Physically Based Rendering读书笔记 part 4

计算机图形学 pbrt

by silver_gp
原文地址：http://www.pbr-book.org/3ed-2018/contents.html

5.6.2 The BSSRDF

5 Color And Radiometry

这个章节开始详细讲解辐射度。
首先引入Spectral Power Distribution(SPD)，一种波长分布函数，用于描述在每个波段的光线的多少。

5.1 Spectral Representation

真实世界物体的SPD非常复杂，下图就是两种。一个渲染器通过SPD函数去渲染，就需要做到足够有效。

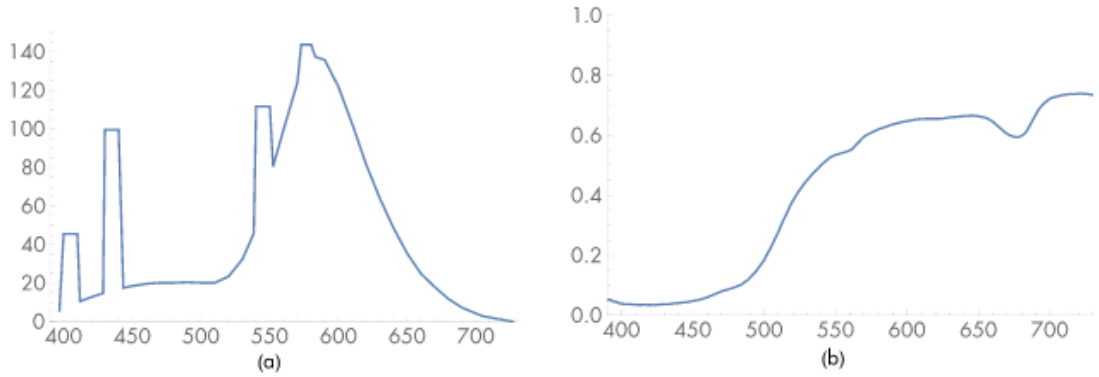


Figure 5.1: (a) Spectral power distributions of a fluorescent light and (b) the reflectance of lemon skin. Wavelengths around 400 nm are bluish colors, greens and yellows are in the middle range of wavelengths, and reds have wavelengths around 700 nm. The fluorescent light's SPD is even spikier than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

一个比较通用的方法去研究这个课题，就是寻找比较好的**基函数**，其核心思想是将无限维空间的可能的SPD函数映射到一系列低维空间的参数$c_i \in R$。例如，一个平凡基函数是一个常量函数$B(\lambda) = 1$。任意SPD函数可以通过一个参数c，用这个基函数表示，其中c是其平均值，因此其近似表示为$cB(\lambda) = c$。这明显是一个糟糕的近似，因为大多数SPD都比这个复杂得多。
有很多种基函数，他们都有对任意SPD函数转化为参数的操作的复杂度的权衡。在这个章节中，将引入两种在pbrt广泛使用的SPD表示方法：RGBSpectrum，将SPD函数表示为rgb参数。SampledSpectrum，将SPD函数表示为一堆在波段内的采样点。

5.1.1 The Spectrum Type

两种表示法都是基于保存固定数量的SPD采样点，因此首先要定义CoefficientSpectrum类，它通过一系列采样点来表示光谱，其中nSpectrumSamples是采样点数量，RGBSpectrum和SampledSpectrum是它的两个派生类。

```
<<Spectrum Declarations>>=
template <int nSpectrumSamples> class CoefficientSpectrum
 {
public:
    <<CoefficientSpectrum Public Methods>>
    <<CoefficientSpectrum Public Data>>
protected:
    <<CoefficientSpectrum Protected Data>>
};

<<CoefficientSpectrum Public Methods>>=
CoefficientSpectrum(Float v = 0.f) {
    for (int i = 0; i < nSpectrumSamples; ++i)
        c[i] = v;
}

<<CoefficientSpectrum Protected Data>>=
Float c[nSpectrumSamples];
```

这个类支持一系列数学操作

```
///简单相加
<<CoefficientSpectrum Public Methods>>+=
CoefficientSpectrum &operator+=(const CoefficientSpectrum
 &s2) {
    for (int i = 0; i < nSpectrumSamples; ++i)
        c[i] += s2.c[i];
    return *this;
}
<<CoefficientSpectrum Public Methods>>+=
CoefficientSpectrum operator+(const CoefficientSpectrum &s
2) const {
    CoefficientSpectrum ret = *this;
    for (int i = 0; i < nSpectrumSamples; ++i)
        ret.c[i] += s2.c[i];
    return ret;
}
```

剩下的减法、乘法、取反都是类似的操作，另外判断相等以及不等的方法也类似。
下面这个方法表示SPD的值处处为0，例如表面没有反射，光线传输流程可以避免生成

反射光线。

```
<<CoefficientSpectrum Public Methods>>+=
bool IsBlack() const {
    for (int i = 0; i < nSpectrumSamples; ++i)
        if (c[i] != 0.) return false;
    return true;
}
```

另外Spectrum的实现还需要提供一系列更复杂的方法，例如提供平方根函数，或者给定能量下，给出特定函数。

```
///开方函数
<<CoefficientSpectrum Public Methods>>+=
friend CoefficientSpectrum Sqrt(const CoefficientSpectrum
 &s) {
    CoefficientSpectrum ret;
    for (int i = 0; i < nSpectrumSamples; ++i)
        ret.c[i] = std::sqrt(s.c[i]);
    return ret;
}
///线性插值
<<Spectrum Inline Functions>>=
inline Spectrum Lerp(Float t, const Spectrum &s1, const Sp
ectrum &s2) {
    return (1 - t) * s1 + t * s2;
}
///截断函数
<<CoefficientSpectrum Public Methods>>+=
CoefficientSpectrum Clamp(Float low = 0, Float high = Infi
nity) const {
    CoefficientSpectrum ret;
    for (int i = 0; i < nSpectrumSamples; ++i)
        ret.c[i] = ::Clamp(c[i], low, high);
    return ret;
}
///参数中是否有NaN
<<CoefficientSpectrum Public Methods>>+=
bool HasNaNs() const {
    for (int i = 0; i < nSpectrumSamples; ++i)
```

```
        if (std::isnan(c[i])) return true;
    return false;
}
```

另外有些类例如TabulatedBSSRDF、HomogeneousMedium以及GridDensityMedium需要取得所有参数，因此要提供获取所有参数的方法。

```
<<CoefficientSpectrum Public Data>>=
static const int nSamples = nSpectrumSamples;
<<CoefficientSpectrum Public Methods>>+=
Float &operator[](int i) {
    return c[i];
}
```

需要指出的是，这些参数隐含的假设是，它线性放大基函数。但如果一个Spectrum的实现方式是通过c_i参数来放大一组高斯函数。

$$S(\lambda) = \sum_i^N c_{2i} e^{-c_{2i+1}}$$

这事情就麻烦，基函数与参数c有关系，因此使用这组参数的代码就要修改了。能正确处理这种情形依然有必要。

5.2 The SampledSpectrum Class

SampledSpectrum类是基于CoefficientSpectrum，在波段的开始和结束，以均匀分布的采样点，来表示SPD。当然这个波长是在400nm到700nm，即可见光范围。采样点数量为60，对于复杂的SPD也够了。这也意味着，第一个采样的波长范围是$[400, 405)$，第二个就是$[405, 410)$。

```
<<Spectrum Utility Declarations>>=
static const int sampledLambdaStart = 400;
static const int sampledLambdaEnd = 700;
static const int nSpectralSamples = 60;
<<Spectrum Declarations>>+=
class SampledSpectrum : public CoefficientSpectrum<nSpectr
alSamples> {
public:
    <<SampledSpectrum Public Methods>>
```

```
private:
    <<SampledSpectrum Private Data>>
};
```

因为是从CoefficientSpecgtrum派生的，所以基本的函数该有都有了，需要增加的函数就是从SPD转化为Spectrum的表示方法。
对于常量SPD初始化非常简单

```
<<SampledSpectrum Public Methods>>=
SampledSpectrum(Float v = 0.f) : CoefficientSpectrum(v) {
 }
```

除此之外，还需要接受一组(λ_i, v_i)进行初始化，其中i表示第i个采样点，v_i是采样点的值，λ_i是对应波长。通常情况下，传入的采样点都不会太均匀，以及肯定比需要的数量要少。
下面的代码接受一系列的sample，来定义分片线性函数(piecewise linear function)，对于每一个SPD sample，都会使用函数AverageSpectrumSamples()来计算当前采样点对应的波段区间内的平均值。

```
///接受一系列sample，来定义分片线性函数
<<SampledSpectrum Public Methods>>+=
static SampledSpectrum FromSampled(const Float *lambda,
                                   const Float *v, int n)
 {
    <<Sort samples if unordered, use sorted for returned s
pectrum>>
    SampledSpectrum r;
    for (int i = 0; i < nSpectralSamples; ++i) {
        <<Compute average value of given SPD over th sampl
e's range>>
    }
    return r;
}
```

AverageSpectrumSamples()函数首先需要采样点是根据波长排序的，SpectrumSamplesSorted()函数来检查这件事，如果发现没有按序，则通过函数SortSpectrumSamples()将其排序。

```
<<Sort samples if unordered, use sorted for returned spect
rum>>=
if (!SpectrumSamplesSorted(lambda, v, n)) {
    //额外分配空间来保存排序后的值
    std::vector<Float> slambda(&lambda[0], &lambda[n]);
    std::vector<Float> sv(&v[0], &v[n]);
    //如果没有排序，则要排序
    SortSpectrumSamples(&slambda[0], &sv[0], n);
    //这里是个递归调用，意思就是排序之后再重新调用自己
    return FromSampled(&slambda[0], &sv[0], n);
}
```

接下来就是计算某小段波段的平均值

```
<<Compute average value of given SPD over th sample's rang
e>>=
Float lambda0 = Lerp(Float(i) / Float(nSpectralSamples),
                     sampledLambdaStart, sampledLambdaEn
d);
Float lambda1 = Lerp(Float(i + 1) / Float(nSpectralSample
s),
                     sampledLambdaStart, sampledLambdaEn
d);
r.c[i] = AverageSpectrumSamples(lambda, v, n, lambda0, lam
bda1);
```

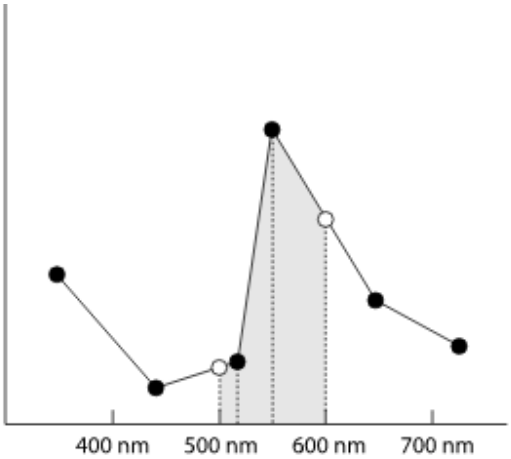


Figure 5.2: When resampling an irregularly defined SPD, we need to compute the average value of the piecewise linear function defined by the SPD samples. Here, we want to average the value from 500 nm to 600 nm—the shaded region under the plot. The `AverageSpectrumSamples()` function does this by computing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

如上图，2个采样点落在了500nm与6000nm之间，于是就采取平均的算法来记录这一区间的v。
下面来拆解这块代码

```
<<Spectrum Method Definitions>>=
Float AverageSpectrumSamples(const Float *lambda, const Fl
oat *vals,
        int n, Float lambdaStart, Float lambdaEnd) {
    <<Handle cases with out-of-bounds range or single samp
le only>>
    Float sum = 0;
    <<Add contributions of constant segments before/after
 samples>>
    <<Advance to first relevant wavelength segment>>
    <<Loop over wavelength sample segments and add contrib
utions>>
    return sum / (lambdaEnd - lambdaStart);
}
```

首先处理边际条件，假如波段完全在给过来的采样点范围外，这里假设spd对于超出范围的波段都是一个常量。

```
<<Handle cases with out-of-bounds range or single sample o
nly>>=
if (lambdaEnd   <= lambda[0])     return vals[0];
if (lambdaStart >= lambda[n - 1]) return vals[n - 1];
if (n == 1) return vals[0];
```

假如波段比采样点范围还要大

```
//实际上就是求超出部分的面积
<<Add contributions of constant segments before/after samp
les>>=
if (lambdaStart < lambda[0])
    sum += vals[0] * (lambda[0] - lambdaStart);
if (lambdaEnd > lambda[n-1])
    sum += vals[n - 1] * (lambdaEnd - lambda[n - 1]);
```

```
//接下来搜索用于计算的起始采样点
<<Advance to first relevant wavelength segment>>=
int i = 0;
while (lambdaStart > lambda[i + 1]) ++i;
```

```
<<Loop over wavelength sample segments and add contributio
ns>>=
在第i个和第i+1个val之间进行插值，插值权重是当前lambda
auto interp = [lambda, vals](Float w, int i) {
    return Lerp((w - lambda[i]) / (lambda[i + 1] - lambda
[i]),
                vals[i], vals[i + 1]);
};
for (; i+1 < n && lambdaEnd >= lambda[i]; ++i) {
    Float segLambdaStart = std::max(lambdaStart, lambda
[i]);
    Float segLambdaEnd =   std::min(lambdaEnd,   lambda[i
 + 1]);
    //算平均值
    sum += 0.5 * (interp(segLambdaStart, i) + in
    terp(segLambdaEnd, i)) *
        (segLambdaEnd - segLambdaStart);
}
```

5.2.1 XYZ Color

人眼能观测到的所有颜色都能表示为3个浮点数，三原色理论（tristimulus theory of color）认为所有可见的SPD都可以被准确表示为$(x_\lambda, y_\lambda, z_\lambda)$，给定一个SPD $S(\lambda)$，通过下面的公式可以计算出三个颜色分量：

$$x_\lambda = \int_\lambda S(\lambda)X(\lambda)d\lambda$$
$$y_\lambda = \int_\lambda S(\lambda)Y(\lambda)d\lambda$$
$$z_\lambda = \int_\lambda S(\lambda)Z(\lambda)d\lambda$$

其中$X(\lambda)$、$Y(\lambda)$、$Z(\lambda)$被称为**光谱匹配曲线（spectral matching curves）**，它是作为**基函数**。而$S(\lambda)$就是SPD，它在基函数$X(\lambda)$、$Y(\lambda)$、$Z(\lambda)$的映射下的向量，就是向量$(x_\lambda, y_\lambda, z_\lambda)$。
这些曲线是由Commission Internationale de l'Eclairage(CIE)这个机构经过一系列的试验得到的（我也不知道这个是什么机构）。总之如下图所示

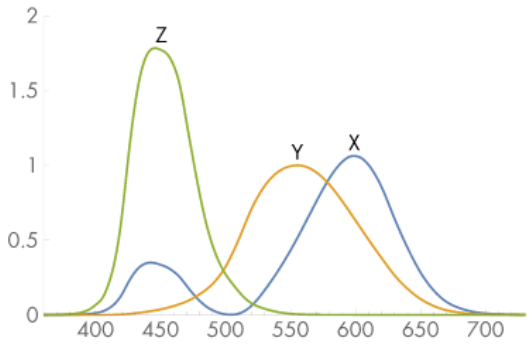


Figure 5.3: Computing the XYZ Values for an Arbitrary SPD. The SPD is multiplied by each of the three matching curves and integrated over their non-zero extent to compute the values x_λ, y_λ, and z_λ, using Equation (5.1).

这些曲线与人的视网膜对于三种颜色的光的敏感程度很类似。另外，非常不同的SPD也许有相似的$(x_\lambda, y_\lambda, z_\lambda)$。这类SPD虽然不同，但是对于人类而言并没什么分别，这一类SPD就叫做**条件等色(metamer)**。
大多数颜色空间尝试通过给人类的可见光建模，通过三原色理论，用3个参数。尽管XYZ这种表示法用于表示人类观察到的颜色很合适，但要进行光谱计算，这就不是一个很好的基函数。例如用XYZ表示柠檬表面或者荧光，直接用XYZ做乘法，就比用SPD光谱做计算得到的结果差的多。
pbrt提供标准的$X(\lambda)$、$Y(\lambda)$、$Z(\lambda)$曲线，以1nm建个来采样，范围在360nm到830nm。

```
<<Spectral Data Declarations>>=
static const int nCIESamples = 471;
///每个采样点的val
extern const Float CIE_X[nCIESamples];
extern const Float CIE_Y[nCIESamples];
```

```
extern const Float CIE_Z[nCIESamples];
///每个采样点的波长
extern const Float CIE_lambda[nCIESamples];
```

在调用AverageSpectrumSamples()函数时可以把上面的参数很容易地传入。SampleSpectrum函数，就是用上述的采样点，去计算XYZ匹配曲线(XYZ matching curve)。

```
//这个是matching curve，是根据人的眼睛来的，常量
<<SampledSpectrum Private Data>>=
static SampledSpectrum X, Y, Z;
```

```
///初始化，在pbrtInit()中调用
<<SampledSpectrum Public Methods>>+=
static void Init() {
    <<Compute XYZ matching functions for SampledSpectrum>>

    <<Compute RGB to spectrum functions for SampledSpectru
m>>
}
<<General pbrt Initialization>>=
SampledSpectrum::Init();
```

```
//这里就是计算每个波段的积分，算出每个波段的强度
<<Compute XYZ matching functions for SampledSpectrum>>=
for (int i = 0; i < nSpectralSamples; ++i) {
    Float wl0 = Lerp(Float(i) / Float(nSpectralSamples),
                     sampledLambdaStart, sampledLambdaEn
d);
    Float wl1 = Lerp(Float(i + 1) / Float(nSpectralSample
s),
                     sampledLambdaStart, sampledLambdaEn
d);
    X.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_X, nCI
ESamples,
                                    wl0, wl1);
    Y.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_Y, nCI
ESamples,
```

```
                                          wl0, wl1);
    Z.c[i] = AverageSpectrumSamples(CIE_lambda, CIE_Z, nCI
ESamples,
                                          wl0, wl1);
}
```

所有Spectrum的实现都会提供SPD到$(x_\lambda, y_\lambda, z_\lambda)$的转换函数，由波函数转化为XYZ分量。
转换函数由黎曼和公式得：

$$x_\lambda = \frac{\lambda_{end} - \lambda_{start}}{N} \sum_{i=0}^{N-1} X_i c_i$$

```
<<SampledSpectrum Public Methods>>+=
void ToXYZ(Float xyz[3]) const {
    xyz[0] = xyz[1] = xyz[2] = 0.f;
    //这里c[i]当前颜色的spd，而X,Y,Z是matching curve，常量
    //这里就是两个函数的乘法做积分的最后一步，实际上是把两个average
乘起来相加，最后乘以一个scale
    //这个就是黎曼和的实现，用这个方法，可以轻易地计算积分
    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];
    }
    //CIE_Y_intgral是一个常量，值为106.856895，在勘误中已经明确
说明，这个系数是不需要的
    //为了明确起见，这里代码中给注释掉
    Float scale = Float(sampledLambdaEnd - sampledLambdaSt
art) /
                  Float(/*CIE_Y_integral **/ nSpectralSamp
les);
    xyz[0] *= scale;
    xyz[1] *= scale;
    xyz[2] *= scale;
}
```

总结一下目前的内容，SPD函数全称叫做Spectral Power Distribution(SPD)，波长能量分布函数，是各个波长的能量分布，不同颜色不一样。而$X(\lambda)$, $Y(\lambda)$, $Z(\lambda)$叫做波长匹配曲线(spectral matching curve)，这个曲线和人的眼睛的敏感程度近似，是**常量**。这两个

函数搭配描述三个分量的强度，转换为XYZ颜色，实际上是做了个积分，这点与傅里叶变换是一样的操作，而离散版本就是卷积计算。
接下来需要一个单独计算Y值的函数，因为它与亮度(Luminance)相关

```
<<SampledSpectrum Public Methods>>+=
Float y() const {
    Float yy = 0.f;
    for (int i = 0; i < nSpectralSamples; ++i)
        yy += Y.c[i] * c[i];

    return yy * Float(sampledLambdaEnd - sampledLambdaStar
t) /
        Float(nSpectralSamples);
}
```

5.2.2 RGB Color

当显示一个RGB颜色时，光谱被基本的三个光谱响应曲线(spectral response curves)所决定，分别是红色、绿色、蓝色。下图展示了LEE和LCD显示器的分布图。

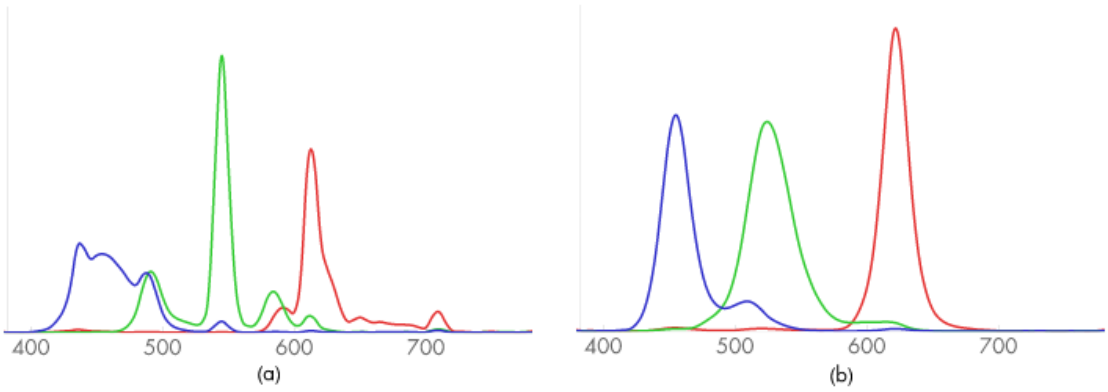


Figure 5.4: Red, Green, and Blue Emission Curves for an LCD Display and an LED Display. The first plot shows the curves for an LCD display, and the second shows them for an LED. These two displays have quite different emission profiles. *(Data courtesy of X-Rite, Inc.)*

下图展示了颜色$(0.6, 0.3, 0.2)$在不同显示器上显示的时候，对应的SPD。很显然的，完全不同。因此我们能得到一个简单的结论，就是使用RGB去描述一个颜色，只对显示器有充分了解的情况下，才有意义。

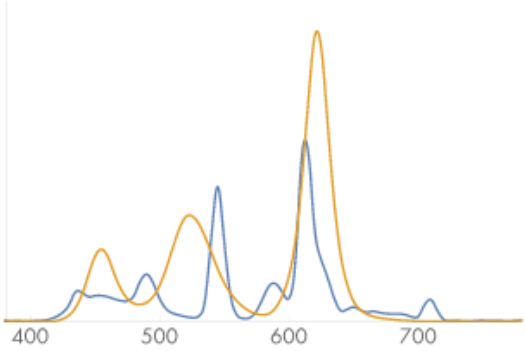


Figure 5.5: SPDs from Displaying the RGB Color $(0.6, 0.3, 0.2)$ on LED and LCD Displays. The resulting emitted SPDs are remarkably different, even given the same RGB values, due to the different emission curves illustrated in Figure 5.4.

现在我们有一个SPD$(x_\lambda, y_\lambda, z_\lambda)$，我们要把它转换为RGB系数。给定了一个SPD，给定**光谱响应曲线**$R(\lambda), G(\lambda), B(\lambda)$，它对应的是特定的显示器，RGB参数可以像上面XYZ参数一样，三条光谱响应曲线与SPD进行积分。

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

而$R(\lambda)X(\lambda)$的积分可以提前算出来，也就是说光谱匹配曲线和光谱响应曲线是已知常量，可以预计算。

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

pbrt给出了给高分辨率电视机用的RGB光谱的转换函数

```
<<Spectrum Utility Declarations>>+=
inline void XYZToRGB(const Float xyz[3], Float rgb[3]) {
    rgb[0] =  3.240479f*xyz[0] - 1.537150f*xyz[1] - 0.4985
35f*xyz[2];
    rgb[1] = -0.969256f*xyz[0] + 1.875991f*xyz[1] + 0.0415
56f*xyz[2];
    rgb[2] =  0.055648f*xyz[0] - 0.204043f*xyz[1] + 1.0573
11f*xyz[2];
```

```
}
```

其逆矩阵就是由RGB到XYZ的转换

```
<<Spectrum Utility Declarations>>+=
inline void RGBToXYZ(const Float rgb[3], Float xyz[3]) {
    xyz[0] = 0.412453f*rgb[0] + 0.357580f*rgb[1] + 0.18042
3f*rgb[2];
    xyz[1] = 0.212671f*rgb[0] + 0.715160f*rgb[1] + 0.07216
9f*rgb[2];
    xyz[2] = 0.019334f*rgb[0] + 0.119193f*rgb[1] + 0.95022
7f*rgb[2];
}
```

有了以上函数，SampledSpectrum类就可以实现：

```
<<SampledSpectrum Public Methods>>+=
void ToRGB(Float rgb[3]) const {
    Float xyz[3];
    ToXYZ(xyz);
    XYZToRGB(xyz, rgb);
}
```

还有对应的ToRGBSpectrum()函数

```
<<SampledSpectrum Public Methods>>+=
RGBSpectrum ToRGBSpectrum() const;
```

如果把事情反过来，想由RGB或者XYZ值推导出SPD就比较难搞了，因为之前说过，有无穷多的SPD都能映射到同一组XYZ，那么反过来的话，也就有无穷多组SPD需要选择。下面有一组这个转换函数的需要满足的必要条件：

- 如果所有RGB的参数值一样，那么最终的SPD是一个常量。
- 一般而言，SPD应该是光滑的。大多数真实世界物体都有一个相对光滑的光谱。

以光滑度作为目标之一的话，直接用显示器的$R(\lambda),G(\lambda),B(\lambda)$来做某种加权平均就不是个很好的主意，因为这几个函数样式本身就不那么光滑，尽管真这么做得到的SPD肯定是给定RGB的条件等色(metamer)，但是这也不是对真实物体SPD的准确表述。
pbrt的实现方式是Smits(1999)提出来的，这个方法是基于观察的，一个很好的开始，是

独立计算对于RGB每个分量对应的SPD，这个SPD是光滑的，然后根据给定的RGB系数，计算这些SPD的加权平均，最后再转换回RGB，这个转换回来的RGB与原始RGB非常接近。
除此之外，Smits还发现了两个额外的改进之处，一个是表示一个常量光谱(就是上面说的RGB三个亮度相同的光谱)，用RGB三个SPD直接相加，得到的光谱未必是常量光谱，不如直接用用对应的常量SPD来表达常量光谱。另外混合颜色，例如黄色（是红色和绿色的混合），是两种主要颜色的混合，但在SPD表示上，表示为它自己的平滑SPD更好一些，而不是不是两个相关颜色的SPD的和。(这块理解起来真绕)
以下SPD满足了以上的要求，它们的采样的波长保存在RGB2SpectLambda[]中，这里是专门用于反射的数据

```
<<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[nRGB2SpectSample
s];
extern const Float RGBRefl2SpectYellow[nRGB2SpectSamples];
extern const Float RGBRefl2SpectRed[nRGB2SpectSamples];
extern const Float RGBRefl2SpectGreen[nRGB2SpectSamples];
extern const Float RGBRefl2SpectBlue[nRGB2SpectSamples];
```

如果给定一个RGB颜色，用于表示光源的亮度，计算转换数据需要用代表光源的SPD来定义白色，而不是上面用于计算反射的数据。RGBIllum2Spect是专门用于描述光源的SPD。**这里我个人理解，虽然都是白色，但是光源应该整体更亮一些吧？反正这里的数据是各个波长采样点的强度，所以强度可以想多大就多大。转换为RGB是显示层面的事。**
在SampledSpectrum::Init()函数中初始化下面的SampledSpectrum

```
<<SampledSpectrum Private Data>>+=
static SampledSpectrum rgbRefl2SpectWhite, rgbRefl2SpectCy
an;
static SampledSpectrum rgbRefl2SpectMagenta, rgbRefl2Spect
Yellow;
static SampledSpectrum rgbRefl2SpectRed, rgbRefl2SpectGree
n;
static SampledSpectrum rgbRefl2SpectBlue;
<<SampledSpectrum Private Data>>+=
```

```
static SampledSpectrum rgbIllum2SpectWhite, rgbIllum2Spect
Cyan;
static SampledSpectrum rgbIllum2SpectMagenta, rgbIllum2Spe
ctYellow;
static SampledSpectrum rgbIllum2SpectRed, rgbIllum2SpectGr
een;
static SampledSpectrum rgbIllum2SpectBlue;
```

SampledSpectrum::FromRGB()将RGB值转换为SPD数据，相关的rgbIllum2Spect或者rgbRefl2Spect用于此转换

```
<<Spectrum Utility Declarations>>+=
enum class SpectrumType { Reflectance, Illuminant };
<<Spectrum Method Definitions>>+=
SampledSpectrum SampledSpectrum::FromRGB(const Float
rgb[3],
                                         SpectrumType typ
e) {
    SampledSpectrum r;
    if (type == SpectrumType::Reflectance) {
        <<Convert reflectance spectrum to RGB>>
    } else {
        <<Convert illuminant spectrum to RGB>>
    }
    return r.Clamp();
}
```

下面展示反射颜色的转换过程，光源颜色的转换过程与其类似。

```
//首先确定rgb三原色中，哪个值最小
<<Convert reflectance spectrum to RGB>>=
if (rgb[0] <= rgb[1] && rgb[0] <= rgb[2]) {
    <<Compute reflectance SampledSpectrum with rgb[0] as m
inimum>>
} else if (rgb[1] <= rgb[0] && rgb[1] <= rgb[2]) {
    <<Compute reflectance SampledSpectrum with rgb[1] as m
inimum>>
} else {
    <<Compute reflectance SampledSpectrum with rgb[2] as m
inimum>>
```

```
}
```

接下来，如果红色是最小的，我们知道绿色和蓝色比红色亮度更大，因此首先红色乘以白色的SPD，剩下要处理的是$(0, g-r, b-r)$，接下来决定g和b谁更小，更小的那个在乘以蓝绿色的spd，加入到最终结果上，剩下的有可能是$(0, g-b, 0)$或者$(0, 0, b-g)$，这个值乘以蓝色的spd，加入到最终结果上，算法结束。**实际上这个算法怎么来的我也不知道，上面那个Smits发明的。**

```
<<Compute reflectance SampledSpectrum with rgb[0] as minim
um>>=
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;
}
```

有了从rgb构造光谱数据，那么转换到XYZ也非常容易

```
<<SampledSpectrum Public Methods>>+=
static SampledSpectrum FromXYZ(const Float xyz[3],
        SpectrumType type = SpectrumType::Reflectance) {
    Float rgb[3];
    XYZToRGB(xyz, rgb);
    return FromRGB(rgb, type);
}
```

最终，是构造函数

```
<<Spectrum Method Definitions>>+=
SampledSpectrum::SampledSpectrum(const RGBSpectrum &r, Spe
ctrumType t) {
    Float rgb[3];
    r.ToRGB(rgb);
    *this = SampledSpectrum::FromRGB(rgb, t);
}
```

小结

这个章节讲的就是如何构造SPD，SPD的横轴是光线的波长，纵轴是强度，它描述的是在各个波长上的能量强度，记为$S(\lambda)$，这个函数定义了光。
接下来介绍的XYZ颜色，是基于光谱匹配曲线（spectral matching curves），记为$X(\lambda)$、$Y(\lambda)$、$Z(\lambda)$，这个曲线是根据人眼的构造来的，用来表示人眼对哪个波长的强度更敏感。有了这两个函数，就能得到XYZ颜色。它的表达式为：

$$
\begin{aligned}
x_\lambda &= \int_\lambda S(\lambda)X(\lambda)d\lambda \\
y_\lambda &= \int_\lambda S(\lambda)Y(\lambda)d\lambda \\
z_\lambda &= \int_\lambda S(\lambda)Z(\lambda)d\lambda
\end{aligned}
$$

其中$X(\lambda)$、$Y(\lambda)$、$Z(\lambda)$被称为**光谱匹配曲线（spectral matching curves）**，它是作为**基函数**。而$S(\lambda)$就是SPD，它在基函数$X(\lambda)$、$Y(\lambda)$、$Z(\lambda)$的映射下的向量，就是向量$(x_\lambda, y_\lambda, z_\lambda)$。
下一个介绍的就是RGB颜色，由此引入了**光谱响应曲线**$R(\lambda),G(\lambda),B(\lambda)$，光谱响应曲线与特定显示器相关。它实际上与XYZ做法没有本质的不同，计算上依然是将$S(\lambda)$映射到光谱响应曲线$R(\lambda),G(\lambda),B(\lambda)$上，得到的向量就是三原色的rgb。

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

很明显的，上面的公式实际上再次引入了XYZ颜色，意图成为连接XYZ和RGB颜色的桥梁。
pbrt中是用SPD来表示颜色的，因此区分了**反射颜色(reflectance)**和**亮度颜色(illumination)**。同样是白色，它们的SPD也是不同的，毕竟SPD只表示能量。
接下来介绍了SPD，XYZ，RGB的相互转换。

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

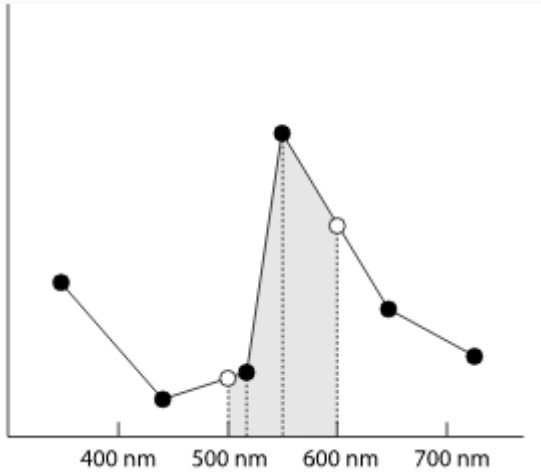
上式就是xyz转换到rgb，值得注意的是，矩阵内的每个元素都是积分项，但是每个积分函数都是常量，与SPD本身是无关的。
最后，积分的计算，用的是**黎曼和**。
对于计算x分量，由积分形式：

$$x_\lambda = \int_\lambda S(\lambda)X(\lambda)d\lambda$$

转化为求和形式

$$x_\lambda = \frac{\lambda_{end} - \lambda_{start}}{N} \sum_{i=0}^{N-1} X_i c_i$$

实际上就是分片求面积。



5.3 RGBSpectrum Implementation

RGBSpectrum代表SPD用rgb三种颜色的加权平均来构成。但不同的显示器构造不同，它们显示同样一个RGB，激发出的SPD一定是不一样的。但是RGB无论计算还是表示颜色，都极其简单，而且已经被广泛使用。RGBSpectrum从CoefficientSpectrum派生得来，系数个数为3个。因此CoefficientSpectrum里的所有数学操作都会继承。

```
<<Spectrum Declarations>>+=
class RGBSpectrum : public CoefficientSpectrum<3> {
public:
    <<RGBSpectrum Public Methods>>
};
<<RGBSpectrum Public Methods>>=
RGBSpectrum(Float v = 0.f) : CoefficientSpectrum<3>(v) { }
RGBSpectrum(const CoefficientSpectrum<3> &v)
    : CoefficientSpectrum<3>(v) { }
```

RGBSpectrum需要提供函数，以便从XYZ到RGB互相转换。其中FromRGB函数的第二个参数SpectrumType并没什么用，所做的操作也就是简单的赋值而已。

```
<<RGBSpectrum Public Methods>>+=
static RGBSpectrum FromRGB(const Float rgb[3],
        SpectrumType type = SpectrumType::Reflectance) {
    RGBSpectrum s;
    s.c[0] = rgb[0];
    s.c[1] = rgb[1];
    s.c[2] = rgb[2];
    return s;
}
```

然后输出到RGB也直截了当

```
<<RGBSpectrum Public Methods>>+=
void ToRGB(Float *rgb) const {
    rgb[0] = c[0];
    rgb[1] = c[1];
    rgb[2] = c[2];
}
//类型转换自己转换到自己
<<RGBSpectrum Public Methods>>+=
const RGBSpectrum &ToRGBSpectrum() const {
    return *this;
}
```

至于RGBSpectrum::ToXYZ()和RGBSpectrum::FromXYZ()，以及RGBSpectrum::y()用到了之前介绍的RGBToXYZ()和XYZToRGB()函数。
接下来就是从任意的采样SPD转换为RGB spectrum。以1-nm的步长分片解析采样光谱，使用函数InterpolateSpectrumSamples()函数，像之前的做法一样做卷积，计算黎曼和。

```
<<RGBSpectrum Public Methods>>+=
static RGBSpectrum FromSampled(const Float *lambda, const
Float *v,
                               int n) {
    <<Sort samples if unordered, use sorted for returned s
pectrum>>
```

```
    Float xyz[3] = { 0, 0, 0 };
    for (int i = 0; i < nCIESamples; ++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];
    }
    //前面的代码已经把CIE_Y_integral给注掉了，这里也许是没改过来
    Float scale = Float(CIE_lambda[nCIESamples-1] - CIE_la
mbda[0]) /
        Float(CIE_Y_integral * nCIESamples);
    xyz[0] *= scale;
    xyz[1] *= scale;
    xyz[2] *= scale;
    return FromXYZ(xyz);
}
```

最后，InterpolateSpectrumSamples()函数输入的可能是不均匀采样波长的SPD值(λ_i, v_i)，输入需要的λ，根据这个值来找左右两个采样点，然后线性插值返回。

```
<<Spectrum Method Definitions>>+=
Float InterpolateSpectrumSamples(const Float *lambda, cons
t Float *vals,
                                 int n, Float l) {
    if (l <= lambda[0])     return vals[0];
    if (l >= lambda[n - 1]) return vals[n - 1];
    int offset = FindInterval(n,
        [&](int index) { return lambda[index] <= l; });
    Float t = (l - lambda[offset]) / (lambda[offset+1] - l
ambda[offset]);
    return Lerp(t, vals[offset], vals[offset + 1]);
}
```

5.4 Radiometry

辐射度法，提供了一系列工具来描述光线传播和反射，它提供了渲染方程的基础推导。辐射传输(Radiative transfer)是对于辐射能量传输的现象研究。它基于辐射度测量原则(radiometric principles)，运行于几何光学这一层面，而宏观的光的属性已经足够满足描述光如何与物体交互，而用不着在波长这个层面。把波动光学模型(wave optics models)引入进来也很平常，不过还是要把这些内容用辐射度传输基础抽象的语言来描述出来。(Presiendorfer(1965))将辐射度传输理论与Maxxwell经典电磁理论联系到了一起，他的框架同时展示了它们的等价，使得可以很容易地从一套理论的结论应用到另一套理论。因此，描述光与物体的交互，这个物体的尺寸甚至与光的波长等长，因此这个模型还有散射和干涉等现象。甚至在更微观的领域，量子机制可以来描述光与原子的交互。但是我们这里并不需要管这事。
在pbrt中，我们假设几何光学已经足够描述光以及光的散射。这里有一些基础的假设：

- 线性(Linearity)：光学系统的同时两个输入的效果永远等价于单独两个输入效果的叠加。
- 能量守恒(Energy conservation)：光从表面或者一个参与介质(participating media)辐射出的能量，永远不会高于输入能量。
- 非极化(No polarization)：我们完全忽略电磁场的极化（然而我并不懂极化是个什么东西），因此唯一相关的光的属性，就是其波长能量的分布（频率）。
- 没有荧光(fluorescence)或者磷光(phosphorescence)：光在一个波长的行为与光在另一个波长的行为或者时间，完全无关。
- 稳态（steady state）：环境中的光假设已经达到均衡状态，就是说不会随着时间的推移而发生变化。磷光实际上违背了这个假设。

使用几何光学模型有一个大问题，就是衍射和干涉效果没法做。

5.4.1 Basic Quantities

有4个基础物理量(Quantities)：通量(flux), 辐射照度(irradiance/ radiance exitance), 辐射强度(intensify)和辐射亮度(radiance)。这些基础物理量都是基于波长的。

Energy

能量的单位是焦耳(J)，是光源发射出光子(photons)，每个光子有一个确定的波长，并且携带一份特定大小的能量。所有的辐射度学物理朗都是用不同的方式来测量光子。一个光子在一个波长λ上携带的能量为：

$$Q = \frac{hc}{\lambda}$$

其中，c是光速，大小为$299,472,458 m/s$，h是普朗克常数，
$h \approx 6.626 \times 10^{-34} m^2 kg/s$。

Flux

能量度量的是一段时间内做的功，在渲染当中，我们假设处于稳态，因此我们更关心在一瞬间的如何度量光。**辐射通量(Radiant Flux)**，也叫作**能量(Power)**，是在一个单位时间内，从一个表面或空间内一个区域通过的能量大小。

$$\Phi = \lim_{\Delta t \to 0} \frac{\Delta Q}{\Delta t} = \frac{dQ}{dt}$$

其单位是焦耳/秒(J/s)，或者瓦特(W)。
例如，给出光$Q = 200,000J$，这是在一个小时内发出的能量，如果在这个小时内每刻发出的能量都是相同的，

$$\Phi = 2000,000J/3600s \approx 55.6W$$

积分表示：

$$Q = \int_{t_0}^{t_1} \Phi(t)dt$$

另外光子实际上是离散的，但这个表达式是个积分形式，并不那么严谨，但对于我们研究的目标而言，这并不重要。

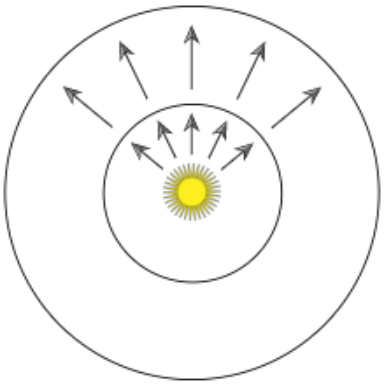


Figure 5.6: Radiant flux, Φ, measures energy passing through a surface or region of space. Here, flux from a point light source is measured at spheres that surround the light.

Irradiance and Radiant Exitance

描述flux的时候，需要一个面积，因此给定一个有限面积A，power的平均密度可以表示为$E = \Phi/A$。这个物理量叫做辐射照度(irradiance)，表示flux到达表面的密度，也表示辐射出照度(radiant exitance(M))，flux离开表面的密度。单位为W/m^2。
上图中，很明显的外圈的能量密度要低于内圈的能量密度，我们可以很容易得到公式：

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

这揭露了一个事实，就是点离光源越远，该点接收到的能量与距离乘平方关系衰减。
下面是严格定义：

$$E(p) = \lim_{\Delta A \to 0} \frac{\Delta \Phi(p)}{\Delta A} = \frac{d\Phi(p)}{dA}$$

相应的Φ也有其积分版本

$$\Phi = \int_A E(p) dA$$

这个等式可以帮助我们理解lambert's law：光到达平面的量，正比于光线方向和法线构成的角度的cos值。假设一束光照到一个表面上，这个光源的面积为A，光通量为Φ，如果这束光垂直打向这个表面，那么这个表面接收到光照的面积就是A_1。明显A_1与A相等。A_1内任一点的Irradiance为：

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

然而，如果光源呈一定角度，那么很明显的表面接收到的光的面积要**更大一些**。

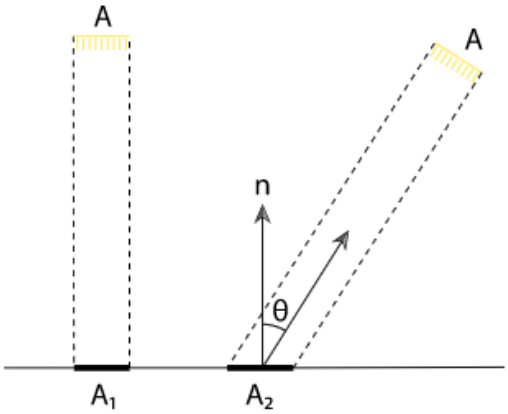


Figure 5.7: Lambert's Law. Irradiance arriving at a surface varies according to the cosine of the angle of incidence of illumination, since illumination is over a larger area at larger incident angles.

根据三角关系，我们能得到

$$\cos\theta A_2 = A$$

因此这时Irradiance变成了

$$E_2 = \frac{\Phi}{A_2} = \frac{\Phi}{\frac{A}{\cos\theta}} = \frac{\Phi \cos\theta}{A}$$

由此式可知，A是固定不变的时候，在θ角越变越大的情况下，E会逐步变为0，也就是说，**光源与平面平行的时候，A_2会变成无限大，此时密度irradiance就会成为0**，这就是$\cos\theta$项的由来。

Solid Angle and Intensify

首先引入立体角(Solid Angle)的概念。立体角是平面上的2D角的扩展版，它是在三维空间中的角度。

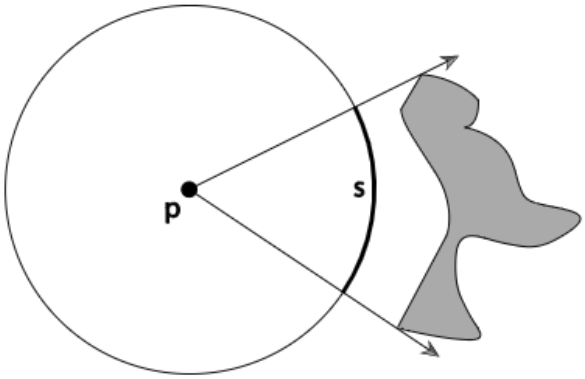


Figure 5.8: Planar Angle. The planar angle of an object as seen from a point p is equal to the angle it subtends as seen from p or, equivalently, as the length of the arc s on the unit sphere.

如上图，P点由一个单位圆包围，把深色的物体投影到这个圆上，得到的弧长s，就是由这个深色物体形成的就是**平面角**，平面角是**弧度制(radian)**的。
现在由2D扩展到3D空间。

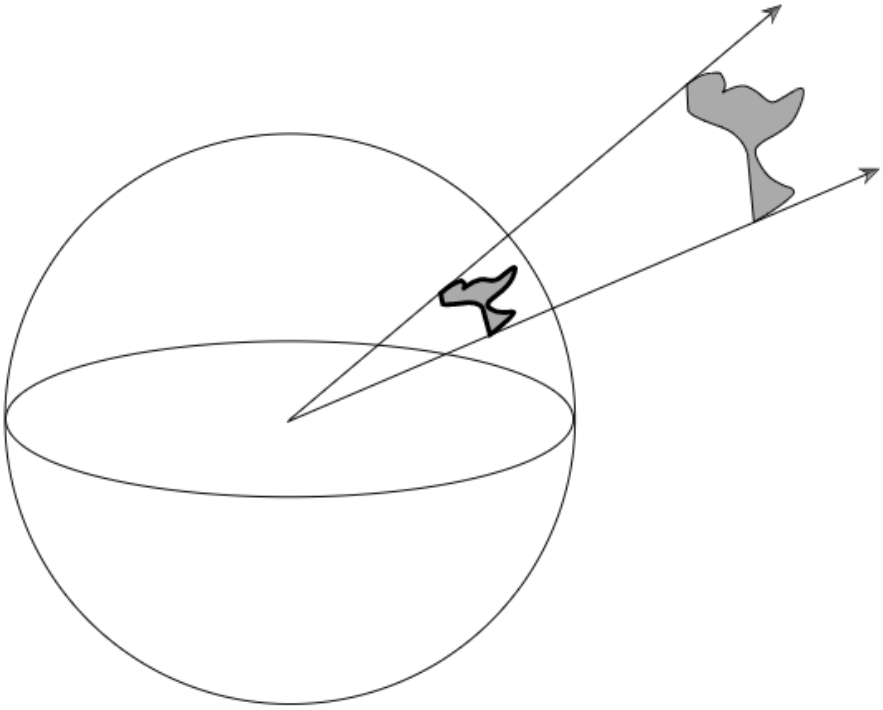


Figure 5.9: Solid Angle. The solid angle s subtended by an object c in three dimensions is computed by projecting c onto the unit sphere and measuring the area of its projection.

由深色物体投影到单位球上的面积，就叫做立体角，单位为**steradian(sr)**。很显然的，单位球的立体角为$4\pi sr$，半球的立体角为$2\pi sr$。
圆球上的点包围着球心p，因此用ω代表球心p到圆球上某一点的方向，很明显的，ω是一个归一化的向量，因为这个球体是单位球体。
假设有一个无穷小的光源正在发射光线，把这个光源放在圆球中心，我们就能计算能量的角密度，**辐射强度(Intensify)**，记为I。单位为W/sr。
对于整个球而言，

$$I = \frac{\Phi}{4\pi}$$

用极限的形式表示，

$$I = \lim_{\Delta\omega \to 0} \frac{\Delta\Phi}{\Delta\omega} = \frac{d\Phi}{d\omega}$$

同样的，积分形式为：

$$\Phi = \int_{\Omega} I(\omega) d\omega$$

Radiance

最后一个物理量，叫**辐射亮度(radiance)**，L。Irradiance或者说radiant exitance，给了我们一个在点p周围，能量微元/面积微元，这么样一个测量方式，但是还欠缺一点，就是并没有区分能量在不同方向上的分布。因此我们在E的基础上再次除以一个方位角的微元，就成为了Radiance：

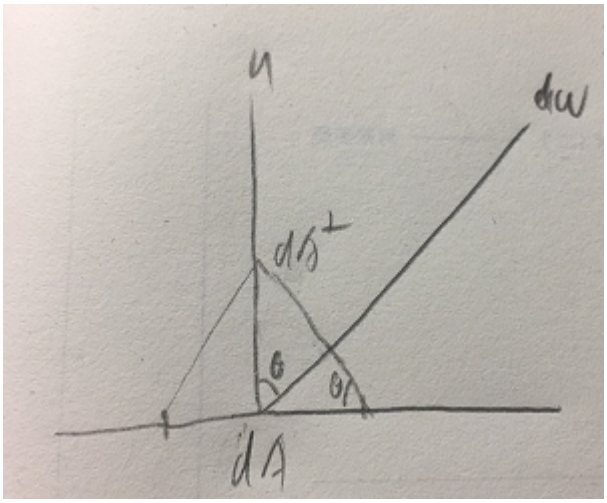
$$L(p, \omega) = \lim_{\Delta\omega \to 0} \frac{\Delta E_{\omega}(p)}{\Delta\omega} = \frac{dE_{\omega}(p)}{d\omega}$$

其中，E_{ω}指的是irradiance，但它对应的平面并不是点所在的那个原始平面dA，而是垂直于方向ω 平面，记为$dA^{\perp}$，也就是说，**L并不考虑光线来的方向**。从表达式上看，所以如果用$dA^{\perp}$这个符号的话，就可以不用引入$\cos\theta$了。
重新整理下公式，radiance代表通量的密度，在每单位面积，每单位立体角下：

$$L = \frac{d\Phi}{d\omega dA^{\perp}}$$

其中$dA^{\perp}$在与ω垂直的平面上，并且其面积是dA在此平面上的投影。因此$dA^{\perp}$与dA有很明显的几何关系：

$$dA^{\perp} = dA \cdot \cos\theta$$



因此这是对于入射光的一种有限度量，一方面ω实际上是一个锥形，其角度$d\omega$是个微

元，另一方面本地平面dA也是一个面积微元。

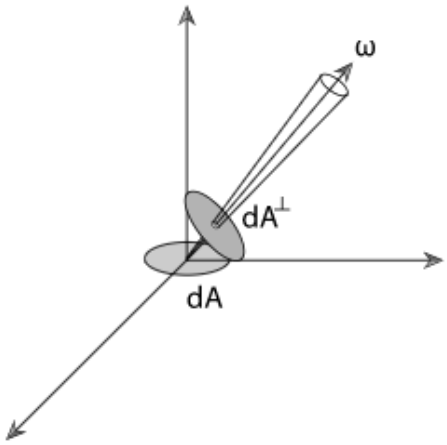


Figure 5.10: Radiance L is defined as flux per unit solid angle $\mathrm{d}\omega$ per unit projected area $\mathrm{d}A^{\perp}$.

接下来，Radiance是一个最常用的物理量，因为它囊括了前面讲的几个物理量，只需要通过积分，就可以把前面的物理量给算出来。另外一个非常好的特性是当沿着射线在空间中传播时，它是一个常量。

5.4.2 Incident and Exitant Radiance Functions

光线与表面相交，在穿过表面前后，radiance函数L并不相同，最极端的例子是镜子，radiance函数在表面上和下是完全无关的。在这里称之为**单侧限位(one-side limit)**。
因此L就变成了2个：表面上的radiance函数和表面下的radiance函数

$$L^{+}(p,\omega)=\lim_{t\to 0^{+}}L(p+tn_p,\omega)$$

$$L^{-}(p,\omega)=\lim_{t\to 0^{-}}L(p+tn_p,\omega)$$

其中，n_p是p点处的法线。**这个表达式意思是，当从正负两个方向趋近于0时，radiance的表达式是完全不同的。**
我们倾向于通过区分radiance到达特定点，以及radiance从特定点离开这两种情况，来解决表达式不同的二义性。
如下图所示，我们在实际计算过程中，区分了L_i（入射光的raidiance）和L_o（出射光的radiance），需要注意的是方向向量ω永远从p点出发，**指向外部离开表面的方向**，但有些作者对于L_i函数，使用的是$-\omega$，使得这个方向是指向点p的。

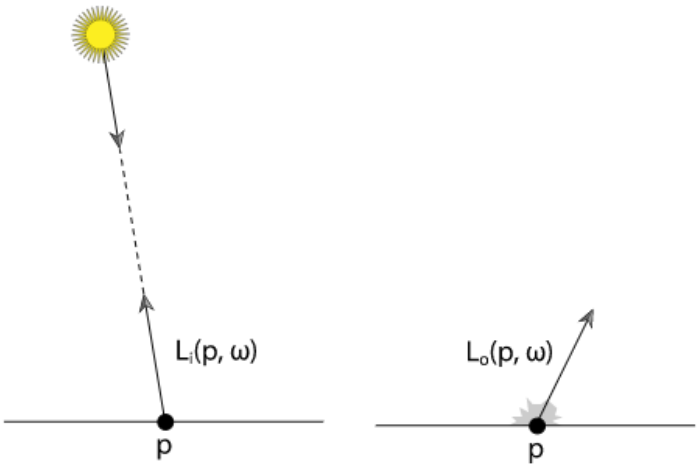
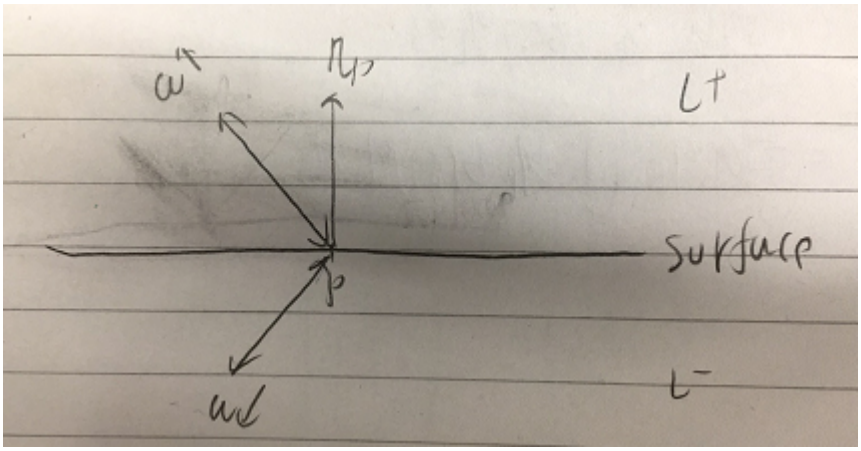


Figure 5.11: (a) The incident radiance function $L_i(p, \omega)$ describes the distribution of radiance arriving at a point as a function of position and direction. (b) The exitant radiance function $L_o(p, \omega)$ gives the distribution of radiance leaving the point. Note that for both functions, ω 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.

用更加直观的式子表示one-side limit，存在如下关系：

$$L_i(p,\omega) = \begin{cases} L^+(p,-\omega), & \omega \cdot n_p > 0 \\ L^-(p,-\omega), & \omega \cdot n_p < 0 \end{cases}$$

$$L_o(p,\omega) = \begin{cases} L^+(p,\omega), & \omega \cdot n_p > 0 \\ L^-(p,\omega), & \omega \cdot n_p < 0 \end{cases}$$



上式中，由于ω代表着离开表面的方向，此外，入射光L_i的两个表达式中，很明显的，$-\omega$是指的是指向p点的方向，就像上面说的一样。

以后我们要通过这个关系来决定在边界的radiance函数。

另外需要记住的是，在空间中的点，没有任何的平面，则L是连续的，即：

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

也就是说，L_i和L_o仅仅是方向上是反向而已。

5.4.3 Luminance and Photometry

上面讲的物理量都与**光度测定(photometry)**有关，所谓光度测定，指的是对于人的视觉系统而言，电磁辐射的可见性研究。每个光谱辐射度测量物理量都会与光谱响应函数(Spectral Response Curve)$V(\lambda)$做积分运算。$V(\lambda)$代表着人眼对于不同波长的光线的敏感程度。
亮度(Luminance)度量的是对于人类观察者而言，光谱能量有多强。例如有一个现象是，一个SPD在绿色波长有一定能量，在人眼看来，其亮度要高于在蓝色波长有相同能量的SPD。
将亮度记为Y，以及光谱辐射亮度(spectral radiance)记为$L(\lambda)$

$$Y = \int_{\lambda} L(\lambda)V(\lambda)d\lambda$$

亮度Luminance和光谱响应曲线$V(\lambda)$与颜色的XYZ表示紧密相关。其中$CIE - Y(\lambda)$曲线选择为与$V(\lambda)$曲线等比缩放，于是就有了：

$$Y = 683 \int_{\lambda} = L(\lambda)Y(\lambda)d\lambda$$

其单位为坎德拉/平方米(cd/m^2)。

5.5 Working with Radiometric Integrals

渲染中最频繁的操作，就是计算辐射度方程的积分。
首先回顾等式

$$L(p,\omega) = \lim_{\Delta\omega \to 0} \frac{\Delta E_{\omega}(p)}{\Delta\omega} = \frac{dE_{\omega}(p)}{d\omega} = \frac{d\Phi}{d\omega dA^{\perp}}$$

前面说过，radiance并不考虑光线的入射方向，因此分母用的是$dA^{\perp}$来正对着$d\omega$的对应方向。那么这里我们要计算E，而不是E_{ω}，而E是在乎入射方向的。

$$\because E_{\omega} = \frac{d\Phi}{dA^{\perp}}, E = \frac{d\Phi}{dA}$$

$$dA^{\perp} = \cos\theta \cdot dA$$

$$\begin{aligned} \therefore E &= \frac{d\Phi}{dA} \\ &= \frac{d\Phi}{dA \cdot \cos\theta} \cdot \cos\theta \\ &= \frac{d\Phi}{dA^{\perp}} \cdot \cos\theta \\ &= E_{\omega} \cdot \cos\theta \end{aligned}$$

另外由于

$$L(p,\omega)=\frac{dE_\omega(p)}{d\omega}$$

其积分形式为

$$E_\omega(p)=\int_\omega L(p,\omega)d\omega$$

因此最终有了如下表达式：

$$E(p,n)=\int_\omega L_i(p,\omega)\left|\cos\theta\right|d\omega$$

$L_i(p,\omega)$为入射radiance函数，$\cos\theta$是因为radiance函数中存在$dA^\perp$项，θ是方向ω与表面法线n的夹角，其中绝对值符号，代表光线强度不能是负数，而且，实际上也只处理半球$H^2(n)$。

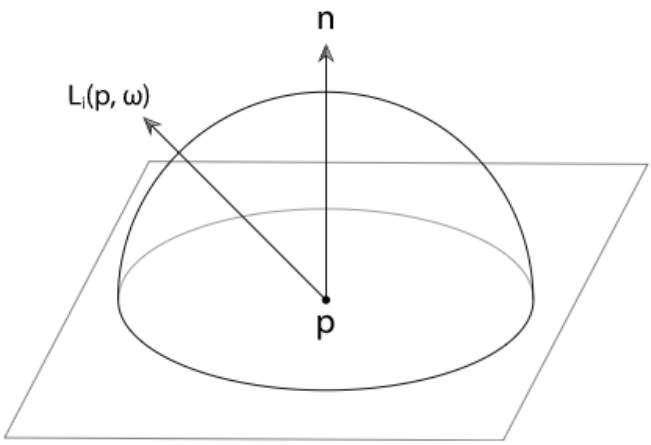


Figure 5.12: Irradiance at a point p is given by the integral of radiance times the cosine of the incident direction over the entire upper hemisphere above the point.

5.5.1 Integrals over Projected Solid Angle

在计算辐射度的时候会出现需要cos系数，这会影响我们对积分式子含义的理解。因此引入新的量：投影立体角（ projected solid angle ）。如下图所示

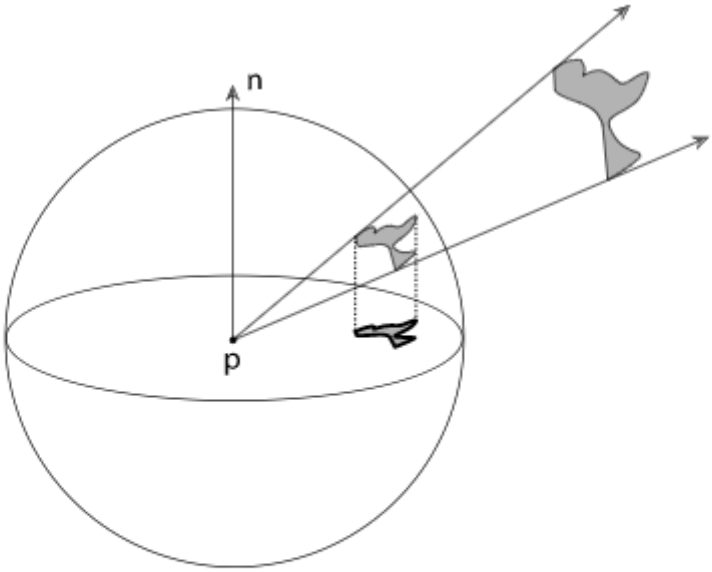


Figure 5.13: The projected solid angle subtended by an object is the cosine-weighted solid angle that it subtends. It can be computed by finding the object's solid angle, projecting it down to the plane perpendicular to the surface normal, and measuring its area there. Thus, the projected solid angle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

先把深色物体投影到单位半球上，得到立体角，再把立体角投影到平面上，得到投影立体角，两步走。
投影立体角的公式为：

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

因此irradiance的积分形式可以简化为：

$$E(p,n) = \int_{H^2(n)} L_i(p,\omega) d\omega^{\perp}$$

在本书里，我们依然使用立体角的表示。
有了irradiance之后，我们可以通过在区域A积分，得到从一些物体通过围绕法线的半球发射而来的总通量。

$$\begin{aligned}\Phi &= \int_A \int_{H^2(n)} L_o(p,\omega) \cos\theta d\omega dA \\ &= \int_A \int_{H^2(n)} L_o(p,\omega) d\omega^{\perp} dA\end{aligned}$$

5.5.2 Integrals over Spherical Coordinates

将立体角积分转化为极坐标(θ,ϕ)是常规操作，下面是转换公式

$$\begin{aligned}x &= \sin\theta\cos\phi \\ y &= \sin\theta\sin\phi \\ z &= \cos\theta\end{aligned}$$

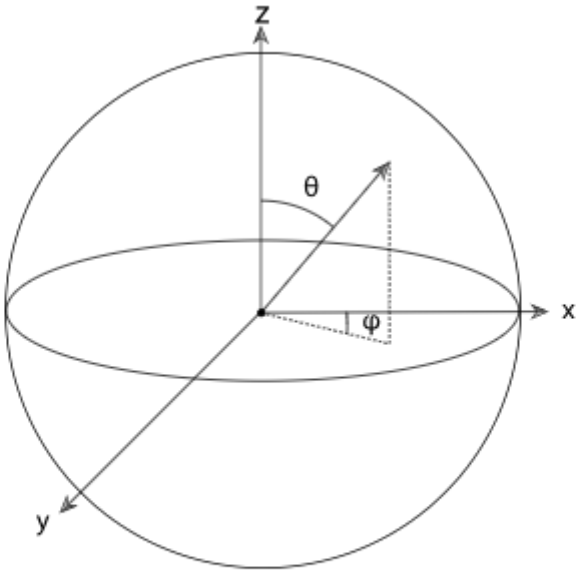


Figure 5.14: A direction vector can be written in terms of spherical coordinates (θ, φ) if the x, y, and z basis vectors are given as well. The spherical angle formulae make it easy to convert between the two representations.

而$d\omega$作为一个单位圆上的面积微元，将其看做一个矩形，它的面积实际上是两条边长度相乘：

$$d\omega = \sin\theta d\theta d\phi$$

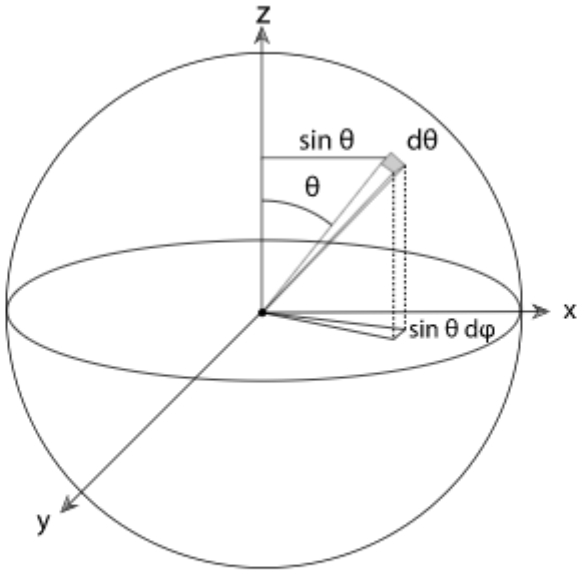


Figure 5.15: The differential area $\mathrm{d}\omega$ subtended by a differential solid angle is the product of the differential lengths of the two edges $\sin\theta\mathrm{d}\phi$ and $\mathrm{d}\theta$. The resulting relationship, $\mathrm{d}\omega = \sin\theta\mathrm{d}\theta\mathrm{d}\phi$, is the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

这个等式是关键的一步，$\sin\theta$那条线将圆球截取为一个横截面，因此横向那条边的长度是$\sin\theta d\phi$，纵向长度还是$d\theta$没变。

因此irradiance积分表达式，在半球$\Omega = H^2(n)$上，可以等价地写为：

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

如果在各个方向的亮度都相等的话，那么可以更加简化的记为

$$E = \pi L_i$$

下面的函数是极坐标系到xyz方向的转换

```
<<Geometry Inline Functions>>+=
inline Vector3f SphericalDirection(Float sinTheta,
        Float cosTheta, Float phi) {
    return Vector3f(sinTheta * std::cos(phi),
                    sinTheta * std::sin(phi),
                    cosTheta);
}
```

接下来的函数定义了一组坐标系，返回的是这个坐标系下的方向在世界当中的方向

```
<<Geometry Inline Functions>>+=
inline Vector3f SphericalDirection(Float sinTheta, Float c
osTheta,
        Float phi, const Vector3f &x, const Vector3f &y,
        const Vector3f &z) {
    return sinTheta * std::cos(phi) * x +
           sinTheta * std::sin(phi) * y + cosTheta * z;
}
```

由xyz转换为极坐标也很简单

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

下面两个函数用来计算xyz转到极坐标

```
<<Geometry Inline Functions>>+=
inline Float SphericalTheta(const Vector3f &v) {
    return std::acos(Clamp(v.z, -1, 1));
}
<<Geometry Inline Functions>>+=
inline Float SphericalPhi(const Vector3f &v) {
    Float p = std::atan2(v.y, v.x);
    return (p < 0) ? (p + 2 * Pi) : p;
```

```
}
```

5.5.3 Integrals over Area

最后一个要介绍的转化那就是如何将对方向的积分，转化为对面积的积分。重新考虑irradiance积分式

$$E(p,n)=\int_{\Omega}L_i(p,\omega)\left|\cos\theta\right|d\omega$$

假设输出的radiance方向上某个距离处有一个四边形，我们想要计算在点p处的irradiance。但计算过程并不那么直接，给定一个特定的方向，但是这个方向上的四边形是否可见却不一定。但是通过计算那个四边形面积下的积分来计算irradiance却容易很多。
从p点看，面积微元与立体角微元的关系，这里不加证明地直接给出关系式：

$$d\omega=\frac{dA\cos\theta}{r^2}$$

其中θ代表dA的法线与dA所在点的夹角，r是p到dA的距离。先不去推导，而是直观地看这个式子，如果dA刚好在距离p为1处，并且刚刚好与$d\omega$垂直，那么很明显的$d\omega=dA$，$\theta=0$，此等式成立。如果dA远离点p，并且有一些旋转，那么r^2项和$\cos\theta$项就会相应地降低$d\omega$。

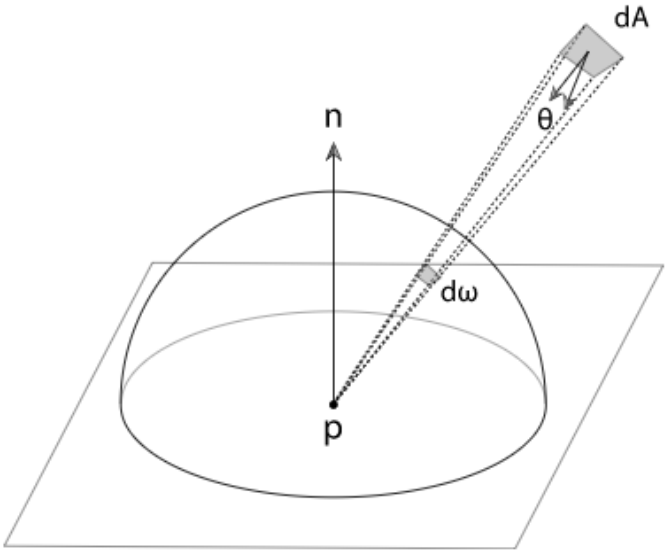


Figure 5.16: The differential solid angle subtended by a differential area $\mathrm{d}A$ is equal to $\mathrm{d}A\cos\theta/r^2$, where θ is the angle between $\mathrm{d}A$'s surface normal and the vector to the point p and r is the distance from p to $\mathrm{d}A$.

因此irradiance积分形式可以改写为来源于dA的版本：

$$E(p,n)=\int_{A}L\cos\theta_i\frac{\cos\theta_o dA}{r^2}$$

其中，L是从四边形表面过来的radiance，θ_i是表面上p点的法线和p点到光源位置p'向量的夹角，θ_o是p'点的法线与p到p'向量的夹角。

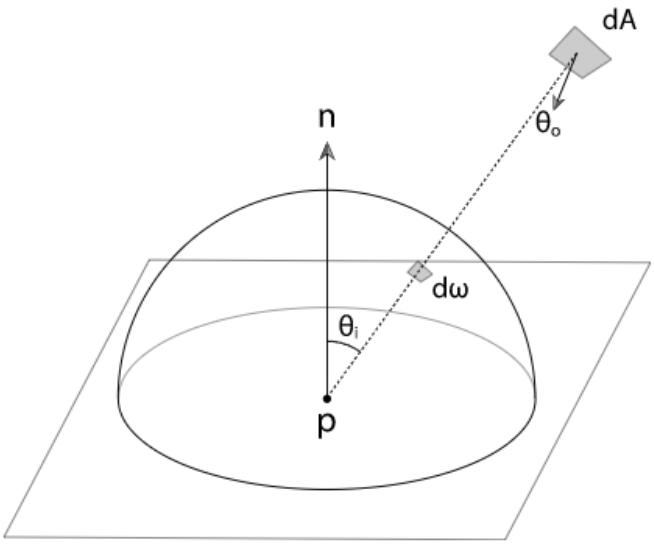


Figure 5.17: To compute irradiance at a point p from a quadrilateral source, it's easier to integrate over the surface area of the source than to integrate over the irregular set of directions that it subtends. The relationship between solid angles and areas given by Equation (5.6) lets us go back and forth between the two approaches.

5.6 Surface Reflection

当光线入射到表面，表面散射光线，反射一部分回到环境中。反射模型中有两个主要效果：反射光线的光谱分布，以及方向分布。例如光线照射到柠檬表皮，白光会反射出黄色（红+绿），而蓝色被吸收掉，与此同时，在某些角度看柠檬皮，会有高光效果。
对于透明表面的反射更加复杂，大量的材质会表现出**次表面光线散射(subsurface light transport)**，光线进入表面的某一点，但从离这点有一定距离的另一点出射出去。
有两个抽象来表达这种光线反射机制：BRDF和BSSRDF。BRDF描述了表面在某一点的反射，而忽略了次表面光线传播，对于大多数材质这种微小的不精确并不重要。而BSSRDF则是BRDF的泛化版本，描述了更加通用的在半透明材质上的光线传播。

5.6.1 The BRDF

双向反射分布函数（bidirectional reflectance distribution function, BRDF）描述了表面的反射。如下图，我们需要在给定入射radiance：$L_i(p, \omega_i)$的情况下，计算出ω_o方向出射的radiance：$L_o(p, \omega_o)$。

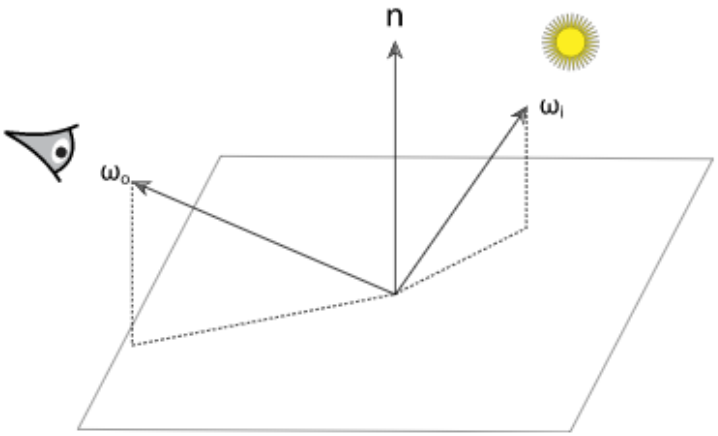


Figure 5.18: The BRDF. The bidirectional reflectance distribution function is a 4D function over pairs of directions ω_i and ω_o that describes how much incident light along ω_i is scattered from the surface in the direction ω_o.

如果ω_i考虑为一个方向的圆锥微元，则p点的irradiance微元为：

$$dE(p,\omega_i) = L_i(p,\omega_i)\cos\theta_i d\omega_i$$

因为有了这个入射光的irradiance微元，我们就会产生一个反射方向ω_o的radiance微元。因为几何光学的线性假设，反射微元radiance与irradiance成正比关系

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

因此这个比例系数定义了表面的BRDF，它和ω_i与ω_o有关。

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

基于物理的BRDFs有两个重要的性质：

- 光路可逆性：对于所有的ω_i和ω_o组合，都有$f_r(p,\omega_i,\omega_o) = f_r(p,\omega_o,\omega_i)$
- 能量守恒：对于反射光的总能量，低于等于入射光的总能量。对于所有的方向ω_o

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

表面的**双向透射分布函数(bidirectional transmittance distribution funciton, BTDF)**，描述了透射光线的分布。BTDF通常表示为$f_t(p,\omega_o,\omega_i)$，值得注意的是ω_o和ω_i在半球的两侧。BTDF并没有光路可逆的性质。
我们通常把BRDF和BTDF统一用$f(p,\omega_o,\omega_i)$来表示，这个函数就叫做**双向散射分布函数(bidirectional scattering distribution function, BSDF)**。

$$dL_o(p,\omega_o) = f(p,\omega_o,\omega_i)L_i(p,\omega_i)\left|\cos\theta_i\right|d\omega_i$$

$\cos\theta_i$项有个绝对值符号。这是因为表面法线方向不会随着ω_i来调整使得它们总在同一边（有些文章里会这么做，但pbrt中不会）。

接下来是将这个式子做积分，得到L_o：

$$L_o(p,\omega) = \int_{S^2} f(p,\omega_o,\omega_i)L_i(p,\omega_i)|\cos\theta_i|d\omega_i$$

这个就是渲染基础方程：它描述了一个点周围分布的入射光线，基于表面的散射属性，是如何转换为出射光线的。它被称为在球体S^2作为域(domain)的情况下的**散射方程(scattering equation)**，或者上半球$H^2(n)$积分的**反射方程(reflection equation)**。

5.6.2 The BSSRDF

双向散射表面反射分布函数(bidirectional scattering surface reflectance distribution function, BSSRDF)描述了从材质散射的光线，表现出了明显的次表面光线传输。用函数$S(p_o,\omega_o,p_i,\omega_i)$表示，它的含义是在点$p_o$处$\omega_o$方向的出射radiance微元，与点$p_i$处$\omega_i$方向的入射通量微元的比值。

$$S(p_o,\omega_o,p_i,\omega_i) = \frac{dL_o(p_o,\omega_o)}{d\Phi(p_i,\omega_i)}$$

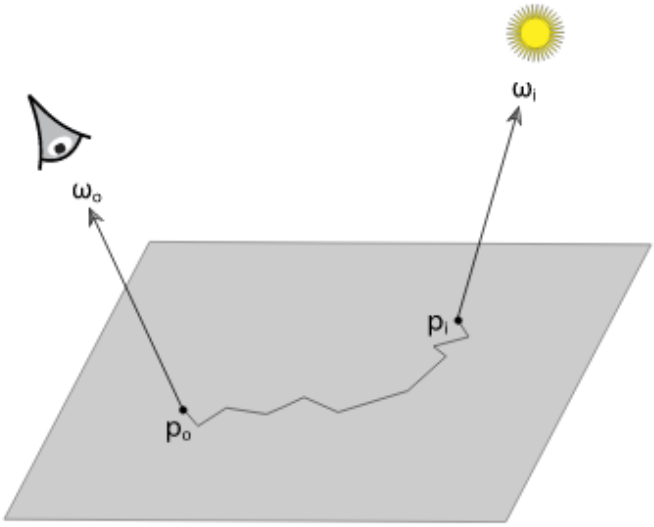


Figure 5.19: The bidirectional scattering surface reflectance distribution function generalizes the BSDF to account for light that exits the surface at a point other than where it enters. It is often more difficult to evaluate than the BSDF, although subsurface light transport makes a substantial contribution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

注意上式中的分子不再是dE，BSSRDF需要两个积分项：面积以及入射方向。将2维积分变成了4维。

$$L_o(p_o,\omega_o) = \int_A \int_{H^2(n)} S(p_o,\omega_o,p_i,\omega_i)L_i(p,\omega_i)|\cos\theta_i|d\omega_i dA$$

相当于与基础的反射方程没有本质变化，但是把dA拉出来做积分了。
需要注意一个特性就是，p_i和p_o离得越远，函数S的值实际上是越小，这也很容易理解，离得越远这种次表面散射的效果应该就越弱，这个特性能帮助我们实现这个函数。