)$ or $(0, 0, b - g)$. Based on whether the green or blue channel is non-zero, the green or blue SPD is scaled by the remainder and the conversion is complete.

⟨*Compute reflectance* SampledSpectrum *with* rgb[0] *as minimum*⟩ ≡ 278
```
r += rgb[0] * rgbRefl2SpectWhite;
if (rgb[1] <= rgb[2]) {
    r += (rgb[1] - rgb[0]) * rgbRefl2SpectCyan;
    r += (rgb[2] - rgb[1]) * rgbRefl2SpectBlue;
}
else {
    r += (rgb[2] - rgb[0]) * rgbRefl2SpectCyan;
    r += (rgb[1] - rgb[2]) * rgbRefl2SpectGreen;
}
```

Given the method to convert from RGB, converting from XYZ color is easy. We first convert from XYZ to RGB and then use the `FromRGB()` method.

⟨*SampledSpectrum Public Methods*⟩ +≡ 266
```
static SampledSpectrum FromXYZ(const float xyz[3],
        SpectrumType type = SPECTRUM_REFLECTANCE) {
    float rgb[3];
    XYZToRGB(xyz, rgb);
    return FromRGB(rgb, type);
}
```

Finally, we provide a constructor that converts from an instance of the `RGBSpectrum` class, again using the infrastructure above.

RGBSpectrum 279
SampledSpectrum 266
SampledSpectrum::FromRGB() 277
SampledSpectrum::rgbRefl2SpectBlue 277
SampledSpectrum::rgbRefl2SpectCyan 277
SampledSpectrum::rgbRefl2SpectGreen 277
SampledSpectrum::rgbRefl2SpectWhite 277
SpectrumType 277
SPECTRUM_REFLECTANCE 277
XYZToRGB() 273

⟨*Spectrum Method Definitions*⟩ +≡
```
SampledSpectrum::SampledSpectrum(const RGBSpectrum &r, SpectrumType t) {
    float rgb[3];
    r.ToRGB(rgb);
    *this = SampledSpectrum::FromRGB(rgb, t);
}
```

5.3 RGBSpectrum IMPLEMENTATION

The RGBSpectrum implementation here represents SPDs with a weighted sum of red, green, and blue components. Recall that this representation is ill defined: given two different computer displays, having them display the same RGB value won't cause them to emit the same SPD. Thus, in order for a set of RGB values to specify an actual SPD, we must know the monitor primaries that they are defined in terms of; this information is generally not provided along with RGB values.

The RGB representation is nevertheless convenient: almost all 3D modeling and design tools use RGB colors, and most 3D content is specified in terms of RGB. Furthermore, it's computationally and storage efficient, requiring just three floating-point values to represent. Our implementation of RGBSpectrum inherits from CoefficientSpectrum, specifying three components to store. Thus, all of the arithmetic operations defined earlier are automatically available for the RGBSpectrum.

⟨*Spectrum Declarations*⟩ +≡
```
class RGBSpectrum : public CoefficientSpectrum<3> {
public:
    ⟨RGBSpectrum Public Methods 279⟩
};
```

⟨*RGBSpectrum Public Methods*⟩ ≡ 279
```
RGBSpectrum(float v = 0.f) : CoefficientSpectrum<3>(v) { }
RGBSpectrum(const CoefficientSpectrum<3> &v)
    : CoefficientSpectrum<3>(v) { }
```

Beyond the basic arithmetic operators, the RGBSpectrum needs to provide methods to convert to and from XYZ and RGB representations. For the RGBSpectrum these are trivial. Note that FromRGB() takes a SpectrumType parameter like the SampledSpectrum instance of this method. Although it's unused here, the FromRGB() methods of these two classes must have matching signatures so that the rest of the system can call them consistently regardless of which spectral representation is being used.

CoefficientSpectrum 264
CoefficientSpectrum::c 264
RGBSpectrum 279
RGBSpectrum::ToRGB() 280
SampledSpectrum 266
SampledSpectrum::FromRGB() 277
SpectrumType 277
SPECTRUM_REFLECTANCE 277

⟨*RGBSpectrum Public Methods*⟩ +≡ 279
```
static RGBSpectrum FromRGB(const float rgb[3],
        SpectrumType type = SPECTRUM_REFLECTANCE) {
    RGBSpectrum s;
    s.c[0] = rgb[0];
    s.c[1] = rgb[1];
    s.c[2] = rgb[2];
    return s;
}
```

Similarly, spectrum representations must be able to convert themselves to RGB values. For the RGBSpectrum, the implementation can sidestep the question of what particular RGB primaries are used to represent the spectral distribution and just return the RGB coefficients directly, assuming that the primaries are the same as the ones already being used to represent the color.

⟨*RGBSpectrum Public Methods*⟩ +≡ 279

```
void ToRGB(float *rgb) const {
    rgb[0] = c[0];
    rgb[1] = c[1];
    rgb[2] = c[2];
}
```

All spectrum representations must also be able to convert themselves to an RGBSpectrum object. This is again trivial here.

⟨*RGBSpectrum Public Methods*⟩ +≡ 279

```
const RGBSpectrum &ToRGBSpectrum() const {
    return *this;
}
```

The implementations of the RGBSpectrum::ToXYZ(), RGBSpectrum::FromXYZ(), and RGBSpectrum::y() methods are based on the utility functions defined above and are not included here.

To create an RGB spectrum from an arbitrary sampled SPD, FromSampled() converts the spectrum to XYZ and then to RGB. It evaluates the piecewise linear sampled spectrum at 1-nm steps, using the InterpolateSpectrumSamples() utility function, at each of the wavelengths where there is a value for the CIE matching functions. It then uses this value to compute the Riemann sum to approximate the XYZ integrals.

⟨*RGBSpectrum Public Methods*⟩ +≡ 279

```
static RGBSpectrum FromSampled(const float *lambda, const float *v,
                               int n) {
    ⟨Sort samples if unordered, use sorted for returned spectrum 267⟩
    float xyz[3] = { 0, 0, 0 };
    float yint = 0.f;
    for (int i = 0; i < nCIESamples; ++i) {
        yint += CIE_Y[i];
        float val = InterpolateSpectrumSamples(lambda, v, n,
                                               CIE_lambda[i]);
        xyz[0] += val * CIE_X[i];
        xyz[1] += val * CIE_Y[i];
        xyz[2] += val * CIE_Z[i];
    }
    xyz[0] /= yint;
    xyz[1] /= yint;
    xyz[2] /= yint;
    return FromXYZ(xyz);
}
```

CIE_lambda 271
CIE_X 271
CIE_Y 271
CIE_Z 271
CoefficientSpectrum::c 264
InterpolateSpectrum Samples() 281
nCIESamples 271
RGBSpectrum 279
RGBSpectrum::FromXYZ() 280

`InterpolateSpectrumSamples()` takes a possibly irregularly sampled set of wavelengths and SPD values (λ_i, v_i) and returns the value of the SPD at the given wavelength λ, linearly interpolating between the two sample values that bracket λ.

⟨*Spectrum Method Definitions*⟩ +≡

```
float InterpolateSpectrumSamples(const float *lambda, const float *vals,
                                 int n, float l) {
    if (l <= lambda[0])   return vals[0];
    if (l >= lambda[n-1]) return vals[n-1];
    for (int i = 0; i < n-1; ++i) {
        if (l >= lambda[i] && l <= lambda[i+1]) {
            float t = (l - lambda[i]) / (lambda[i+1] - lambda[i]);
            return Lerp(t, vals[i], vals[i+1]);
        }
    }
}
```

5.4 BASIC RADIOMETRY

Radiometry provides a set of ideas and mathematical tools to describe light propagation and reflection. It forms the basis of the derivation of the rendering algorithms that will be used throughout the rest of this book. Interestingly enough, radiometry wasn't originally derived from first principles using the physics of light, but was built on an abstraction of light based on particles flowing through space. As such, effects like polarization of light do not naturally fit into this framework, although connections have since been made between radiometry and Maxwell's equations, giving radiometry a solid basis in physics.

Radiative transfer is the phenomenological study of the transfer of radiant energy. It is based on radiometric principles and operates at the *geometric optics* level, where macroscopic properties of light suffice to describe how light interacts with objects much larger than the light's wavelength. It is not uncommon to incorporate phenomena from wave optics models of light, but these results need to be expressed in the language of radiative transfer's basic abstractions.[3] In this manner, it is possible to describe interactions of light with objects of approximately the same size as the wavelength of the light, and thereby model effects like dispersion and interference. At an even finer level of detail, quantum mechanics is needed to describe light's interaction with atoms. Fortunately, direct simulation of quantum mechanical principles is unnecessary for solving rendering problems in computer graphics, so the intractability of such an approach is avoided.

In `pbrt`, we will assume that geometric optics is an adequate model for the description of light and light scattering. This leads to a few basic assumptions about the behavior of light that will be used implicitly throughout the system:

3 Preisendorfer (1965) has connected radiative transfer theory to Maxwell's classical equations describing electromagnetic fields. His framework both demonstrates their equivalence and makes it easier to apply results from one worldview to the other. More recent work was done in this area by Fante (1981).

- *Linearity:* The combined effect of two inputs to an optical system is always equal to the sum of the effects of each of the inputs individually.
- *Energy conservation:* When light scatters from a surface or from participating media, the scattering events can never produce more energy than they started with.
- *No polarization:* We will ignore polarization of the electromagnetic field; therefore, the only relevant property of light is its distribution by wavelength (or, equivalently, frequency).
- *No fluorescence or phosphorescence:* The behavior of light at one wavelength is completely independent of light's behavior at other wavelengths or times. As with polarization, it is not too difficult to include these effects, but they would add relatively little practical value to the system.
- *Steady state:* Light in the environment is assumed to have reached equilibrium, so its radiance distribution isn't changing over time. This happens nearly instantaneously with light in realistic scenes, so it is not a limitation in practice. Note that phosphorescence also violates the steady-state assumption.

The most significant loss from adopting a geometric optics model is that diffraction and interference effects cannot easily be accounted for. As noted by Preisendorfer (1965), this is a hard problem to fix because, for example, the total flux over two areas isn't necessarily equal to the sum of flux over each individual area in the presence of those effects (p. 24).

5.4.1 BASIC QUANTITIES

There are four radiometric quantities that are central to rendering: flux, irradiance/radiant exitance, intensity, and radiance. All of these quantities are generally wavelength dependent. For the remainder of this chapter, we will not make this dependence explicit, but this property is important to keep in mind.

Flux

Radiant flux, also known as *power*, is the total amount of energy passing through a surface or region of space per unit time. Its units are joules/second (J/s), or more commonly watts (W), and it is normally denoted by the symbol Φ. Total emission from light sources is generally described in terms of flux. Figure 5.6 shows flux from a point light source measured by the total amount of energy passing through an imaginary sphere around the light. Note that the total amount of flux measured on either of the two spheres in Figure 5.6 is the same—although less energy is passing through any local part of the large sphere than the small sphere, the greater area of the large sphere means that the total flux is the same.

Irradiance and Radiant Exitance

Irradiance (E) is the area density of flux arriving at a surface, and *radiant exitance* (M) is the area density of flux leaving a surface. These measurements have units of W/m^2. (The term "irradiance" is sometimes also used to refer to flux leaving a surface, but for clarity we'll use different terms for the two cases.)

For the point light source example in Figure 5.6, irradiance at a point on the outer sphere is less than the irradiance at a point on the inner sphere, since the surface area of the outer

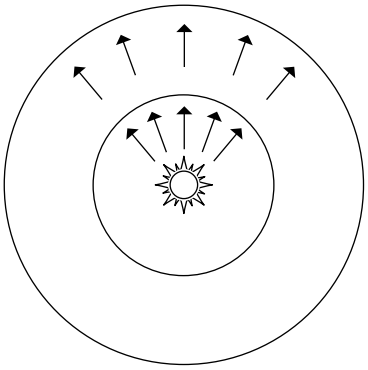

Figure 5.6: Radiant flux, Φ, measures energy passing through a surface or region of space. Here, flux from a point light source is measured at spheres that surround the light.

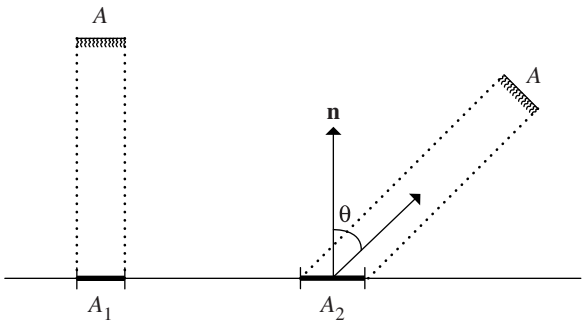

Figure 5.7: Lambert's Law. Irradiance (E) arriving at a surface varies according to the cosine of the angle of incidence of illumination, since illumination is over a larger area at smaller incident angles.

sphere is larger. In particular, for a sphere in this configuration that has radius r,

$$E = \frac{\Phi}{4\pi r^2}.$$

This fact explains why the amount of energy received from a light falls off with the squared distance from the light.

The irradiance equation can also help us understand the origin of *Lambert's law*, which says that the amount of light arriving at a surface is proportional to the cosine of the angle between the light direction and the surface normal (Figure 5.7). Consider a light source with area A and flux Φ that is illuminating a surface. If the light is shining directly down

on the surface (as on the left side of the figure), then the area on the surface receiving light A_1 is equal to A. Irradiance at any point inside A_1 is then

$$E_1 = \frac{\Phi}{A}.$$

However, if the light is at an angle to the surface, the area on the surface receiving light is larger. If A is small, then the area receiving flux, A_2, is roughly $A/\cos\theta$. For points inside A_2, the irradiance is therefore

$$E_2 = \frac{\Phi \cos\theta}{A}.$$

More formally, to cover the general case where the emitted flux distribution is not constant, irradiance at a point is defined as

$$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A},$$

where the differential flux from the light is computed over the differential area receiving flux.

Solid Angle and Intensity

In order to define *intensity*, we first need to define the notion of a *solid angle*. Solid angles are just the extension of two-dimensional angles in a plane to an angle on a sphere. The *planar angle* is the total angle subtended by some object with respect to some position (Figure 5.8). Consider the unit circle around the point p; if we project the shaded object onto that circle, some length of the circle s will be covered by its projection. The arc length of s (which is the same as the angle θ) is the angle subtended by the object. Planar angles are measured in *radians*.

The solid angle extends the 2D unit circle to a 3D unit sphere (Figure 5.9). The total area s is the solid angle subtended by the object. Solid angles are measured in *steradians*. The entire sphere subtends a solid angle of 4π, and a hemisphere subtends 2π.

The set of points on the unit sphere centered at a point p can be used to describe the vectors anchored at p. We will frequently use the symbol ω to indicate these directions, and we will use the convention that ω is a normalized vector.

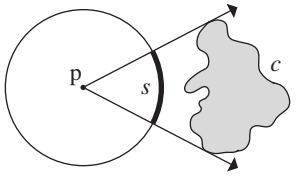

Figure 5.8: Planar Angle. The planar angle of an object c as seen from a point p is equal to the angle it subtends as seen from p, or equivalently as the length of the arc s on the unit sphere.

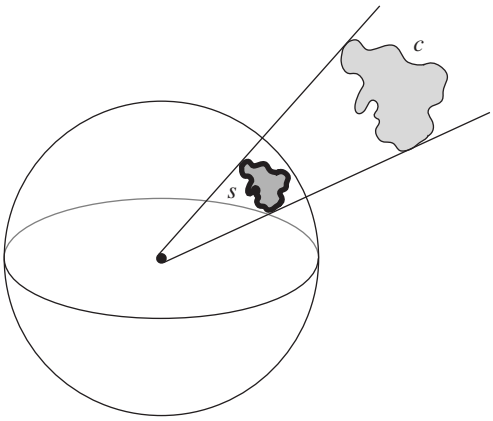

Figure 5.9: Solid Angle. The solid angle s subtended by an object c in three dimensions is computed by projecting c onto the unit sphere and measuring the area of its projection.

Given these definitions, we can now define intensity, which is flux density per solid angle:

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

Intensity describes the directional distribution of light, but it is only meaningful for point light sources.

Radiance

The final, and most important, radiometric quantity is *radiance*, L. Radiance is the flux density per unit area, per unit solid angle. In terms of flux, it is

$$L = \frac{\mathrm{d}\Phi}{\mathrm{d}\omega\, \mathrm{d}A^{\perp}}, \tag{5.2}$$

where $\mathrm{d}A^{\perp}$ is the projected area of $\mathrm{d}A$ on a hypothetical surface perpendicular to ω (Figure 5.10). Thus, it is the limit of the measurement of incident light at the surface as a cone of incident directions of interest $\mathrm{d}\omega$ becomes very small, and as the local area of interest on the surface $\mathrm{d}A$ also becomes very small.

Of all of these radiometric quantities, radiance will be the one used most frequently throughout the rest of the book. An intuitive reason for this is that in some sense it's the most fundamental of all the radiometric quantities; if radiance is given, then all of the other values can be computed in terms of integrals of radiance over areas and directions. Another nice property of radiance is that it remains constant along rays through empty space. It is thus a natural quantity to compute with ray tracing.

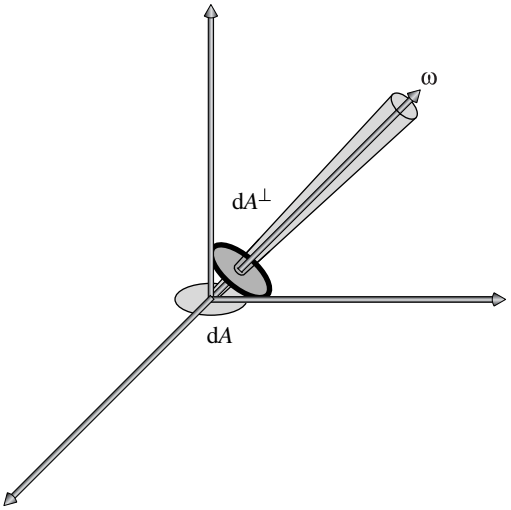

Figure 5.10: Radiance L is defined as flux per unit solid angle $d\omega$ per unit projected area $dA^{\perp}$.

5.4.2 INCIDENT AND EXITANT RADIANCE FUNCTIONS

When discussing radiance at a point in an environment and when writing equations based on radiance, it is useful to make a distinction between radiance arriving at the point (for example, due to illumination from a light source) and radiance leaving that point (for example, due to reflection from a surface). In `pbrt` we will use the idea of incident and exitant radiance functions to distinguish between these two cases. The approach used here is based on Section 3.5 of Veach (1997).

Consider a point p on the surface of an object. There is some distribution of radiance arriving at the point that can be described mathematically by a function of position and direction. This function is denoted by $L_i(p, \omega)$ (Figure 5.11). The function that describes the outgoing reflected radiance from the surface at that point is denoted by $L_o(p, \omega)$. Note that in both cases the direction vector ω is oriented to point away from p, but be aware that some authors use a notation where ω is reversed for L_i terms so that it points toward p.

It is important to see that, in general,

$$L_i(p, \omega) \neq L_o(p, \omega).$$

Another property to keep in mind is that at a point in space where there is no surface and where there is no participating media causing scattering, these functions are related by

$$L_o(p, \omega) = L_i(p, -\omega).$$

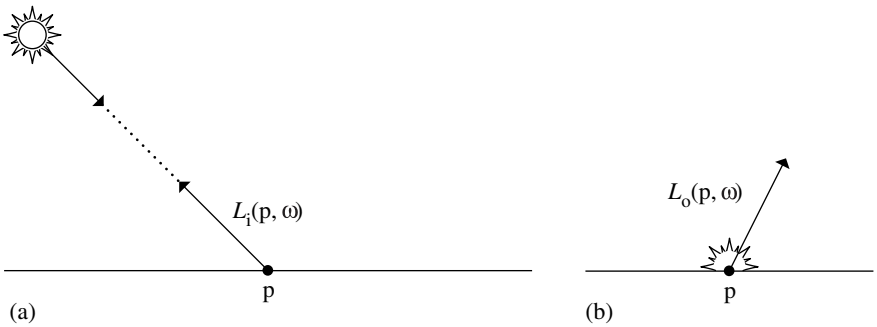

Figure 5.11: (a) The incident radiance function $L_i(p, \omega)$ describes the distribution of radiance as a function of position and direction. (b) The exitant radiance function $L_o(p, \omega)$ gives the distribution of radiance leaving the surface. Note that for both functions, ω is oriented to point away from the surface and, thus for example, $L_i(p, -\omega)$ gives the radiance arriving on the other side of the surface than the one where ω lies.

5.4.3 LUMINANCE AND PHOTOMETRY

All of the radiometric measurements like flux, radiance, and so forth have corresponding photometric measurements. *Photometry* is the study of visible electromagnetic radiation in terms of its perception by the human visual system. Each spectral radiometric quantity can be converted to its corresponding photometric quantity by integrating against the spectral response curve $V(\lambda)$, which describes the relative sensitivity of the human eye to various wavelengths.[4] *Luminance* measures how bright a spectral power distribution appears to a human observer. For example, luminance accounts for the fact that an SPD with a particular amount of energy in the green wavelengths will appear much brighter to a human than an SPD with the same amount of energy in blue.

We will denote luminance here by Y; it related to spectral radiance $L(\lambda)$ by

$$Y = \int_\lambda L(\lambda)\, V(\lambda)\, d\lambda.$$

Luminance and the spectral response curve $V(\lambda)$ are closely related to the XYZ representation of color (Section 5.2.1). The CIE $Y(\lambda)$ tristimulus curve was chosen to be proportional to $V(\lambda)$ so that

$$Y = 683 \int_\lambda L(\lambda)\, Y(\lambda)\, d\lambda.$$

The units of luminance are candelas per meter squared (cd/m^2), where the candela is the photometric equivalent of radiant intensity. The quantity cd/m^2 is often referred to as a *nit*. Some representative luminance values are given in Table 5.1.

4 The spectral response curve model is based on experiments done in a normally illuminated indoor environment. Because sensitivity to color decreases in dark environments, it doesn't model the human visual system's response well under all lighting situations. Nonetheless, it forms the basis for the definition of luminance and other related photometric properties.

Table 5.1: Representative Luminance Values for a Number of Lighting Conditions.

Condition	Luminance (cd/m^2, or nits)
Sun at horizon	600,000
60-watt light bulb	120,000
Clear sky	8,000
Typical office	100–1000
Typical computer display	1–100
Street lighting	1–10
Cloudy moonlight	0.25

5.5 WORKING WITH RADIOMETRIC INTEGRALS

One of the most frequent tasks in rendering is the evaluation of integrals of radiometric quantities. In this section, we will present some tricks that can make this task easier. To illustrate the use of these techniques, we will use the computation of irradiance at a point as an example. Irradiance at a point p with surface normal **n** due to radiance over a set of directions Ω is

$$E(\mathrm{p}, \mathbf{n}) = \int_{\Omega} L_i(\mathrm{p}, \omega) \, |\cos\theta| \, \mathrm{d}\omega, \tag{5.3}$$

where $L_i(\mathrm{p}, \omega)$ is the incident radiance function (Figure 5.12) and the $\cos\theta$ term in this integral is due to the $\mathrm{d}A^{\perp}$ term in the definition of radiance. θ is measured as the angle between ω and the surface normal **n**. Irradiance is usually computed over the hemisphere $\mathcal{H}^2(\mathbf{n})$ of directions about a given surface normal **n**.

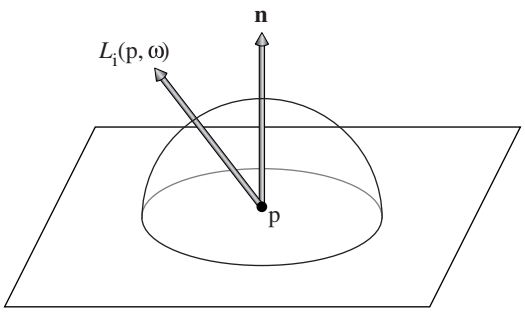

Figure 5.12: Irradiance at a point p is given by the integral of radiance times the cosine of the incident direction over the entire upper hemisphere above the point.

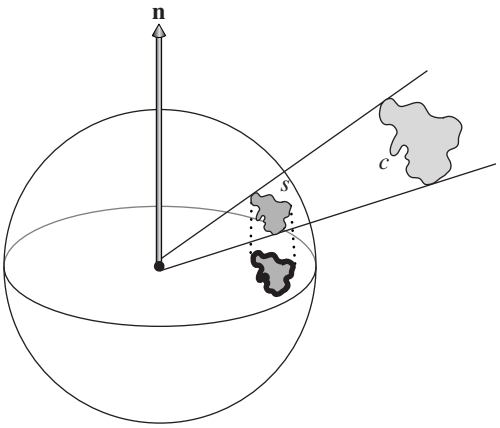

Figure 5.13: The projected solid angle subtended by an object c is the cosine-weighted solid angle that it subtends. It can be computed by finding the object's solid angle s, projecting it down to the plane perpendicular to the surface normal, and measuring its area there. Thus, the projected solid angle depends on the surface normal where it is being measured, since the normal orients the plane of projection.

5.5.1 INTEGRALS OVER PROJECTED SOLID ANGLE

The various cosine terms in the integrals for radiometric quantities can often distract from what is being expressed in the integral. This problem can be avoided using *projected solid angle* rather than solid angle to measure areas subtended by objects being integrated over. The projected solid angle subtended by an object is determined by projecting the object onto the unit sphere, as was done for the solid angle, but then projecting the resulting shape down onto the unit disk (Figure 5.13). Integrals over hemispheres of directions with respect to cosine-weighted solid angle can be rewritten as integrals over projected solid angle.

The projected solid angle measure is related to the solid angle measure by

$$\mathrm{d}\omega^{\perp} = |\cos\theta|\,\mathrm{d}\omega,$$

so the irradiance-from-radiance integral over the hemisphere can be written more simply as

$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L_{\mathrm{i}}(\mathrm{p}, \omega)\,\mathrm{d}\omega^{\perp}.$$

For the rest of this book, we will write integrals over directions in terms of solid angle, rather than projected solid angle. In other sources, however, projected solid angle can be common, and it is always important to be aware of the integrand's actual measure.

Just as we found irradiance in terms of incident radiance, we can also compute the total flux emitted from some object over the hemisphere surrounding the normal by

integrating over the object's surface area A:

$$\Phi = \int_A \int_{\mathcal{H}^2(\mathbf{n})} L_o(\mathrm{p}, \omega) \cos\theta \, \mathrm{d}\omega \, \mathrm{d}A$$
$$= \int_A \int_{\mathcal{H}^2(\mathbf{n})} L_o(\mathrm{p}, \omega) \, \mathrm{d}\omega^{\perp} \, \mathrm{d}A.$$

5.5.2 INTEGRALS OVER SPHERICAL COORDINATES

It is often convenient to transform integrals over solid angle into integrals over spherical coordinates (θ, ϕ). Recall that an (x, y, z) direction vector can also be written in terms of spherical angles (Figure 5.14):

$$x = \sin\theta \cos\phi$$
$$y = \sin\theta \sin\phi$$
$$z = \cos\theta.$$

In order to convert an integral over a solid angle to an integral over (θ, ϕ), we need to be able to express the relationship between the differential area of a set of directions $\mathrm{d}\omega$ and the differential area of a (θ, ϕ) pair (Figure 5.15). The differential area $\mathrm{d}\omega$ is the product of the differential lengths of its sides, $\sin\theta \, \mathrm{d}\phi$ and $\mathrm{d}\theta$. Therefore,

$$\mathrm{d}\omega = \sin\theta \, \mathrm{d}\theta \, \mathrm{d}\phi. \quad [5.4]$$

We can thus see that the irradiance integral over the hemisphere, Equation (5.3) with

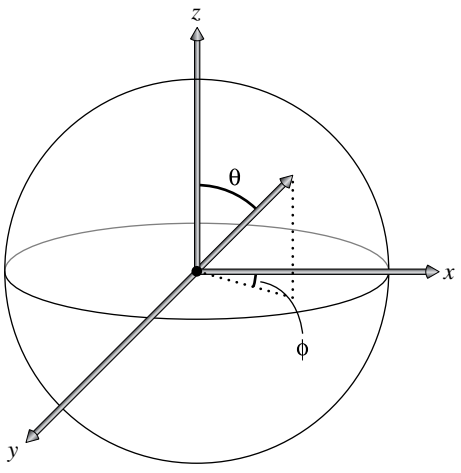

Figure 5.14: A direction vector can be written in terms of spherical coordinates (θ, ϕ) if the x, y, and z basis vectors are given as well. The spherical angle formulae make it easy to convert between the two representations.

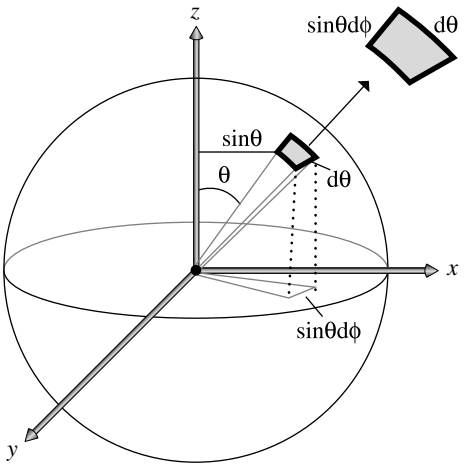

Figure 5.15: The differential area $\mathrm{d}A$ subtended by a differential solid angle is the product of the differential lengths of the two edges $\sin\theta\,\mathrm{d}\phi$ and $\mathrm{d}\theta$. The resulting relationship, $\mathrm{d}\omega = \sin\theta\,\mathrm{d}\theta\mathrm{d}\phi$, is the key to converting between integrals over solid angles and integrals over spherical angles.

$\Omega = \mathcal{H}^2(\mathbf{n})$, can equivalently be written as

$$E(\mathrm{p}, \mathbf{n}) = \int_0^{2\pi} \int_0^{\pi/2} L_\mathrm{i}(\mathrm{p}, \theta, \phi) \cos\theta \ \sin\theta \ \mathrm{d}\theta \, \mathrm{d}\phi$$

If the radiance is the same from all directions, this simplifies to $E = \pi L_\mathrm{i}$.

For convenience, we'll define two functions that convert θ and ϕ values into (x, y, z) direction vectors. The first function applies the earlier equations directly. Notice that these functions are passed the sine and cosine of θ, rather than θ itself. This is because the sine and cosine of θ are often already available to the caller. This is not normally the case for ϕ, however, so ϕ is passed in as is.

⟨*Geometry Inline Functions*⟩ +≡
```
inline Vector SphericalDirection(float sintheta,
                                 float costheta, float phi) {
    return Vector(sintheta * cosf(phi),
                  sintheta * sinf(phi),
                  costheta);
}
```

Vector 57

The second function takes three basis vectors representing the x, y, and z axes and returns the appropriate direction vector with respect to the coordinate frame defined by them:

⟨*Geometry Inline Functions*⟩ +≡
```
inline Vector SphericalDirection(float sintheta, float costheta,
                                 float phi, const Vector &x,
                                 const Vector &y, const Vector &z) {
    return sintheta * cosf(phi) * x +
           sintheta * sinf(phi) * y + costheta * z;
}
```

The conversion of a direction (x, y, z) to spherical angles can be found by

$$\theta = \arccos z$$
$$\phi = \arctan \frac{y}{x}.$$

The corresponding functions follow. Note that `SphericalTheta()` assumes that the vector `v` has been normalized before being passed in.

⟨*Geometry Inline Functions*⟩ +≡
```
inline float SphericalTheta(const Vector &v) {
    return acosf(Clamp(v.z, -1.f, 1.f));
}
```

⟨*Geometry Inline Functions*⟩ +≡
```
inline float SphericalPhi(const Vector &v) {
    float p = atan2f(v.y, v.x);
    return (p < 0.f) ? p + 2.f*M_PI : p;
}
```

5.5.3 INTEGRALS OVER AREA

One last transformation of integrals that can simplify computation is to turn integrals over directions into integrals over area. Consider the irradiance integral in Equation (5.3) again, and imagine there is a quadrilateral with constant outgoing radiance and we'd like to compute the resulting irradiance at a point p. Computing this value as an integral over directions is not straightforward, since given a particular direction it is nontrivial to determine if the quadrilateral is visible in that direction. It's much easier to compute the irradiance as an integral over the area of the quadrilateral.

Differential area is related to differential solid angle (as viewed from a point p) by

$$\mathrm{d}\omega = \frac{\mathrm{d}A \ \cos\theta}{r^2}, \tag{5.5}$$

where θ is the angle between the surface normal of $\mathrm{d}A$ and the vector to p, and r is the distance from p to $\mathrm{d}A$ (Figure 5.16). We will not derive this result here, but it can be understood intuitively: If $\mathrm{d}A$ is at distance 1 from p and is aligned exactly so that it is perpendicular to $\mathrm{d}\omega$, then $\mathrm{d}\omega = \mathrm{d}A$, $\theta = 0$, and Equation (5.5) holds. As $\mathrm{d}A$ moves farther away from p, or as it rotates so that it's not aligned with the direction of $\mathrm{d}\omega$, the r^2 and $\cos\theta$ terms compensate accordingly to reduce $\mathrm{d}\omega$.

M_PI 1002
SphericalTheta() 292
Vector 57

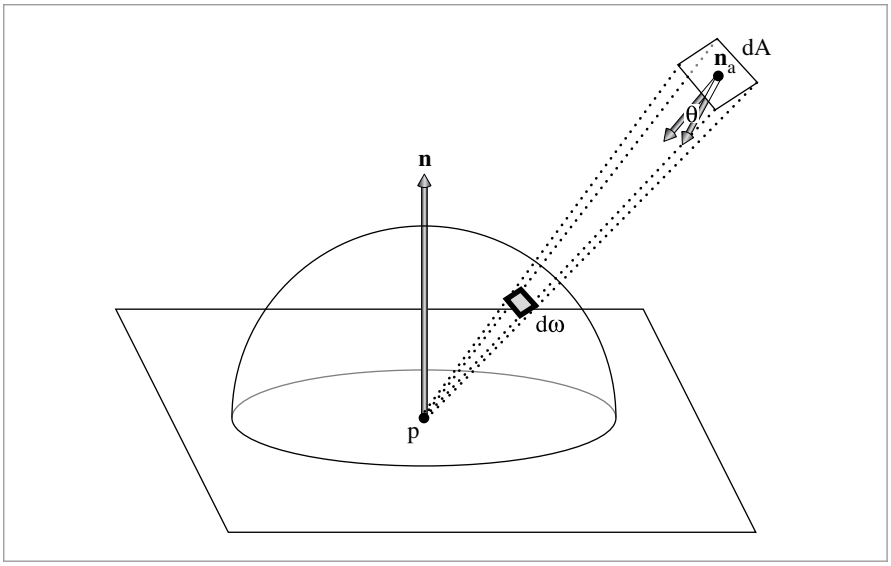

Figure 5.16: The differential solid angle subtended by a differential area $\mathrm{d}A$ is equal to $\mathrm{d}A \cos\theta / r^2$, where θ is the angle between $\mathrm{d}A$'s surface normal and the vector to the point p and r is the distance from p to $\mathrm{d}A$.

Therefore, we can write the irradiance integral for the quadrilateral source as

$$E(\mathrm{p}, \mathbf{n}) = \int_A L \cos\theta_\mathrm{i} \frac{\cos\theta_\mathrm{o}\, \mathrm{d}A}{r^2},$$

where L is the emitted radiance from the surface of the quadrilateral, θ_i is the angle between the surface normal at p and the direction from p to the point p′ on the light, and θ_o is the angle between the surface normal at p′ on the light and the direction from p′ to p (Figure 5.17).

5.6 SURFACE REFLECTION

When light is incident on a surface, the surface scatters the light, reflecting some of it back into the environment. There are two main effects that need to be described to model this reflection: the spectral distribution of the reflected light and its directional distribution. For example, the skin of a lemon mostly absorbs light in the blue wavelengths, but reflects most of the light in the red and green wavelengths (recall the lemon skin reflectance SPD in Figure 5.1). Therefore, when it is illuminated with white light, its color is yellow. The skin has pretty much the same color no matter what direction it's being observed from, although for some directions a highlight—a brighter area that is more white than yellow—is visible. In contrast, the light reflected from a point in a mirror depends almost entirely on the viewing direction. At a fixed point on the mirror, as the viewing angle changes, the object that is reflected in the mirror changes accordingly.

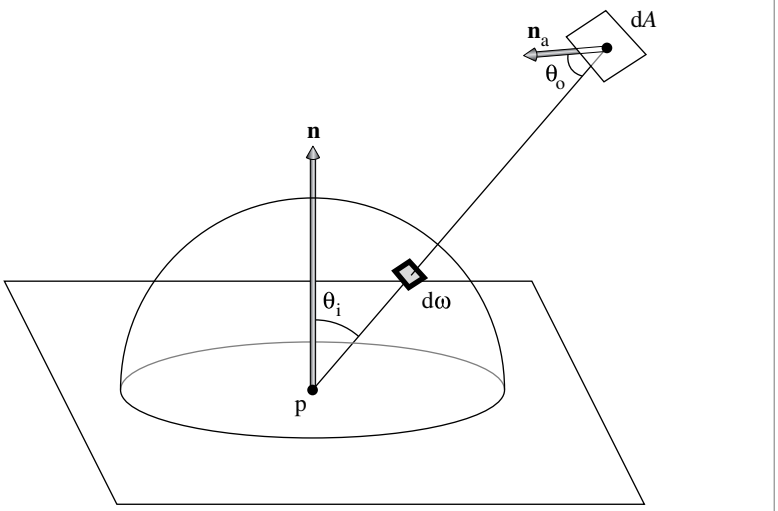

Figure 5.17: To compute irradiance at a point p from a quadrilateral source, it's easier to integrate over the surface area of the source than to integrate over the irregular set of directions that it subtends. The relationship between solid angles and areas given by Equation (5.5) lets us go back and forth between the two approaches.

Reflection from translucent surfaces is more complex; many materials ranging from skin and leaves to wax and dense liquids exhibit *subsurface light transport*, where light that enters the surface at one point exits it some distance away. (Consider, for example, how shining a flashlight in one's mouth makes one's cheeks light up, as light that enters the inside of the cheeks passes through the skin and exits the face.)

There are two abstractions for describing these mechanisms for light reflection: the BRDF and the BSSRDF, described in Sections 5.6.1 and 5.6.2, respectively. The BRDF describes surface reflection at a point neglecting the effect of subsurface light transport; for materials where this transport mechanism doesn't have a significant effect, this simplification introduces little error and makes the implementation of rendering algorithms much more efficient. The BSSRDF generalizes the BRDF and describes the more general setting of light reflection from translucent materials.

5.6.1 THE BRDF

The *bidirectional reflectance distribution function* (BRDF) gives a formalism for describing reflection from a surface. Consider the setting in Figure 5.18: we'd like to know how much radiance is leaving the surface in the direction ω_o toward the viewer, $L_o(p, \omega_o)$, as a result of incident radiance along the direction ω_i, $L_i(p, \omega_i)$.

If the direction ω_i is considered as a differential cone of directions, the differential irradiance at p is

$$dE(p, \omega_i) = L_i(p, \omega_i) \cos\theta_i \, d\omega_i. \quad [5.6]$$

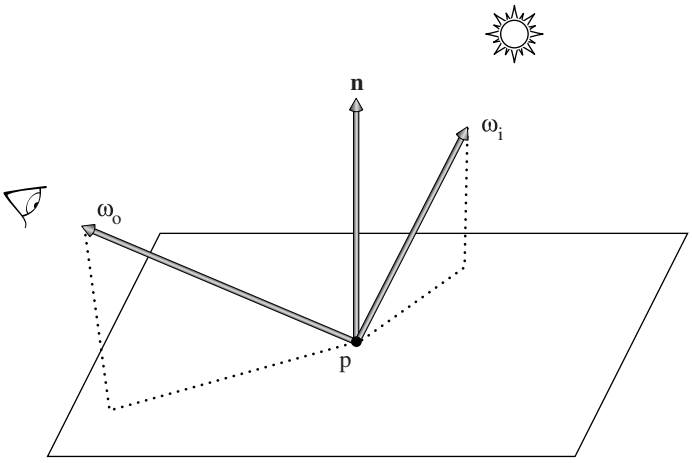

Figure 5.18: The BRDF. The bidirectional reflectance distribution function is a four-dimensional function over pairs of directions ω_i and ω_o that describes how much incident light along ω_i is scattered from the surface in the direction ω_o.

A differential amount of radiance will be reflected in the direction ω_o due to this irradiance. Because of the linearity assumption from geometric optics, the reflected differential radiance is proportional to the irradiance

$$dL_o(p, \omega_o) \propto dE(p, \omega_i).$$

The constant of proportionality defines the surface's BRDF for the particular pair of directions ω_i and ω_o:

$$f_r(p, \omega_o, \omega_i) = \frac{dL_o(p, \omega_o)}{dE(p, \omega_i)} = \frac{dL_o(p, \omega_o)}{L_i(p, \omega_i) \cos\theta_i \, d\omega_i}. \quad [5.7]$$

Physically based BRDFs have two important qualities:

1. *Reciprocity:* For all pairs of directions ω_i and ω_o, $f_r(p, \omega_i, \omega_o) = f_r(p, \omega_o, \omega_i)$.
2. *Energy conservation:* The total energy of light reflected is less than or equal to the energy of incident light. For all directions ω_o,

$$\int_{\mathcal{H}^2(\mathbf{n})} f_r(p, \omega_o, \omega') \cos\theta' \, d\omega' \leq 1.$$

The surface's *bidirectional transmittance distribution function* (BTDF), which describes the distribution of transmitted light, can be defined in a manner similar to that for the BRDF. The BTDF is generally denoted by $f_t(p, \omega_o, \omega_i)$, where ω_i and ω_o are in opposite hemispheres around p. Remarkably, the BTDF does not obey reciprocity; we will discuss this issue in detail in Section 8.2.

For convenience in equations, we will denote the BRDF and BTDF when considered together as $f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})$; we will call this the *bidirectional scattering distribution function* (BSDF). Chapter 8 is entirely devoted to describing BSDFs that are used in graphics.

Using the definition of the BSDF, we have

$$\mathrm{d}L_\mathrm{o}(\mathrm{p}, \omega_\mathrm{o}) = f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}.$$

Here an absolute value has been added to the $\cos\theta_\mathrm{i}$ term. This is done because surface normals in `pbrt` are not reoriented to lie on the same side of the surface as ω_i (many other rendering systems do this, although we find it more useful to leave them in their natural orientation as given by the `Shape`). Doing so makes it easier to consistently apply conventions like "the surface normal is assumed to point outside the surface" elsewhere in the system. Thus, applying the absolute value to $\cos\theta$ terms like these ensures that the desired quantity is actually calculated.

We can integrate this equation over the sphere of incident directions around p to compute the outgoing radiance in direction ω_o due to the incident illumination at p from all directions:

$$L_\mathrm{o}(\mathrm{p}, \omega_\mathrm{o}) = \int_{\mathbb{S}^2} f(\mathrm{p}, \omega_\mathrm{o}, \omega_\mathrm{i})\, L_\mathrm{i}(\mathrm{p}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}. \tag{5.8}$$

This is a fundamental equation in rendering; it describes how an incident distribution of light at a point is transformed into an outgoing distribution, based on the scattering properties of the surface. It is often called the *scattering equation* when the sphere $\mathbb{S}^2$ is the domain (as it is here), or the *reflection equation* when just the upper hemisphere $\mathcal{H}^2(\mathbf{n})$ is being integrated over. One of the key tasks of the integration routines in Chapter 15 is to evaluate this integral at points on surfaces in the scene.

5.6.2 THE BSSRDF

Many materials exhibit a significant amount of subsurface light transport: light that enters them at one point may exit quite far away. In general, subsurface light transport is one of the key mechanisms that gives objects like wax, candles, marble, skin, and many biological tissues their characteristic appearances.

The *bidirectional scattering-surface reflectance distribution function* (BSSRDF) is the formalism that describes these kinds of scattering processes. It is a distribution function $S(\mathrm{p_o}, \omega_\mathrm{o}, \mathrm{p_i}, \omega_\mathrm{i})$ that describes the ratio of exitant differential radiance at point $\mathrm{p_o}$ in direction ω_o to the differential irradiance at $\mathrm{p_i}$ from direction ω_i:

$$S(\mathrm{p_o}, \omega_\mathrm{o}, \mathrm{p_i}, \omega_\mathrm{i}) = \frac{\mathrm{d}L_\mathrm{o}(\mathrm{p_o}, \omega_\mathrm{o})}{\mathrm{d}E(\mathrm{p_i}, \omega_\mathrm{i})}.$$

The generalization of the scattering equation for the BSSRDF requires integration over surface area *and* incoming direction, turning the 2D scattering Equation (5.8) into a 4D integral. With two more dimensions to integrate over, it is substantially more complex to use in rendering algorithms.

$$L_\mathrm{o}(\mathrm{p_o}, \omega_\mathrm{o}) = \int_A \int_{\mathcal{H}^2(\mathbf{n})} S(\mathrm{p_o}, \omega_\mathrm{o}, \mathrm{p_i}, \omega_\mathrm{i})\, L_\mathrm{i}(\mathrm{p_i}, \omega_\mathrm{i})\, |\cos\theta_\mathrm{i}|\, \mathrm{d}\omega_\mathrm{i}\, \mathrm{d}A. \tag{5.9}$$

Shape 108

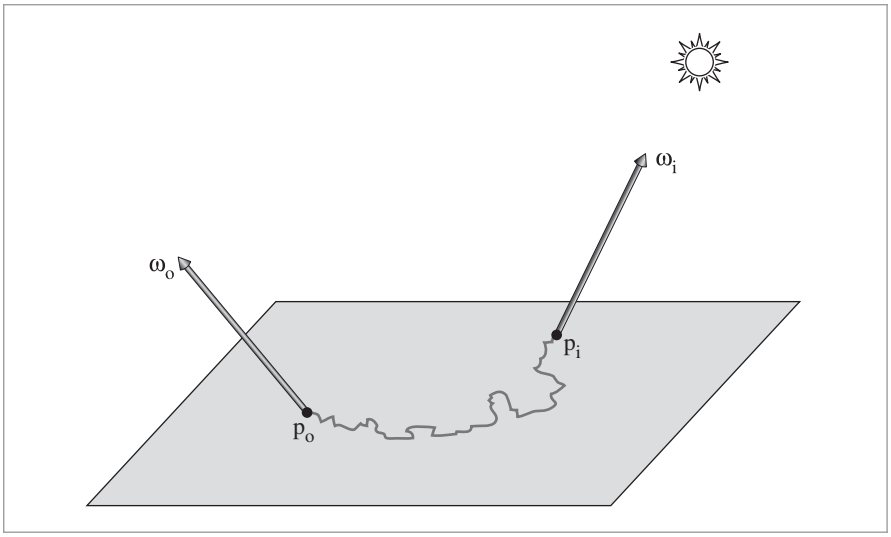

Figure 5.19: The bidirectional scattering-surface reflectance distribution function generalizes the BSDF to account for light that exits the surface at a point other than where it enters. It is often more difficult to evaluate than the BSDF, although subsurface light transport can make a substantial contribution to the appearance of many real-world objects.

However, as the distance between points p_i and p_o increases, the value of S generally diminishes. This fact can be a substantial help in implementations of subsurface scattering algorithms.

Light transport beneath a surface is described by the same principles as volume light transport in participating media and is described by the equation of transfer, which is introduced in Section 16.1. Subsurface scattering is thus based on the same effects as light scattering in clouds and smoke, just at a smaller scale. The `DipoleSubsurfaceIntegrator` in Section 16.5 implements an algorithm that applies an approximation to efficiently approximate this integral.

FURTHER READING

Meyer was one of the first researchers to closely investigate spectral representations in graphics (Meyer and Greenberg 1980; Meyer et al. 1986). Hall (1989) summarized the state of the art in spectral representations through 1989, and Glassner's *Principles of Digital Image Synthesis* (1995) covers the topic through the mid-1990s. Survey articles by Hall (1999), Johnson and Fairchild (1999), and Delvin et al. (2002) are good resources on this topic.

`DipoleSubsurfaceIntegrator` 887

Borges (1991) analyzed the error introduced from the tristimulus representation when used for spectral computation. A polynomial representation for spectra was proposed by Raso and Fournier (1991). Peercy (1993) developed a technique based on choosing

basis functions in a scene-dependent manner: by looking at the SPDs of the lights and reflecting objects in the scene, a small number of basis functions that could accurately represent the scene's SPDs were found using characteristic vector analysis. Rougeron and Péroche (1997) projected all SPDs in the scene onto a hierarchical basis (the Haar wavelets), showing that this adaptive representation can be used to stay within a desired error bound. Ward and Eydelberg-Vileshin (2002) developed a method for improving the spectral results from a regular RGB-only rendering system by carefully adjusting the color values provided to the system before rendering.

Another approach to spectral representation was investigated by Sun et al. (2001), who partitioned SPDs into a smooth base SPD and a set of spikes. Each part was represented differently, using basis functions that worked well for each of these parts of the distribution. Drew and Finlayson (2003) applied a "sharp" basis, which is adaptive but has the property that computing the product of two functions in the basis doesn't require a full matrix multiplication as many other basis representations do.

Evans and McCool (1999) introduced stratified wavelength clusters for representing SPDs: the idea is that each spectral computation uses a small fixed number of samples at representative wavelengths, chosen according to the spectral distribution of the light source. Subsequent computations use different wavelengths, such that individual computations are relatively efficient (being based on a small number of samples), but, in the aggregate over a large number of computations, the overall range of wavelengths is well covered. Related to this approach is the idea of computing the result for just a single wavelength in each computation and averaging the results together: this was the method used by Walter et al. (1997) and Morley et al. (2006), who showed that this approach converges reasonably quickly.

Glassner (1989b) has written an article on the underconstrained problem of converting RGB values (e.g., as selected by the user from a display) to an SPD. Smits (1999) developed an improved method that is the one we have implemented in Section 5.2.2.

McCluney's book on radiometry is an excellent introduction to the topic (McCluney 1994). Preisendorfer (1965) also covered radiometry in an accessible manner and delved into the relationship between radiometry and the physics of light. Nicodemus et al. (1977) carefully defined the BRDF, BSSRDF, and various quantities that can be derived from them. See Moon and Spencer (1936, 1948) and Gershun (1939) for classic early introductions to radiometry. Finally, Lambert's seminal early writings about photometry from the mid-18th century have been translated into English by DiLaura (Lambert 1760).

EXERCISES

❷ 5.1 Implement a new spectral basis functions representation in `pbrt`. Compare both image quality and rendering time to the `RGBSpectrum` and `SampledSpectrum` representations implemented in this chapter. Be sure to include tricky situations like fluorescent lighting.

5.2 Compute the irradiance at a point due to a unit-radius disk h units directly above its normal with constant outgoing radiance of 10 J/m^2 sr. Do the computation twice, once as an integral over solid angle and once as an integral over area. (Hint: If the results don't match at first, see Section 13.6.2.)

5.3 Similarly, compute the irradiance at a point due to a square quadrilateral with outgoing radiance of 10 J/m^2 sr that has sides of length 1 and is 1 unit directly above the point in the direction of its surface normal.