Physically Based Rendering Notes 基于物理渲染笔记

Lingheng Tony Tao

2024 年 6 月 11 日

目录

前言

这份笔记的作用是帮助复习基于 CMU 的课程 15-468/668/868 (Physically Based Rendering) 以及英文教材 Physically Based Rendering, From Theory to Implementation（`pbrt`）的 PBR（基于物理渲染）相关知识点。笔记中肯定会有不严谨的描述，也可能会出现纰漏，希望读者也可以帮助纠错，并反馈至作者。

文中出现的伪代码主要以 C/C++ 风格出现。与 `pbrt` 相似，在这份笔记中也会重点关注 C++ 中的实现。

本笔记的主要参考书籍有：

- Pharr, M., Jakob, W., & Humphreys, G. (2023b). *Physically Based Rendering, fourth edition: From Theory to Implementation*. MIT Press.
- York, S. U. M. I. N., & Shirley, P. (2022a). *Fundamentals of Computer Graphics: International Student Edition*. CRC Press.
- Dutre, P., Bala, K., & Bekaert, P. (2006). *Advanced Global Illumination,* second edition. A K Peters/CRC Press.
- Mitzenmacher, M., & Upfal, E. (2017). *Probability and computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis.* Cambridge University Press.

Part I

基于物理渲染基础

Chapter 1

光线追踪概述

本章节主要是对光线追踪进行一些简单的概述，以便后面进行几何表示法的讨论。

1.1 与光栅化的比较

光栅化（Rasterization）与**光线追踪**（Ray Tracing）都是在历史上作为 3D 渲染的手段中占据重要篇幅、具有重要作用的两种渲染手段。

在光栅化中，我们将物体上的点最终从模型空间中转移至屏幕空间。这是一个由 3D 开始至 2D 结束的手段。一个很简单的伪代码算法如下：

```
for (each triangle)
    for (each pixel)
        if(triangle covers pixel)
            keep closest hit
```

这是一种从三角形出发的算法。注意第一行和第二行 for 循环嵌套的顺序。

而在光线追踪中，我们将从屏幕空间出发，从像素上打出一根光线，并决定这根光线会与哪些 3D 物体产生交互。

```
for (each pixel or ray)
    for (each triangle)
        if(ray hits triangle)
            keep closest hit
```

两种渲染手段都各有千秋。

光栅化并不需要时刻跟踪整个场景的数据资源，并且支持并行计算，也有各种各样的内存优化，因此，光栅化整体而言给人更快的感觉。但是，光栅化尤其不擅长处理全局光照效果，例如阴影、反射、透明度等。

而光线追踪则可以说是光栅化的另一面，擅长处理全局光照，能够给出非常真实、接近照片的渲染结果，但是代价就是开销高，计算慢，硬件支持差（但是，随着科技的进步，越来越多的 GPU 已经开始支持甚至青睐光线追踪，所以这也不是一个缺陷了）。

因此，在本笔记中，我们将几乎不讨论光栅化，主要去讨论如何更加真实地将光线追踪的效果推至登峰造极。

1.2 基本的光线追踪算法

在写实渲染中，我们最终的目的是要渲染出一张肉眼看上去与照片无法区分的计算机生成图。在光线追踪中，我们模拟光的实际物理行为（当然，也有一些限制）。

从概念上来说，光线追踪的原理不难，所谓的追踪，就是跟随一条光线，让它与场景交互、弹射，最后计算颜色。但是，不论光线追踪算法被如何拓展，其基本的骨架中都会有以下的部分。

- **相机**：相机决定观察者的位置，所以有的时候也被称为**人眼**。
- **射线-物体求交**：光线的数学本质是**射线**。我们必须要精确地表达光线与物体相交的位置，有时也需要维护一些额外的信息，例如交点所在面的表面法线或者材质等。一个光线追踪算法通常会让光线与多个物体相交，然后返回最近的那个交点。
- **光源**：光线追踪算法不仅要记录光源的位置，也要记录它们发光（或者说发出光子能量）的方式，通过辐射度量学的物理量精确描述这种方式。
- **表面散射**：当光线到达一个物体的表面时，我们要精确地分析它们接下来该往哪去——有多少进入了表面，有多少离开了表面，离开的光线又去了哪个方向。我们通常需要一个参数化函数来完成这件事。
- **间接光**：所谓的间接光就是经过弹射之后到达表面的光。光发射到其它表面上时，可能会弹射出一部分，这部分又到达了其它表面，就成为了间接光。光线追踪算法也需要考虑间接光的存在。
- **光线传播**：虽然我们假设在真空中光线的能量在这条射线上保持不变，但是现实中并没有这么多的真空情况；恰恰相反，类似于雾、烟、霾甚至是地球大气之类的现象都很常见，我们也需要考虑这个部分。

在这份笔记中，我们对光线的行为做出以下的一些假设。

- 光线只有粒子的特性而没有波的特性，即没有**衍射** (diffraction)，没有**偏振** (polarization)，也没有**干涉** (interference)。
  - 衍射：指波遇到障碍物时偏离原来直线传播的物理现象。
  - 偏振：指的是横波能够朝着不同方向振荡的性质。
  - 干涉：指的是两列或两列以上的波在空间中重叠时发生叠加，从而形成新波形的现象。
- 光线在真空中沿直线传播。
- 颜色可以被一个 $C \in [0,1]^3$ 描述，即 (R,G,B)。

在上文与光栅化的比较中，我们给出过一个非常简单的光线追踪算法，从像素出发，投射一根光线，与物体求交。其实光线追踪万变不离其宗，我们可以写一个稍微更偏向于实现的基本算法。

```
RayTraceImage() {
    Parse(); // Parsing the scene

    for (each pixel) {
        Ray ray = GenerateRayFromCamera(pixel);
        pixel.Color = Trace(ray);
    }
}

Trace(Ray ray) {
    HitInfo hit = FindIntersection();

    return Shade(hit); // might trace more rays in Shade()
}
```

从这个简单的伪代码中，我们就可以看出来我们需要在光线追踪中解决的一些问题。如何计算与场景的交点？如何投射出射线？颜色又到底是怎么计算出来的？这些问题在后面来一一解决。

1.3 射线-几何体求交

这是本课程中我们需要解决的第一个问题，同时，这个问题我们在这里也只会先解决一部分。这个问题其实分成了两个部分，**射线**和**几何体**。我们必须要能够描述射线和几何体，才能解决这个问题。我们先关注三个最基本的求交，射线-球、射线-平面以及射线-三角形求交。

定义（射线）对于给定的**起点** $\mathbf{o}$，以及一个单位方向向量 $\mathbf{d}$，**射线**（ray）的参数化方程为

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

其中，$\mathbf{o}$ 是一个齐次坐标中 $w \neq 0$ 的四维向量，$\mathbf{d}$ 是一个齐次坐标中 $w = 0$ 的四维单位向量。

在这里，$\mathbf{d}$ 定义为单位向量是约定，虽然也可以用非单位向量，但是在之后的计算中可能会出现问题。这是射线的**显式**（explicit）几何表达。

区分一个方程是一个几何的显式表达还是隐式（implicit）表达的方式很简单：如果我们有一个点 (x, y, z)，我们可以将其插入方程判断等号是否成立，则这个表达式是隐式的。如果我们可以通过使用参数，生成出一个位于这个几何上的点，那么这个表达式就是显式的。所以显然，在这里我们通过输入不同的 t，可以找到在射线上不同的点，这个射线方程就是显式的。

1.3.1 射线-球求交

首先先考虑第一个最简单的情况。射线与球的交点。对于球而言，我们可以用一个隐式方程来定义球。

定义（球）对于给定的**球心** $\mathbf{c}$，以及一个所求点 $\mathbf{x}$，一个半径为 r 的**球**（sphere）的隐式方程为

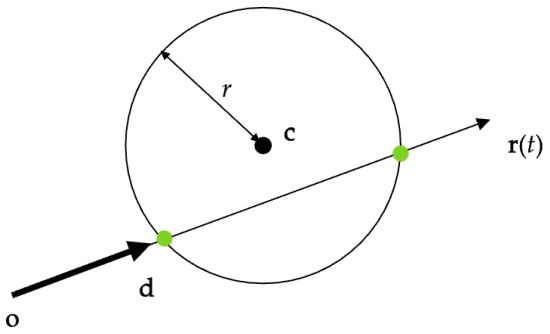
$$||\mathbf{x} - \mathbf{c}||^2 - r^2 = 0$$

这样的话，我们只需要把这个点 $\mathbf{x}$ 带入射线方程即可，因为如果有交点的话，交点也同时在射线上。也就是求解

$$||\mathbf{o} + t\mathbf{d} - \mathbf{c}||^2 - r^2 = 0$$

展开之后，

$$(\mathbf{o}_x + t\mathbf{d}_x - \mathbf{c}_x)^2 + (\mathbf{o}_y + t\mathbf{d}_y - \mathbf{c}_y)^2 + (\mathbf{o}_z + t\mathbf{d}_z - \mathbf{c}_z)^2 - r^2 = 0$$



在这里，$\mathbf{o}_x, \mathbf{o}_y, \mathbf{o}_z, \mathbf{d}_x, \mathbf{d}_y, \mathbf{d}_z, \mathbf{c}_x, \mathbf{c}_y, \mathbf{c}_z, r$ 全部都是常数，因此这是一个简单的一元 (t) 二次方程，根据该方程的根的数量，就可以判断射线与球是否相交（或相切）了。由一元二次方程的求根公式 $t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$，如果判别式 $b^2 - 4ac < 0$，我们就知道没有交点。否则，有一个或是两个不同的交点。在这里，$a = \mathbf{d} \cdot \mathbf{d}, b = 2\mathbf{d} \cdot (\mathbf{o} - \mathbf{c}), c = (\mathbf{o} - \mathbf{c})^2 - r^2$。

1.3.2 射线-平面求交

与球类似，我们也可以使用平面的隐式定义，然后带入其中求解。

定义（平面）对于给定的平面上的一个**点** $\mathbf{p}$，以及一个所求点 $\mathbf{x}$，一个**法向量**为 $\mathbf{n}$ 的**平面**（plane）的隐式方程为

$$(\mathbf{x} - \mathbf{p}) \cdot \mathbf{n} = 0$$

类似地，如果有交点，这个点也肯定在平面上，因此我们用射线方程替代 $\mathbf{x}$。

$$(\mathbf{o} + t\mathbf{d} - \mathbf{p}) \cdot n = 0$$

$$t\mathbf{d} \cdot \mathbf{n} + (\mathbf{o} - \mathbf{p}) \cdot \mathbf{n} = 0$$

$$t = -\frac{(\mathbf{o} - \mathbf{p} \cdot \mathbf{n})}{\mathbf{d} \cdot \mathbf{n}}$$

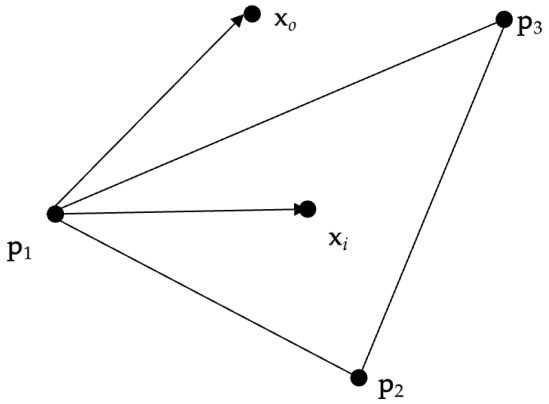
从结果上来看，当且仅当 $\mathbf{n}$ 与射线方向 $\mathbf{d}$ 垂直时，t 的值无意义；形象地理解，这也确实是正确的——只要射线不是与平面平行（即与平面法线垂直），那么射线所在的直线就一定和平面有交点。

但是，如果所求出 t 的值为负值，那意味着这个点如果沿着射线方向走，需要走到射线起点的反方向。对于射线而言，这样的解也没有意义。因此，当且仅当 $t \geq 0$ 时，我们认为射线和平面有交点。

1.3.3 射线-三角形求交

三角形肯定在一个平面上，因此，射线与三角形有交点的必要条件是射线与三角形所在平面有交点。射线与三角形有交点的充分必要条件还需要再加上一条，即这个交点在三角形内。因此这个交点满足下面的三个条件。

- 交点在射线上。
- 交点在平面内。
- 交点在三角形内。



现在我们观察上图中在三角形 $\Delta\mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$ 中的点 $\mathbf{x}_i$ 以及在其外面的点 $\mathbf{x}_o$。

叉乘法

首先，我们先找到一条三角形所在平面内的法线。这个法线可以通过叉积获得。即

$$\mathbf{n} = (\mathbf{p}_2 - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$

在上图中，根据右手定则，这根法线应该伸向纸外。下面，比较以下两个法线的方向。

$$\mathbf{n}_o = (\mathbf{x}_o - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$
$$\mathbf{n}_i = (\mathbf{x}_i - \mathbf{p}_1) \times (\mathbf{p}_3 - \mathbf{p}_1)$$

注意到，$\mathbf{n}_o$ 指向纸面内，而 $\mathbf{n}_i$ 与 $\mathbf{n}$ 相同，指向纸面外。这样我们就找到了判断交点在不在三角形内的第一个方法，也被称作*叉乘法*。

```
// Requires: triangle is a 3-element array
IsPointInTriangle(Vec3f P, Vec3f[] triangle) {
    Vec3f AP, BP, CP;
    AP = P - triangle[0];
    BP = P - triangle[1];
    CP = P - triangle[2];

```

```
    Vec3f AB, BC, CA;
    AB = triangle[1] - triangle[0];
    BC = triangle[2] - triangle[1];
    CA = triangle[0] - triangle[2];

    Vec3f n1, n2, n3;
    n1 = Vec3f.Cross(AB, AP);
    n2 = Vec3f.Cross(BC, BP);
    n3 = Vec3f.Cross(CA, CP);

    bool check1 = n1.z >=0 && n2.z >= 0 && n3.z >= 0;
    bool check2 = n1.z < 0 && n2.z < 0 && n3.z < 0;
    return check1 || check2;
}
```

重心坐标法

第二个方法被称作重心坐标法。注意到，如果所求点在三角形内，那么这个点和三角形每另外两个顶点组成的小三角形（因此共有三个）的面积总和应该与三角形的面积相同。首先，我们需要明确三角形的重心坐标的概念。

定义（重心坐标）对于一个三角形 ABC，设其顶点坐标分别为 $\mathbf{a}, \mathbf{b}, \mathbf{c}$。一个与三角形共面的点 $\mathbf{p}$ 可以被写作 $\alpha\mathbf{a}+\beta\mathbf{b}+\gamma\mathbf{c}$ 的形式，其中 (α,β,γ) 为点 $\mathbf{p}$ 关于该三角形的**重心坐标**（Barycentric coordinates）。

性质 若点 $\mathbf{p}$ 关于三角形 ABC 的重心坐标 (α,β,γ) 满足 $\alpha+\beta+\gamma=1$，则点 $\mathbf{p}$ 在三角形内。

此处，我们先不说明如何求得三角形的重心坐标。假设我们已经获得了 (α,β,γ)，根据之前的那三个条件，我们也就可以直接将重心坐标带入方程了。

$$\begin{aligned}\mathbf{o}+t\mathbf{d} &= \alpha\mathbf{a}+\beta\mathbf{b}+(1-\alpha-\beta)\mathbf{c}\\ \alpha(\mathbf{a}-\mathbf{c})+\beta(\mathbf{b}-\mathbf{c})-t\mathbf{d} &= \mathbf{o}-\mathbf{c}\\ \alpha\mathbf{A}+\beta\mathbf{B}-t\mathbf{d} &= \mathbf{C}\end{aligned}$$

也就是说，我们只需要求解

$$\begin{bmatrix}-\mathbf{d} & \mathbf{A} & \mathbf{B}\end{bmatrix}\begin{bmatrix}t\\ \alpha\\ \beta\end{bmatrix}=\mathbf{C}$$

即可，这也许会快很多。

Möller-Trumbore 算法

第三个方法是工业中使用的比较多的 *Möller-Trumbore* 算法。这个方法我们直接将射线的方向作为信息使用，其计算速度相对而言较快。在这里我们不介绍算法的原理，也不讨论算法的证明，仅提供算法过程，感兴趣的同学可以自行查询。

```
// Requires: triangle is a 3-element array
// Moller-Trumbore
```

```
DoesRayHitTriangle(Ray ray, Vec3f P, Vec3f p0, Vec3f p1, Vec3f p2) {
    Vec3f e1 = p1-p0;
    Vec3f e2 = p2-p0;
    Vec3f p = cross(ray.direction, e2);

    // Determinant
    float a = dot(e1, p);
    // Fail Case 1
    if(a is close to 0) return false;

    // t is the hit parameter for the ray
    Vec3f t = ray.origin - p0;
    float w2 = dot(t,p) / a;

    // Fail Case 2
    if(w2 < 0 || w2 > 1) return false;

    Vec3f q = cross(t, e1);
    float w3 = dot(ray.direction, q) / a;

    // Fail Case 3
    if(w3 < 0 || w2 + w3 > 1) return false;

    return true;
}
```

至此，关于光线追踪的简单概述就差不多了。接下来我们就可以进入一些更细节的内容，来讨论如何完善整个光线追踪的算法，以及如何通过 C++ 实现。

Chapter 2

基本几何数据结构

在上一章中，我们已经看到了射线与一些基本几何体求交的过程。在本章节中，我们从如何用严格的数学和计算机语言描述几何、以及我们可能会在几何上进行的一些操作行为开始，进一步描述几何图形。本章中我们会使用来自达特茅斯的 Dartmouth Introductory Ray Tracer（DIRT）作为框架，从实现的角度出发。

2.1 几何与变换

2.1.1 坐标系

坐标系的存在毫无疑问是一切的根基，否则，我们将根本无从以数学的语言来描述点、向量这些最基本的概念。首先，我们先了解仿射空间的概念。

定义（仿射空间）一个原点 $\mathbf{o}$ 以及 n 个互相线性独立的基向量 $\{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_n\}$ 可以定义一个 n 维的**仿射空间**（affine space）。在这个空间中，任何向量 $\mathbf{v}$ 都可以写作

$$\mathbf{v} = s_1\mathbf{v}_1 + s_2\mathbf{v}_2 + ... + s_n\mathbf{v}_n$$

的形式，且任何点 $\mathbf{p}$ 都可以写作

$$\mathbf{p} = \mathbf{o} + s_1\mathbf{v}_1 + s_2\mathbf{v}_2 + ... + s_n\mathbf{v}_n$$

的形式。

然而，为了确定一个坐标系的存在，我们需要先于这个坐标系确定点和基向量的概念，因这就会出现先有鸡还是先有蛋的矛盾问题。为了解决这个问题，我们会规定一个**标准坐标系**，（在三维的情况下）它的原点为 $(0, 0, 0)$，基向量为 $(1, 0, 0), (0, 1, 0), (0, 0, 1)$。其它的所有坐标系都可以通过这个标准化坐标系来定义，这个标准化坐标系就被我们称为**世界空间**（world space）。

2.1.2 向量

无论是点还是向量，它们本质上都是一个 n 维数组。由于数组有可能是整数的数组、有可能是浮点数或是 `double`，我们不希望为每一种类型单独写一整套方法。我们可以采用 C++ 的泛型策略，利用 `template` 关键字，创建模板结构体，来保存向量和点的基本数据结构。

```
/// An array of N values of type T
template <size_t N, typename T> struct Vec {
    // To Be Added
}

template <typename T> using Vec2 = Vec<2, T>;
//... OMIT, do the same for 3d and 4d, as well as Color3, Color4

using Vec2i = Vec2<std::int32_t>;
//... OMIT, do the same for all the other frequently used types
```

在泛型编程中，我们并不在意这个向量内存储的实际类型；模板中，我们以 `T` 来指代这个类型。我们可以用一个数组存储这个向量的每一个维度，然后用方括号操作符读取。

```
template <size_t N, typename T> struct Vec {
    std::array<T, N> dimensions;

    T operator[](size_t i) const { return dimensions[i]; }
    T &operator[](size_t i) {return dimensions[i]; }
}
```

为了防止编译器自动的隐式转换，我们使用一个 `explicit` 关键字来限定我们的构造函数。

```
template <size_t N, typename T> struct Vec {
    std::array<T, N> dimensions;

    explicit Vec(T e0) {
        for(size_t i = 0; i < N; ++i) {
            e[i] = e0;
        }
    }

    Vec(const std::initializer_list<T> &list) {
        int i = 0;
        for (auto &element : list) {
            e[i] = element;
            if (++i > N) break;
        }

    }

    T operator[](size_t i) const { return dimensions[i]; }
    T &operator[](size_t i) {return dimensions[i]; }
}
```

我们也可以在头文件中定义一些内联函数或接口以提供有关向量的基础计算，例如点乘、叉乘、返回最大元素、基本的加减乘除、向量模等。这里就不再赘述每个函数的实现方式。

2.1.3 矩阵

矩阵的实现和向量也非常类似，只是在元素的索引上我们有多重选择，这也取决于我们如何看待矩阵。从图形学的角度，通常我们有以下几种对矩阵的理解方式。

- 矩阵由 n^2 个元素组成。

- 矩阵由 n 个 n 维行向量组成。
- 矩阵由 n 个 n 维列向量组成。
- 矩阵由数个维度更小的矩阵组成。

这意味着我们可能需要有多种构造矩阵类型的方式。反映到构造函数上，上面的理解方式中，第四种我们暂时先不考虑，先考虑前面三种的代码意义。

- 矩阵 `M` 应该可以看做一个长度为 n^2 的一维数组。
- 矩阵 `M` 也应该可以看做一个长度为 n 的一维数组，其中的每个元素都是一个 `VecN` 行向量。此时，如果索引 `M[r][c]`，返回的就应该是 `M` 中第 `r` 个向量的第 `c` 个元素。
- 矩阵 `M` 也应该可以看做一个长度为 n 的一维数组，其中的每个元素都是一个 `VecN` 列向量。此时，如果索引 `M[r][c]`，返回的就应该是 `M` 中第 `c` 个向量的第 `r` 个元素。

因此，矩阵本身的数据结构我们可以有以下的实现方式（以 4×4 的 `Mat44` 为例）：

```
template <typename T> struct Mat44 {
    union {
        // Type - 1, struct of 16 elements
        struct {
            T m00, m10, m20, m30;
            T m01, m11, m21, m31;
            T m02, m12, m22, m32;
            T m03, m13, m23, m33;
        };

        // Type - 2, struct of arrays of 4 Vec4, aliasing for cols/rows
        std::array<Vec4, 4> m;
        struct { Vec4 x,y,z,w; }
        struct { Vec4 a,b,c,d; }
    }
}
```

回顾一下，在 C++ 中 `union` 关键字的用法。在 C++ 中，`union` 是一种特殊的数据类型，允许我们在相同的内存位置存储不同的数据类型。`union` 的主要特点是所有成员共享同一块内存，`union` 的大小等于这里最大成员的大小（加上一些可能的填充）。但是，`union` 在任何时候都只能存储其中的一种类型。例如在上面的声明中，`Mat44` 就只能是 16 个 `T` 类型元素的数组，或者是一个 4 个 `Vec4` 元素的向量数组。

考虑完矩阵的数据结构之后，我们需要考虑一下矩阵的构造和索引。构造函数也需要去考虑使用什么样的数据结构。

```
template <typename T> struct Mat44 {
    // OMIT union, see above
    // Constructors
    explicit Mat44(T s);
    Mat44(const Vec4 &A, const Vec4 &B, const Vec4 &C, const Vec4 &D);
    Mat44(const T a[4][4]);

    // Element access.
```

```
    const Vec4 &operator[] (int col) const { return m[col]; }
    Vec4 &operator [](int col) { return m[col]; }
    T &operator()(int row, int col) { return m[col][row]; }
    T operator()(int row, int col) const { return m[col][row]; }
}
```

至此，关于矩阵的结构、构造和索引基本已经完成了。和向量类似，我们也可以给矩阵提供一些基本运算、快速填充、转置、求逆、返回行列式之类的内联操作，在这里也就不再赘述每个操作的实现方式了，有兴趣的话可以去钻研 DIRT 中的 `vec.h` 头文件。

2.1.4 变换

在计算机图形学的实现中，我们几乎不去考虑非齐次坐标下仿射变换与线性变换的区别，因为我们肯定都会使用齐次坐标来解决变换的问题。因此，在三维运算中，我们实际上操作的都是 4×4 的齐次坐标变换矩阵。另外，在没有特别的说明下，我们默认变换与变换矩阵是同义的。因此，存储变换的数据结构显而易见，就是我们上面刚刚声明的 `Mat44`。

```
struct Transform {
    using Mat44f = Mat44<float>;
    Mat44f m;
    Mat44f mInv;

    Transform(const Mat44f &m) : m(m), mInv(::inverse(m)) {}
    Transform(const Mat44f &t, const Mat44f &it) : m(t), mInv(it) {}
}
```

为了方便我们的使用，我们直接在变换的结构体中声明了逆变换矩阵，由于矩阵中我们已经实现了 `inverse()` 方法，所以这里在声明矩阵的时候我们也不需要额外提供什么信息。当然，如果逆矩阵已经计算好了，也可以直接用第二个构造函数代入构造。

在变换结构体中，我们也可以提供一些基本的方法，以供之后使用。例如，

```
Vec3f vector(const Vec3f &v) const {};
Vec3f normal(const Vec3f &n) const {};
Vec3f point(const Vec3f &p) const {};
Ray3f ray(const Ray3f &r) const {};
Box3f box(const Box3f &box) const{};
static Transform translate(const Vec3f &t) {};
```

例如，我们就可以使用 `vt = T.vector(v)` 来对 `v` 使用变换 `T`，并将其存入变量 `vt` 中了。

特别地，我们需要关注一下法线变换。我们知道，法线与平面垂直的关系在变换中并不会保留。通常，对于一个变换 $\mathbf{M}$，我们需要额外地对法线去施加不同的变换 $(\mathbf{M}^{-1})^{\mathrm{T}}$，即变换矩阵的逆矩阵的转置矩阵。这也是为什么我们需要在 `Transform` 结构体中总是保存着逆矩阵的原因。

2.1.5 射线

在有了之前的数据结构之后，我们终于可以开始去实现具体的、实际的几何图形了。我们从最基本的元素开始——射线。回顾一下之前的射线定义。

定义（射线）对于给定的**起点 o**，以及一个单位方向向量 **d**，**射线**（ray）的参数化方程为

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}$$

其中，**o** 是一个齐次坐标中 $w \neq 0$ 的四维向量，**d** 是一个齐次坐标中 $w = 0$ 的四维单位向量，$t \geq 0$。

因此，对于一条射线，其需要定义的内容也就非常清晰了。

```
template<size_t N, typename T> struct Ray {
    Vec<N, T> o;
    Vec<N, T> d;
    float mint, maxt;
    Vec<N, T> operator () (T t) const { return o + t * d; }
}
```

这里的 `mint` 和 `maxt` 是射线实际的有效参数范围。我们认为小于 `mint` 和大于 `maxt` 的部分都不在射线内（虽然这听起来让射线变成了线段，但是很快我们就会在后面看到这么做的用意）。构造射线自然而然也就应该利用起点和方向。

```
template<size_t N, typename T> struct Ray {
    // OMIT, see above
    Ray(const Vec<N,T> &o, const Vec<N,T> &d) : o(o), d(d) {}
}
```

2.1.6 轴对齐包围盒

在第一章的部分中，我们已经看到了射线与三角形求交并不是一个简单的过程。随着模型复杂起来，射线可能会需要和很多的三角形求交。所以，为了加速我们的渲染，我们可以将几何体先用一个与 x, y, z 三条轴对齐的包围盒包裹。当这条射线与包围盒有交点的时候，射线才有可能和物体有交点，这个时候我们再去做射线-三角形求交。

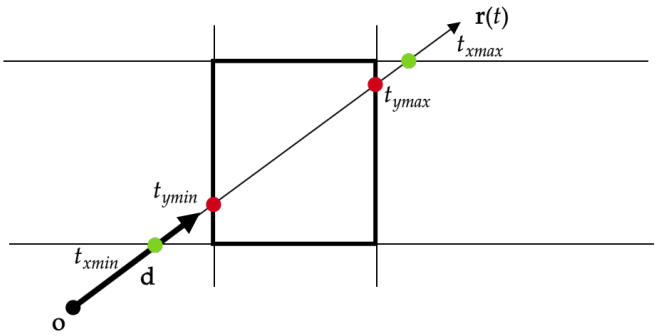
因此，我们可以将**轴对齐包围盒**（axis-aligned bounding box, AABB）作为一个基本的几何类型。本质上它们就是长方体，而且因为它们是轴对齐的，我们并不需要去记录每条边的方向。有两种方法可以限定一个 AABB：要么我们可以记录它的一个顶点，并记录三条边的边长；要么我们可以记录两个对角点（例如左下角和右上角），我们称它们为最小点和最大点。在这里，我们以最小点最大点的记录方法为例。

```
template<size_t N, typename T> struct Box {
    Vec<N, T> pMin, pMax; // lower-bound and upper bound
}
```

在 AABB 中，我们可以去实现各种方便的功能，例如包围另一个包围盒，判断包围盒是否为空等。这里，我们重点关注一个功能的实现，就是之前没有提到的**射线-轴对齐包围盒求交**。

射线-轴对齐包围盒求交

我们先从更简单的 2D 情况开始讨论。3D 情况实际上只要对 xy, xz, yz 三个平面分别做一次 2D 求交即可。以 xy 平面为例。



要求得射线与 $x = x_{\min}$ 的交点很简单。假设射线起点 $\mathbf{o} = (o_x, o_y, o_z)$，射线方向 $\mathbf{d} = (d_x, d_y, d_z)$，则

$$o_x + t_{x\,\min} d_x = x_{\min}$$
$$t_{x\,\min} = \frac{x_{\min} - o_x}{d_x}$$

类似地，我们也可以得到其它的几个交点对应的 t 值。注意，这个 t 是射线的参数。

$$t_{x\,\min} = (x_{\min} - o_x)/d_x$$
$$t_{x\,\max} = (x_{\max} - o_x)/d_x$$
$$t_{y\,\min} = (y_{\min} - o_y)/d_y$$
$$t_{y\,\max} = (y_{\max} - o_y)/d_y$$

观察上图，我们观察到，只有在 $[t_{x\,\min}, t_{x\,\max}] \cap [t_{y\,\min}, t_{y\,\max}]$ 中，射线是处于包围盒内的。所以，射线与包围盒有交点的充要条件是这个区间不为空集。另外，在上面的代码中，我们也默认了 d_x 是大于 0 的；否则，射线就会先击中 $x_{\max}$，而不是 $x_{\min}$。

还有一个问题要解决。在上面的 t 值计算中，d_x, d_y 是有可能是 0 的，那么这个值就可能出现除零错误。根据 IEEE 的浮点数规则，我们知道 $\forall a > 0$,

$$+a/0 = +\infty;$$
$$-a/0 = -\infty.$$

我们以 x 方向为例，考虑 $x_d = 0, y_d > 0$ 的情况（一条竖直向上的射线）。

$$t_{x\,\min} = (x_{\min} - o_x)/0$$
$$t_{x\,\max} = (x_{\max} - o_x)/0$$

考虑这三种情况。

- $o_x \leq x_{\min}$: 这种情况下，射线出现在 $x_{\min}$ 的左侧，必然与包围盒没有交集。
- $o_x \geq x_{\max}$: 这种情况下，射线出现在 $x_{\max}$ 的右侧，必然与包围盒没有交集。
- $o_x \in (x_{\min}, x_{\max})$: 这种情况下，射线在包围盒内部，必然与包围盒有交集。

观察这三种情况中，$t_{x\,\min}$ 和 $t_{x\,\max}$ 对应的数值。

- $t_{x\,\min} = +\infty, t_{x\,\max} = +\infty$，因此，$[t_{x\,\min}, t_{x\,\max}] \cap [t_{y\,\min}, t_{y\,\max}] = [\infty, \infty] \cap [t_{y\,\min}, t_{y\,\max}] = \emptyset$.
- $t_{x\,\min} = -\infty, t_{x\,\max} = -\infty$，因此，$[t_{x\,\min}, t_{x\,\max}] \cap [t_{y\,\min}, t_{y\,\max}] = [-\infty, -\infty] \cap [t_{y\,\min}, t_{y\,\max}] = \emptyset$.
- $t_{x\,\min} = -\infty, t_{x\,\max} = +\infty$，因此，$[t_{x\,\min}, t_{x\,\max}] \cap [t_{y\,\min}, t_{y\,\max}] = [-\infty, \infty] \cap [t_{y\,\min}, t_{y\,\max}] = [t_{y\,\min}, t_{y\,\max}]$.

因此，我们并不需要额外的检验 0 的操作；IEEE 的浮点数下它的行为正合我们的要求。到这里，我们已经基本上完成了整个算法、包括实现的一些细节。我们可以有下面的实现方式。

```
template<size_t N, typename T> struct Box {
    Vec<N, T> pMin, pMax; // lower-bound and upper bound

    bool intersect(const Ray<N,T> &ray) const {
        T minT = ray.mint;
        T maxT = ray.maxt;

        // test for all 3 dimensions
        for(size_t i = 0; i < N; ++i) {
            T a = T(1)  / ray.d[i];
            T tmin = (pMin[i] - ray.o[i]) * a;
            T tmax = (pMax[i] - ray.o[i]) * a;

            if (a < 0.0f) std::swap(tmin, tmax);

            minT = (tmin > minT) ? tmin : minT;
            maxT = (tmax < maxT) ? tmax : maxT;
            if (maxT < minT) return false;
        }
        return true;
    }
}
```

在上述算法中，我们注意到我们使用了之前提到过的 `ray.mint` 或者 `ray.maxt` 这两个属性。这两个属性在这里可以用来记录射线实际上可能在包围盒中的部分的那一个线段，它们还有另一个作用是为了处理阴影的化整误差（shadow rounding error），我们在第四章中会继续探讨这个问题。

```
template<size_t N, typename T> struct Ray {
    Vec<N, T> o;
    Vec<N, T> d;
    T mint, maxt;
    Vec<N, T> operator () (T t) const { return o + t * d; }
}
```

在包围盒求交算法中，我们维护 `minT` 和 `maxT` 两个变量，时刻去求不同轴向的 $[t_{\min}, t_{\max}]$ 区间，并求交。我们不去直接除以可能为 0 的 d_x，而是将其先求倒数再用乘法。这是为了防止 $d_x = -0$ 时，我们可能会错误地在第 14 行置换 `minT` 和 `maxT` 的值。

2.2 空间加速结构

2.2.1 包围盒层级

注意到，虽然当一根射线与三角形没有交点时，通过包围盒的方法可以非常方便，但是，若射线与三角形有交点，则反而会比正常的 brute force 方法还要昂贵，毕竟射线-包围盒求交也并不是免费的操作。为了

进一步优化时间性能，我们可以采用**层级式包围盒**（hierarchical bounding boxes）结构。概念上，我们将一个大包围盒进一步分为两个小的包围盒，当射线与包围盒有交点时，我们就进一步与这个大包围盒中的小包围盒求交，确认尽可能少的可能相交的三角形。我们可以迭代式地一直分到一个包围盒只包含一个三角形为止。这个方法在很多地方也被称作**包围体层级**（Bounding Volume Hierarchy, BVH）。

```
class BBHNode : public Surface {
    Box3f m_bounds;

    // need a class for saving the geometry nodes
    shared_ptr<SurfaceBase> m_left;
    shared_ptr<SurfaceBase> m_right;
}
```

在 DIRT 的包围盒层级实现中，它将 `BBHNode` 继承自 `Surface`。在 2.7.4 我们讨论到 `Surface` 时，我们将再回过头来讨论这里的选择。

2.2.2 包围盒层级遍历

假设我们有一个已经分好层级的包围盒集合。每个包围盒都包含两个小包围盒，一个被称为 *left*，另一个被称为 *right*。一个简单的迭代算法就应该是:

```
void intersectBVH(Ray ray, HitInfo &hit) {
    if (m_bounds.hit(ray)) {
        if (isLeaf) intersect(ray, m_bounds);
        else {
            leftChild.intersectBVH(ray, hit);
            rightChild.intersectBVH(ray, hit);
        }
    }
}
```

概念上非常简单。一言以蔽之，就是如果当前有交，就继续往小包围盒上求交，直到包围盒不能再向下划分。

2.2.3 包围盒层级划分

如果包围盒的层级划分得好，那么使用 BVH 来加速射线求交的问题就会取得更好的效果。诚然，现实中的几何体肯定不可能总是非常乖巧地长在各个包围盒的中间，无论我们采取怎样的策略，都一定会遇到一些较慢的情况。我们一般有这么两种策略:

- **自顶向下**（Top-down）：沿着一条轴将几何体们分成两个子集。
- **自下而上**（Bot-up）：迭代地将靠近的物体组成一个子集。

有很多具体的策略。我们会在以后开始讨论各种建造 BVH 的算法时再进一步讨论。

2.2.4 基本几何 `SurfaceBase` 类

那么，对于 BVH 的各个层级，它们需要有一个能够指代其存储的是什么几何图形的数据结构。这个结构本身应该作为一个可以承载子节点的结构，同时也能够快速返回其包围盒。因此，我们先使用 `SurfaceBase` 类来达成这种比较基础的目的。

```cpp
// surface.h
class SurfaceBase {
  public:
    virtual ~SurfaceBase() {}

    // Return the surface's local-space AABB.
    virtual Box3f localBBox() const = 0;

    // Return the surface's world-space AABB, by default just returns the local bounding box.
    virtual Box3f worldBBox() const { return localBBox(); }

    // Add a child surface (if this is an aggregate).
    virtual void addChild(shared_ptr<SurfaceBase> surface) {};

    // Perform any necessary precomputation before ray tracing.
    virtual void build(){};

    virtual bool intersect(const Ray3f &ray, HitInfo &hit) const = 0;
};
```

在这样的写法下，让基础几何本身继承自这个类就是很好的选择。基础几何本身也应该支持求交，并且都应该与其自身的包围盒先进行求交。如果基础几何能够自己携带包围盒的信息，就能使得这些算法变得更方便。因此，我们使用 `Surface` 类去继承 `SurfaceBase`，让它作为基础几何的数据结构。

```cpp
// surface.h
class Surface : public SurfaceBase {
  public:
    Surface(const Transform &xform = Transform()) : m_xform(xform) {}

    virtual ~Surface() {}

    Box3f worldBBox() const override;

  protected:
    Transform m_xform = Transform(); // Local-to-world Transformation
};
```

与 `SurfaceBase` 不同，`Surface` 指代的实际的空间中的几何体，因此它必须携带一个 `Transform m_xform` 以将其模型空间的坐标转换到世界空间。这样，我们可以认为在局部上，每个几何体都是以 (0,0,0) 作为中心的。那么，以向量为例，当我们拿到一个世界空间下的向量 `v`，要将其转换至这个几何的局部空间，我们就可以通过

```cpp
Vec3f localV = m_xform.inverse().vector(v);
```

进行转换。要将一个处于当前几何体局部空间的向量 `u` 转换至世界空间时，我们就可以通过

```cpp
Vec3f worldU = m_xform.vector(u);
```

进行转换。

2.3 交互记录

从上面的讨论中我们已经看出来，整个光线追踪几乎就是基于光线与三角形求交这一基础问题的展开。但是，求交点的目的是为了了解到交点的很多情报——布尔值是否相交只是其中的一个。如果相交了，我们希望知道这个交点在三角形的哪里（这可以通过重心坐标记录），我们希望知道交点位置的法线（通过插值），这个交点在射线的哪个位置（通过参数）等等。

因此，我们需要一个专门的数据结构 `HitInfo` 来存储求交结果，而并非只是返回一个是否有交点的简单的布尔值。

```
// surface.h
struct HitInfo {
    float t;              /// Ray parameter t for the hit
    Vec3f p;              /// Hit Position
    Vec3f n;              /// Geometric normal
    Vec3f sn;             /// Interpolated shading normal
    Vec2f uv;             /// Triangle barycentric coord for texturing

    // Default constructor
    HitInfo() = default;

    /// Param constructor
    HitInfo (float t, const Vec3f &p, const Vec3f &n, const Vec3f &sn, const Vec2f uv):
        t(t), p(p), n(n), sn(sn), uv(uv) {}
};
```

这提供了射线与表面相交的结果。在之后的第十一章中，引入参与介质后，射线求交的对象可能不再是表面而是介质——那个时候我们也会有响应的交互记录数据结构。

2.3.1 射线与球求交

现在我们就可以去考虑如何利用上面的 `HitInfo` 去实现光线与球的求交问题。对于几何形状球，作为一种基本几何，我们有理由给它一个单独的类去处理关于它的构造和方法。显然，球 `Sphere` 应该也是继承于基本图形 `Surface` 的一个派生类。除了 `Surface` 该有的功能以外，球还需要它的半径。

因此，它的头文件定义应该如下：

```
// sphere.h
class Sphere : public Surface {
  public:
    // Constructor
    Sphere(float radius = 1.f, const Transform &xform = Transform());

    // Override base class functions
    Box3f localBBox() const override;
    bool intersect(const Ray3f &ray, HitInfo &hit) const override;

  protected:
    float m_radius = 1.0f; // radius of the sphere
};
```

那么，下面我们就可以在 `sphere.cpp` 中去实现 `Sphere` 类的 `intersect` 方法以计算射线与球求交。

求交点

首先，我们先使用我们在第一章的时候说过的方法，求解二次函数的根以求得交点。

```
// sphere.cpp
bool Sphere::intersect(const Ray3f &ray, HitInfo &hit) const {
    // transform ray to local coordinate
    Ray3f localRay = m_xform.inverse().ray(ray);
    Vec3f oc = localRay.o - Vec3f(0.f,0.f,0.f);
    auto a = length2(localRay.d);
    auto b = dot(oc, localRay.d);
    auto c = length2(oc) - m_radius * m_radius;
    auto discriminant = b * b - 4 * a * c;
    // Discriminant < 0, no root
    if (discriminant < 0) return false;
    // Otherwise solve for roots
    float sq_discriminant = sqrt(discriminant);
    float t1 = (-b - sq_discriminant)/(2 * a);
    // float t2 = (-b + sq_discriminant)/(2 * a); // need the second root in several cases

    hit.p = m_xform.point(localRay(t1));
    // ... other code
    return true;
}
```

在这里，我们要面临几个选择。首先，对于一根直线来说，其求解出来的两只值中，总是更小的那一个才是新的交点。然而，光线是射线，很有可能第一个交点并不存在于射线的正向。这种情况就意味着光线是从球的内部出发的。那么，这个时候我们应该直接返回 `false` 吗？通常情况下并不能这么做：在之后我们会见到的透明物体以及支持散射的物体，光线是很有可能从球的内部走到球的外侧的。因此，在 `t1` 返回的值在射线负向时，我们也依然需要考虑结果 `t2`。对于射线而言，我们总是会存储其 `mint` 和 `maxt` 作为其有效范围，因此，我们只需要检查结果是否在这个范围内即可。

```
// sphere.cpp
bool Sphere::intersect(const Ray3f &ray, HitInfo &hit) const {
    //... OMIT the same part
    float t1 = (-b - sq_discriminant)/(2 * a);
    float t2 = (-b + sq_discriminant)/(2 * a);

    // Intersection parameter
    float hitT = 0.f;
    if(t1 <= ray.mint || t1>= ray.maxt) {
        // check for t2
        if(t2 <= ray.mint || t2 >= ray.maxt) return false;
        else hitT = t2;
    } else hitT = t1;

    hit.p = m_xform.point(localRay(hitT));
    // ... other code
    return true;
}
```

处理交点法线

由于球上任何一点，其表面法线都应该与从球心出发指向该点的向量同向。而我们又假设我们的球心一直都处于原点。因此，要求任何一点的顶点法线，只需要用该点坐标的归一化值即可。但是，需要注意的是，如果光线是从求的内部射出，那么交点表面法线应该是从交点指向球心的方向。

```cpp
// sphere.cpp
bool Sphere::intersect(const Ray3f &ray, HitInfo &hit) const {
    //... OMIT the same part
    Vec3f localNormal = localRay(hitT) / m_radius;
    Vec3f worldNormal = normalize(m_xform.normal(localNormal));
    bool fromInterior = dot(normalize(localRay.d, localNormal)) > 0;

    if(fromInterior) worldNormal *= -1;
    hit.gn = worldNormal;
    // ... other code
    return true;
}
```

处理求交信息的 uv 坐标

2.3.2 射线与四边形求交

```cpp
// quad.cpp
bool Quad::intersect(const Ray3f &ray, HitInfo &hit) const {
    // compute ray intersection (and ray parameter), continue if not hit
    auto tray = m_xform.inverse().ray(ray);
    if (tray.d.z == 0) {
        return false;
    }
    auto t = -tray.o.z / tray.d.z;
    auto p = tray(t);

    if (m_size.x < p.x || -m_size.x > p.x || m_size.y < p.y || -m_size.y > p.y) {
        return false;
    }

    // check if computed param is within ray.mint and ray.maxt
    if (t < tray.mint || t > tray.maxt) {
        return false;
    }

    // project hitpoint onto plane to reduce floating-point error
    p.z = 0;

    Vec3f gn = normalize(m_xform.normal({0, 0, 1}));
    Vec2f uv(p.x / (2 * m_size.x) + 0.5f, p.y / (2 * m_size.y) + 0.5f);

    // if hit, set intersection record values
    hit = HitInfo(t, m_xform.point(p), gn, gn, uv, m_material.get(), m_medium_interface.get(),
    this);
    return true;
```

```
}
```

2.3.3 射线与三角形求交

```
// mesh.cpp
bool singleTriangleIntersect(const Ray3f &ray, const Vec3f &p0, const Vec3f &p1, const Vec3f &p2, const Vec3f *n0,
                             const Vec3f *n1, const Vec3f *n2, const Vec2f *t0, const Vec2f *t1, const Vec2f *t2,
                             HitInfo &hit, const Material *material, const MediumInterface *mi,
                             const SurfaceBase *surface) {
    // Find vectors for two edges sharing v[0]
    Vec3f edge1 = p1 - p0;
    Vec3f edge2 = p2 - p0;

    // Begin calculating determinant. Also used to calculate U parameter
    Vec3f pvec = cross(ray.d, edge2);

    // If determinant is near zero, ray lies in plane of triangle
    float det = dot(edge1, pvec);
    if (abs(det) < 1e-8f) {
        return false;
    }
    float inv_det = 1.0f / det;

    // Calculate distance from v[0] to ray origin
    Vec3f tvec = ray.o - p0;

    // Calculate U parameter and test bounds
    float u = dot(tvec, pvec) * inv_det;
    if (u < 0.0 || u > 1.0) {
        return false;
    }

    // Prepare to test V parameter
    Vec3f qvec = cross(tvec, edge1);

    // Calculate V parameter and test bounds
    float v = dot(ray.d, qvec) * inv_det;
    if (v < 0.0 || u + v > 1.0) {
        return false;
    }

    // Ray intersects triangle. Compute t
    float hitT = dot(edge2, qvec) * inv_det;
    if (hitT < ray.mint || hitT > ray.maxt) {
        return false;
    }

    // Compute geometric and shading normals
    Vec3f gn = normalize(cross(p1 - p0, p2 - p0));
```

```
    Vec3f bary(1 - u - v, u, v);
    Vec3f sn;
    if (n0 != nullptr && n1 != nullptr && n2 != nullptr) { // Do we have per-vertex normals
    available?
        // We do -> dereference the pointers
        sn = normalize(bary.x * (*n0) + bary.y * (*n1) + bary.z * (*n2));
    } else {
        // We don't have per-vertex normals - just use the geometric normal
        sn = gn;
    }

    Vec2f uv;
    if (t0 != nullptr && t1 != nullptr && t2 != nullptr) {
        uv = bary.x * (*t0) + bary.y * (*t1) + bary.z * (*t2);
    } else {
        uv = {u, v};
    }

    // Because we've hit the triangle, fill in the intersection data
    hit = HitInfo(hitT, ray(hitT), gn, sn, uv, material, mi, surface);
    return true;
}
```

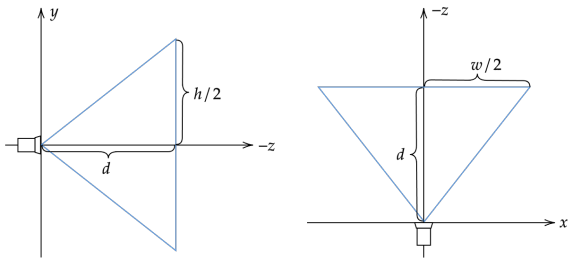
```
bool Triangle::intersect(const Ray3f &ray, HitInfo &hit) const {
    INCREMENT_INTERSECTION_TESTS;

    auto i0 = m_mesh->F[m_faceIdx].x, i1 = m_mesh->F[m_faceIdx].y, i2 = m_mesh->F[m_faceIdx].z;
    auto p0 = m_mesh->V[i0], p1 = m_mesh->V[i1], p2 = m_mesh->V[i2];
    const Vec3f *n0 = nullptr, *n1 = nullptr, *n2 = nullptr;
    if (!m_mesh->N.empty()) {
        n0 = &m_mesh->N[i0];
        n1 = &m_mesh->N[i1];
        n2 = &m_mesh->N[i2];
    }
    const Vec2f *t0 = nullptr, *t1 = nullptr, *t2 = nullptr;
    if (!m_mesh->UV.empty()) {
        t0 = &m_mesh->UV[i0];
        t1 = &m_mesh->UV[i1];
        t2 = &m_mesh->UV[i2];
    }

    return singleTriangleIntersect(ray, p0, p1, p2, n0, n1, n2, t0, t1, t2, hit, m_mesh->material
    .get(),
                                   m_mesh->medium_interface.get(), this);
}
```

2.4 相机

整个渲染流程中，还有一个需要被表示的数据结构就是我们的相机。本笔记中，相机和人眼是同义词，且都指代的是一个虚拟的**小孔相机**（pinhole camera）。

2.4.1 Camera 类

相机类 Camera 是另一个单独的类，它主要负责生成用以追踪的光线。我们认为它自己有一个 Transform，定义了它与世界坐标的关系。在相机空间（camera space）中，它看向 $-z$ 轴，并且在 $-d$ 处有一个大小为 $w \times h$ 的成像平面，也就是我们的**图像**（image）。



我们先提供一个透视相机的实现。感兴趣的读者可以以此 Camera 类作为基类，然后编写正交相机、透视相机等相机的派生类。

```
// camera.h
class Camera {
  public:
    /// Construct a camera from json parameters.
    Camera(const json &j = json()) { }

    /// Return the camera's image resolution
    Vec2i resolution() const { return m_resolution; }

    Ray3f generateRay(float u, float v, const Vec2f &lensSample) const;

    bool hitSensor(const Ray3f &ray, Vec2f &pixelPos) const;

    float focalDistance() const { return m_focalDistance; }

  private:

    Transform m_xform = Transform();        ///< Local coordinate system
    Vec2f m_size = Vec2f(1, 1);             ///< Physical size of the image plane
    float m_focalDistance = 1.f;            ///< Distance to image plane along local z axis
    Vec2i m_resolution = Vec2i(512, 512);   ///< Image resolution
    float m_apertureRadius = 0.f;           ///< The size of the aperture for depth of field
    std::shared_ptr<const Medium> m_medium; ///< Volumetric medium that camera is looking at
};
```

2.4.2 构造相机

确认高度

首先是通过 json 文件作为参数的构造函数。我们从 json 文件中读取的信息决定了我们相机的参数。其中，vfov 参数决定了相机的**垂直视场**（vertical field of view），这是一个角度，其定义为

$$\texttt{vfov} = 2 \cdot \arctan\left(\frac{h/2}{d}\right)$$

换句话说，我们也可以认为

$$h = 2 \times \tan\left(\frac{\texttt{vfov}}{2}\right) \times d.$$

确认宽度

相机提供了图像平面的分辨率，而分辨率是一个二维整数向量。我们可以定义一个**长宽比** (aspect ratio)，

$$\text{aspect ratio} = \frac{w}{h} = \frac{\texttt{m\_resolution.x}}{\texttt{m\_resolution.y}}$$

那么，使用长宽比我们就可以计算出宽度。

$$w = \texttt{aspect ratio} \times h$$

实现

有了上面的这些算式，我们就可以轻松地写出 `camera.cpp` 中的构造函数了。

```
// camera.cpp
Camera::Camera(const json &j = json()) {
    m_xform = j.value("transform", m_xform);
    m_resolution = j.value("resolution", m_resolution);
    m_focalDistance = j.value("fdist", m_focalDistance);
    m_apertureRadius = j.value("aperture", m_apertureRadius);

    if (j.contains("medium")) {
        m_medium = parseMedium(j.at("medium"));
    }

    float vfov = 90.f; // Default vfov value.
    vfov = deg2rad(j.value("vfov", vfov)); // Override this with the value from json

    m_size.y = 2 * tan(vfov / 2) * m_focalDistance;
    m_size.x = float(m_resolution.x) / m_resolution.y * m_size.y;
}
```

2.4.3 生成光线

我们希望相机类提供一个这样的功能：我们给定图像平面上的一个位置 (u, v)，相机能够生成一条从相机出发经过该位置的射线。

即使相机是小孔相机，我们依然认为相机的**光圈** (aperture) 具有实际的大小。关于理想的点相机与基于物理的小孔相机的区别我们将在本章的附加小节中介绍。因此，相机发射出来的光线不应该总是由代表光圈中心的点 (这个点在相机空间中始终处于原点)，而是应该以原点为圆心，在平面 $z = 0$ 上，以光圈半径为半径进行一个圆盘采样，然后以采样点作为起点。在圆盘上均匀采样的方式我们将在第八章中详述，在这里我们先假设我们有一个函数 `randomInUnitDisk()`，其用一个 `Vec2f` 作为输入，返回一个圆盘上均匀的随机采样点。

由此，我们可以写出我们的实现。

```cpp
// camera.cpp
Ray3f Camera::generateRay(float u, float v, const Vec2f &lensSample) const {
    Vec2f disk = m_apertureRadius * randomInUnitDisk(lensSample);
    Vec3f origin(disk.x, disk.y, 0.f);
    Vec3f direction(m_size.x * (u / static_cast<float>(m_resolution.x) - 0.5f),
                    m_size.y * (0.5f - v / static_cast<float>(m_resolution.y)),
                    -m_focalDistance);
    return m_xform.ray(Ray3f(origin, direction - origin)).withMedium(m_medium);
}
```

2.4.4 光线图像求交 *

在第十二章中，我们将会接触光追踪以及双向路径追踪。之所以我们要给相机提供这个功能，是因为在这两个光线追踪算法中，光线是从光源出发，最终到达人眼的。因此，我们也需要相机类提供 `hitSensor()` 功能：给定一条光线 `ray`，我们检查其是否击中图像平面，如果击中图像平面，则修改传入的参数 `pixelPos`，并返回 `true`，否则则不修改，返回 `false`。

一个重要的观察是：如果图像平面与光线起点的 z 值的差值为正，这说明射线起点比图像平面在 z 轴方向上更远离相机，因此是从远离相机的那一侧进入图像平面的。在这种情况下，光线从远处照向相机，因为相机始终看向 $-z$ 轴，所以此时光线方向的 z 值应该就是正的。因此，定义数值

$$t = (d - \mathbf{o}_{\text{ray,z}})/\mathbf{d}_{\text{ray},z}.$$

如果 t 为负值，则说明上述的情况不成立，那么就说明要么光线是从图像平面远离相机的一侧射向更远的一点，要么就是在图像平面靠近相机的一侧射向相机。两种情况下，光线都不会集中图像平面。

另外，我们注意到，如果光线和图像平面有交点，那么上面的 t 值实际上就是该交点在射线上的时间参数 t。因此，交点的位置可以直接通过 $\mathbf{o} + t\mathbf{d}$ 获得。

有了以上的观察，我们就可以完成 `hitSensor` 的实现了。

```cpp
// camera.cpp
bool Camera::hitSensor(const Ray3f &ray, Vec2f &pixelPos) const {
    // transfer the world position to camera position
    float nearPlane = -m_focalDistance;
    float ar = float(m_resolution.x) / float(m_resolution.y);

    Ray3f localRay = m_xform.inverse().ray(ray);

    if(localRay.d.z >= nearPlane && localRay.d.z < 0.f) return false;

    float t = (nearPlane - localRay.o.z) / localRay.d.z;
    if(t <= 0) return false;

    // else we record the pixel position
    Vec3f hitPoint = localRay(t);
    if(hitPoint.x < -m_size.x / 2 || hitPoint.x > m_size.x / 2||
       hitPoint.y < -m_size.y / 2 || hitPoint.y > m_size.y / 2) {
        return false;
```

```
    }

    pixelPos.x = (hitPoint.x + m_size.x / 2) / m_size.x * m_resolution.x;
    pixelPos.y = (hitPoint.y + m_size.y / 2) / m_size.y * m_resolution.y;

    return true;
}
```

2.5 景深以及基于物理的相机 *

Chapter 3

纹理

这一节中我们将会运用各类数学工具来解构关于纹理的深入知识点。

3.1 纹理

3.1.1 定义

在 PBR 中，纹理拥有严格的数学定义。

定义（纹理）将三维空间中的点 $\mathbf{p}=(x,y,z)$ 映射到二维单位正方形 $[0,1]^2$ 上的坐标 (u,v) 上的函数 $f:\mathbb{R}^3\to[0,1]^2$ 被称为**纹理**（texture）。即形同

$$f(x,y,z)=(u,v),x,y,z\in\mathbb{R},u,v\in[0,1]$$

的函数就是纹理。另外，纹理本身也是一个函数。它将二维单位正方形 $[0,1]^2$ 上的点映射至某一类型的值上。纹理的数学定义就说明了纹理所对应的不一定要是颜色。将三维上的点映射到二维图像上取色的过程显然是一种纹理采样，但是二维图像上理论上可以存储各种各样的值，顶点偏移、法线偏移、法线倍率、金属度、粗糙度、甚至是皮肤敏感度……只要是可以用数据来记录的本质上都可以。

3.1.2 实现

对于不同种类的纹理，它们肯定都有不同的作用。但是基本地，无论是什么类型的纹理，我们都是将颜色作为信息编码进纹理的结构中的。所以，对于一个纹理的基类，我们需要它能根据纹理坐标 (u,v) 返回这个颜色。

```
class Texture {
    public:
        // basic destructor
        virtual ~Texture() = default;

        // basic color return
        virtual Color3f value(float u, float v, const Vec3f &p) const = 0;
}
```

3.2 纹理采样率

3.3 纹理坐标生成

接下来我们要做的就是清楚如何将一个物体上的点 $\mathbf{p}$ 映射到二维单位正方形上，换言之，也就是纹理映射函数的具体内容。

3.3.1 球面映射

球天生具有很好的参数化方式，也就是极坐标。对于一个点 $\mathbf{p}=(\theta,\varphi)$，应用**球面映射**（spherical mapping）

$$f(\mathbf{p})=\begin{bmatrix}\frac{\varphi}{2\pi}\\ \frac{\theta}{\pi}\end{bmatrix}$$

3.3.2 圆柱映射

圆柱体展开后是一张方形。对于一个在圆柱体上的点 $\mathbf{p}=(x,y,z)$，我们单独处理它的上下顶面。如果在圆柱体身上，那么我们应用**圆柱映射**（cylindrical mapping）

$$f(\mathbf{p})=\begin{bmatrix}\frac{\varphi}{2\pi}\\ \frac{y}{h}\end{bmatrix}$$

其中，φ 是该点处于横截面圆上的角度。

3.3.3 平面映射

沿着 xy 平面将点 $\mathbf{p}=(x,y,z)$ 映射至大小为 $w\times h$ 的纹理，应用**平面映射**（planar mapping），

$$f(\mathbf{p})=\begin{bmatrix}\frac{x}{w}\\ \frac{y}{h}\end{bmatrix}$$

基于平面映射，我们也可以将物体置于方块内，对每个面实行一次平面映射，也称作进行**六面方体映射**（6-side Cubical Mapping）。

3.4 纹理贴图处理

3.4.1 纹理寻址

寻址算法

在着色操作中，当你需要从纹理中读取出当前顶点/像素需要的数据时，我们就得知道在纹理的哪里寻找。上面的映射函数已经给了我们将 (x,y,z) 转化到 (u,v) 坐标的方法了。但是，纹理本身并不是连续定义的，纹理也是一个 $n_x\times n_y$ 的方阵，其中的每一个元素被称为一个**纹素**（texel）。因此，我们还需要将 $(u,v)\in[0,1]^2$ 坐标转化到 $(i,j)\in[0,n_x]\times[0,n_y]$ 坐标，以获得纹理 `tex[i][j]` 上的数据。

一个比较简单的方法就是采用纹素的中心点指代它所处的这个 1×1 的方格。对于一个 $n_x \times n_y$ 的纹理，我们可以采用

$$i = un_x - 0.5,$$
$$j = un_y - 0.5$$

的计算方式。显然，这里得出的 i, j 依然可能不是整数。我们有各种方法进一步处理，例如

- 直接取整。这种方法也被称作**最近点查找**（nearest neighbor lookup）。
- 做**双线性插值**（bilinear interpolation）（甚至是**三线性插值** (trilinear interpolation)）。

我们会在之后详细描述这些方法。

特殊情况处理

特别地，如果我们计算出来的 i, j 超出了其取值范围，我们也有几个不同的选择，例如

- 直接**限制**（clamp）至范围。超出取值范围的值我们当做 0 处理。
- 将纹理当成一种周期信号——当 i, j 超出取值范围时，我们就认为是在该纹理四周的一样的纹理上采样。

第二种做法也常被称作纹理的**平铺**（tile）。为了使得平铺看起来平滑，尝尝会使用左右看起来可以无缝连接的纹理。

现在已经知道对于一个像素，我们具体要去纹理上的哪个方格取值了。可是，现实中我们几乎不可能遇到像素和纹素一一对应的情况。有两种情况是我们需要考虑的。

1. **纹理过大**（Magnification）：一个纹素的大小比一个像素要大。这时，我们就会有多个像素被映射到同一个纹素上。
2. **纹理过小**（Minification）：一个纹素的大小比一个像素要小。这时，我们的一个像素就可能会覆盖到多个纹素。

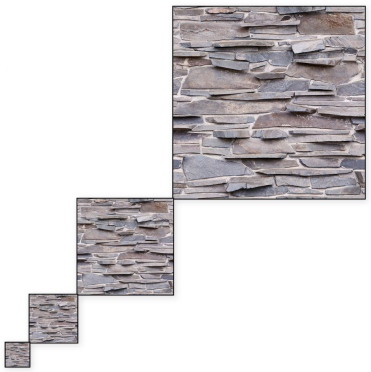
3.4.2 MIPMap

MIPMap 用来解决纹理过小的问题，也就是一个像素会覆盖到多个纹素的问题。当像素覆盖到多个纹素时，一个直观的想法就是计算它们覆盖到的值的（高斯）均值。然而，如果每次都要进行这样的计算，性能就会非常低下，因为求高斯卷积是非常昂贵的操作。对于纹理而言，我们可以先将高斯核滤波过的结果存储在内存中，并减小其尺寸，然后使用对应大小的卷积结果采样，这个方式就叫做 **MIPmap**。

MIPMap 常用图像金字塔的方式来构建，即存储 $2^n \times 2^n$ 大小的处理过的纹理贴图。使用这种方式多占用的空间至多只有原始最高清贴图的 1/3，其中每一个 n 所对应的就是一个 MIPMap 层级。

计算 MIPMap 层级

我们的核心思想是在采样时，采样区域可以覆盖当前层级 MIPMap 上的四个纹素。因此，假设我们使用纹理采样时，映射到的原最高清晰度纹理上的区域为 $[i_{00}, i_{10}] \times [j_{00}, j_{10}]$，则水平对 i 采样率为 $\frac{\Delta i}{\Delta x} = i_{10} - i_{00}$，



对 j 采样率为 $\frac{\Delta j}{\Delta x} = j_{10} - j_{00}$，垂直对 i 采样率为 $\frac{\Delta i}{\Delta y} = i_{01} - i_{00}$，对 j 采样率为 $\frac{\Delta j}{\Delta x} = j_{01} - j_{00}$。设

$$L_x^2 = (\frac{\Delta i}{\Delta x})^2 + (\frac{\Delta j}{\Delta x})^2, L_y^2 = (\frac{\Delta i}{\Delta y})^2 + (\frac{\Delta j}{\Delta y})^2,$$

则我们所需要的层级为

$$D = \log_2 \sqrt{\max(L_x^2, L_y^2)}$$

3.4.3 纹理滤波器

双线性插值

在完成采样之后，我们获得了四个纹素，最直观的方法就是对它们进行**双线性插值**。

三线性插值

上述的双线性插值会面临一个问题，在纹理使用不同层级的 MIPMap 时，层级与层级之间会有清晰的区分；为了消除这种明显的层级清晰度区分，我们会采用**三线性插值**，即在不同层级之间再应用一次线性插值。

3.4.4 RIPMap

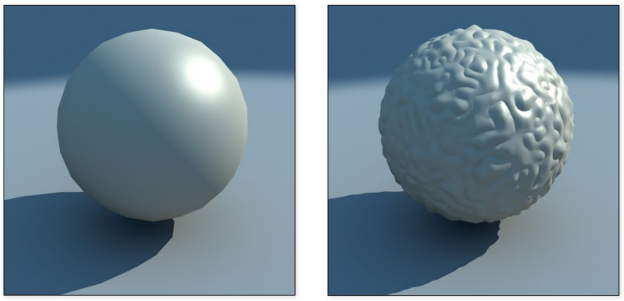
还有一种改进 MIPMap 的方式——RIPMap。在现实工业中，我们很少是以一个从顶向下垂直观看纹理的方式，因此当纹理处于倾斜角度时，总有一个方向会受到压缩。解决纹理压缩时导致的采样可以使用各向异性滤波，也可以使用 RIPMap，即存储不同比例的、在不同方向上采用不同压缩率的纹理。

3.5 几何修饰纹理

上面说过，纹理贴图可存储的数据绝不止有颜色一种。而且事实上，工业界使用纹理来修饰几何体本身的顶点或者表面法线数据的做法是非常常见的。这里主要研究两种不同的纹理。

3.5.1 位移贴图

将表面沿着（插值后的）法线方向进行位移的距离编码成颜色然后存入纹理映射中的贴图就叫做**位移贴图**（Displacement Mapping）。



观察上述示意图[1]中物体表面的轮廓，物体上的点的位置确实有被更新到新的位置。

假设原表面可被参数化为 $\mathbf{p}(u,v)$，其上某一点 $\mathbf{p}$ 对应的值为 h，其插值后的法线 $\mathbf{n}$，则该点在实际参与运算时，我们认为它的位置为

$$\mathbf{p}_d = \mathbf{p} + h\mathbf{n}$$

在更新位置之后，我们还要进一步更新新的点对应的法线的位置，通常这个法线没有一个固定解，我们认为它与该点在 u,v 上的偏导数的叉乘成正比，即

$$\mathbf{n}_d \propto \frac{\partial \mathbf{p}_d}{\partial u} \times \frac{\partial \mathbf{p}_d}{\partial v}.$$

注意，上述的方式并不一定总能找到解，因为没有任何条件限制 $\frac{\partial \mathbf{p}_d}{\partial u}$ 与 $\frac{\partial \mathbf{p}_d}{\partial v}$ 不会平行。如果它们平行，则其对应的叉乘也就成为零向量，那就没有意义了。事实上，这两个偏导数的叉乘具有明确的几何意义。假设该曲面的方程为 $F(\mathbf{p})$，则其对应的曲面积分为

$$\int_S F(\mathbf{p}) dA(\mathbf{p})$$

对于面积的微分，通过微积分原理我们得到

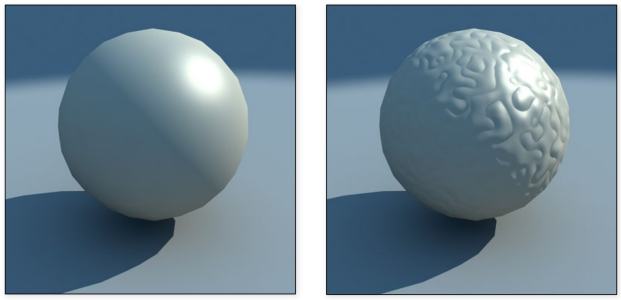
$$dA(\mathbf{p}) = \left\| \frac{\partial \mathbf{p}_d}{\partial u} \times \frac{\partial \mathbf{p}_d}{\partial v} \right\| dudv$$

其中，$\left\| \frac{\partial \mathbf{p}_d}{\partial u} \times \frac{\partial \mathbf{p}_d}{\partial v} \right\|$ 就是积分换元中的雅可比项。雅可比项告诉我们，当我们在参数空间上取一个微小区域并将其映射到曲面上时，这个区域的面积会如何变化。具体来说，雅可比项的绝对值是从参数空间到曲面的面积扩张因子。

3.5.2 凹凸贴图

凹凸贴图（Bump Mapping）虽然也对法线进行修改，但是它并不修改物体上点的位置。

[1] www.spot3d.com



可以观察上述示意图的表面轮廓，可以注意到物体上的点的位置并没有被更新。

凹凸贴图虽然在真实度上略逊于位移贴图，但是它可以简化光线与物体求交的过程，因为物体上的点的位置并没有被更新。

3.5.3 环境贴图

环境/反射贴图（Environment/Reflection Mapping）是将一个图片作为反射光、环境光的来源，用它来简化环境建模的步骤。它们将预渲染的背景环境或是反射记录在贴图上，然后在物体表面需要使用环境光或者反射光的时候的采用。

3.6 程序生成式纹理

除了使用预制的纹理贴图，我们也可以让程序在运行时生成等价的纹理函数，例如，

$$C(x) = 0.5\sin(x) + 0.5.$$

然后应用这个函数来做纹理采样。这样的纹理被称作**生成式纹理**（procedural texture）。这样做有几个好处：

- *不用占用内存*。预制的贴图无论多小，它都必须是一个图片格式的文件，那么都会占据空间。
- *理论上可以有无限大的分辨率*。预制的贴图是一个 $n_x \times n_y$ 的方阵，但用程序化生成的贴图则不用限定其长宽具体数值。因此，理论上它可以是无限大的。而且，它们也可以无限地放大或缩小，不会出现走样的情况。

3.6.1 生成式纹理示例

下面是一些简单的生成式纹理的案例。

3D 条纹图

生成（垂直向）黑白相间条纹的纹理算法可以按照如下步骤。

```
Color Stripe (point P) {
    return sin(P.x) >= 0 ? 0 : 1;
}
```

根据 sin 函数的性质，如果我们想用一个变量控制条纹间隔周期，我们可以调整三角函数的角频率。对于三角函数 $\sin(\omega x)$ 而言，其周期为 $2\pi/\omega$。因此，如果我们的函数增加一个 w 输入，我们就可以调整条纹的宽度。

```
Color Stripe (point P, float width) {
    return sin(pi * P.x/width) >= 0 ? 0 : 1;
}
```

在这里，width 越大，条纹的宽度也就越宽。当 width 为 1 时，其周期为 $2\pi/\pi = 2$。更进一步地，如果我们想要一个边缘柔和、模糊的条纹图，而不是黑白分明的条纹图，我们可以使用 lerp 函数，而不是一个简单的取二值。

```
Color Stripe (point P, float width) {
    float t = (1 + sin(pi * P.x/width))/2;
    return lerp(0,1,t);
}
```

2D 棋盘格纹理

黑白相间的、每格边长为 1 的棋盘格也可以用类似的方法生成。

```
Color Checkerboard (point P) {
    int a = floor(P.x);
    int b = floor(P.y);
    return isEven(a+b) ? 0 : 1;
}
```

读者可以自行思考一下如果边长为 w，上面的代码应该怎么修改。

3.7 噪声函数

上述的生成式纹理形式上都非常简单，但是也太过“完美”。现实中，我们希望材质表面有各种各样的磨损、做旧，才会显得更真实。而生成式纹理的一个重要应用就是生成**噪声函数**（noise function）。

定义（噪声函数）噪声函数指的是一个形同 $rand : \mathbb{R}^n \to [0,1]$ 的函数，其中 $n = 1, 2, 3, ...$。

一个良好的噪声函数的性质包括无法观察到明显的重复、旋转不变性和非带限。后面的两个概念我们会在之后介绍。噪声函数通常用作于提升画面的纹理质感、添加随机性，它是在实时渲染以及 PBR 中生成云、山、树等多样且重复的元素中重要的技术手段。我们主要研究几种噪声。

3.7.1 白噪声

白噪声在原理实现上无比简单，不能再简单了。仅仅只有一行代码。

```
Color WhiteNoise (point P) {
    return randf();
}
```

从代码上就能看出来，白噪声的生成与点的位置无关。无论在哪里生成白噪声，我们都只取完全随机的结果。视觉上来看，白噪声不平滑，具有明显的颗粒感。

但是，这样带来的结果就是白噪声不是带限的。我们首先要理解什么事**带限信号**。

定义（带限信号）仅在某个频率区间之内有非零值，而在这个区间之外的频率均为 0 的信号就被称为**带限信号**（band-limited signal）。

考虑到频率的物理意义，任何信号我们都认为下限是 0Hz，所以我们在说一个信号是不是带限的，主要就看它有没有频率上限。

那么，为什么白噪声不是带限的呢？首先，回顾在计算机图形学中我们曾经体积过的二维图像的频率的物理意义。在图像中，我们说图像中的平坦的大色块、缓慢变化的渐变背景等是低频信号，而锐利的边缘、剧烈的颜色变化、细节的纹理等是高频信号。当我们观察一个无限大的含有白噪声的图像时，直观上我们可以认为它势必会拥有各种程度的平坦（或剧烈）变化，因为每个像素都是完全随机的。因此，我们说白噪声是一个非带限信号。

3.7.2 值噪声

下面介绍第一种带限噪声信号，**值噪声**（Value Noise）。值噪声的生成原理也很简单，我们以二维值噪声为例。假设我们有 $n \times n$ 个格子。

1. 首先，现在每个格子中填入一个 $[0, 1]$ 中的随机值。
2. 然后，对于网格内的任意位置，假设这个位置不是刚好落在格子中心。
3. 接下来，利用这个格子周围的 4 个格子，做双线性插值。

值噪声分析

那么，这个噪声为什么是带限的呢？首先，我们分析一下这个信号的频率上限，也就是变化最剧烈的可能性。无非就是相邻的两个格子，一个纯黑，一个纯白。但即使如此，纯黑的像素在其格子内没有变化，纯白的像素在其格子内也没有变化，因此，网格的尺寸就决定了噪声的最高频率成分。另外，在插值过程中，无论是双线性插值还是三线性插值，都有一个很平滑的过渡，也不会出现颜色的剧变。因此，这个噪声是有上限的。

下面提供一个值噪声实现的算法。

```
// randomly permuted array of [0,255], duplicated
const unsigned char values [256*2] = [2, 234, ...];

float noise1D(float x) {
    // &255: bit operation to keep the index to be bounded within in 0-255
    int xi = (int)floor(x) & 255;
    return lerp(values[xi], values[xi+1], x-xi)/128.0 - 1;
}
```

值噪声的缺陷

值噪声已经是工业界用的很多的噪声函数了。但是它也依然有一些明显的缺陷。

首先，值噪声依赖于格状的数据结构，因此其能够表示的细节程度就受到网格分辨率的限制。这意味着在细节层面上可能会缺乏一些多样性，特别是在更大的尺度上观察的时候。

其次，当进行多次查找的时候，值噪声要进行多次插值的特性就会拖慢查询速度。如果是三线性插值，我们就要在一次插值中进行 8 次值查询，而如果是三立方插值，则要进行 64 次。这是很可怕的性能开销。

3.7.3 柏林噪声

下面介绍**柏林噪声**（Perlin noise）。它的基本思想是在格子上记录向量（或是梯度），我们以二维柏林噪声为例。假设我们有 $n \times n$ 个格子。

1. 首先，在每个格子的每个顶点上随机生成一个向量。
2. 然后，对于像素点所在格子的每个顶点，计算该像素点的位置向量与顶点的梯度向量的点积，每个格子就有四个点积值。
3. 使用插值函数（例如三次插值）在四个点积值之间进行插值。
4. 将计算出的噪声值归一化到 [0,1]。

这里不再赘述柏林噪声的实现，与之前值噪声其实很类似。柏林噪声可以生成出更加平滑的噪声，它在空间上是连续的。另外，它可以通过调整频率和振幅来控制噪声的粗细和强度。因此，它非常适合用来生成类似于云彩、水面和山地之类的自然界纹理。

同样类似地，因为柏林噪声也是定义在网格上的，所以它依然有频率上限，是带限信号。即使它可以调整振幅和频率，但是它的上限是明确存在的。

柏林噪声的缺陷

首先，与值噪声类似的格状数据结构会导致查询缓慢的问题。虽然比值噪声快很多（以三次插值为例，值噪声需要查询 4^n 个值，而柏林噪声需要 2^n），但绝对上来说依然很慢。另外，柏林噪声并没有旋转不变的特性。

3.7.4 分形噪声

在数学上，如果我们有一个复杂的函数 $f_s(\mathbf{p})$，它可以通过不同比例的同一个函数 $f(\mathbf{p})$ 缩放相加而成，即

$$f_s(\mathbf{p}) = \sum_i w_i f(s_i \mathbf{p})$$

并且，如果满足当其中一项有更高的频率（即 s_i 更大）时，其振幅则会降低（即 w_i 更小），我们就说 f_s 是 f 的**分形和**（fractal sum）。特别地，当 $s_i = 2^i$，且对应的 $w_i = 2^{-i}$ 时，我们就把上面这个累加式中的每一项都叫做一个**八度**（octave）。

在图形学中，我们将柏林噪声的分形和称作**分形布朗运动**（fractional Brownian motion，fBm）。与分形布朗运动有着同样的计算方式，只不过在计算的过程中取绝对值，得到的噪声就被称为**湍流**（turbulence）。

3.7.5 沃利噪声

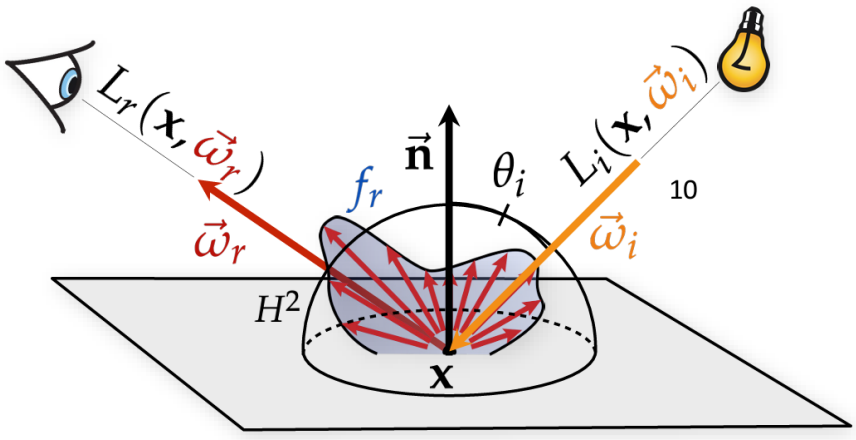
Chapter 4

散射模型

在光线追踪的步骤中，处于射线-物体求交之后的一部就是计算实际的光照了。这一步就被称为**着色**（Shading）。具体地说，当光线击中某个表面，我们就要计算这个位置显示的颜色。同时，我们必须尊重物理——真实世界中，一个物体的表面有可能吸收光，有可能反射光，有可能折射光，也有可能自发光。因此，这些我们都需要加入考虑。

4.1 反射方程

渲染方程是我们用来描述最终颜色的一个表达式，其中有一项用以描述反射，也被我们成为**反射方程**（The Reflection Equation）。反射方程描述的是被反射的辐射亮度（radiance），这个辐射亮度是从各个角度来的入射辐射亮度的半球积分。关于辐射亮度的具体定义我们会在之后的辐射度量学一章严格描述，但是各位读者应该已经了解这个概念。



反射方程具有如下的形式：

$$L_r(\mathbf{x}, \vec{\omega_r}) = \int_{H^2} f_r(\mathbf{x}, \vec{\omega_i}, \vec{\omega_r}) L_i(\mathbf{x}, \vec{\omega_i}) \cos\theta_i d\vec{\omega_i}$$

其中，$\mathbf{x}$ 是我们关注的点，L_r 研究的是从点 $\mathbf{x}$ 出发沿着方向 $\vec{\omega_r}$ 反射的光的辐射亮度。f_r 在反射方程中，通常指的就是**双向反射分布函数**（Bidirectional Reflection Distribution Function），简称 BRDF。BRDF 描述

了在点 $\mathbf{x}$ 上，光线从入射方向 $\vec{\omega_i}$ 到出射方向（也就是 $\vec{\omega_r}$）的反射特性，我们稍后会详细描述。对各个入射方向反射出去的亮度求一个半球积分，就能得到我们的反射出去的亮度了。另外，我们还会见到**双向透射分布函数**（Bidirectional Transmittance Distribution Function），简称 BTDF，用自然语言描述就是在一个点 $\mathbf{x}$ 上，光线从入射方向 $\vec{\omega_i}$ 来到另一个出射方向 $\vec{\omega_o}$ 上，材质将光线透射了多少，BRDF 则描述了将光线反射了多少。我们还会看到**双向次表面散射分布函数**（Bidirectional Surface Scattering Reflectance Distribution Function），简称 BSSRDF。

所有这些 BRDF，BTDF，BSSRDF 被统称为**双向散射分布函数**（Bidirectional Scattering Distribution Functions），简称 BSDF。我们对光线的研究都要依赖各类 BSDF 提供分布。在第五章中我们会详细介绍 BSDF 的计算。

4.2 漫反射模型

在第一种模型中，我们对材质做出一种理想化的抽象：

- 光在各个方向上都被反射。
- 由表面的颜色决定颜色。

在现实中，光在很多材质上并不是在各个方向上均匀地反射的，但是，在一些材质中，比如纸、哑光漆上，漫反射模型几乎可以当成是精确的描述。我们认为光是均匀地被反射到各个方向的。我们也可以说假设光接触的材质表面为**朗伯表面**（Lambertian Surface）。

不同的模型在光线追踪算法中产生的影响主要出现在光线击中物体之后。根据不同的材质着色模型，我们会认为光线接下来走的方向应该要受到影响。下面要考虑的问题有两个：

1. 接下来要从哪个方向追踪下一条光线？
2. 接下来需要追踪几根光线？

首先解决第二个问题。在光线追踪的迭代过程中，我们只能选择一条随机的方向继续我们的迭代。否则，如果每个 `trace()` 都会调用 $n > 1$ 次自己，下一条光线击中另一个物体后又调用 n 次 `trace()`，我们就有可能会调用指数级别的 `trace()`。这是肯定不行的。解决这个问题的方案只有每次只迭代一次，总调用次数才不会指数增长。

接下来回到第一个问题。对于漫反射模型，这个问题也很简单——只需要随机选择一个方向即可，因为漫反射模型中，各个方向的权重其实应该是一样的，所以以重要性采样为策略，每个方向都是等价的，采样其中一个方向即可。但是，随之而来的问题也必须要解决——怎么在一个半球上均匀地随机选择一个方向？

4.2.1 采样朗伯散射

4.2.2 阴影的化整误差

4.3 镜面反射模型

镜面反射（Specular Reflection / Mirror）模型所描述的内容很简单：

- 入射光线与反射光线处于同一个平面。
- 入射角等于反射角。

其中，入射角指的是入射光线与接触面法线的夹角，反射角是反射光线和接触面法线的夹角。在入射光线方向 ω_i 与法线 $\mathbf{n}$ 都是单位向量时，它们的夹角余弦值就是 $\mathbf{n} \cdot \omega_i$。

镜面反射模型认为，反射光的辐射亮度为从接触点 $\mathbf{p}$ 沿着反射光方向 ω_r 做光线追踪得到的固定比例。即

$$L_r = k_r \texttt{trace}(\mathbf{p}, \omega_r).$$

另外，在描述 ω_i 与 ω_r 的时候，我们都使用光线方向。因此，ω_i 是从光源指向接触点的，而 ω_r 是从接触点指向反射方向的。通过简单的几何，我们可以得到 ω_r，ω_i 与 $\mathbf{n}$ 之间的关系。

$$\omega_r = -2\mathbf{n}(\mathbf{n} \cdot \omega_i) + \omega_i.$$

关于镜面反射模型也就没有什么好多说的了。下面是镜面反射模型的伪代码。

```
Color trace(Point p, Ray r) {
    HitInfo hit = surfaces.intersect(r);
    if(hit.isHit) {
        [col, sRay] = hit -> mat -> scatter(r);
        return col * trace(hit.position, sRay);
    }
    return backgroundColor;
}
```

4.4 镜面折射模型

在反射模型之上，更为通用的描述应该是**镜面折射**（specular refraction）模型。为了完全了解镜面折射，我们需要先了解一些背后的光学原理。

4.4.1 物理描述

电磁理论认为，光是电磁场的振动（oscillation）。当一束光到达物质表面时，它刺激组成该物质材质的原子周围围绕着的电子，让它们快速地振动。这些电子导致电磁场内的次级振动，而这些次级振动的叠加也就导致了相长干涉（constructive interference）和相消干涉（destructive interference）。这些就形成了原子反射光的基本机制，并且也将物质根据它们原子的行为划分为基本的三大类：

- **电介质**（dielectrics）。电介质描述的是各种不导电的物质（无论固液气态），例如玻璃、水[1]、矿物质油、空气等。这些物质的电子会与原子紧紧地联系在一起。
- **导体**（conductor）。包括各种金属、合金和半金属（例如石墨）。这些物质的电子相对而言非常自由，并且在金属材质中，光在到达材质表面后很快就被吸收转化为热能（大约在 $0.1\mu m$ 之内）。只有非常非常薄的金属才能让光线通过，因此我们可以在实现时将金属材质视作不透明物质。
- **半导体**（semiconductor）。例如硅和锗。它们有一些电介质和导体的性质。

[1] 虽然可以往水中添加离子（ion）使其导电，但是水分子本身是不导电的。

4.4.2 折射率

如上文所说，当入射光接触物体表面，刺激原子的电子的时候会产生振动。这些振动会导致光在物质内部的行动会被减慢一些，因此，相比于理想真空中的光速 c_0，光线进入物质内部后，速度都会比原来慢一些。减慢的程度就被我们称作**折射率**。

定义（折射率）物质的**折射率**（index of refraction, IOR）η_M 指的是真空中的光速 c_0 与该物质中的光速 c_M 的比值，即

$$\eta_M = c_0/c_M.$$

光进入的物质我们也可以称其为**介质**（medium）。光从一种介质进入另一种 IOR 差距很大的介质中时，会产生相比于差距小时很明显的反射，例如从空气进入钻石中（ΔIOR = 2.42）就会比从空气进入玻璃中（ΔIOR = 1.5）产生明显得多的反射。

关于反射光线的计算和行为，我们已经在 4.3 的镜面反射模型中描述过，这里就不再赘述。

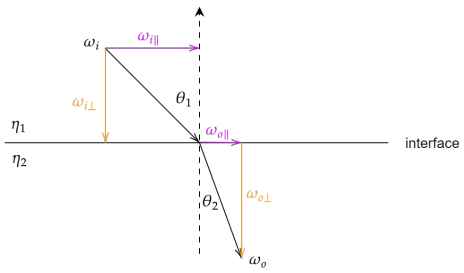
4.4.3 斯涅尔定律

光线从一种介质进入另一种介质时会发生偏折，这种现象就叫做**折射**（refraction）。描述折射光线方向的定律就是**斯涅尔定律**。

定义（斯涅尔定律）光线从（与法线 $\mathbf{n}$ 呈）(θ_1, φ_1) 的角度从 IOR 为 η_1 的介质中进入另一种 IOR 为 η_2 的材质中时，折射光线与法线 $\mathbf{n}$ 呈 (θ_2, φ_2)，它们满足

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2, \text{ 且} \varphi_2 = \varphi_1 + \pi.$$

的关系被称为**斯涅尔定律**（Snell's Law）。



另外，不同波长的光通常也会有不同的被折射的方向，这就会导致**色散**（dispersion）的情况，例如雨后的彩虹，或是白光通过棱镜时产生的光谱。

计算折射方向

首先，我们认为 ω_i 和 ω_o 是两条单位向量[2]，因此 $\omega_o = \omega_i = 1$。那么，根据斯涅尔定律，我们可以得到 ω_o 的平行分量 $\omega_{o\parallel}$

$$\omega_{o\parallel} = \omega_o \sin\theta_2 = \omega_o \frac{\eta_1 \sin\theta_1}{\eta_2} = \frac{\eta_1}{\eta_2}\omega_i \sin\theta_1 = \frac{\eta_1}{\eta_2}\omega_{i\parallel}.$$

[2]在这里上下文明确时，我们用 ω_i 来指代 $||\vec{\omega_i}||$，ω_o 同理。

又因为 $\vec{\omega_{i\parallel}} = \vec{\omega_i} - \vec{\omega_{i\perp}}$，而

$$\vec{\omega_{i\perp}} = (\omega_i \cdot \mathbf{n})\mathbf{n}.$$

因此，

$$\vec{\omega_{t\perp}} = \frac{\eta_1}{\eta_2}(\omega_i - (\omega_i \cdot \mathbf{n})\mathbf{n})$$

折射光线的垂直分量 $\vec{\omega_{o\parallel}}$ 很好计算，它的长度为 1，所以大小就是 $\cos\theta_2$，方向则沿着负的法线。

$$\vec{\omega_{o\parallel}} = -\cos\theta_2\mathbf{n}.$$

最后，我们将垂直与平行分量相加，即可获得折射方向 ω_o.

$$\begin{aligned}\vec{\omega_o} &= \vec{\omega_{o\perp}} + \vec{\omega_{o\parallel}} \\ &= \frac{\eta_1}{\eta_2}(\omega_i - (\omega_i \cdot \mathbf{n})\mathbf{n}) - \cos\theta_2\mathbf{n}.\end{aligned}$$

最后，我们再通过三角函数的关系，以及单位向量的夹角余弦等于其点积的性质，

$$\begin{aligned}\cos\theta_2 &= \sqrt{1-\sin^2\theta_2} \\ &= \sqrt{1-\frac{\eta_1^2}{\eta_2^2}\sin^2\theta_1} \\ &= \sqrt{1-\frac{\eta_1^2}{\eta_2^2}(1-\cos^2\theta_1)} \\ &= \sqrt{1-\frac{\eta_1^2}{\eta_2^2}(1-(\vec{\omega_i}\cdot\mathbf{n})^2)}\end{aligned}$$

因此，折射方向 ω_t、入射方向 ω_i、接触面法线 $\mathbf{n}$ 以及折射率 η_1, η_2 之间的关系为

$$\vec{\omega_o} = \frac{\eta_1}{\eta_2}(\vec{\omega_i} - (\vec{\omega_i}\cdot\mathbf{n})\mathbf{n}) - \mathbf{n}\sqrt{1-\frac{\eta_1^2}{\eta_2^2}(1-(\vec{\omega_i}\cdot\mathbf{n})^2)}$$

其中的所有向量都是单位向量。

全反射

上述的方程中，为了使得 $\sqrt{1-\frac{\eta_1^2}{\eta_2^2}(1-(\omega_i\cdot\mathbf{n})^2)}$ 有意义，根号内的内容必须要非负，换句话说，仅当

$$\frac{\eta_1^2}{\eta_2^2}(1-(\omega_i\cdot\mathbf{n})^2) \leq 1$$

时，这个方向才有意义。而没有意义的情况会发生在当入射角 θ_1 大于临界角 θ_c 时，其中 $\theta_c = \sin^{-1}(\eta_1/\eta_2)$。此时，我们就完全不会看到折射光，而仅只能观察到反射光。这种情况就被称作**全反射**(total internal reflection)。

因此，在最终计算折射的时候，可以参考以下的伪代码。

```
// Return: true if there is a refraction
// Refraction direction is written into the param wo
bool Refract(Vec3f wi, Vec3f n, Vec3f *wo, float eta_1, float n_2) {
```

```
    float eta_r = eta_1 / eta_2;
    float cosTheta_i = Dot(n, wi);

    // Compute Snell's Law
    float sin2Theta_i = static_cast<float>(0, 1 - sqrt(cosTheta_i));
    float sin2Theta_o = sin2Theta_i / sqrt(eta_r);

    // Handle Total Internal Reflection -> No refraction, return false
    if (sin2Theta_o >= 1) return false;

    float cosTheta_o = sqrt(1-sin2Theta_t);
    *wo = -wi/eta_r + (cosTheta_i / eta_r - cosTheta_o) * n;

    return true;
}
```

4.4.4 菲涅尔方程

之前我们的讨论集中在折射光的方向上，还有一个问题没有解决：有多少光会被折射。而**菲涅尔方程**就解决了这个问题。

定义（菲涅尔方程）假设入射光与接触面的法线夹角为 θ_i，折射光与接触面的法线（的负方向）的夹角为 θ_t。光线在入射介质的折射率为 η_i，光线在折射介质的折射率为 η_t。则定义 $r_\perp$ 与 $r_\parallel$ 分别为

$$r_\perp = \frac{\eta_i \cos\theta_i - \eta_t \cos\theta_t}{\eta_i \cos\theta_i + \eta_t \cos\theta_t}.$$
$$r_\parallel = \frac{\eta_t \cos\theta_i - \eta_i \cos\theta_t}{\eta_t \cos\theta_i + \eta_i \cos\theta_t}.$$

菲涅尔方程（Fresnel Equation）指出，此时有 F_r 的光线被反射，$1 - F_r = F_t$ 的光线被折射，其中，

$$F_r = \frac{1}{2}\left(r_\parallel^2 + r_\perp^2\right).$$

Chapter 5

辐射度量学

在第一章的引入中，我们就提及 PBR 的主题是精准地描述物理世界中的光。虽然我们在很多时候忽略光的波动性质，但是对于光的粒子性质我们会尽可能地精准地描述。对于图形学的**辐射度量学**（radiometry）而言，我们重点研究以下几个物理量：光通量、辐射照度和辐射亮度。

5.1 物理量

我们假设光都是由可以用一个三维数对 $(\mathbf{x}, \omega, \lambda)$ 来描述的光子组成的。其中，$\mathbf{x}$ 表示该物质的位置，ω 表示该物质运动的方向（的单位向量），λ 为该物质的波长。

5.1.1 光子与辐射能量

在 PBR 中，我们假设我们研究的对象是光子。在图形学中，我们对光子做出如下定义。

定义（光子）**光子**（photon）是一个沿着直线移动的能量体，不具有波的性质。它具有位置 $\mathbf{e}$，方向 $\hat{\mathbf{d}}$，以及波长 λ。每个光子携带的能量为

$$Q_i = \frac{hc}{\lambda}[J]$$

其中，h 为普朗克常量[1]，c 为真空下的光速[2]，光子能量的单位为 $J = \text{kg} \cdot m^2/s^2$。

在现实中，我们不可能通过 Brute Force 暴力地去追踪每一个光子的行为，因此，我们研究光子成群的宏观行为，这种研究也就是图形学范畴的辐射度量学。所以我们首先关注的就是辐射能量。

定义（辐射能量）**辐射能量**（radiant energy）指的是被（所有波长的）光线 $\mathbf{r}$ 辐射的总能量，即

$$Q = \sum_{i \in \mathbf{r}} Q_i$$

在很多时候，我们也关注一个特定波段（或者是特定波长）上的辐射能量。因此，我们也定义光谱能量。

[1] $h = 6.62606993489 \times 10^{-34} J \cdot s$

[2] $c = 299792458 m/s$

定义（光谱能量）**光谱能量**（spectral energy）指的是处于一定波长范围内的光线 $\mathbf{r}$ 辐射的总能量，即

$$Q_\lambda = \frac{\Delta Q}{\Delta \lambda}$$

当 $\Delta\lambda \to 0$ 时，我们研究的就是特定波长的光谱能量。

$$Q_\lambda = \lim_{\Delta\lambda \to 0} \frac{\Delta Q}{\Delta \lambda} = \frac{dQ}{d\lambda}[J \cdot mm^{-1}]$$

在本笔记中，我们均研究单位波长上的光子行为，即各类光谱物理量。在上下文清晰时，我们会在描述中忽略波长。直观上来说，辐射能量就是被光子击中的总次数乘以单个的光子能量，因为单个光子能量总是常数，所以辐射能量就反映了被光子击中的总次数。

5.1.2 辐射通量

我们接下来定义单位时间内的辐射。

定义（辐射通量）**辐射通量**（radiant flux）指的是单位时间内的辐射能量，即

$$\Phi = \frac{dQ}{dt}[J/s = W] \Leftrightarrow Q = \int_{t_0}^{t_1} \Phi(t)dt$$

直观上来说，辐射通量（不产生歧义时，我们也简称通量）指的就是每秒内被击中的次数。

5.1.3 辐射通量密度

无论如何测量光通量，我们都要面对一个区域去测量通过的光子数量。因此，定义单位面积的光通量——即辐射通量密度，是很符合直觉的。在定义单位面积的通量时，我们除了考虑通过的量，我们也会考虑通过的方向。对于单位时间到达单位面积的光，以及单位时间离开单位面积的光，虽然它们显然有着相同的单位，但确实会有不同的应用场景。

定义（辐射照度）**辐射照度**（irradiance）指的是到达单位面积内的辐射通量，即

$$E(\mathbf{p}) = \frac{\partial \Phi}{\partial A}[W/m^2]$$

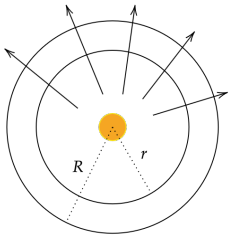
简称辐照度，其中，$\mathbf{p}$ 是我们关注的点（一个极小的接触面）。

定义（辐射出射度）**辐射出射度**（radiant exitance）指的是离开单位面积内的辐射通量。即

$$B(\mathbf{p}) = \frac{\partial \Phi}{\partial A}[W/m^2]$$

简称辐射度（radiosity），其中，$\mathbf{p}$ 是我们关注的点（一个极小的接触面）。

直观上来说，辐射照度指的就是被光照射的表面上每平方米在每秒内被击中的次数（单位面积、单位时间内的辐射能量）。我们通过一个点光源的例子来理解辐射照度。



点光源辐照度

假设我们有一个点光源。如上图所示。

由于能量是在球面内均匀分布，所以我们观察到，在以 r 为半径的球面上的任何一点 $\mathbf{p}_i$，辐照度为

$$E(\mathbf{p}_i) = \frac{\Phi}{4\pi r^2}$$

在以 R 为半径的球面上的任何一点 $\mathbf{p}_o$，辐照度为

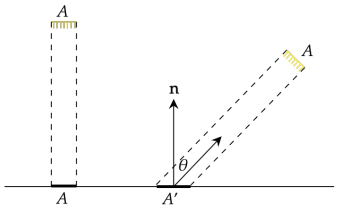
$$E(\mathbf{p}_o) = \frac{\Phi}{4\pi R^2}$$

由此，我们可以观察到，光线的辐照度在点光源上以距离的平方衰减。

朗伯定律

从直觉上来说，光线毕竟反应的是光子的行为，所以如果光线倾斜了，那么光与表面接触的等效面积应该会不同。**朗伯定律**（Lambert's Law）描述的就是这个现象。

定义（朗伯定律）如果光线与接触面不垂直，同样面积的光照射在接触面上就会有更大面积的区域被照亮。



假设光源的面积为 A，光线与接触面夹角为 θ，则接触面被照亮区域的大小就为 $A' = A/\cos\theta$。因此，接触面的入射辐照度为

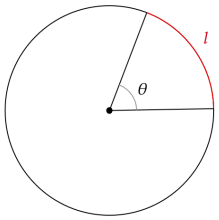
$$E = \frac{\Phi \cos\theta}{A}.$$

5.1.4 立体角与微分立体角

立体角的概念

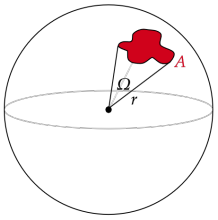
在定义下一个概念辐射亮度之前，我们需要先有一个立体的几何概念，叫做**立体角**。在研究立体角之前，我们先确认一下平面角的定义。

定义（平面角）**平面角**（angle）指的是该角对应圆弧的弧长对圆半径的比值，即 $\theta = l/r$ 或 $l = r\theta$。



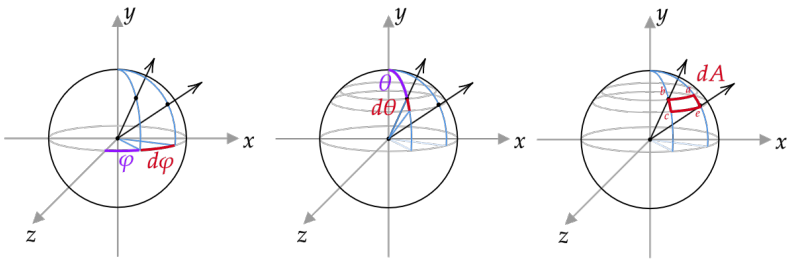
这也是为什么 360° 的弧长为 2π。我们利用类似的方法定义立体角。

定义（立体角）**立体角**（solid angle）指的是该角对应不规则面的面积对球半径的平方的比值，即 $\Omega = A/r^2$ 或 $A = \Omega r^2$。立体角拥有形式单位 steradian，简写为 sr。这是一个无量纲量，量纲为 1，但是不可忽略。



微分立体角

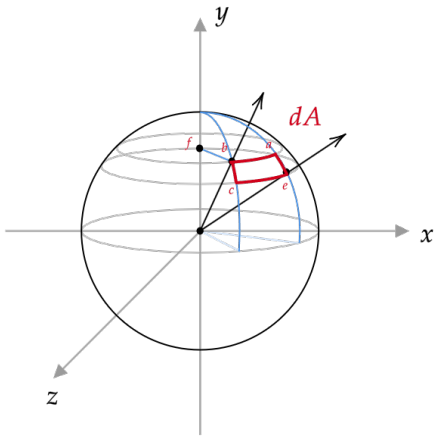
注意到，当一个立体角极小时，其对应的面积也极小，我们可以将此时的立体角看作是一个方向。那么我们就需要知道极小的立体角需要如何定义，也就是需要推导出**微分立体角**（differential solid angle）的表达式。如同下图。



首先，我们要确定出什么样的面积是极小的面积。我们假设从球心指出的一条射线在球面上扫过极小的距离。这个距离在经度范围上扫过 $d\varphi$，在纬度范围上扫过 $d\theta$，如上图。这里，$d\varphi, d\theta \to 0$。在移动很小的情

况下，我们可以认为扫过的这个面积就是一个长方形（图中的 $abce$）。因此，$dA = ||ab|| \cdot ||ce||$.。

根据弧长的定义，$||bc|| = rd\theta$。假设过 ab 的弧所在的球的水平截面（是个圆）与 y 轴的交点为 f，如下图所示。



在这里，$||bf|| = r\sin\theta$。再根据弧长定义，我们就可以得到 $||ab|| = r\sin\theta d\varphi$。

因此，$dA = rd\theta r\sin\theta d\varphi$。根据立体角的定义，$d\omega = r^2\sin\theta d\theta d\varphi / r^2 = \sin\theta d\theta d\varphi$。
我们可以用积分验证我们的思路。对整个球面的所有微分立体角进行积分，我们有

$$\Omega_{sphere} = \int_{H^2} d\omega = \int_0^{2\pi}\int_0^{\pi} \sin\theta d\theta d\varphi = 4\pi.$$

因此，在上下文清晰的时候，我们通常都用方向一词来指代微分立体角。我们描述一个方向上的光（或者说光线上的光），实际上就是在描述一个极小的立体角内的光子的行为。

有了立体角和微分立体角的概念，我们就可以讨论剩下的两个物理量了。

5.1.5 辐射亮度

之前的辐射照度已经我们已经了解了到达一个点的光有多少——单位时间内、单位面积内的辐射能量。但是，我们依然不知道这些光来自于哪个方向。为了精确描述光线，我们还希望知道来自一个特定方向的光线的“数量”。有了立体角以及微分立体角的概念，描述方向就变得清晰了起来。下面就是最后、也是最重要的物理量——辐射亮度。

定义（辐射亮度）**辐射亮度**（radiance）描述的是辐射通量密度的立体角密度。对于一个点 $\mathbf{p}$，来自方向 $\vec{\omega}$ 的入射辐射亮度定义为辐射照度的立体角密度，即

$$L(\mathbf{p} \leftarrow \vec{\omega}) = \lim_{\Delta\omega\to 0} \frac{\Delta E(\mathbf{p})}{\Delta\omega\cos\theta}\left[\frac{W}{m^2\cdot sr}\right]$$

或者等价地，辐射照度是入射辐射亮度对立体角的半球积分。

$$E = \int_{H^2} L(\mathbf{p}\leftarrow\omega)\cos\theta d\omega$$

同理，对于点 $\mathbf{p}$，自方向 $\vec{\omega}$ 离开的出射辐射亮度定义为辐射出射度的立体角密度，即

$$L(\mathbf{p} \to \vec{\omega}) = \lim_{\Delta\omega \to 0} \frac{\Delta B(\mathbf{p})}{\Delta\omega \cos\theta} \left[\frac{W}{m^2 \cdot sr}\right]$$

在上下文清晰时，我们用**辐射亮度**一次统称出射辐射亮度和入射辐射亮度，可以统一地写作 $L(\mathbf{p}, \omega)$。在特别描述从点 $\mathbf{a}$ 到点 $\mathbf{b}$ 的辐射亮度时，我们也可以简写为 $L(\mathbf{a} \to \mathbf{b})$。

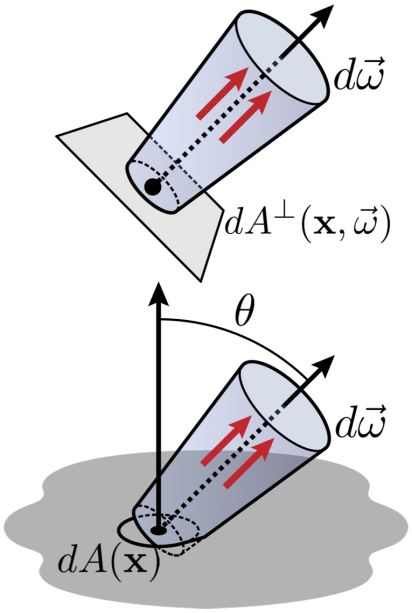
直觉上来说，辐射亮度（以下简称亮度）是通过 $\mathbf{p}$ 和方向 ω 定义的沿着一条光线上的辐射能量。在真空中，对于一个给定的方向，亮度都不会发生变化。我们在下一小节将会给出证明。

通量视角

如果以光通量来描述辐射亮度的话，它是单位立体角、单位垂直区域的光通量。即

$$L(\mathbf{p}, \vec{\omega}) = \frac{d^2\Phi(A)}{d\vec{\omega} dA^{\perp}(\mathbf{p}, \vec{\omega})}$$

如果我们认为微小区域 dA 的法向量为 $\mathbf{n}$，$d\omega$ 与这个法线的夹角为 θ，如下图[3]，



则根据朗伯定律，我们认为

$$L(\mathbf{p}, \vec{\omega}) = \frac{d^2\Phi(A)}{d\vec{\omega} dA \cos\theta}$$

辐照亮度视角描述辐照度与通量

将辐照度用辐射亮度来描述，也就是所有方向照射到某一点上的辐射亮度的积分：

$$E(\mathbf{p}) = \int_{H^2} L(\mathbf{p}, \vec{\omega}) |\cos\theta| d\vec{\omega}$$

其中，$|\cos\theta|$ 是为了处理 $dA^{\perp}$。类似地，再对单位面积上所有点进行积分，就可以得到通量，

$$\Phi(A) = \int_A \int_{H^2} L(\mathbf{p}, \vec{\omega}) |\cos\theta| d\vec{\omega} dA(\mathbf{p})$$

[3]图来自 15-668 2024 Spring Lecture 06 演示文稿。

5.2 物理量性质

5.2.1 辐射度量学中的积分

球面积分计算辐照度

在上文中计算微分立体角时，我们已经验证了

$$d\omega = \sin\theta d\theta d\varphi$$

利用这个关系，我们也就可以将辐照度写成球面坐标下辐射亮度的积分：

$$\begin{aligned} E(\mathbf{p}) &= \int_{H^2} L(\mathbf{p},\omega) d\omega \\ &= \int_0^{2\pi}\int_0^{\pi} L(\mathbf{p},\theta,\varphi)\cos\theta\sin\theta d\theta d\varphi \end{aligned}$$

面积积分计算辐照度

假设在半球外有一个四边形的光源稳定地照射（半球的球面）。这种情况下，使用球面坐标计算积分是不合适的，因为给定球面坐标 (θ,φ)，要计算这个点能否看到光源是一件并不轻松的事情。考虑到立体角的定义本身和面积有关（立体角在半球上的面积占总半球面积的比值），我们完全可以考虑用面积作为积分的换元方式。观察到对于给定光源 A 和所求点 $\mathbf{p}$，

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

其中，dA 是光源上的一个很小的面积（可视作一个点），θ 是 dA 的法线与向量 $dA-\mathbf{p}$ 的夹角。r 是 dA 与 $\mathbf{p}$ 的距离。因此，对于四边形光源，设其上一点为 $\mathbf{p}'$，我们可以将其辐照度定义为

$$E(\mathbf{p}) = \int_A B\cos\theta_i \frac{\cos\theta_o dA}{r^2}$$

其中，B 是光源的辐射出射度（按照上述情况描述应该是定值），θ_i 是 $\mathbf{p}$ 点法线与 $\mathbf{p}'$-$\mathbf{p}$ 的夹角，θ_o 是 $\mathbf{p}'$ 法线与 $\mathbf{p}$-$\mathbf{p}'$ 的夹角。

5.2.2 辐射亮度的性质

辐射亮度是用于生成图像的基本辐射度量物理量。在上文中我们已经定义了辐射亮度，下面我们将讨论辐射亮度的重要性质，它们决定了只有辐射亮度能够成为最终记录为图像上像素颜色的参考值。

沿直线路径不变

在 5.1.3 中的点光源辐照度的案例中，我们看到光线的辐射照度在沿着一条直线向前时，由于对应相同立体角内面积会越来越大，因此会随着距离而衰减。然而，辐射亮度却没有这样的性质。对于单位立体角，在真空中辐射亮度会保持不变。假设被照射的点为 $\mathbf{x}$，光源为 $\mathbf{p}$。数学上，这个性质也被表达为

$$L(\mathbf{x}\to\mathbf{p}) = L(\mathbf{p}\leftarrow\mathbf{x})$$

我们可以通过能量守恒来验证这个性质。

光感设备测量辐射亮度

5.3 辐射度量物理量关系表

5.4 颜色

研究以上这些复杂的物理量的目的依然还是为了我们的渲染结果，而渲染最后总还是要确定屏幕上的像素的颜色。而颜色这个属性反映的就是光的波长。任何的光感设备——无论是数字的、模拟的还是人眼，对于不同波长的光都有不同的敏感程度，我们可以通过光感设备的**光谱灵敏度函数**（spectral sensitivity function, SSF）$f(\lambda)$ 描述。当我们测量入射的光谱通量时，光感设备会产生一个标量的颜色响应，

$$R = \int_{\lambda} \Phi(\lambda) f(\lambda) d\lambda$$

其中，$\Phi(\lambda)$ 是对于特定波长的光的光通量。这个数值被我们称作三**刺激值**（tristimulus value）。

5.4.1 人眼

对我们来说，最真实的渲染需要反映人眼的感知。虽然颜色可以被光的光谱分布描述，但是颜色最终还是人类的主观感知，而并非物理的客观描述。

三**色刺激理论**（tristimulus theory）指出，由于人眼中存在三种不同的**锥状细胞**（cone cells），每一种都能响应特定波长的光，因此人眼能看到的颜色可以被三个三刺激值表征。我们可以用上面提到的颜色响应来记录这三种不同的波长反映，其中，

- 短波长对应光谱上的蓝色频段，$S = \int_{\lambda} \Phi(\lambda) m_1(\lambda) d\lambda$。
- 中波长对应光谱上的绿色频段，$M = \int_{\lambda} \Phi(\lambda) m_2(\lambda) d\lambda$。
- 长波长对应光谱上的红色频段，$S = \int_{\lambda} \Phi(\lambda) m_3(\lambda) d\lambda$。

其中 $m_i(\lambda)$ 是三种锥状细胞的光谱灵敏度函数。

人眼上，短中长锥状细胞的分布是 4:32:64，因此，我们肯定要有一种颜色模型来反映这种区别。通常，我们有两种设计方案：要么我们使用不同的 $m_i(\lambda)$ 来描述这种标量颜色响应，要么我们也可以在同一个像素上用不同数量的红绿蓝马赛克格（这种方案也被称作 Bayer 马赛克，其中绿色最多）。

5.5 双向散射分布函数的计算

在第三章中，我们曾经介绍过 BSDF 的概念，但是忽略了 BSDF 的计算。对于不同的反射、透射、散射模型（换言之，不同的材质），显然都会有不同的 BSDF。在本节中，我们介绍各类 BSDF 的计算。

5.5.1 基本概念

为了能够描述我们观察到的材质，我们按照外观将材质分为下面的四类：

- 漫反射材质（diffuse）。漫反射材质理论上将入射光均匀地散射到环境的每个方向上去。虽然不存在物理上的理想漫反射材质，但是我们可以用漫反射材质去模拟类似于黑板、磨砂表面、纸等物体的材质。

- 光滑镜面材质（glossy specular）。光滑镜面材质会将入射光更多地反射到镜面反射方向上，但也会有反射到其他方向上的光。我们可以用这种材质去模拟塑料、车体表面等物体的材质。
- 理想镜面材质（perfect specular）。理想镜面材质只会将入射光反射到镜面反射方向上，所以主要用于模拟镜子、玻璃等物体的材质。
- 逆反射材质（retroreflective）。逆反射材质主要将光反射到入射光的方向上，一般用来模拟天鹅绒、月球、安全服等物体的材质。

描述材质反射分布的分布函数可能是**各向同性**（isotropic）或是**各向异性**（anisotropic）的。各向同性指的是如果你选择物体表面的一个点，将其绕着表面法线在表面上转动，反射分布不会变化的性质。如果反射分布发生了变化，那就是各向异性。

5.5.2 双向散射分布函数的属性

对于 BSDF 而言，它也有以下重要的物理性质：

- 能量守恒。因此，我们有
$$\int_{H^2} f_r(\mathbf{x}, \omega_i \to \omega_r) \cos\theta_r d\omega_r \leq 1, \forall \omega_i.$$
- 光路可逆。因此，我们有
$$f_r(\mathbf{x}, \omega_i, \omega_r) = f_r(\mathbf{x}, \omega_r, \omega_i)$$

综合二者，我们才能得到对于 BSDF 的一个物理性质：

$$\int_{H^2} f_r(\mathbf{x}, \omega_i \to \omega_r) \cos\theta_i d\omega_i \leq 1, \forall \omega_r.$$

5.5.3 狄拉克 δ 分布

狄拉克 δ 分布的定义与理解

对于很多镜面模型，狄拉克 δ 函数是唯一的可以用来描述反射分布的函数。

定义（狄拉克 δ 分布）若函数 $\delta(x)$ 拥有以下性质：

- $\delta(x) = \begin{cases} +\infty, & x = 0 \\ 0, & \text{Otherwise.} \end{cases}$
- $\int \delta(x) dx = 1.$

则我们称该函数为**狄拉克 δ 函数**（Dirac delta function），也叫做**狄拉克 δ 分布**（Dirac delta distribution）。

狄拉克 δ 函数具有一个重要的性质——采样性质。它可以挑选出在某一点的函数值。假设我们有一个普通的函数 $f(x)$，它在数学上定义良好，可以是任何形式的函数，正弦波、多项式、指数函数，等等。现在，我们想要知道这个函数在某一点 a 的数值。

在数学上，我们通常无法通过直接积分来得到函数在某一点的值。积分主要用来求面积和累加，很少用来找单个点的值。而观察积分

$$\int_{-\infty}^{\infty} f(x)\delta(x-a)dx$$

注意到：

- 当 $x \neq a$ 时，$\delta(x-a)=0$。这是根据狄拉克 δ 函数的性质决定的，因此这些函数值都不会对该积分的结果有任何贡献。
- 当 $x=a$ 时，$\delta(x-a)$ 的值按照定义式无限大。但实际上因为狄拉克 δ 函数的第二个性质，我们知道它的在数轴上的整个积分的结果是 1。

因此，上述积分的结果实际上就是 $f(a)$。

BSDF 中的应用

在镜面反射和折射模型中，我们知道光线只会被反射或折射到一个特定的方向，而不是在多个方向上散射。这种情况下的光线分布可以用狄拉克 δ 函数来理想化地表示，我们会在下面的小节探讨镜面反射折射模型时深入研究。

5.5.4 漫反射模型

漫反射模型的描述就是光线会被均匀地反射到所有的方向上去。因此，对于漫反射模型，BRDF 应该是一个常数值，与入射方向、出射方向、接触点都没有关系。那么，在反射方程中，我们就应该可以将 BRDF 项 f_r 取到积分的外面。

$$\begin{aligned} L_r(\mathbf{x},\omega_r) &= \int_{H^2} f_r(\mathbf{x},\omega_i,\omega_r)L_i(\mathbf{x},\omega_i)\cos\theta_i d\omega_i \\ \Longrightarrow L_r(\mathbf{x},\omega_r) &= f_r\int_{H^2} L_i(\mathbf{x},\omega_i)\cos\theta_i d\omega_i \end{aligned}$$

根据辐射亮度的定义，该积分计算的就是辐照度。

$$\begin{aligned} L_r(\mathbf{x},\omega_r) &= \int_{H^2} f_r(\mathbf{x},\omega_i,\omega_r)L_i(\mathbf{x},\omega_i)\cos\theta_i d\omega_i \\ \Longrightarrow L_r(\mathbf{x},\omega_r) &= f_r\int_{H^2} L_i(\mathbf{x},\omega_i)\cos\theta_i d\omega_i \\ \Longrightarrow L_r(\mathbf{x},\omega_r) &= f_r E(\mathbf{x}) \end{aligned}$$

因为我们知道，L_r 与出射方向无关，仅与入射点的位置有关，因此上式可以进一步简化为

$$L_r(\mathbf{x}) = f_r E(\mathbf{x}). \tag{5.1}$$

假设所有入射光都被反射了，那么我们就知道辐照度就是辐射度。

$$E(\mathbf{x}) = B(\mathbf{x}).$$

再根据辐射度的定义，

$$\begin{aligned} E(\mathbf{x}) &= B(\mathbf{x}). \\ \Longrightarrow E(\mathbf{x}) &= \int_{H^2} L_r(\mathbf{x})\cos\theta d\omega. \end{aligned}$$

而我们知道对于这个积分而言，$L_r(\mathbf{x})$ 的取值是一个常数，因此它也可以从积分中被取出。

$$\begin{aligned} & E(\mathbf{x}) = B(\mathbf{x}). \\ \Longrightarrow & E(\mathbf{x}) = \int_{H^2} L_r(\mathbf{x}) \cos\theta d\omega. \\ \Longrightarrow & E(\mathbf{x}) = L_r(\mathbf{x}) \int_{H^2} \cos\theta d\omega. \end{aligned}$$

最后，此时的积分仅是对半球内所有立体角的积分，其取值为 π。因此，

$$E(\mathbf{x}) = L_r(\mathbf{x})\pi. \tag{5.2}$$

将（5.2）代入（5.1）式中，我们得到

$$f_r = \frac{L_r(\mathbf{x})}{E(\mathbf{x})} = \frac{L_r(\mathbf{x})}{L_r(\mathbf{x})\pi} = \frac{1}{\pi}. \tag{5.3}$$

在现实中，光线并不会全部被反射，有部分可能会被漫反射表面吸收。因此，漫反射的 BRDF 实际上被定义为

$$f_r = \frac{\rho}{\pi}, \rho \in [0, 1].$$

其中，ρ 就是漫反射的**反射率**（albedo）。

5.5.5 镜面反射折射模型

Chapter 6

微表面理论

在 BRDF 的发展过程中，从早期 1970 年代裴祥风提出 Phong 经验模型，到之后慢慢出现基于物理的模型，我们可以研究在这个过程中，光照模型都发生过什么样的变化。其中，**微表面模型**（microfacet models）的出现是很重要的一个进步。微表面模型的核心思想是：一个粗糙的表面可以被视为许多微小的、定向各异的平面（即微表面）的集合，每个微表面都按照镜面反射的规则独立地反射光线。通过考虑这些微表面的分布和方向，微表面模型能够更真实地模拟光线如何在真实世界中的粗糙表面上散射和反射。

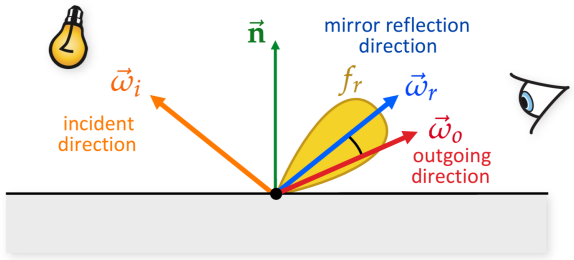
6.1 经验 BRDF 模型

我们可以先从早期的 BRDF 开始研究。注意，在这里的 BRDF 一词本身并不代表严谨的物理关系，这个函数只是一种反应某个方向反射光与某个方向入射光的关系的函数。直到物理模型的出现，BRDF 才逐渐走向科学和严谨。而且事实上，经验 BRDF 模型中的 BRDF 通常并没有能量守恒的性质。注意，在以下模型描述中，我们都认为入射方向 ω_i 和出射方向 ω_o 均从表面上被照亮的点指向光源。

6.1.1 Phong BRDF

Phong 模型指出：反射光会在理想反射方向附近呈现余弦指数随机分布，用数学语言表达，即

$$f_r(\omega_i, \omega_o) = \frac{e+2}{2\pi}(\omega_o \cdot \omega_r)^e$$



其中，e 被称作高光项（specular term）。观察到，随着 e 变大，镜面高光区域会变得更加集中和尖锐。这意味着反射光在一个更小的区域内集中，从而使得高光看起来更亮且更集中，同时，也会使得整个表面看起来

更光滑。反射光方向 ω_r 可以用我们 3.3 中得出的结论计算，

$$\omega_r = 2\mathbf{n}(\mathbf{n} \cdot \omega_i) - \omega_i$$

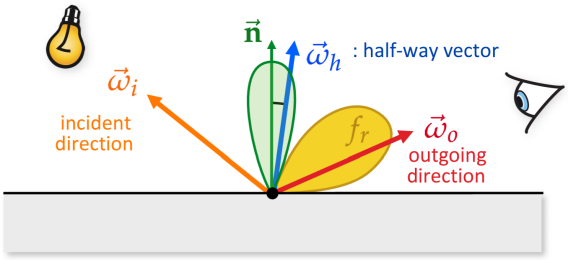
6.1.2 Blinn-Phong BRDF

根据我们计算机图形学中的知识，我们知道 Blinn-Phong 与 Phong 相比最大的进步在于半程向量的提出。

定义（半程向量）**半程向量**（halfway vector）是入射光与反射光形成的夹角的角平分线方向的单位向量，即

$$\omega_h = \frac{\omega_i + \omega_o}{||\omega_i + \omega_o||}.$$

Blinn-Phong 模型指出：出射方向的光受到法线的影响，出射方向并不会随机分布，会随机动的是法线。



因此，Blinn-Phong 的 BRDF 形如

$$f_r(\omega_i, \omega_o) = \frac{e+2}{2\pi}(\omega_h \cdot \mathbf{n})^e.$$

半程向量与镜面反射方向的对比

以上两种 BRDF 在表面接近球形以及接近正视的角度时候区别其实很小。但是，当我们观察地面、墙壁等需要从一个比较刁钻的角度观看的对象时，半程向量表达的 BRDF 就会更加真实。半程向量表达的 BRDF 在当光线几乎与表面平行的时候，会渲染出很狭长的高光，因此会显得更加真实。

6.1.3 局限性

传统的经验模型有以下的限制：

- 并非基于物理。因此有可能会出现类似于能量不守恒、光亮不稳定无迹可寻之类的问题。
- （通常）没有菲涅尔效果，无法反映出光线在不同角度下与物体表面交互时反射率的变化。
- 无法精确模拟很多反光表面的材质。

Blinn-Phong 模型在一定程度上已经走在了正确的方向上，考虑了真正的物理效果。但是显然可以做得更好。

6.2 微表面理论

微表面理论指出，物体表面是由很多小的微表面（tiny facets）组成的。在微表面理论中我们需要解决的问题是在一个宏观表面上，有多少微表面参与了反射。自然，如果我们有一台算力无穷大的计算机，我们可以通过直接把表面建模成微表面然后参与光照计算，但显然目前对我们来说这不是可行的方案。我们的常规想法是通过统计学的结果来描述这种物理现象。

6.2.1 两种基本模型

Torrance-Sparrow 模型

Torrance-Sparrow 模型是微表面理论中的一个重要模型，它被用于模拟光在粗糙表面的反射。这个模型认为表面是由很多微小的凹槽（micro-grove）组成，其中的每一个凹槽都是理想镜面。在 1980 年代，这个模型能够大大提高对金属的渲染质量。

通用微表面模型

通用微表面模型（General Microfacet Model）将 BRDF 分为了三个部分：菲涅尔系数 F、微表面法线分布 D 以及阴影遮罩 G。其形式为

$$f_r(\omega_i, \omega_o) = \frac{F \cdot D \cdot G}{4|(\omega_i \cdot \mathbf{n})(\omega_o \cdot \mathbf{n})|}$$

我们将在下面的三个小节中详细介绍这三个部分。

6.2.2 菲涅尔系数函数

菲涅尔系数和我们之前在散射部分提到的电介质菲涅尔项很相似，但也有些许不同。菲涅尔系数函数定义为散射出射方向 ω_o 与（随机采样的）半程向量（附近的近似法向量）ω_h 的函数。

$$F(\omega_h, \omega_o) = \frac{R_s + R_p}{2}.$$

其中，假设 ω_o 与 ω_h 皆为单位向量，

$$R_s = \frac{(\eta^2 + k^2) - 2\eta(\omega_h \cdot \omega_o) + (\omega_h \cdot \omega_o)^2}{(\eta^2 + k^2) + 2\eta(\omega_h \cdot \omega_o) + (\omega_h \cdot \omega_o)^2}.$$
$$R_p = \frac{(\eta^2 + k^2)(\omega_h \cdot \omega_o)^2 - 2\eta(\omega_h \cdot \omega_o) + 1}{(\eta^2 + k^2)(\omega_h \cdot \omega_o)^2 + 2\eta(\omega_h \cdot \omega_o) + 1}.$$

其中，η 为当前材质的折射率，k 被称为**消光系数**（extinction coefficient）。

6.2.3 微表面法线分布函数

微表面法线分布函数（Microfacet Normal Distribution Function, NDF）是用来描述微表面中的假想法线分布情况的函数。它给出了在特定方向上找到微表面法线的概率，是一个对于归一化投影立体角的概率密度函数[1]。它是一个关于半程向量 ω_h 的函数，满足

$$\int_{H^2} D(\omega_h) \cos\theta_h d\omega_h = 1.$$

[1] 关于概率密度函数的概念我们将会在下一章中介绍。

其中，θ_h 是半程向量 ω_h 与表面法线 $\mathbf{n}$ 的夹角。我们有很多不同的分布函数模型，在这里我们讨论一下 Beckmann 分布模型。

Beckmann 分布模型

Beckmann 分布模型假设表面的微观凸起或凹陷的斜率服从高斯分布。在这个模型中，表面的每一个微表面的法线分布被建模成具有高斯分布的随机变量，其中平均法线方向对应于宏观表面法线。

具体来说，Beckmann 分布使用高斯分布的平方来描述微表面的法线分布——因为斜率是方向的导数，因此是一个二次项。它的形式为

$$D(\omega_h) = \frac{1}{\pi\alpha^2 \cos^4 \theta_h} e^{-\frac{\tan^2 \theta_h}{\alpha^2}},$$

其中，θ_h 是半程向量 ω_h 与表面法线 $\mathbf{n}$ 的夹角。参数 α 描述了分布的宽度，我们可以将其视作粗糙度的度量。当 α 值小时，它表示一个较窄的分布，这样也就对应着更平滑的表面；而较大的 α 值则表示一个较宽的分布，对应于较粗糙的表面。

6.2.4 阴影遮罩函数

最后一项被称作**阴影遮罩**（Shadow Masking）函数。因为我们假设表面有微小隆起和凹陷，所以，当入射光呈一定角度照射在表面上时，一定会有一些微表面处于一些微小隆起与光产生的阴影之中。对于不同的法线分布模型，当然也会有不同的阴影遮罩函数模型，在这里我们依然仅讨论 Beckmann 分布的阴影遮罩。

Beckmann 分布的阴影遮罩模型

服从 Beckmann 分布的阴影遮罩函数是入射光方向 ω_i 与散射光方向 ω_o 的函数，其形式为

$$G(\omega_i, \omega_o) = G_1(\omega_i) \cdot G_1(\omega_o),$$

其中，函数 G_1 是一个关于某一方向的函数，其形式为

$$G_1(\omega) = \frac{2}{1 + \text{erf}(s) + \frac{1}{s\sqrt{\pi}} e^{-s^2}},$$

其中，$s = \frac{1}{\alpha \tan\theta}$，$\theta$ 是观察方向 ω 与微表面法线 $\mathbf{n}$ 的夹角。这里的 erf 函数是高斯函数的积分，其被定义为

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

又看到了喜闻乐见的积分。这样的模型在数学上来说也许很准确，但是在计算机图形学来说只会增加我们计算的复杂度。因此，我们只会使用其近似形式。

$$G(\omega) \approx \begin{cases} \frac{3.535s + 2.181s^2}{1 + 2.276s + 2.577s^2}, & s < 1.6 \\ 1, & \text{Otherwise.} \end{cases}$$

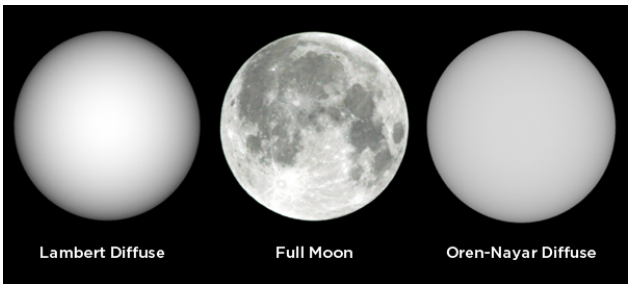
至此，服从 Beckmann 分布模型的 F, D, G 的计算已经全部完成，我们就可以写出 Beckmann 分布模型下的通用微表面模型的 BRDF 了。

6.3 Oren-Nayar 模型

6.3.1 月球的 BRDF

我们考虑一个问题：描述月球的 BRDF 应该是什么样的？

是漫反射模型吗？虽然月球看起来是平滑的，但是通常漫反射模型会在周围出现一个黑色的边缘（rim）。月球并没有在边缘附近颜色变黑的情况，相反几乎是全亮的。在这种时候，我们会使用 Oren-Nayar 模型来描述月球，通常也被称为**粗糙漫反射**（rough diffuse）模型。



6.3.2 粗糙漫反射

在 Oren-Nayar 粗糙漫反射模型中，大部分的概念与微表面模型是相同的，不同点在于我们会假设所有的微表面也是漫反射的。

在经典的朗伯漫反射模型中，表面被假设为完美的漫反射体。当光线射入时，它会被均匀地散射到所有方向，因此散射方向与入射方向实际上是没有关系的。在粗糙漫反射模型上，我们会同时考虑入射光和观察方向相对于表面法线的角度差异，以及它们对光线反射的影响。粗糙漫反射并没有一个分析解，我们可以用下面的近似来计算。

$$f_r(\omega_o, \omega_i) = \frac{\rho}{\pi}(A + B\max(0, \cos\phi_i - \phi_o))\sin\alpha\tan\beta)$$

其中，

$$A = 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)} \qquad B = \frac{0.45\sigma^2}{\sigma^2 + 0.09}$$

$$\alpha = \max(\theta_i, \theta_o) \qquad \beta = \min(\theta_i, \theta_o)$$

以上表达式中，ρ 是表面的反射率，θ_i 与 θ_o 分别是入射光与散射光与表面法线的夹角，ϕ_i 与 ϕ_o 分别是入射光与反射光在表面平面上的投影方向，σ 是表面的粗糙度参数。当 $\sigma = 0$ 时，我们就可以得到理想的朗伯漫反射模型。

6.4 `Material` 类

最后，我们来考虑一下如何将 BRDF 等数值最终反映到我们的计算当中。我们提及的任何一种 BRDF 最终都是对于一类材质的总结，无论是否基于物理。因此，将这些内容编码进 `Material` 类是非常合理的选择。我们从一个 `Material` 基类开始。

```cpp
// material.h
class Material {
public:
    virtual ~Material() = default; // deconstructor

    // return a pointer to a global default material
    static shared_ptr<const Material> defaultMaterial();

    virtual bool scatter(const Ray3f &ray, const HitInfo &hit, ScatterRecord &srec) const {
    return false; }

    // return the amount of emitted light at the surface hitpoint.
    virtual Color3f emitted(const Ray3f &ray, const HitInfo &hit) const { return Color3f(0,0,0);
    }

    // return whether or not this Material is emissive.
    // Base material not emissive.
    virtual bool isEmissive() const { return false; }

    virtual Color3f BSDF (const Vec3f &dirIn, const Vec3f &scattered, const HitInfo &hit) const {
        return Color3f(0.f);
    }

    virtual float pdf (const Vec3f &dirIn, const Vec3f &scattered, const HitInfo &hit) const {
    return 0.f; }

    virtual bool sample(const Vec3f &dirIn, const HitInfo &hit, ScatterRecord &srec) const {
    return false; }
}
```

其中，srec 是一个用来存储与散射相关的信息数据结构。它会记录当前击中的表面的以下信息：

```cpp
// material.h
struct ScatterRecord {
    Color3f attenuation; // the color takeaway from the current scatter
    Vec3f scattered;  // direction of the scattered ray
    bool isSpecular;  // if the material is specular
}
```

在这个基类中，我们重点观察四个函数，其中的几个函数我们会放在之后的章节修改或实现。

- scatter() 函数：这个函数会将 ray 击打在该材质表面上时的散射方向以及此时带上的固有色存入 srec 中。注意，散射意味着出射光线方向可能是反射方向，也可能是折射方向。
- BSDF() 函数：这个函数会计算 BSDF 返回的颜色。
- pdf() 函数：这个函数会计算 sample() 函数采样遵循的概率密度。这个部分我们会放在下一章再详细描述。
- sample() 函数：在表面上被光线击中的点上采样一个方向。根据不同的模型，我们有不同的采样逻辑。

我们先从最简单的模型朗伯漫反射模型为例上手，开始对 Material 派生类的实现。在下面的实现中，我们不考虑构造函数和析构函数这类与 Material 类实现本身没有强关联的内容，实际上我们可以通过各种方法

构造，例如在 DIRT 中，我们是通过读取 json 文件进行构造的。因此，在以下的讨论中，我们忽略对构造函数、析构函数、复制函数、赋值操作符等的讨论。

6.4.1 Lambertian 类

类声明

对于朗伯漫反射模型来说，我们唯一需要知道的就是表面的固有色。我们可以通过 `albedo` 贴图完成这个目的。其余内容与父类几乎没有区别，实现父类定义的虚函数即可。

```
// material.h
class Lambertian : public Material {
public:
    // constructor/deconstructor/... omitted
    shared_ptr<const Texture> albedo;

    bool scatter(const Ray3f &ray, const HitInfo &hit, ScatterRecord &srec) const override;

    Color3f BSDF(const Vec3f &dirIn, const Vec3f &scattered, const HitInfo &hit) const override;

    float pdf(const Vec3f &dirIn, const Vec3f &scattered, const HitInfo &hit) const override;

    bool sample(const Vec3f &dirIn, const HitInfo &hit, ScatterRecord &srec) const override;
};
```

`scatter()` 函数

我们知道在漫反射中，反射的方向是所有方向均匀的。而我们的 `scatter()` 函数会在光线击打到表面时被调用，用以确认下一轮迭代的散射方向。

```
// material.cpp
bool Lambertian::scatter(const Ray3f &ray, const HitInfo &hit, ScatterRecord &srec) const {
    srec.scattered = normalize(randomCosineHemisphere());
    srec.attenuation = albedo->value(hit);
    srec.scattered *= length(ray.d);
    return true;
}
```

保持入射光与散射光的模长相等是保证能量守恒的方法之一，因此我们将散射方向归一化后乘上了入射光线的长度。朗伯漫反射模型中，我们只需要在半球中以某个方式随机采样一个方向即可。这里的 `randomCosineHemisphere()` 函数可以以余弦项加权随机返回半球内的一个方向。我们会在第 8 章中详细讨论为什么要余弦项加权，以及这个函数的实现方式。

`BSDF()` 函数

在第五章讨论朗伯漫反射模型的时候，我们知道其 BRDF 为

$$f_r(\omega_i, \omega_o) = \frac{\rho}{\pi}$$

其中，ρ 为反射率。根据这个公式，就能很快写出 `Lambert::BSDF()` 函数了。需要注意根据朗伯定律，反射光的强度与入射光线和表面法线的夹角余弦成正比，因此我们需要为其乘上一个余弦项，表示了散射方向相对于表面法线的倾斜程度。

```cpp
// material.cpp
bool Lambertian:::BSDF(const Vec3f &dirIn, const Vec3f &scattered, const HitInfo &hit) const {
    Color3f alb = albedo->value(hit);
    return alb * max(0.f, dot(scattered, hit.sn)) / M_PI;
}
```

Chapter 7

蒙特卡洛积分

目前学习的概念中，有许多都涉及大量的积分计算。本章中我们将会介绍通过蒙特卡洛积分的方式来计算积分，并讨论其中的一些变形。

7.1 数值积分

7.1.1 动机

积分并不是初等数学，很多时候我们很难找到一个便于计算的形式将函数的不定积分计算出来。而且，对于大部分的应用场景，我们很少真正需要去计算出函数的不定积分。如果需要计算的对象是给定范围的定积分的话，我们就可以考虑使用一些更简单的方法去近似地计算积分，用累加的手段来代替积分的计算。这种近似方式就被我们叫做**数值积分**（numerical integration）。

7.1.2 方法

梯积法则

首先是最简单的**梯积法则**（trapezoid rule）。在 2D 下，我们可以用多段折线来近似原来的函数曲线，这样的话折线下方的面积就可以看作是多个梯形的面积。用这个面积来模拟原来的曲线下方的面积。

我们假设折线的步长（也就是每次采样的间隔）为 h，折线纵坐标就是原函数在该位置的值。当步数 $n \to \infty$，则误差为 $O(h^2) = O(1/n^2)$。

对于多元函数，我们可以应用 Fubini's Theorem，然后重复使用梯积法则。但是，随着维度变高，虽然误差并不会提高，依然保持 $O(h)$，但是需要的计算量会越来越大。对于 kD 维度，我们需要 $O(n^k)$ 的计算量。

蒙特卡洛方法

蒙特卡洛积分（Monte Carlo integration）是一种基于随机的近似方式，我们在下面会具体介绍。它的一个很优秀的性质是我们仅需要在给定区间的随机位置去计算被积函数 $f(x)$ 的值，就能估算积分 $\int f(x)dx$ 的

值。因此，它不仅便于计算，而且对被积函数本身几乎没有任何要求。另外，它从平均上来说是正确的，我们很快就会解释这句话的意思。

7.2 概率论回顾

在光线追踪中，我们很快会看到随机采样这一工具的强大能力。因此，为了方便接下来的讨论，我们先对概率论进行一些简单的回顾。

7.2.1 概率

虽然**概率**(probability)拥有严格的数学定义,在这里我们还是使用自然语言描述。如果我们知道一个集合 S 包括了随机事件所有可能发生的结果，这些结果分别是 $E_1, E_2, ...$。我们可以认为其中的一个事件 E_i 发生的概率就是当我们进行足够多(接近无穷多)的实验次数时,结果(会接近)为 E_i 的次数占总实验次数的比值。

直观地说就是在进行一次实验时，结果 E_i 发生的可能性就是它的概率。事件 E_i 发生的概率被记作 $\Pr\{E_i\}$。

7.2.2 随机变量

生活中的一些变量它的取值并不是固定的，而是随机的，这些变量被称作**随机变量**（random variables）。例如，我们随机从箱子中取出 X 个球，这个 X 就是随机变量[1]，因为它的取值是有几率不固定的。

在上述这个例子中，X 的取值只会是整数。像这样的随机变量我们也称之为**离散型随机变量**（discrete random variables）。有些随机变量的取值是连续的，取值范围内的任何实数都有几率成为该随机变量的取值，例如明天的气温 T。这样的随机变量我们就称之为**连续型随机变量**（continuous random variables）。

我们通常说随机变量的概率，对于离散型随机变量，指的是事件 $\{X = a\}$ 的概率，其中 a 是某个取值，我们将概率 $\Pr\{X = a\}$ 简写为 $\Pr(X = a)$，或在上下文明确时，直接写作 $\Pr(a)$。对于连续型随机变量而言，对于任意给定值 a，事件 $\{X = a\}$ 发生的概率都是 0，只有讨论对于某一区间 $[a_1, a_2]$ 时，事件 $\{X \in [a_1, a_2]\}$ 的概率才有意义。类似地，我们将概率 $P(\{a_1 \leq X \leq a_2\})$ 简写为 $\Pr(a_1 \leq X \leq a_2)$。

对于两个随机变量，我们认为如果它们自身的概率互不影响对方的取值概率，那么就说它们是*独立的*(independent)。此时，我们可以用联合概率 $\Pr(x, y)$ 来表示事件 $\{X = x, Y = y\}$。它满足

$$\Pr(x, y) = \Pr(X = x) \Pr(Y = y)$$

对于两个不独立的随机变量，它们的概率受到彼此的影响，因此我们要用*条件概率*（conditional probability）来计算这种情况。它满足

$$\Pr(x, y) = \Pr(x) \Pr(y|x)$$

其中，概率 $\Pr(y|x)$ 的意义是：在给定 $X = x$ 已经发生的前提下，事件 $\{Y = y\}$ 的概率。

[1]对于随机变量，我们通常使用斜体大写字母表示。

7.2.3 概率质量函数

对于离散型随机变量，给出其每个取值的概率的函数就叫做**概率质量函数**（probability mass function, pmf）。pmf 在图形学中的应用价值有限，但是能够帮助我们理解有关连续型随机变量的概念。

7.2.4 概率密度函数

概率密度函数的概念可以说是概率论中最基础、同时也是最重要的概念。无论是从严谨的数学语言还是从直观的自然语言描述，我们都需要对概率密度函数有最扎实的理解。我们从累积分布函数的概念开始。

定义（累积分布函数）设 X 是一个随机变量。对于任意实数 $x \in (-\infty, \infty)$，函数

$$F(x) = \Pr\{X \leq x\}$$

就叫做随机变量 X 的**累积分布函数**（cumulative distribution function, cdf）。对于事件 $x_1 < X \leq x_2$，其对应的概率则为

$$\Pr\{x_1 < X \leq x_2\} = F(x_2) - F(x_1).$$

有了累积分布函数的概念，我们就可以定义连续型随机变量 X 的概率密度函数了。

定义（概率密度函数）设 $F(x)$ 为连续型随机变量 X 的累积分布函数。假设存在非负的可积函数 $f(x)$，使得对于任意实数 x 有

$$F(x) = \int_{-\infty}^{x} f(t)dt,$$

则称 $f(x)$ 为 X 的**概率密度函数**（probability density function, pdf），记作 $X \sim f(x)$。

首先注意在英文表达中 pdf 以及 cdf 对应的 d 的意义不同。cdf 中的 d 是分布，而 pdf 中对应的则是密度。概率密度函数用数学语言描述其实并不直观。在理解之前，我们要注意对于连续型随机变量来说几个基本的事实：

1. 对于任何实数 a，$\Pr\{X = a\} = 0$。

2. 我们无法描述单个可能取值对应的概率，但我们可以通过 cdf 计算某一区间 $(x_1, x_2]$ 的概率。谨记，一旦你的脑海中出现了“我想求 $X = x_i$ 的概率”的时候，就要反省。连续型随机变量对应单个取值的概率永远是 0，求这样的概率没有任何实际的意义。

3. 对于 cdf，我们有 $F(x_2) - F(x_1) = \int_{x_1}^{x_2} f(t)dt$，因此 pdf 的定积分对应的则是该区间发生的概率。

4. 对于 pdf，固定的一个点 x_0 其对应的函数值 $f(x_0)$ 并没有明确的物理意义。一定要理解的话，可以想象成随机变量的取值落在 x_0 一个极小的邻域上的概率。有物理意义的只有定积分。

所以，形象地理解概率密度函数：它定义了随机变量 X 取值落在某个区间内的概率，可以将概率密度函数想象为描述随机变量值分布的“形状”或“轮廓”。它也能告诉你随机变量取值在不同区域的相对可能性。

7.2.5 期望

由于我们在计算机图形学中需要处理的随机变量大多数都是连续型的，因此我们这里只讨论连续型随机变量的期望。

定义（期望）对于连续型随机变量 X，设其概率密度函数为 $p(x)$，若积分

$$\int_{-\infty}^{\infty} xp(x)dx$$

绝对收敛，则称该积分为 X 的**期望**（expectation）或**均值**（mean），记作 $E(X)$。

直观上理解期望，我们要从它的别名——均值来理解。期望本质就是事件结果以其发生的概率为权重进行加权平均。如果是离散型随机变量，期望的理解非常简单，但是连续型牵涉到取单一值概率为无穷小的问题，因此，我们对 $xp(x)$ 的值进行积分。

另外，期望有一个很好的性质：如果一个随机变量 X 的取值服从概率密度函数 $p(x)$，那么关于 x 的函数 $f(X)$，其期望为

$$E[f(X)] = \int f(x)p(x)dx$$

也可以将 $E[f(x)]$ 简写为 $Ef(x)$。观察到若 $f(x) = x$ 则我们也能得到上面期望的定义。

线性性质

期望有一个很好的线性性质，即

$$E[X + Y] = E[X] + E[Y],$$

这是对于任何的随机变量 X 和 Y 来说的，随机变量之间不需要相互独立。更一般地，

$$E\left[\sum_{i=1}^{n} f(X_i)\right] = \sum_{i=1}^{n} E[f(X_i)]$$

7.2.6 方差

方差（variance）记作 $V(X)$，有两种理解方式，一种是随机变量与期望的差（注意，这本身也是一个随机变量）的平方的期望。

$$V[X] \equiv E[X - E[X]^2]$$

另一种理解方式是随机变量的平方的期望与期望的平方的差。

$$V[X] \equiv E[X^2] - [E[X]]^2$$

这两个定义可以通过代数过程证明，在此不赘述。选择比较容易理解的方式记住就行。前者可能概念上比较直观，后者会更便于计算。

方差的性质

对于任何随机变量，我们有

$$V[aX] = a^2V[X]$$

对于 n 个独立随机变量，我们有

$$V\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} V[X_i]$$

7.2.7 多维随机变量

既然有了将实数作为随机变量取值的想法，自然也会有将向量作为随机变量取值的想法。在这里，我们只从概括的层面建立一个**多维随机变量**（multidimensional random variable）的直觉，并不会严格定义。

假设多维空间 S（例如，二维平面、三维体积）可以通过某种方式测量（例如二维的面积、三维的体积），其中的一个区域其规模为 μ（例如二维球面上一个面积为 μ 的区域）。我们可以定义一个 pdf，$p(\mathbf{x}) : S \to \mathbb{R}$，那么，随机点 $\mathbf{x} \sim p$ 处于 $S_i \subset S$ 中的概率

$$\Pr(x \in S_i) = \int_{S_i} p(\mathbf{x}) d\mu.$$

由于这个函数的计算完全取决于 S 是什么（比如如果是个面积，则 $d\mu$ 可以认为是 $dxdy$，又或者比如是个微分立体角，则可以认为是 $\sin\theta d\theta d\phi$），因此这里不对多维随机变量做过多解释。

如果有一个多元函数 $f(\mathbf{x})$ 服从概率密度函数 $p(\mathbf{x})$ 的分布，我们也可以得到类似于一元时的关系。

$$Ef(\mathbf{x}) = \int_S f(\mathbf{x}) p(\mathbf{x}) d\mu.$$

7.2.8 期望估计

假设我们有 N 个相互独立但是同时服从同一个概率密度函数的随机变量 $X_1, X_2, ..., X_N$。这样的随机变量们也被称为**独立同分布随机变量**（independent identically distributed random variable, iid）。我们可以通过 N 次随机采样获得的值的平均估计这个随机变量的期望，即

$$E(X) \approx \frac{1}{N} \sum_{i=1}^{N} X_i.$$

定义（大数定律）当 N 趋于无穷大时，上式的值与实际期望相等的概率为 1，即

$$\Pr\left\{E(X) = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} X_i\right\} = 1.$$

这一规律被称为**大数定律**（Law of Large Number）。

用自然语言描述就是，只要采样数量足够多，随机采样获得的均值就是该随机变量的期望。这也符合期望的定义。

7.3 蒙特卡洛积分

下面我们介绍积分的另一种数值估计法——**蒙特卡洛方法**（Monte Carlo）。

7.3.1 蒙特卡洛方法

蒙特卡洛积分（Monte Carlo Integration）是一种基于随机的数值积分计算方式，它可以用来估算任意函数的积分，即使这个函数不连续。由期望估计部分的知识，我们知道，

$$Eg(\mathbf{x}) = \int_{x\in S} g(\mathbf{x})p(\mathbf{x})d\mu \approx \frac{1}{N}\sum_{i=1}^{N} g(\mathbf{x}_i).$$

但是，上式有一个比较大的问题。通常我们需要估算的积分是单独的一个函数，例如 f 的积分，而不会是两个函数 gp 的积的积分。我们可以通过 $f = gp$ 的代换方式，修改上面的积分，

$$\int_{x\in S} f(\mathbf{x})d\mu \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(\mathbf{x}_i)}{p(\mathbf{x}_i)}.$$

由上，我们得出了**蒙特卡洛估值**的计算方式。下面以一元函数 $f(x)$ 为例。

定义（蒙特卡洛估值）对于函数 $f(x)$ 在区间 $[a,b]$ 的定积分，我们可以通过**蒙特卡洛估值**（Monte Carlo Estimator）进行估算。该估值通过以下方式计算：
定积分：$\int_a^b f(x)dx$;
随机变量：在区间 $[a,b]$ 上随机采样 $X_i \sim p(x)$;
蒙特卡洛估值[2]：$F_N = \langle F\rangle = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}$，其中 $p(x)$ 被我们称作**采样分布函数**（sampling distribution function）。

注意，在蒙特卡洛估值中，有一个隐含的限制条件——当 $f(X_i) \neq 0$ 时，$p(X_i) \neq 0$。否则就会出现除零错误。除此之外，我们可以选择任何 $p(x)$ 作为蒙特卡洛估值的随机变量分布。

Example 7.1. 编写函数 *integrate()* 计算函数

$$F = \int_a^b e^{\sin(3x^2)}dx$$

在区间 *[a,b]* 上服从均匀分布的蒙特卡洛估值。

Solution. 在区间 $[a,b]$ 上均匀分布的随机变量的密度函数为

$$p(x) = \frac{1}{b-a}$$

因此，其蒙特卡洛估值为

$$F_N = \frac{b-a}{N}\sum_{i=1}^{N} f(X_i) = \frac{b-a}{N}\sum_{i=1}^{N} e^{\sin(3X_i^2)}.$$

[2]在本笔记中，我们会使用 F_N 和 $\langle F\rangle$ 两种记法。

因此，我们的代码如下：

```
double integrate(int N, double a, double b) {
    double Xi, sum = 0.0;
    for (int i = 0; i < N; i++) {
        // randomly generate a Xi in [a,b]
        Xi = a + randf() * (b-a);
        sum += exp(sin(3*x*x));
    }

    return (b-a) * sum / double(N);
}
```

■

7.3.2 正确性分析

要判断这个估计是否合理，我们可以从其期望来讨论。如果平均上来说它的计算是正确的，那么我们就有理由相信这是正确的估计。下面我们来计算这个估值的期望。

$$
\begin{aligned}
E[F_N] &= E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] && \text{（根据蒙特卡洛估值的定义）} \\
&= \int_a^b \frac{1}{N}\sum_{i=1}^{N}\frac{f(x)}{p(x)}p(x)dx && \text{（根据期望的定义）} \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b \frac{f(x)}{p(x)}p(x)dx && \text{（积分的性质）} \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b f(x)dx && \text{（假设 } p(x)\neq 0\text{）} \\
&= \int_a^b f(x)dx
\end{aligned}
$$

因此，F_N 的期望的确就是 $f(x)$ 在区间 $[a,b]$ 上的定积分。像这样的期望与实际值相同的估计值，我们说它是**无偏**（unbiased）的。这样的估计值并不依赖于样本数量的大小，即使我们使用小量的样本，结果的均值依然是正确的。

另外，蒙特卡洛估计值还有一个特征：随着样本数量的增加，它最终会越来越接近真实值。对于一些估计，它们的期望不一定是正确的，但是也依然可以通过增大样本容量来提高准确性，这样的估计被我们叫做**一致**（consistent）的。显然，这两种概念并不冲突，一个估计可以同时一致和无偏——蒙特卡洛估值就是无偏且一致的。

另外，即使蒙特卡洛估计值是无偏且一致的，其采样结果肯定也会存在与均值不同的情况，那么也就会出现方差。在之后的章节中，我们也会讨论减少这个方差的方法。

7.3.3 误差分析

同样，根据定义来计算这个估值的方差。

$$\begin{aligned} V[F_N] &= V\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] \\ &= \frac{1}{N^2}\sum_{i=1}^{N}V\left[\frac{f(X_i)}{p(X_i)}\right] \\ &= \frac{1}{N^2}\sum_{i=1}^{N}V[Y_i] \\ &= \frac{1}{N}V[Y] \end{aligned}$$

在这里，我们用随机变量 $Y_i = \frac{f(X_i)}{p(X_i)}$ 来分析方差。可以注意到蒙特卡洛估值方差的几个性质。

- 误差本身与维度无关。无论维度提升多少，蒙特卡洛估值的误差都是相同的。
- 当 $N \to \infty$ 时，误差会趋近于 0。也就是我们上面所说的一致性。

7.3.4 收敛分析

7.3.2 的正确性分析中，我们已经验证了：

$$E[F_N] = F = \int_a^b f(x)dx$$

并且我们也提及蒙特卡洛估计值也是一个一致的估计，因此，提高样本容量可以有助于结果收敛至正确答案。随之而来的一个问题就是，如何使得使得结果更接近正确答案。

随机变量加权和

一般地，假设有函数 G，它是 N 个随机变量 $g(x_1), g(x_2), ..., g(x_N)$ 的加权和，其中每个 x_i 具有相同的概率密度分布函数 $p(x)$。这样的 i 个随机变量 x_i 称为**独立同分布**（independent identically distribution, i.i.d.）变量。设 $g_i(x)$ 表示函数 $g(x_i)$，则

$$G = \sum_{j=1}^{N} w_j g_j$$

根据期望的线性属性，我们知道

$$E[G(x)] = \sum_j w_j E[g_j(x)]$$

考虑权重 w_j 都相同，并且和为 1 的情况（即 $w_j = 1$），那么我们有

$$\begin{aligned} G(x) &= \sum_{j=1}^{N} w_j g_j(x) \\ &= \sum_{j=1}^{N} \frac{1}{N} g_j(x) \\ &= \frac{1}{N}\sum_{j=1}^{N} g_j(x) \end{aligned}$$

那么，$G(x)$ 的期望值为：

$$\begin{aligned} E[G(x)] &= \frac{1}{N}\sum_{j=1}^{N} E[g_j(x)] \\ &= \frac{1}{N}\sum_{j=1}^{N} E[g(x)] \\ &= \frac{1}{N} N E[g(x)] \\ &= E[g(x)] \end{aligned}$$

因此，G 与 $g(x)$ 拥有相同的期望，我们可以用 G 来估计 $g(x)$ 的期望值，G 也就是 $g(x)$ 的**估计值**（estimator）。

根据定义，G 的方差为

$$\mathrm{Var}[G(x)] = \mathrm{Var}\left[\sum_{i=1}^{N} \frac{g_i(x)}{N}\right]$$

下面，我们再给出协方差的概念。

定义（协方差）随机变量 X, Y 的**协方差**（covariance）为

$$\mathrm{Cov}[X, Y] = E[XY] - E[X] \cdot E[Y].$$

一般地，方差满足

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y] + 2\mathrm{Cov}[X, Y].$$

在独立随机变量的情况下，随机变量彼此之间的协方差为 0，也就使得上式变成形如线性的形式，

$$\mathrm{Var}[X + Y] = \mathrm{Var}[X] + \mathrm{Var}[Y].$$

对于若干个独立随机变量的线性组合，我们也就会有

$$\mathrm{Var}[ax] = a^2\mathrm{Var}[x]$$

基于 G 中的 x_i 彼此都是 i.i.d. 变量的事实，我们可以得到 G 的方差为

$$\begin{aligned} \mathrm{Var}[G(x)] &= \sum_{i=1}^{N} \mathrm{Var}\left[\frac{g_i(x)}{N}\right] \\ &= N\frac{\mathrm{Var}[g(x)]}{N^2} \\ &= \frac{\mathrm{Var}[g(x)]}{N} \end{aligned}$$

也就是说，随着 N 的提升，G 的方差是会减小的。

蒙特卡洛方法的收敛

对于蒙特卡洛估计值 F_N 而言，根据我们刚才的结论，

$$\mathrm{Var}[F_N] = \frac{1}{N}\int \left(\frac{f(x_i)}{p(x_i)} - F\right)^2 p(x)dx$$

随着 N 的增加，方差将随 N 线性减小。而估计量中的误差与标准差 $\sigma = \sqrt{\mathrm{Var}[F_N]}$ 成正比，因此标准差随着 $\sqrt{N}$ 而减小。事实上，蒙特卡洛技术的问题之一正是这个缓慢的收敛速度——我们需要 4 倍的样本数量才能将蒙特卡洛计算的误差减小一半。

Chapter 8

采样策略

屏幕上的像素最终是入射光线的辐射亮度 L_i 的表征。然而我们知道辐射亮度是一个定义在相机成像空间的连续函数，而我们的屏幕却是一个二维方阵，是离散的。因此，屏幕上显示的内容与实际的光照一定会有偏差。这个偏差的程度就决定了我们成像的质量。另外，在上一章中，我们也曾经提到过蒙特卡洛积分中，选用不同的 pdf 去采样被积函数所得到的方差会有所不同，在这一章中，我们会重点关注减少方差的手段。

8.1 采样理论

8.1.1 信号处理

我们首先需要很好地理解采样和走样两个概念在图形学中的意义。这两个词都是信号处理的专有名词，也有严格的定义，所以我们在这一部分先学习信号处理的基本概念。

我们目前处理过很多的连续函数。然而，我们知道计算机用 bit 作为单位，本质上是不可能完美呈现连续函数的。因此，现实中我们最常用的方法就是在这条连续函数上选择足够多的点，再通过这些点去重建（reconstruct）这个函数。首先，先让我们了解一系列的概念：

定义（时域）描述物理信号对时间的关系，就是**时域**（Time Domain）。在时域分析中，自变量为时间 t，因变量为待分析数据 S。

定义（频域）描述物理信号对频率的关系，就是**频域**（Frequency Domain）。在频域分析中，自变量为频率 f，因变量为待分析数据 S。图 8.1[1]中，上为时域信号，下为频域信号。

定义（采样）对于一个连续的模拟信号 S，从连续时域上将其转换到离散时域上成为离散信号的过程叫做**采样**（sampling）。

定义（重建）对于一个连续的模拟信号 S，将其离散的采样点转换成一个连续函数的过程叫做**重建**（reconstruction）。

[1] https://upload.wikimedia.org/wikipedia/commons/4/4f/Triangle-td_and_fd.png

定义（走样）对于两个不同的连续的模拟信号 S_1, S_2，若用相同的采样方式将它们从连续时域上转换到离散时域时获得的两个离散信号 $\Sigma_1 = \Sigma_2$，则称发生了信号**走样**（aliasing）。

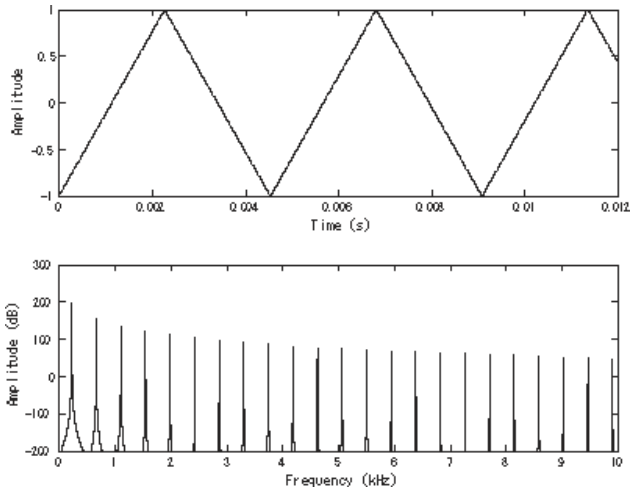


图 8.1: 时域与频域信号的区别

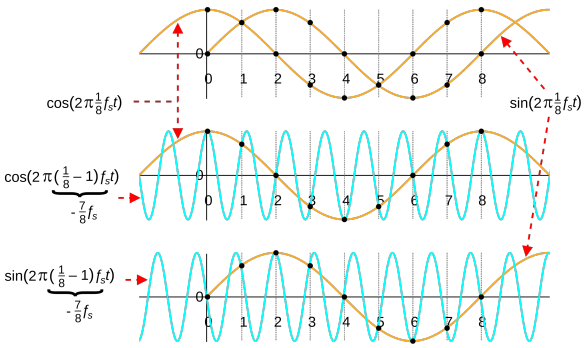


图 8.2: 信号走样

例如，在图 8.2[2]中，橙色信号和蓝色信号是两条不同的信号，但是如果我们采用每 1 秒采样的采样方式的话，它们获得的离散值就是相同的。在重建时，我们就无法区分这两个信号，这就是走样。

8.1.2 傅里叶变换

在本笔记的范围中，我们不对**傅里叶变换**（Fourier Transform）做过多的介绍。傅里叶变换是傅里叶分析的基础，它的作用是将函数从时域转换至频域。通常，它会将函数变换成为正弦信号（sinusoid）的叠加，而我们知道正弦信号是有频率的，因此其可以将一个原来定义在时域上的函数转换为不同频率信号的叠加。

[2]https://en.wikipedia.org/wiki/Aliasing#/media/File:Aliasing_between_a_positive_and_a_negative_frequency.svg

对于一个一维函数 $f(x)$，其傅里叶变换为

$$F(\omega) = \int_{-\infty}^{\infty} f(x) e^{-i2\pi\omega x} dx,$$

其中，我们有 $e^{ix} = \cos x + i \sin x, i = \sqrt{-1}$。

8.1.3 理想采样与重建

有了频域分析的基础，我们就可以来定义什么是理想的采样以及理想的重建了。

8.2 随机均匀采样

当我们提及随机采样（random sampling）时，我们实际上也需要考虑采样服从的概率密度函数。在图形学中，均匀采样是最为重要的一个话题，我们主要考虑以下的均匀采样：

1. 在 $[0,1]$ 区间上的均匀采样。这是其余采样的基础。
2. 在 $[a,b]$ 区间上的均匀采样。
3. 在圆盘或球上的均匀采样。
4. 在半球上的均匀采样。
5. 在更复杂的几何体内的均匀采样。

其中，1. 和 2. 虽然在概念上是非常简单的，但是在计算机上涉及**伪随机数生成**（pseudo-random number generator, PRNG）[3]的问题。在这里，我们假设函数 `randf()` 能够生成一个 $[0,1]$ 区间的、足够随机的伪随机数，对于 `randf()` 的实现原理暂时先不讨论，有兴趣的同学可以阅读本章附加节随机数生成部分。PRNG 的一个主要限制是它只能生成 [0,1] 区间上的均匀分布，因此，对于 3. 及之后的均匀分布，我们就需要借助其它的理论手段。

8.2.1 拒绝采样

拒绝采样（rejection sampling）是一种基于概率的统计技术，用于从复杂的概率分布中生成随机样本。这种方法特别适用于那些难以直接采样的分布。它的核心思想是我们根据某种简单的分布随机采样一些点，然后拒绝其中不满足复杂分布条件的点。

圆盘上随机选点

在这里我们不证明我们做法的正确性。但是从直观上来思考：如果我们希望我们采样的点在圆盘面上是均匀分布的，那么它也应该在圆的外接正方形上均匀分布。这样的话，我们就可以通过先在外接正方形上均匀采样，然后拒绝其中不在圆盘内的点即可。

```
Vec2f v;
do{
    v.x = 1-2*randf();
    v.y = 1-2*randf();
} while (dot(v,v) > 1)
```

[3]虽然很多随机算法的前提都是真随机，但是良好的伪随机函数已经足以满足我们的要求，当然，如果一定要真随机数，我们也可以从 www.random.org 获得基于原子衰变或大气噪声的真随机数。

注意，我们这里使用的 `do-while` 循环中，循环的条件实际上是我们拒绝的条件——只要不达成这个条件，我们就一直采样，直到采样到符合这个条件的点。

单位球内随机选点

在单位球内的随机均匀选点的方式与圆盘极其相似，我们只需要在球的外切立方体内均匀采样，然后拒绝其中不在球内的点即可。

```
Vec3f v;
do{
    v.x = 1-2*randf();
    v.y = 1-2*randf();
    v.z = 1-2*randf();
} while (dot(v,v) > 1)
```

单位球面上随机选点

要在球面上选点，实际上只需要把从球心指向上一步中球内均匀选择的点的向量归一化即可。这样，就可以将点“移动”到球面上了。同样，我们不证明这么做的正确性。

```
Vec3f v;
do{
    v.x = 1-2*randf();
    v.y = 1-2*randf();
    v.z = 1-2*randf();
} while (dot(v,v) > 1)

v /= length(v);
```

xz 平面截开的单位半球内随机选点

也很简单，我们只需要拒绝处于下半球的点即可。

```
Vec3f v;
do{
    v.x = 1-2*randf();
    v.y = 1-2*randf();
    v.z = 1-2*randf();
} while (dot(v,v) > 1 || v.z < 0)
```

任意平面截开的单位半球内的随机选点

稍微与上面有一些区别，主要是要找到描述处于另外半球的点的条件即可。我们可以简单地使用该平面的法线 $\mathbf{n}$ 来判断。与其点积小于 0，就意味着它们不在同一个半球。

```
Vec3f v;
do{
    v.x = 1-2*randf();
    v.y = 1-2*randf();
    v.z = 1-2*randf();
} while (dot(v,v) > 1 || dot(v, n) < 0)
```

以上就是一些常用的拒绝采样的实现方式了。但是，很显然，一个重要的问题也伴随着 `do-while` 循环而来：我们无法估计循环需要进行的轮数。当运气极度不好时，这个循环可能会一直进行下去，从而导致严重的超时。因此，在实际项目中，我们很少会真的去采用拒绝采样；取而代之的是被称为*直接采样*的方式，将采样点显式地计算出来。要了解如何计算，我们从分布转换开始讨论。

8.3 分布转换

如果我们将采样点转换到极坐标上，我们就会考虑是否可以通过 (r, ϕ) 在圆盘上随机均匀选点：均匀随机选择两个点 ξ_1, ξ_2，然后令 $r = \xi_1, \phi = 2\pi\xi_2$。

事实证明，这种方法并不均匀。按照这种采样方式，采样结果在集中圆盘中心。直观来说，这是因为采样点虽然在 (r, ϕ) 的极坐标空间是均匀的，但是在 (x, y) 坐标下却不均匀：当采样结果中的 r 均匀地减小时，对应的内圈面积也会越来越小，可是小面积获得采样点的可能性却是一样大的，因此我们会看到在圆盘中心附近集中了很多采样结果。

然而，如果令 $r = \sqrt{\xi_1}, \phi = 2\pi\xi_2$，则结果就是正确的，直觉上来说这是因为考虑了面积随半径的平方变化的特性。一般地，我们应该如何选择在不同的坐标系之间进行分布转换呢？下面我们来讨论这个问题。

8.3.1 反密度函数方法

设 $X \sim f(x)$，其中 $f(x)$ 是一个一维的、在区间 $[x_1, x_2]$ 上有定义的 pdf。如果我们随机让 `randf()` 生成 i 个数 $\xi_1, \xi_2, ..., \xi_i$，然后，计算

$$\alpha_i = P^{-1}(\xi_i)$$

其中，

$$P(x) = \int_{x_1}^{x} f(t) d\mu(t).$$

那么，这 i 个 α_i 是服从 $f(x)$ 分布的，其中 P^{-1} 是 P 的反函数。在这里我们不证明这个结论，感兴趣的同学可以概率论的相关书籍文献中轻松找到它的相关资料。

Example 8.1. *在 x 轴上选取 i 个随机点 x_i，使得它们在 $[-1, 1]$ 上服从 $p(x) = \frac{3x^2}{2}$。*

Solution. 观察到，

$$\begin{aligned} P(x) &= \int_{-1}^{x} \frac{3t^2}{2} d\mu(t) \\ &= \int_{-1}^{x} \frac{3t^2}{2} d(t \cdot 1) \\ &= \int_{-1}^{x} \frac{3t^2}{2} dt \\ &= \frac{x^3 + 1}{2}. \end{aligned}$$

我们可以将 $P(x) = y$，然后

$$y = \frac{x^3 + 1}{2}$$
$$x^3 = 2y - 1$$
$$x = \sqrt[3]{2y - 1}$$

因此我们得到

$$P^{-1}(x) = \sqrt[3]{2x - 1}$$

利用 `randf()` 随机生成 i 个输入值 $\xi_1, \xi_2, ...\xi_i$，则

$$(\alpha_1, \alpha_2, ..., \alpha_i) = (P^{-1}(\xi_1), P^{-1}(\xi_2), ..., P^{-1}(\xi_i))$$

就是我们需要的 i 个随机点的 x 坐标。 ■

总结一下，使用 `randf()` 生成在区间 $[a, b]$ 上服从 $f(x)$ 采样的一维随机变量步骤为：

1. 计算 cdf $P(x) = \int_0^a f(t)dt$。
2. 计算其反函数 $P^{-1}(y)$。
3. 使用 `randf()` 生成在 [0,1] 上均匀分布的一个随机变量 ξ。
4. 计算随机变量 $X_i = P^{-1}(\xi)$，则随机变量 $X_i \sim f(x)$。

8.3.2 直接采样

一般地，对于随机 d 维变量 $X_i \sim p_X(\mathbf{x})$，以及随机 d 维变量 $Y_i = T(X_i)$。我们考虑 Y_i 的 pdf。

$$p_Y(\mathbf{y}) = p_Y(T(\mathbf{x})) = \frac{p_X(\mathbf{x})}{|J_T(\mathbf{x})|}$$

其中，$|J_T(\mathbf{x})|$ 是 T 的雅可比行列式的绝对值。对于二元函数来说，假设 $\mathbf{x} = (x, y)$，则其雅可比矩阵

$$J_T(\mathbf{x}) = \begin{bmatrix} \frac{\partial T_1}{\partial x} & \frac{\partial T_1}{\partial y} \\ \frac{\partial T_2}{\partial x} & \frac{\partial T_2}{\partial y} \end{bmatrix}$$

需要强调的是：对于直接采样而言，我们容易获得的应该是随机变量 X_i，而我们容易写出的 pdf 却应该是 Y_i 的 pdf。换句话说，当我们需要采样服从 p_Y 的密度函数的随机变量 Y_i 时，我们可以先想办法用 X_i 表达 Y_i，然后根据某种方式采样 X_i。

圆盘上随机采样

以变换 $T(r, \phi) = [r\cos\phi, r\sin\phi]$，我们可以将坐标从极坐标系转换到 xy 坐标系。我们先计算雅可比矩阵。

$$J_T(r, \phi) = \begin{bmatrix} \frac{\partial(r\cos\phi))}{\partial r} & \frac{\partial(r\cos\phi)}{\partial\phi} \\ \frac{\partial(r\sin\phi))}{\partial r} & \frac{\partial(r\sin\phi)}{\partial\phi} \end{bmatrix} = \begin{bmatrix} \cos\phi & -r\sin\phi \\ \sin\phi & r\cos\phi \end{bmatrix}$$

因此，行列式为 $r(\cos^2\phi + \sin^2\phi) = r$。所以，$p_Y(x, y) = p_X(r, \phi)/r$。

现在，考虑到在 xy 坐标系下，均匀分布意味着对于每一个满足 $x^2+y^2 \leq 1$ 的数对 (x,y)，它们对应的密度函数都是 $\frac{1}{\pi r^2} = \frac{1}{\pi}$。因此，我们有

$$p_X(r,\phi) = p_Y(T(r,\phi)) \cdot |J_T(r,\phi)| = \frac{1}{\pi} \cdot r = \frac{r}{\pi}.$$

注意到，r 与 ϕ 的采样是可以独立进行的，它们本身也是互不影响的。因此，我们有 $p_X(r,\phi) = p(r)p(\phi) = r/\pi$。这样，我们就可以使用下面的步骤：

- 随机生成一个 $\phi \sim \frac{1}{2\pi}$，其中 $\phi \in [0, 2\pi]$。
- 随机生成一个 $r \sim 2r$，其中 $r \in [0,1]$。

现在问题就变成了如何生成一个一维函数、区间固定服从给定 pdf 的随机变量了。这里就可以直接遵从我们上面提到的反密度函数方法中均匀采样的步骤。

随机生成区间 $[0,1]$ 上服从 $p(r) = 2r$ 的随机变量：
首先，获取 cdf，

$$P(r) = \int_0^r 2t dt = r^2$$

然后，令 $P(r) = x$，则

$$x = r^2$$
$$r = \sqrt{x}$$

因此我们得到

$$P^{-1}(r) = \sqrt{r}$$

随机生成区间 $[0, 2\pi]$ 上服从 $p(\phi) = 1/2\pi$ 的随机变量：
首先，获取 cdf，

$$P(r) = \int_0^\phi (1/2\pi) dt = \frac{\phi}{2\pi}$$

然后，令 $P(\phi) = y$，则

$$y = \frac{\phi}{2\pi}$$
$$\phi = 2\pi y$$

因此我们得到

$$P^{-1}(\phi) = 2\pi\phi$$

因此，使用 `randf()` 生成在 [0,1] 上均匀分布的两个随机值 ξ_1, ξ_2，然后令 $r = \sqrt{\xi_1}, \phi = 2\pi\xi_2$ 就可以完成在圆盘上随机均匀采样的目标了。这与我们上面靠直觉得出的结论是相同的。

球面上随机采样

在圆盘上的随机采样是从二维极坐标 $r\theta$ 转换到二维 xy 平面的。但是如果我们要用球坐标 $\theta\phi$ 去采样，然后转换到三维 xyz 空间时，就会面临一个问题：雅可比项不存在行列式，因为变换矩阵并不是方阵。对于这一类不同维度间的变换，我们使用不同的变换法则。

$$p_{\mathbf{x}}(\mathbf{x}(u,v)) = \frac{p_{(u,v)}(u,v)}{||\mathbf{x}_u(u,v) \times \mathbf{x}_v(u,v)||}$$

其中，$\mathbf{x}(u,v)$ 将二维向量 (u,v) 转换至 $n \geq 2$ 维空间。那么，这个雅可比行列式是哪里来的呢？它来自于微分的表达式

$$dA(\mathbf{x}) = ||\mathbf{x}_u(u,v) \times \mathbf{x}_v(u,v)||dudv$$

以上，我们就有球面上随机采样的步骤了。

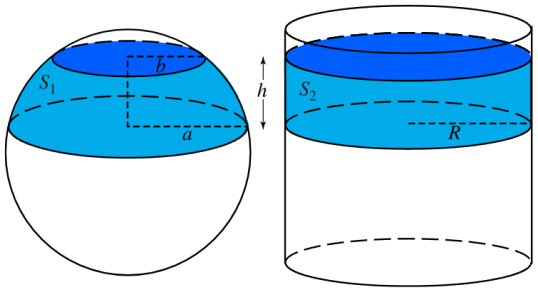
球面上随机采样（$\theta\phi$ 坐标）：
首先，使用 `randf()` 在 [0,1] 上生成均匀分布的两个随机值 ξ_1, ξ_2。
然后，令 $\theta = \cos^{-1}(2\xi_1 - 1), \phi = 2\pi\xi_2$。
然后，计算三维坐标 $(x,y,z) = (\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta)$。

阿基米德帽盒定理

在进一步讨论与球面相关的随机之前，我们先了解一个古代的定理。

定义（阿基米德帽盒定理）**阿基米德帽盒定理**（Archimedes' Hat-Box Theorem）指出，如果用两个水平面将球隔开，那么在这两个水平面之间的球的表面积与该球的外切圆柱体被相同的水平面截开后对应的截面面积相同。例如，在下图[4]中，浅蓝色的面积符合

$$S \equiv S_1 = S_2 = 2\pi Rh.$$



阿基米德帽盒定理意味着，如果我们在球面上均匀地采样点，其纬度的分布应该是均匀的。这样，我们也可以换一个思路：均匀采样高度、均匀采样方位角。我们可以使用性能更好的 $r\phi z$ 坐标，也叫做柱面坐标系（cylindrical coordinates）。

球面上随机采样（柱面坐标）：
首先，使用 `randf()` 在 [0,1] 上生成均匀分布的两个随机值 ξ_1, ξ_2。
然后，令 $z = 2\xi_1 - 1, \phi = 2\pi\xi_2$。
然后，计算 $r = \sqrt{1-z^2}$ 然后，计算三维坐标 $(x,y,z) = (r\cos\phi, r\sin\phi, z)$。

[4] https://mathworld.wolfram.com/ArchimedesHat-BoxTheorem.html

8.4 重要性采样

在上一章的误差分析中，我们知道即使蒙特卡洛估值是无偏和一致的，其依然存在方差。显示在我们的渲染结果中，方差就意味着噪点，降低方差也就意味着提高渲染准确率，即渲染效果。虽然用以估计的概率密度函数 $p(x)$ 没有明确的要求，但是如果它与 $f(x)$ 有着相似的分布，则显然 F_N 会更接近原积分。挑选一个与 $f(x)$ 相似的概率密度函数的蒙特卡洛估值也被叫做**重要性采样**（importance sampling），它可以显著地降低 f/p 的方差。

8.4.1 理想的重要性采样

例如，对于积分 $\int f(x)dx$，其蒙特卡洛估值为

$$\langle F\rangle = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)},$$

其中我们采样的概率密度为 $p(x)$。我们之前已经证明过这个估值是无偏的，其期望就是积分的实际值。能够使得方差最低的最优密度函数 $p(x)$ 可以通过**拉格朗日乘数法**（Lagrange Multiplier）来实现。我们需要找到一个使得下面这个关于 $p(x)$ 的函数 L 达到最小值的标量 λ。

$$L(p) = \int_D \left(\frac{f(x)}{p(x)}\right)^2 p(x)dx + \lambda\int_D p(x)dx,$$

唯一的边界条件是 $p(x)$ 在积分域内的积分为 1，即

$$\int_D p(x)dx = 1.$$

这类最小值问题可以被通过**欧拉-拉格朗日微分方程**（Euler-Lagrange differential equation）来求解：

$$L(p) = \int_D \left(\frac{f(x)^2}{p(x)} + \lambda p(x)\right)dx$$

为了使得这个函数最小化，我们对 $L(p)$ 求关于 $p(x)$ 的微分，使结果为 0，求解其对应的 $p(x)$。

$$\begin{aligned} 0 &= \frac{\partial}{\partial p}\left(\frac{f(x)^2}{p(x)} + \lambda p(x)\right) \\ 0 &= -\frac{f^2(x)}{p^2(x)} + \lambda \\ p(x) &= \frac{1}{\sqrt{\lambda}}|f(x)|. \end{aligned}$$

常数 $\frac{1}{\sqrt{\lambda}}$ 是一个比例系数，它使得 $p(x)$ 可以满足边界条件。在这种情况下，最优的 $p(x)$ 为

$$p(x) = \frac{|f(x)|}{\int_D f(x)dx}$$

如果我们使用这个 $p(x)$，那么方差就一定会是 0。但是，在现实中，这是没有意义的：在假设 $p(x) = \frac{|f(x)|}{\int_D f(x)dx}$ 时，我们必须要先有对积分本身的求值，而我们本身就是在用采样的方式求解积分。不过，以上步骤足以说明如果我们采用的 PDF 与被积函数很接近，那么方差也会显著减小，这就是重要性采样的原理。

8.4.2 重要性采样的步骤

在实施重要性采样时，我们遵从以下的步骤。

坐标系统转换

将分布通过一个能够简化采样和计算的坐标系统来表达。比如，如果目标分布在极坐标下表现得更为规则或易于采样，那么我们可能希望将分布转换到极坐标系。

在进行坐标转换时，我们需要计算雅可比矩阵。雅可比矩阵是一个坐标系统的微元到另一个坐标系统的微元缩放比例。

计算条件一维 pdf

在多维问题中，我们可能需要首先处理边缘分布，即忽略其他变量的分布。在这种情况下，**条件概率密度函数**（conditional probability distribution function）描述了在给定一个或多个变量的值的情况下，其他变量的分布情况。这些一维 pdf 用于顺序采样，即一次采样一个变量，每个变量的采样可能依赖于前面变量的值。

反密度函数法采样一维 pdf

再利用 8.3.1 中提及的反密度函数方法采样即可。

8.5 多重重要性采样

我们经常会遇到一个积分是两个函数的乘积的情况，例如 $\int f_a(x)f_b(x)dx$，且经常我们可以单独地对 $f_a(x)$ 或是 $f_b(x)$ 进行重要性采样，可以采样出来的结果却与 $f_a(x)f_b(x)$ 的形状大相径庭。在渲染方程中，我们会将辐射亮度写成 BSDF、入射辐射亮度和余弦因子的乘积形式，因此，在渲染中想办法解决这个问题是非常重要的。

例如，假设我们要使用蒙特卡洛积分来估算出射辐射亮度的值，且假设我们通过某种方式得到了两个采样分布函数 p_a 和 p_b，它们完美地匹配了真实的分布函数 f_a 和 f_b[5]。也就是说，$f_a(x) = c_a p_a(x), f_b(x) = c_b p_b(x)$。现在我们有两个选择：

- 使用 p_a 进行重要性采样。这样的估计值为
$$\frac{f(X)}{p_a(X)} = \frac{f_a(X)f_b(X)}{p_a(X)} = c_a f_b(X)$$
- 使用 p_b 进行重要性采样。这样的估计值为
$$\frac{f(X)}{p_b(X)} = \frac{f_a(X)f_b(X)}{p_b(X)} = c_b f_a(X)$$

也就是说，如果使用与其中一种分布函数匹配的采样函数，那么方差就会与另一种分布函数成正比，因此这个方差本身可以非常大。现实中，我们甚至没法做到使得采样函数与其中任意一个函数完美匹配，方差可能比我们预计的还要糟糕。

不幸的是，我们也没有办法通过简单的平均两个估计值来降低方差，因为方差是叠加性的 (additive)。我们需要提出新的采样方案来处理这种需要同时考虑多个函数的重要性采样，这个方法就叫做**多重重要性采样**（Multiple Importance Sampling, MIS）。

[5]实际上现实中不太可能做到，但是这里就先这么假设

8.5.1 多重重要性采样的步骤

MIS 的基本思想在于，当估算一个积分的时候，我们应该从多个采样分布中抽取样本，抱着其中势必有一个与实际被积函数的分布匹配的希望，选取其中一个样本，即使我们也许并不知道是哪一个样本。然后，MIS 通过给采集到的样本进行加权处理，来降低方差。

以图形学中最常见的双重重要性采样为例。假设我们有采样分布函数 p_a 和 p_b。假设我们从两个分布中各采样一次，$X \sim p_a, Y \sim p_b$。那么该积分的**多重重要性蒙特卡洛估计值**（MIS Monte Carlo estimator）就是

$$\langle F \rangle = w_a(X)\frac{f(X)}{p_a(X)} + w_b(Y)\frac{f(Y)}{p_b(Y)} \tag{8.1}$$

其中，w_a 和 w_b 是**权重函数**（weighting function），它们通过加权，使得公式 (8.1) 中的估计值的期望与实际被积函数相同。

一般地，给定 n 个采样分布函数 $p_1, p_2, ..., p_n$，其中第 i 个分布采集 n_i 个样本，那么其对应的 MIS 蒙特卡洛估值为

$$\langle F \rangle = \sum_{i=1}^{n} \frac{1}{n_i} \sum_{j=1}^{n_i} w_i(X_{i,j}) \frac{f(X_{i,j})}{p_i(X_{i,j})} \tag{8.2}$$

8.5.2 平衡启发式权重函数

一个不错的权重函数的选择被称作**平衡启发式**（balance heuristic）权重函数。平衡启发式尝试考虑所有样本可能被生成的不同方式，而不仅仅只是考虑其中特定的一个。第 i 次采样时的平衡启发式的权重函数形如

$$w_i(x) = \frac{n_i p_i(x)}{\sum_j n_j p_j(x)}. \tag{8.3}$$

在使用平衡启发式的权重函数时，我们注意到 (8.1) 中的双重重要性采样的估计值就变成了

$$\frac{f(X)}{p_a(X) + p_b(X)} + \frac{f(Y)}{p_a(Y) + p_b(Y)}$$

注意，每一次 f 的采样都被除以了所有取值为当前随机变量的 PDF 的和，而不仅仅是只有生成了这个随机变量的那个 PDF。直观地说，如果 p_a 通过低概率生成了一个样本，而 p_b 相对来说有高概率，则除以它们的和就可以有效地降低这个样本的贡献。

8.5.3 幂启发式权重函数

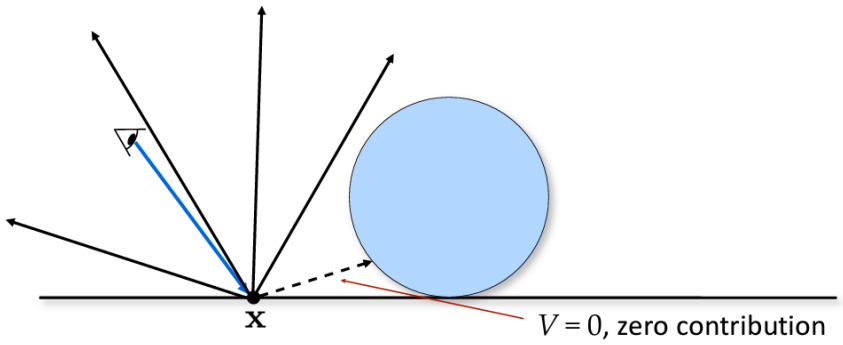
在实际的实现中，**幂启发式**（power heuristic）权重函数甚至可以进一步降低方差。与 (8.3) 的形式非常相似，只不过幂启发式函数要再多一个指数 β：

$$w_i(x) = \frac{(n_i p_i(x))^\beta}{\sum_j (n_j p_j(x))^\beta}. \tag{8.4}$$

注意 β 在分母部分中的位置，是累加所有 β 次方的 $n_j p_j$，而不是累加完之后再一起取 β 次方。经验告诉我们，$\beta = 2$ 通常能起到最好的效果。

8.6 环境光遮罩

环境光遮罩（Ambient Occlusion，AO）是一种现实中普遍存在的现象。当两个表面相互接近时，靠近的部分通常受到较少的环境光，因此会产生局部阴影。现在，我们假设在一个平面上有一点 $\mathbf{x}$，我们求这一点的散射辐射亮度 $L_r(\mathbf{x}, \cdot)$。



如上图所示，根据反射方程我们有

$$L_r(\mathbf{x}, \omega) = \int_{H^2} f_r(\mathbf{x}, \omega_i, \omega_r) L_i(\mathbf{x}, \omega_i) \cos\theta_i d\omega_i$$

假设表面是朗伯表面，即 BRDF 为 $\frac{\rho}{\pi}$，且我们知道任何一点出射亮度应该与方向无关，因此，上述方程可以简化为

$$L_r(\mathbf{x}) = \frac{\rho}{\pi} \int_{H^2} V(\mathbf{x}, \omega_i) \cos\theta_i d\omega_i$$

其中，函数 $V(\mathbf{x}, \omega_i)$ 被称作**可见性函数**（visibility function），它被定义为：如果从 $\mathbf{x}$ 看向 ω_i 的方向，环境被遮挡，则其值为 0；否则则为 1。在上图中，虚线方向的 V 就会被计算为 0。

使用蒙特卡洛估值计算环境光遮罩下的辐射亮度，我们应该有

$$L_r(\mathbf{x}) \approx \frac{\rho}{\pi} \frac{V(\mathbf{x}, \omega_i) \cos\theta_i}{p(\omega_i)}$$

其中，$p(\omega_i)$ 是我们采样方向的 pdf。

8.6.1 环境光遮罩的重要性采样

那么下面就是喜闻乐见的选择 pdf 的时间了。我们自然可以直接使用半球内均匀采样的方式从点 $\mathbf{x}$ 出发选择方向。在这种情况下，$p(\omega_i) = \frac{1}{2\pi}$。在环境光遮罩的计算公式中，我们可以对余弦项进行重要性采样。我们不再在半球内均匀采样，而是根据余弦项加权采样。

半球面上余弦项加权采样

首先，根据余弦项加权的概率密度分布函数是

$$p(\theta, \phi) = \frac{\cos\theta}{\pi}.$$

其累积分布函数可以由下计算：

$$
\begin{aligned}
F &= \frac{1}{\pi}\int_{\mathcal{H}}^{2}\cos\theta d\omega \\
F(\theta,\phi) &= \frac{1}{\pi}\int_0^{\phi}\int_0^{\theta}\cos\theta'\sin\theta' d\theta' d\phi' \\
&= \frac{1}{\pi}\int_0^{\phi} d\phi' \int_0^{\theta}\cos\theta'\sin\theta' d\theta' \\
&= \frac{\phi}{\pi}(-\cos^2\theta'/2)|_0^{\theta} \\
&= \frac{\phi}{2\pi}(1-\cos^2\theta).
\end{aligned}
$$

在这里，与 ϕ 和 θ 有关的累积分布函数实际上是可以分离的：

$$
\begin{aligned}
F_\phi &= \frac{\phi}{2\pi} \\
F_\theta &= 1-\cos^2\theta \\
F(\theta,\phi) &= F_\phi \cdot F_\theta
\end{aligned}
$$

因此，假设使用 `randf()` 采样两个均匀分布的样本 ξ_1 和 ξ_2，并且有

$$
\begin{aligned}
\phi_i &= 2\pi\xi_1 \\
\theta_i &= \cos^{-1}\sqrt{\xi_2}
\end{aligned}
$$

这些 θ_i 和 ϕ_i 的值将根据余弦概率密度分布函数分布。

8.7 采样模式

虽然蒙特卡洛积分的核心思想是通过随机来处理计算结果，但是有的时候任意的随机可能是有害的。真随机的情况下，我们无法保证在有限次的次数内一定能够采样到更为均匀的结果，又或者说是在有限的次数内可以采样出我们需要的某一种模式。我们可以在采样时稍加一些限制或是手段，就能大大提高采样的效率。

8.7.1 独立采样与等距采样

独立采样（independent sampling）是最简单的采样模式，因为没有什么好说的：每次采样的时候在采样范围内独立选择。它的优点很明显：

- 能够很直接地拓展到高维情况。
- 概念上非常简单，而且不消耗多余的内存空间。

但是它的缺点也很致命：

- 在有限次（少量）采样的过程中，可能在样本彼此之间存在巨大的空隙。
- 同理，也可能会有很多样本团集在一起，分布很有可能不均匀。

这两个缺点实际上在讲同一件事：独立采样的结果可能是包含所有频率的。换句话说，采样的结果就是白噪声。因此，在实际的渲染中，我们几乎不去使用独立采样。

另一种思路是使用**等距采样**（regular sampling）：每个相邻采样点之间与的距离是完全相同的。如果我们使用等距采样，确实可以解决上面独立采样不是带限信号的问题，且同时也可以保证独立采样的优势，但是，均匀的采样会很容易产生走样问题。

8.7.2 分层采样

独立采样的问题在于，在采集样本较少时，我们无法保证这些样本能够覆盖到一个合理的大范围内。一个优化的思路是：我们先将样本均匀分成多个层次，然后再在这些层次中随机均匀采样。这样的方法叫做**分层采样**（stratefied sampling）。对于在 $[0,1]^d$ 上的采样，我们需要 $O(N^d)$ 个采样点。

分层采样的优点在于：

- 我们可以证明它不会增加采样结果的方差。
- 继承独立采样的优势。

但是，分层采样也有比较明显的缺点。

- 对于 d 维的 N 层情况，我们需要 $O(N^d)$ 个采样点。
- 每个层次需要做相同次数的采样，因此如果要额外做少数个采样点时，我们需要对所有层次都再进行一次采样。对于这种情况，我们也说其**粒度**（granularity）比较大。

Example 8.2. 采样一个 *5D* 图像。其中，除了 *RGB* 三个维度之外，还有时间和镜头的数据。如果我们把每一个数据分四层采样，那么每个像素的采样点总数就是 $4^5 = 1024$。通常，我们也许不会在每个维度上都做这么多采样，会减少其中几个维度的采样——但是这样的话分层采样的优势也就体现不出来了。这种现象也被大家称作**维度的诅咒**（*the curse of dimensionality*）。

8.7.3 N-Rooks 采样法

8.7.4 差异性

差异性是一种用来描述采样点聚集程度的属性。基本的想法是，对于一个 d 维空间 $[0,1)^d$ 中的一个点集，它的质量可以通过观察 $[0,1)^d$ 中的一些区域，数出这些区域中的点的数量，然后比较每个区域的 “体积” 和其中的采样点数量来描述。严格来说，其定义如下：

定义（差异性）对于 $[0,1)^d$ 的一组子集 B，满足

$$B = \{[0, v_1] \times [0, v_2] \times ... \times [0, v_d]\}$$

其中，$0 \leq v_i \leq 1$，一组 [0,1) 范围内的 n 个采样点集合 $P = \{x_1, ..., x_n\}$，它对于 B 的**差异性**（discrepancy）为

$$D_n(B, P) = \sup_{b \in B} \left| \frac{\#\{x_i \in b\}}{n} - V(b) \right|,$$

其中，$\#\{x_i \in b\}$ 是集合 b 中的的采样点数量，$V(b)$ 是 b 的容量。sup 运算符被称作最低上限，其给出定义域范围内最低的取值上限。

从直觉上来理解这个定义。在采样点按照理想状态均匀分布时，$\#\{x_i \in b\}/n$ 应该能够很好地估计其所在的集合 b 的容量。所以，差异性指的就是通过这种方式估算容量时最高的误差。注意，实际上 B 中所选择的子集并不一定要从原点出发。通过从原点出发的形状组成的 B 而得出的差异性也叫做**星形差异性**（star discrepancy），记作 $D_n^*(P)$。

低差异性

如果一个 d 维点集合的差异性满足

$$D_n^*(P) \in O\left(\frac{(\log n)^d}{n}\right)$$

则我们说这个集合是低差异（low discrepancy）的。低差异的点集和序列通常都是通过确定性（deterministic）而非随机性（random）的算法来计算采样的。用这样确定的算法来采样，然后计算蒙特卡洛积分的方式被我们称作**拟蒙特卡洛方法**（quasi-Monte Carlo Method, QMC）。在 8.7.5 中我们将会讨论基于确定性算法的霍尔顿采样以及对应的拟蒙特卡洛方法。

Koksma-Hlawka 不等式

差异性与积分的误差关系满足 **Koksma-Hlawka 不等式**。对于函数 f，该不等式为

$$\left|\int f(u)du - \frac{1}{n}\sum_i f(x_i)\right| \leq D*(P)\mathcal{V}_f$$

其中，$\mathcal{V}_f$ 是被积函数 f 的**总变化**（total variation），其定义为

$$\mathcal{V}_f = \sup_{0=y_1<y_2<\ldots<y_m=1} \sum_{i=1}^{m} |f(y_i) - f(y_{i+1})|$$

简单来说，总变化描述了函数值在采样点之间的变化快慢，而差异性则描述了采样点反应函数变化的能力。

根据 Koksma-Hlawka 不等式，我们能够看出随着维度 d 增加，低差异性采样的数值积分误差来到了 $O(n^{-1})$，这比蒙特卡洛积分的 $O(n^{-1/2})$ 要好很多。然而，因为 QMC 是确定性的算法，用方差来描述估值的质量是不可能的。一个想法是采样点依然可以在不破坏差异性的前提下被随机采样。这样基于 QMC 的随机算法也被我们称为**随机拟蒙特卡洛算法**（Randomized quasi-Monte Carlo, RQMC）。

8.7.5 霍尔顿采样与拟蒙特卡洛方法

分层采样背后的逻辑是使得采样点尽可能均匀地分布在整个采样面上，相邻两个采样点不要相距太远或太近，同时保持随机。另外，分层采样依然得需要通过随机获得采样点。而**霍尔顿采样**（Halton sampling）则是一种可以直接确定地生成低差异采样点的算法，是一种拟蒙特卡洛方法。

基础逆运算

首先，一个事实是：对于一个正整数 a，它可以被唯一地表达为一个以 b 为基数的数位序列，

$$a = \sum_{i=1}^{m} d_i(a) b^{i-1}.$$

其中，$d_i(a) \in [0, b-1] \cap \mathbb{Z}$。这样我们就可以定义基础逆函数了。

定义（基础逆函数）以 b 为基数的非负整数 a 的**基础逆函数**（radical inverse function）被定义为

$$\Phi_b(a) = 0.d_1(a)d_2(a)...d_m(a) = \sum_{i=1}^{m} d_i(a) b^{-i}.$$

Example 8.3. 计算 $b = 2$ 的情况下，*10* 的基础逆函数值。

Solution. 我们可以按照如下的步骤计算 10 的基础逆函数值。

1. 将 $n = 10$ 转换为基数 $b = 2$d 的表示形式，也就是二进制。$10_{(10)} = 1010_{(2)}$。
2. 反转这个数字的顺序。1010 反转后成为 0101。
3. 将反转后的序列视为 $b = 2$ 基数下的小数部分。这意味着我们将 0101 视为二进制小数 0.0101。
4. 计算这个小数的十进制值。

$$0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} = 0.3125.$$

因此，10 的基础逆函数值 $\Phi_2(10) = 0.3125$。 ■

根据上面的描述，我们可以写出 C++ 中的基础逆运算的实现。

```
float RadicalInverse(int n, int base, float inv) {
    float v = 0.f;
    for(float p = inv; n != 0; p *= inv, n /= base) {
        v += (n % base) * p;
    }
    return ;
}
```

范德科皮特序列

最简单的低差异性序列之一就是**范德科皮特序列**（van der Corput Sequence）。这是一个一维序列，

$$x_a = \Phi_2(a)$$

其中，$a = 0, 1, ...$。这个序列的差异性为

$$D_n^*(P) = O(\frac{\log n}{n})$$

这也是最优的差异。我们并不需要提前知道总共需要采样多少个点就可以生成这个序列。

霍尔顿序列

d 维的**霍尔顿序列**（Halton Sequence）也是基于基础逆函数的定义，只不过对于每一个维度，它有不同的基数。每一个基数必须是互质的，所以一个自然的想法就是使用第 n 个质数。

$$x_k = (\Phi_2(k), \Phi_3(k), \Phi_5(k), ..., \Phi_{p_d}(k))$$

其中，p_i 是第 i 个质数。和范德科皮特序列一样，我们也不需要提前知道采样点的总数就可以生成任意长度的霍尔顿序列了。这个序列的差异性为

$$D_n^*(P) = O\left(\frac{(\log n)^d}{n}\right)$$

也是最优的。

哈默斯莱序列

如果我们提前知道采样总数，那么我们就可以使用**哈默斯莱序列**（Hammersley Sequence），

$$x_k = (k/N, \Phi_2(k), \Phi_3(k), \Phi_5(k), ..., \Phi_{p_d}(k))$$

其中，p_i 是第 i 个质数。它的差异性略低于霍尔顿序列。

8.7.6 低差异性序列的问题

低差异性序列的问题来自于其确定性的算法。我们没办法通过多次渲染求平均的方式来降噪，因为没有随机性——不管运行多少次，每次渲染出来的结果都会是完全一致的。另外，随着维度的增加，采样值在低维的投影可能会显示出一种有规则的范式，如图 8.3 所示[6]。

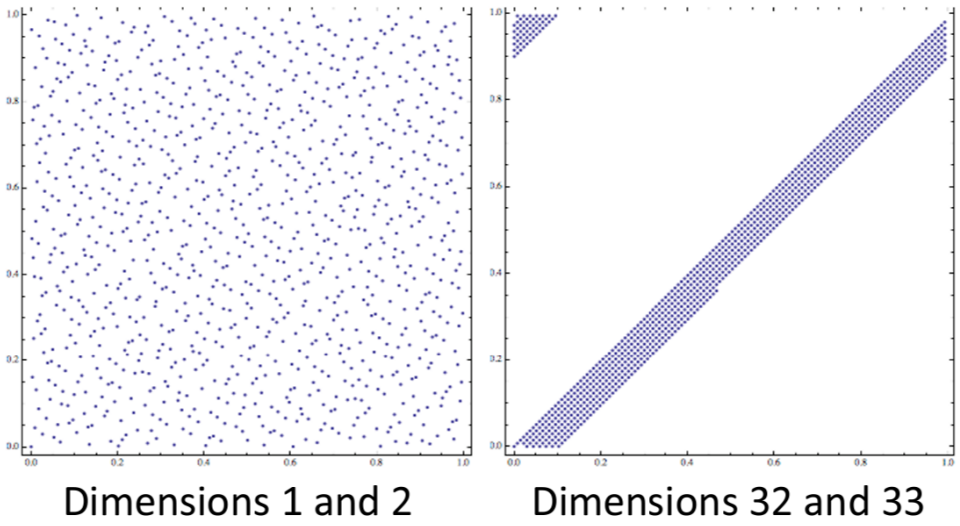


图 8.3: 霍尔顿序列在低维和高维的表现

为了解决这些问题，我们采用的方式是在不破坏低差异性的前提下，继续加入随机，也就是前文提及的随机拟蒙特卡洛方法。

[6]图来自CMU 15668 Physically Based Rendering Lecture 9

加扰序列

加扰（Scrambling）是一种常用的手段，它的核心想法是去重新随机排列每一个采样点的每个坐标。我们创建一个排列表 π，给每一个数位一个顺序，然后在计算基础逆时去使用这个排列表。也就是说，

$$\Phi_b(n) = 0.\pi(d_1)\pi(d_2)...\pi(d_m)$$

一个基数为 2 的随机基础逆函数可以通过 XOR 操作很高效地计算，在后面的具体实现中我们会讨论这个算法。

8.8 Sampler 类

通过以上的讨论，我们已经清楚了采样策略的重要性。因此，在生成随机数时，我们不应该再简单地通过 `randf()` 去生成 [0,1] 内均匀分布的随机值了。我们应该诉诸于 `Sampler` 类，其任务就是要在 [0,1] 内按照某一种采样策略生成 d 维的采样点。

8.8.1 Sampler 接口

从基础的父类 `Sampler` 开始。

```
class Sampler{
public:
    // constructors and destructors OMITTED
    virtual void startPixel();
    virtual bool startNextPixelSample();
    virtual float get1D() = 0;
    virtual Vec2f get2D() = 0;

    size_t samplesPerPixel;
    size_t dimension;
protected:
    size_t currentPixelSample = 0;
    size_t current1DDimension = 0, current2DDimension = 0;
}
```

在有了 `Sampler` 的接口后，我们就可以修改之前的一些随机采样函数。例如，对于使用两次 `randf()` 的随机采样函数，我们则可以使用采样器的 `get2D()` 方法生成一个二维向量 `sample`，然后使用 `sample.x` 和 `sample.y` 分别替换之前的两个 `randf()`。

8.8.2 StratefiedSampler 类

```
class StratifiedSampler : public Sampler {
  public:
    StratifiedSampler(const json &j = json::object());

    void startPixel() override;

    float next1D() override;

```

```
    Vec2f next2D() override;

  private:
    void stratifiedSample1D(std::vector<float> &samples);

    void stratifiedSample2D(std::vector<Vec2f> &samples);

    // Indexed as samples[dimIndex][imageSampleIndex], where imageSampleIndex denotes the
    // current image sample and dimIndex denotes which random value in the current image sample
    std::vector<std::vector<float>> samples1D;
    std::vector<std::vector<Vec2f>> samples2D;
};
```

```
// sampler.cpp
StratifiedSampler::StratifiedSampler(const json &j) {
    // samplesPerPixel must be a perfect square (e.g. 1, 4, 9, 16, etc.)
    samplesPerPixel = roundToPerfectSquare(j.value("image_samples", 4));
    dimension = j.value("dimension", 4);

    // Initialize samples1D and samples2D arrays
    samples1D.resize(dimension);
    samples2D.resize(dimension);
    for (size_t i = 0; i < dimension; i++) {
        samples1D[i].resize(samplesPerPixel);
        samples2D[i].resize(samplesPerPixel);
    }
}

void StratifiedSampler::startPixel() {
    for (size_t i = 0; i < samples1D.size(); i++) {
        stratifiedSample1D(samples1D[i]);
        shuffle<float>(samples1D[i]);
    }

    for (size_t i = 0; i < samples2D.size(); i++) {
        stratifiedSample2D(samples2D[i]);
        shuffle<Vec2f>(samples2D[i]);
    }

    Sampler::startPixel();
}

void StratifiedSampler::stratifiedSample1D(std::vector<float> &samples) {
    float invNSamples = 1.f / samples.size();
    for (size_t x = 0; x < samples.size(); x++) {
        samples[x] = min((x + randf()) * invNSamples, ONE_MINUS_EPSILON);
    }
}

void StratifiedSampler::stratifiedSample2D(std::vector<Vec2f> &samples) {
    int i = 0;
    float sqrtN = sqrt(samples.size());
    float invSqrtN = 1.f / sqrtN;
```

```
    for (int y = 0; y < static_cast<int>(sqrtN); y++) {
        for (int x = 0; x < static_cast<int>(sqrtN); x++) {
            samples[i].x = min((x + randf()) * invSqrtN, ONE_MINUS_EPSILON);
            samples[i].y = min((y + randf()) * invSqrtN, ONE_MINUS_EPSILON);
            i++;
        }
    }
}

float StratifiedSampler::next1D() {
    if (current1DDimension < samples1D.size() && currentPixelSample < samplesPerPixel) {
        float out = samples1D[current1DDimension][currentPixelSample];
        current1DDimension++;
        return out;
    } else {
        return randf();
    }
}

Vec2f StratifiedSampler::next2D() {
    if (current2DDimension < samples2D.size() && currentPixelSample < samplesPerPixel) {
        Vec2f out = samples2D[current2DDimension][currentPixelSample];
        current2DDimension++;
        return out;
    } else {
        return Vec2f(randf(), randf());
    }
}
```

8.8.3 HaltonSampler 类

```
// sampler.h
class HaltonSampler : public Sampler {
  public:
    HaltonSampler(const json &j = json::object());

    float next1D() override;

  private:
    static float scrambledRadicalInverse(const std::vector<uint64_t> &perm, uint64_t a, uint64_t
    base);

    std::vector<std::vector<uint64_t>> perms;
};
```

```
// sampler.cpp
HaltonSampler::HaltonSampler(const json &j) {
    dimension = j.value("dimension", 4);
    if (dimension > 0) {
        dimension = std::min(dimension, static_cast<size_t>(PrimeTableSize));
    } else {
```

```
        dimension = PrimeTableSize;
    }

    perms.resize(dimension);
    for (size_t i = 0; i < dimension; i++) {
        int base = Primes[i];
        perms[i].resize(base);
        for (int j = 0; j < base; j++) {
            perms[i][j] = j;
        }
        shuffle<uint64_t>(perms[i]);
    }
}

float HaltonSampler::scrambledRadicalInverse(const std::vector<uint64_t> &perm, uint64_t a,
    uint64_t base) {
    const double invBase = 1.f / static_cast<float>(base);
    uint64_t reversedDigits = 0;
    double invBaseN = 1.;
    while (a != 0) {
        uint64_t next = a / base;
        uint64_t digit = a - next * base;
        reversedDigits = reversedDigits * base + perm[digit];
        invBaseN *= invBase;
        a = next;
    }
    return min(invBaseN * (reversedDigits + invBase * perm[0] / (1 - invBase)), ONE_MINUS_EPSILON
    );
}

float HaltonSampler::next1D() {
    if (current1DDimension < dimension) {
        int base = Primes[current1DDimension];
        float out = scrambledRadicalInverse(perms[current1DDimension], currentGlobalSample, base)
    ;
        current1DDimension++;
        return out;
    } else {
        return randf();
    }
}
```

8.9 Integrator 类

```
// integrator.h
class Integrator {
  public:
    virtual ~Integrator() = default;

    /// Return a pointer to a global default integrator
```

```
    static shared_ptr<Integrator> defaultIntegrator();

    /**
        Sample the incident radiance along a ray

        \param scene   A pointer to the underlying scene
        \param sampler A pointer to a sampler
        \param ray     The ray in question
        \return        An estimate of the radiance in this direction
     */
    virtual Color3f Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const {
        // The default integrator just returns magenta
        return Color3f(1, 0, 1);
    }
};
```

8.9.1 NormalIntegrator 类

```
// integrator.h
/// Render the shading normals of the visible geometry in the scene
class NormalIntegrator : public Integrator {
  public:
    NormalIntegrator(const json &j = json::object());

    Color3f Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const override;
};
```

```
// integrator.cpp
Color3f NormalIntegrator::Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const {
    HitInfo hit;
    if (!scene.intersect(ray, hit)) {
        return Color3f(0.f);
    }

    // Return the component-wise absolute value of the normal as a color
    return Color3f(abs(hit.sn.x), abs(hit.sn.y), abs(hit.sn.z));
}
```

8.9.2 AmbientOcclusionIntegrator 类

```
// integrator.h
/// Render an ambient occlusion image, where diffuse surfaces receive uniform illumination
class AmbientOcclusionIntegrator : public Integrator {
  public:
    AmbientOcclusionIntegrator(const json &j = json::object());

    Color3f Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const override;
};
```

```cpp
// integrator.cpp
Color3f AmbientOcclusionIntegrator::Li(const Scene &scene, Sampler &sampler, const Ray3f &ray)
    const {
    HitInfo hit;
    if (!scene.intersect(ray, hit)) {
        return Color3f(0.f);
    }

    ScatterRecord srec;
    Vec2f sample = sampler.next2D();
    if (!hit.mat->sample(ray.d, hit, sample, srec)) {
        return Color3f(0.f);
    }

    Ray3f shadowRay(hit.p, srec.scattered);
    if (scene.intersect(shadowRay, hit)) {
        return Color3f(0.f);
    }
    return Color3f(1.f);
}
```

8.9.3 PathTracerIntegrator 类

```cpp
// integrator.h
/// Render an image using recursive path tracing
class PathTracerMaterials : public Integrator {
  public:
    PathTracerMaterials(const json &j = json::object());

    Color3f Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const override;

    Color3f recursiveColor(const Scene &scene, Sampler &sampler, const Ray3f &ray, int depth)
    const;

    int m_maxBounces = 64;
};
```

```cpp
// integrator.cpp
Color3f PathTracerMaterials::recursiveColor(const Scene &scene, Sampler &sampler, const Ray3f &
    ray,
                                            int depth) const {
    HitInfo hit;
    if (!scene.intersect(ray, hit)) {
        return scene.background(ray);
    }

    ScatterRecord srec;
    Color3f emitted = hit.mat->emitted(ray, hit);
    // If we have no more bounces, just return the emitted color
    if (depth == m_maxBounces) {
        return emitted;
```

```
    }

    // Compute illumination by sampling the material BSDF
    Vec2f sample = sampler.next2D();
    if (!hit.mat->sample(ray.d, hit, sample, srec)) {
        return emitted;
    }
    Ray3f scattered(hit.p, srec.scattered);

    // If the sampled direction is specular, don't use Monte Carlo
    if (srec.isSpecular) {
        return emitted + srec.attenuation * recursiveColor(scene, sampler, scattered, depth + 1);
    }

    // Recursively trace a ray
    float pdf = hit.mat->pdf(ray.d, srec.scattered, hit);
    if (pdf == 0.f) {
        return emitted;
    }
    Color3f eval = hit.mat->eval(ray.d, srec.scattered, hit);
    return emitted + eval * recursiveColor(scene, sampler, scattered, depth + 1) / pdf;
}

Color3f PathTracerMaterials::Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const {
    return recursiveColor(scene, sampler, ray, 0);
}
```

8.10 采样函数的实现

在拥有完整的 `Sampler` 类之后，我们也就可以基于采样器去实现我们的采样函数了。

8.10.1 均匀圆盘内采样

```
// sampling.h
inline Vec2f randomInUnitDisk(const Vec2f &sample) {
    float r = sqrt(sample.x);
    float phi = 2.f * M_PI * sample.y;
    float x = r * cos(phi);
    float y = r * sin(phi);
    return Vec2f(x, y);
}
```

8.10.2 均匀球面采样

```
// sampling.h
inline Vec3f randomOnUnitSphere(const Vec2f &sample) {
    float phi = 2.f * M_PI * sample.x;
    float cosTheta = 2.f * sample.y - 1;
    float sinTheta = sqrt(1.f - cosTheta * cosTheta);
```

```
    float x = cos(phi) * sinTheta;
    float y = sin(phi) * sinTheta;
    float z = cosTheta;
    return Vec3f(x, y, z);
}
```

8.10.3 均匀球内采样

```
// sampling.h
inline Vec3f randomInUnitSphere(const Vec2f &sample) {
    float r = cbrt(randf());
    return r * randomOnUnitSphere(sample);
}
```

8.10.4 均匀半球上采样

```
// sampling.h
inline Vec3f randomOnUnitHemisphere(const Vec2f &sample) {
    float phi = 2.f * M_PI * sample.x;
    float cosTheta = sample.y;
    float sinTheta = sqrt(1.f - cosTheta * cosTheta);
    float x = cos(phi) * sinTheta;
    float y = sin(phi) * sinTheta;
    float z = cosTheta;
    return Vec3f(x, y, z);
}
```

8.10.5 余弦项加权半球上采样

```
// sampling.h
inline Vec3f randomCosineHemisphere(const Vec2f &sample) {
    float phi = 2.f * M_PI * sample.x;
    float cosTheta = sqrt(sample.y);
    float sinTheta = sqrt(1.f - cosTheta * cosTheta);
    float x = cos(phi) * sinTheta;
    float y = sin(phi) * sinTheta;
    float z = cosTheta;
    return Vec3f(x, y, z);
}
```

8.10.6 余弦项加权指数半球上采样

```
// sampling.h
inline Vec3f randomCosinePowerHemisphere(float exponent, const Vec2f &sample) {
    float phi = 2.f * M_PI * sample.x;
    float cosTheta = powf(sample.y, 1.f / (exponent + 1.f));
    float sinTheta = sqrt(1.f - cosTheta * cosTheta);
```

```
    float x = cos(phi) * sinTheta;
    float y = sin(phi) * sinTheta;
    float z = cosTheta;
    return Vec3f(x, y, z);
}
```

Chapter 9

直接光照

为了使得场景可见，场景中必须要有一个光源。除了环境光以外，有的时候自发光的光源也是提供光照的一部分。本章中我们主要讨论与发光物体有关的知识点。

9.1 光源的重要性采样

在渲染方程

$$L_r(\mathbf{x},\omega_r)=\int_{H^2} f_r(\mathbf{x},\omega_i,\omega_r)L_i(\mathbf{x},\omega_i)\cos\theta_i d\omega_i$$

中，在之前的章节我们已经介绍了

- 对余弦项 $\cos\theta_i$ 的重要性采样：例如环境光遮罩中对半球内方向的采样。
- 对 BRDF 项 $f_r(\mathbf{x},\omega_i,\omega_r)$ 的重要性采样：例如 Phong 与 Blinn-Phong 模型中对反射方向的采样。

那么，我们有没有可能对入射辐射亮度项 $L_i(\mathbf{x},\omega_i)$ 做重要性采样呢？实际上，这是几乎不可能的。我们无法判断从各个方向来的入射光中，哪个方向来的入射光偏多。

如图 9.1[1]所示，入射辐射亮度 $L_i(\mathbf{x},\omega_i)$ 可能来自于以下的部分：

- **直接光照**（direct illumination），也就是光源不经过散射直接将光子打在所求点 $\mathbf{x}$ 上。
- **间接光照**（indirect illumination），自光源出发的光子经过墙面、遮挡物等遮挡、反射、折射后，光子最终抵达所求点 $\mathbf{x}$ 上。

难以重要性采样是因为间接光照的存在，这个部分与场景的配置直接相关，只要场景中的物体位置发生变化，间接光照的布局就有可能不同。

然而，如果我们无视间接光照，假设只存在直接光照的话，就可以进行重要性采样了。换句话说，假设入射辐射亮度 $L_i(\mathbf{x},\omega)$ 等于从光源出发的出射辐射亮度 $L_e(r(\mathbf{x},\omega),-\omega)$。这样，我们就可以将渲染方程改写为

$$L_r(\mathbf{x},\omega_r)=\int_{H^2} f_r(\mathbf{x},\omega_i,\omega_r)L_e(r(\mathbf{x},\omega_i),-\omega_i)\cos\theta_i d\omega_i.$$

[1]图选自CMU 15668 Physically Based Rendering Lecture 10

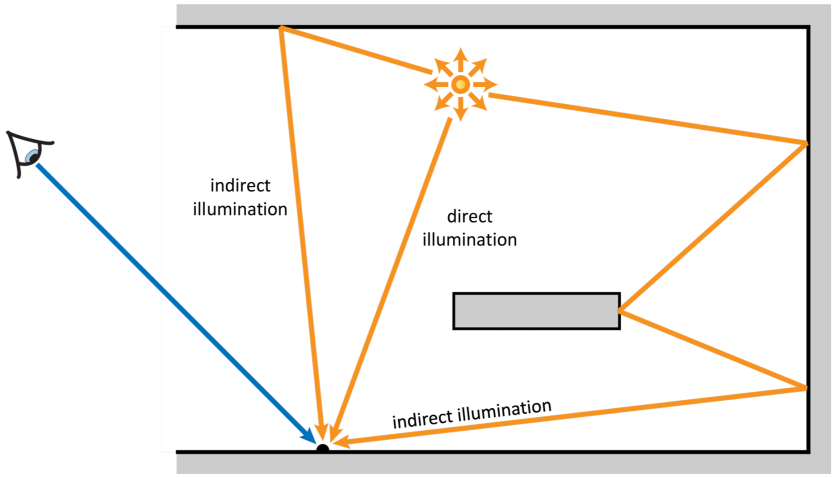


图 9.1: 直接间接光照的区别

对这个方程进行蒙特卡洛估值，对于所选方向的重要性采样决定我们的 pdf 是什么样的。然而，观察到如果我们对半球采样，我们会浪费很多的采样结果，因为实际上从光源进入半球的立体角只有很小的一部分，如图 9.2 所示。

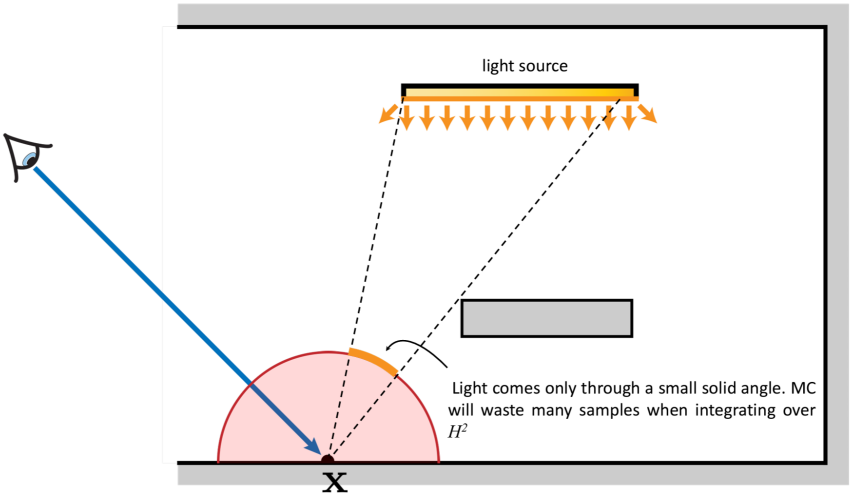


图 9.2: 光源的有效范围

因此，我们不如直接从光源进行采样。

9.1.1 改写反射方程

之前，我们需要所求点 $\mathbf{x}$，入射方向 ω_i 和出射方向 ω_r 来确定 BRDF，辐射亮度等物理量。改写到光源之后，我们假设 $\mathbf{x}$ 到光源上一点 $\mathbf{y}$ 的单位方向即 ω_i，$\mathbf{x}$ 到被反射光照射的点 $\mathbf{z}$ 的单位方向即 ω_r。这样，

我们就可以改写原来的记法：

$$L_i(\mathbf{x}, \omega_i) = L_i(\mathbf{x}, \mathbf{y})$$
$$L_r(\mathbf{x}, \omega_r) = L_r(\mathbf{x}, \mathbf{z})$$
$$f_r(\mathbf{x}, \omega_i, \omega_r) = f_r(\mathbf{x}, \mathbf{y}, \mathbf{z}).$$

微分立体角与光源上的面积微元满足以下的关系（推导见附录 B）：

$$d\omega_i = \frac{|\cos\alpha|}{||\mathbf{x}-\mathbf{y}||^2} dA$$

其中，α 是光源上的点 $\mathbf{y}$ 处法线与向量 $\mathbf{x}$-$\mathbf{y}$ 的夹角。

9.1.2 反射方程的面积形式

根据刚才的内容，我们可以将反射方程从半球形式

$$L_r(\mathbf{x}, \omega_r) = \int_{H^2} f_r(\mathbf{x}, \omega_i, \omega_r) L_i(\mathbf{x}, \omega_i) \cos\theta_i d\omega_i$$

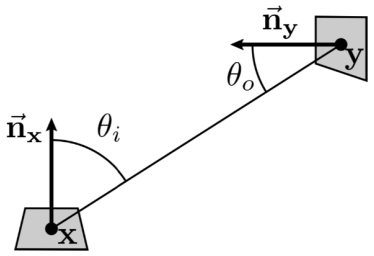
改写成面积形式

$$L_r(\mathbf{x}, \mathbf{z}) = \int_A f_r(\mathbf{x}, \mathbf{y}, \mathbf{z}) L_i(\mathbf{x}, \mathbf{y}) G(\mathbf{x}, \mathbf{y}) dA(\mathbf{y})$$

其中，$G(\mathbf{x}, \mathbf{y})$ 被称作**几何项**（geometry term），它的定义如下：

$$G(\mathbf{x}, \mathbf{y}) := V(\mathbf{x}, \mathbf{y}) \frac{|\cos\theta_i||\cos\theta_o|}{||\mathbf{x}-\mathbf{y}||^2}.$$

这里的 $V(\mathbf{x}, \mathbf{y})$ 是 8.5 中曾介绍过的可见性函数，其中的 θ_i, θ_o 如图所示。



理解几何项

对于几何项中

$$\frac{|\cos\theta_i||\cos\theta_o|}{||\mathbf{x}-\mathbf{y}||^2}$$

的部分，我们可以这么理解：光子从一个面积微元出发，击中另一个面积微元的概率会随着以下两个方面减少：

- 当两个面积微元与彼此远离，$||\mathbf{x}-\mathbf{y}||^2$ 变大（分母部分），概率则降低。
- 当两个面积微元朝向与彼此原理，$|\cos\theta_i||\cos\theta_o|$ 变小（分子部分），概率则降低，如图 9.3 所示。

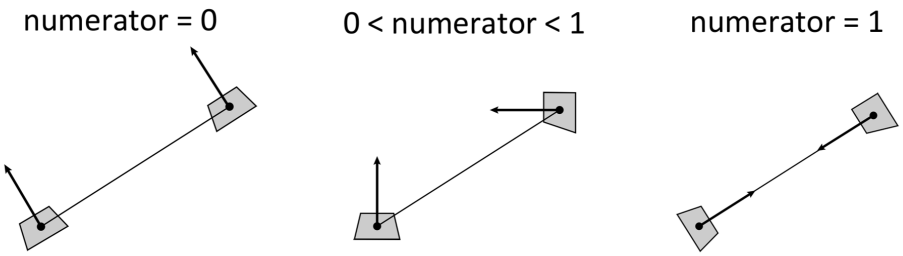


图 9.3: 分子部分描述朝向与概率的关系

那么，最终，我们如何决定什么时候使用面积形式的反射方程，什么时候又使用立体角形式的呢？任何时候，对于光源都是面积形式更合理吗？显然不是。我们应该关注光源本身的形状。在下面的小节中我们会讨论不同形式的光源，但是在进入细节的讨论之前，我们先从与它们有一些不同的环境光出发。

9.2 环境光

在之前的光线追踪部分，在光线不与任何场景物体产生交互的时候，我们就返回白色——意味着光线什么也没有碰到。然而实际上，我们可以把远处的景象做成一张贴图，然后包裹住整个半球，当光线接触到半球的某一个位置时，我们再做纹理查询即可。这就是我们常说的**环境贴图**（Environment Mapping）。

来自环境贴图的反射方程的半球形式可以写成

$$\begin{aligned} L_r(\mathbf{x}, \omega_r) &= \int_\Omega f_r(\omega_i, \omega_r) L_i(\mathbf{x}, \omega_i) \cos\theta_i d\omega_i \\ &= \int_\Omega f_r(\omega_i, \omega_r) L_{\text{env}}(\omega_i) V(\mathbf{x}, \omega_i) \cos\theta_i d\omega_i \end{aligned}$$

9.2.1 环境光的重要性采样

对于半球形式，我们希望我们的采样 pdf 与 L_{env} 正相关。我们依然使用 $\theta - \phi$ 极坐标系来进行参数化，也就是说采样概率密度函数 $p(\theta, \phi) \propto L_{\text{env}}(\theta, \phi) \sin\theta$。

9.2.2 基于表面的采样函数

球表面采样

我们可以使用之前 8.10.2 中实现过的均匀球面采样来完成球面上一点的采样。

```
// sampling.h
inline Vec3f sampleSphere(Vec3f center, float radius, const Vec2f &sample) {
    Vec3f pointOnUnitSphere = randomOnUnitSphere(sample);
    pointOnUnitSphere *= radius;
    pointOnUnitSphere += center;
    return pointOnUnitSphere;
}
```

长方形表面采样

对于一个中心为 center，两边的向量分别为 v0 和 v1 的长方形，我们可以通过如下方式在表面上采样。

```
// sampling.h
inline Vec3f sampleRect(Vec3f center, Vec3f v0, Vec3f v1, const Vec2f &sample) {
    float a = sample.x * 2 - 1;
    float b = sample.y * 2 - 1;
    return center + a * v0 + b * v1;
}
```

三角形表面采样

对于三个顶点分别为 v0, v1, v2 的三角形而言，其表面采样函数可以通过如下方式实现。

```
inline Vec3f sampleTriangle(Vec3f v0, Vec3f v1, Vec3f v2, const Vec2f &sample) {
    float a = sample.x;
    float b = sample.y;
    if(a+b >= 1) {
        a = 1-a;
        b = 1-b;
    }
    float c = 1-a-b;
    return v0 * a + v1 * b + v2 * c;
}
```

球帽表面采样

球帽（spherical cap）指的是球的一个横截面一侧的部分。如果以经纬度坐标来表示球的话，球帽通常指的就是某个纬度以上的部分。因此，对于单位球而言，我们可以通过最大天顶角的余弦值 $\cos\theta_{\max}$ 来表达球帽的大小，以此进行采样。

```
inline Vec3f randomSphericalCap(float cosThetaMax, const Vec2f &sample) {
    float phi = 2.f * M_PI * sample.x;
    float cosTheta =  lerp(cosThetaMax, 1.0f, sample.y);
    float sinTheta = sqrt(1.f - cosTheta * cosTheta);
    float x = cos(phi) * sinTheta;
    float y = sin(phi) * sinTheta;
    float z = cosTheta;
    return Vec3f(x, y, z);
}
```

9.3 环境光的多重重要性采样

尽管目前我们已经了解如何对 BRDF、光源以及余弦项做重要性采样，但我们依然要面对我们应该如果选择重要性采样的问题。具体的采样策略与上下文高度相关，我们无法下一个通用的定论。实际上，更多时候我们可能会组合不同的采样方式，然后以一定的方式将它们的结果合并在一起。

9.3.1 动机

在第七章中，我们曾经讨论过在蒙特卡洛积分中，如果 PDF 与被积函数不成比例，就可能会出现方差较高的情况。这样的方差在渲染图中表现出来的样子就是像萤火虫（fireflies）一样的白噪点。

最差的情况下，对于那些对辐射亮度贡献很大的区域，我们只采样了极少的样本，换句话说，在估计值

$$\langle F^N \rangle = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$

中，$f(x_i)$ 可能是一个很大的值，但是 $p(x_i)$ 是一个很小的值，就导致估计值变得相当大，也就显示出了接近白色的颜色，这就是我们在图中看到的萤火虫噪点。

因此，我们经常会使用多种采样策略，目的就是要减少图中的萤火虫噪点。

9.3.2 混合采样

一种很符合直觉且概念上简单的采样策略就是简单地平均两个不同的估计值。

$$\frac{1}{2} \left(\frac{1}{N_1} \sum_{i=1}^{N_1} \frac{f(x_i)}{p_1(x_i)} + \frac{1}{N_2} \sum_{i=1}^{N_2} \frac{f(x_i)}{p_2(x_i)} \right)$$

但是这个方法实际上并没有真的帮助我们减少方差。方差是可叠加的，我们需要一种策略给这两种估计值标记权重。

更有帮助的策略是通过**平均 PDF**（average PDF）来采样。

$$\frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{0.5(p_1(x_i) + p_2(x_i))}$$

实现

假设我们有两个采样函数以及其对应的 PDF。

```
float sample1(Sampler sampler);
flaot sample2(Sampler sampler);
float pdf1(float x);
float pdf2(float x);
```

我们需要实现一个新的平均采样函数 `sampleAvg()` 以及平均 PDF 函数 `pdfAvg()`。

```
float sampleAvg(float rnd) {
    float Prob1 = 0.5f; //< 也许之后需要调整这个概率
    if(rnd < Prob1) return sample1(sampler / Prob1);
    return sample2((rnd-Prob1)/(1-Prob1));
}

float pdfAvg(float x) {
    return 0.5 * (pdf1(x) + pdf2(x));
}
```

如果我们要调整第一个估计值被采样的概率，我们也要相应地调整 `pdfAvg()` 的内容。

```
float sampleAvg(float rnd) {
    float Prob1 = 0.25f;
    if(rnd < Prob1) return sample1(sampler / Prob1);
    return sample2((rnd-Prob1)/(1-Prob1));
}

float pdfAvg(float x) {
    return 0.25f * pdf1(x) + 0.75f * pdf2(x);
}
```

9.3.3 多重重要性采样

在混合采样中我们已经有了多重重要性采样的意识，对于不同的估计值我们给予不同的权重。对于多个估计值，我们也可以分别给予一个总和为 1 的权重。

$$\langle F^{\sum N_s} \rangle = \sum_{s=1}^{M} \frac{1}{N_s} \sum_{i=1}^{N_s} w_s(x_i) \frac{f(x_i)}{p_s(x_i)}$$

其中

$$\sum_{s=1}^{M} w_s(x) = 1.$$

特别地当 $N = 2$ 时，这是我们一般在光线追踪中多重采样的规模。一份采样直接光照，另一份采样间接光照，我们很快会在第十章中讨论实现中的二重重要性采样。

$$\langle F^{\text{MIS}} \rangle = w_1(x_1) \frac{f(x_1)}{p_1(x_1)} + w_2(x_2) \frac{f(x_2)}{p_2(x_2)}$$

其中，$w_1(x) + w_2(x) = 1$。

下面的问题就是我们应该如何选择我们的权重。在上一章我们曾经介绍过，比较常用的两种权重选择为**平衡启发式**（balance heuristic）以及**幂启发式**（power heuristic）。

平衡启发式选择权重

$$w_s(x) = \frac{p_s(x)}{\sum_j p_j(x)}$$

幂启发式选择权重

$$w_s(x) = \frac{p_s(x)^\beta}{\sum_j p_j(x)^\beta}$$

9.3.4 采样模型

多重采样模型

在**多重采样**（Multi-sample）模型策略下，我们确定性地为第 s 种采样策略分布 N_s 个采样点，然后计算估计值

$$\langle F^{\sum N_s} \rangle = \sum_{s=1}^{M} \frac{1}{N_s} \sum_{i=1}^{N_s} w_s(x_i) \frac{f(x_i)}{p_s(x_i)}.$$

其中，$\sum N_s = N$ 是我们一共拥有的采样点数量。

单一采样模型

如果我们只想要一个采样点，多重采样的策略就不可用了。这个时候，我们可以随机地从我们的策略中选择出其中一个策略，使用**单一采样**（1-sample）模型。假设其为第 s 个策略，则估计值

$$\langle F^1 \rangle = w_s(x_i) \frac{f(x)}{q_s p_s(x_i)}.$$

其中，q_s 是使用策略 s 的概率，并且我们有 $\sum_{s=1}^{N} q_s = 1$。在单一采样中，如果我们使用平衡启发式选择权重，注意到

$$\langle F^1 \rangle = \frac{q_s p_s(x)}{\sum_j q_j p_j(x)} \frac{f(x)}{q_s p_s(x)} = \frac{f(x)}{\sum_j q_j p_j(x)}.$$

这与我们在混合采样中得到的平均 PDF 拥有一样的形式。

9.3.5 Background 类

目前我们已经讨论了直接光照的采样问问题，但是在目前的讨论中，我们经常会预设如果光线没有击中任何物体，就默认返回白色。但事实上，光线没有击中物体的描述实际上对应的是光线最终与无限远处的某个物体相交，因此，我们需要一个 `Background` 类，来提供这个无限远处的物体的信息。它可以提供单一颜色，也可以是环境贴图，所以一个通用的 `Background` 类是很重要的。

```
// background.h
class Background {
  public:
    /// Return a pointer to a global default background
    static shared_ptr<Background> defaultBackground();

    virtual ~Background() = default;

    virtual Color3f value(const Ray3f &ray) const = 0;
};
```

对于返回固定颜色的环境，我们可以简单地提供一个默认的 `ConstantBackground` 类。显然，它也可以作为 `Background` 类中的默认指针类型。

```
// background.h
class ConstantBackground : public Background {
  public:
    ConstantBackground(const Color3f &c) : color(c) {}

    ConstantBackground(const json &j = json::object());

```

```
    Color3f value(const Ray3f &ray) const override { return color; }

    Color3f color;
};
```

9.3.6 环境贴图与 ImageBackground 类

显然，另一种非常常用且必要的无限远环境就是通过贴图提供的环境贴图。我们可以使用一个单独的 ImageBackground 类来承载这个任务。

```
// background.h
class ImageBackground : public Background {
  public:
    ImageBackground(const json &j = json::object());

    Color3f value(const Ray3f &ray) const override;

    Image3f tex;
};
```

对于环境贴图而言，最重要的就是 value 函数。由于我们是处于一个无限大的天空球内，因此，我们只需要一个射线方向，然后根据这个射线方向，返回对应的贴图采样值即可。因此，这个实现这个函数实际上也只是简单的坐标变换。

```
// background.cpp
Color3f ImageBackground::value(const Ray3f &ray) const {
    Vec3f dir = normalize(ray.d);
    float phi = atan2(dir.y, dir.x);
    float theta = asin(dir.z);
    float u = (phi + M_PI) / (2 * M_PI);
    float v = (theta + M_PI / 2) / M_PI;

    float x = (1 - u) * (tex.sizeX() - 1);
    float y = (1 - v) * (tex.sizeY() - 1);
    int i = clamp(static_cast<int>(round(x)), 0, tex.sizeX() - 1);
    int j = clamp(static_cast<int>(round(y)), 0, tex.sizeY() - 1);
    return tex(i, j);
}
```

9.4 硬阴影光源

硬阴影光源指的是会产生没有边缘过度、棱角分明的硬阴影的光源。其中比较常见的是点光源、聚光灯光源和方向光源。下面我们会一一介绍。

9.4.1 点光源

首先是通过自一个概念上的点向外发射光线的光源，这种光源通常被我们称作**点光源**（point light）。点光源向四面八方（omnidirectional）发射光线，通常可以通过中心点 $\mathbf{p}$ 和向外的发光通量 Φ 来定义。

点光源需要通过一个狄拉克函数来描述，这是因为点光源在单一方向上只会照亮一个点。这个说法听起来有点不符合直觉，但是事实上就是如此。来自点 $\mathbf{p}$ 出发虽然有无数的方向，但是在单一方向上，它只会击中一个目标点 $\mathbf{z}$。因此，我们可以将点光源 $\mathbf{x}$ 至发光点 $\mathbf{z}$ 的辐射亮度表达为

$$\begin{aligned} L_r(\mathbf{x},\mathbf{z}) &= \int_{\mathcal{A}_e} f_r(\mathbf{x},\mathbf{y},\mathbf{z}) L_e(\mathbf{y},\mathbf{x}) V(\mathbf{x},\mathbf{y}) \frac{|\cos\theta_i||\cos\theta_o|}{||\mathbf{x}-\mathbf{y}||^2} dA(\mathbf{y}) \\ &= \int_{\mathcal{A}_e} f_r(\mathbf{x},\mathbf{y},\mathbf{z}) \frac{\Phi}{4\pi} \delta(\mathbf{y}-\mathbf{p}) V(\mathbf{x},\mathbf{y}) \frac{|\cos\theta_i||\cos\theta_o|}{||\mathbf{x}-\mathbf{y}||^2} dA(\mathbf{y}) \\ &= \frac{\Phi}{4\pi} f_r(\mathbf{x},\mathbf{p},\mathbf{z}) V(\mathbf{x},\mathbf{p}) \frac{|\cos\theta_i|}{||\mathbf{x}-\mathbf{p}||^2} \end{aligned}$$

同理，因为点光源本身只是一个点，并没有面积，所以严格来说场景内任意随机光线击中点光源的概率应该是 0。在对光源进行重要性采样时，如果目标光源是一个点光源，那么概率密度函数 `pdf` 应该总是返回 0。

像这类本身大小为 0，现实中不存在、仅作为理想模型的光源就被我们称作 **Delta 光源**（delta light）。

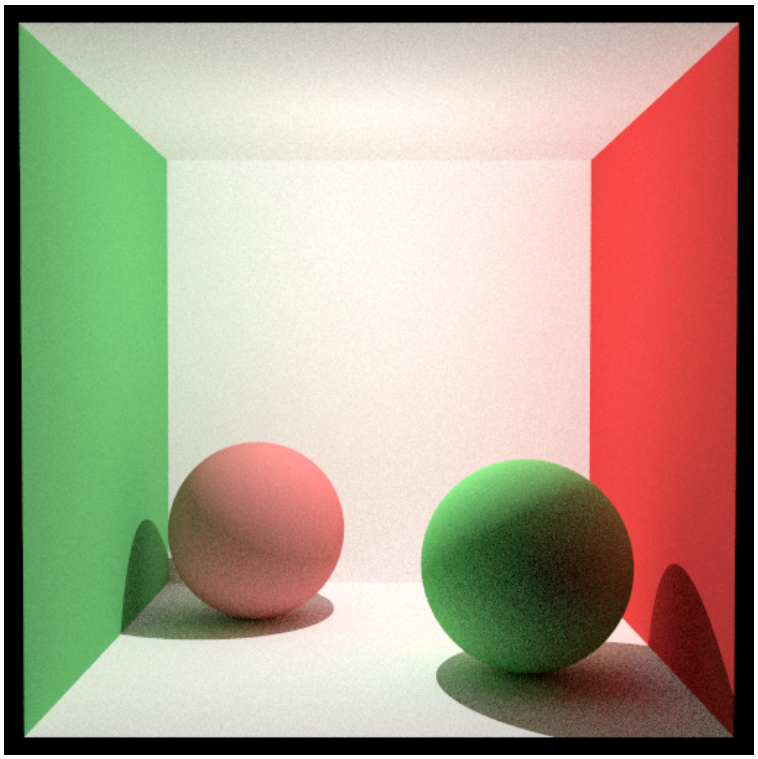


图 9.4: 点光源场景（PathTracerMIS 积分器，128 采样点/像素，深度 64）

上图中为使用多重重要性采样的路径追踪器渲染的点光源康奈尔方盒。如图所示，球的阴影产生了硬的边缘，且并不能在图中看到一个具体的光源位置。点光源在该图中位于图像中心。

9.4.2 聚光灯光源

聚光灯光源（spotlight）与点光源的相同之处在于它们都假设光发射自一个抽象的点，因此它也是一个 Delta 光源。但是在聚光灯光源中，光线的辐射亮度显然应该是与方向有关的。通常，我们通过一个发光点

$\mathbf{p}$ 以及一个与方向有关的辐射强度函数 I 来定义。

$$L_r(\mathbf{x}, \mathbf{z}) = I(\mathbf{p}, \mathbf{x}) f_r(\mathbf{x}, \mathbf{p}, \mathbf{z}) V(\mathbf{x}, \mathbf{p}) \frac{|\cos\theta_i|}{||\mathbf{x} - \mathbf{p}||^2}.$$

与点光源类似，在光源的重要性采样中，如果目标是聚光灯光源，那么概率密度函数 `pdf` 也应该总是返回 0。

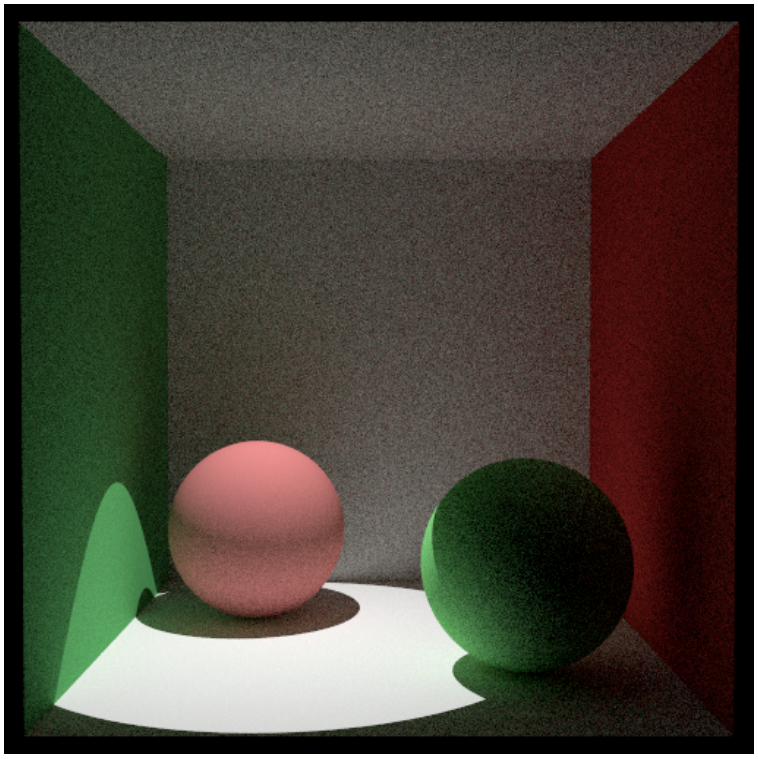
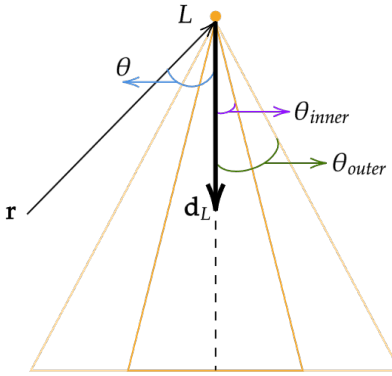


图 9.5: 无衰减聚光灯光源场景（PathTracerMIS 积分器，128 采样点/像素，深度 64）

上图中显示的是与点光源同场景下其余配置不变的聚光灯场景。为了一个更真实的聚光灯效果，在实现中，我们也可以考虑通过位置 $\mathbf{p}$、光源方向 $\mathbf{d}_L$ 以及两个角度 θ_{inner} 和 θ_{outer} 来定义聚光灯，以控制聚光灯的角度以及距离衰减。如下图所示。



对于入射光线 $\mathbf{r} = \mathbf{o} + t\mathbf{d}$ 而言，如果光线的反方向 $-d$ 与光源的方向 $\mathbf{d}_L$ 的夹角小于 θ_{outer}，我们就可以认为这条光线来自于聚光灯的光锥外。θ_{inner} 和 θ_{outer} 定义了聚光灯的衰减范围。在内角度的范围内，聚光灯光照不发生衰减，而在内角到外角的范围内，聚光灯的光强逐渐衰减至 0。

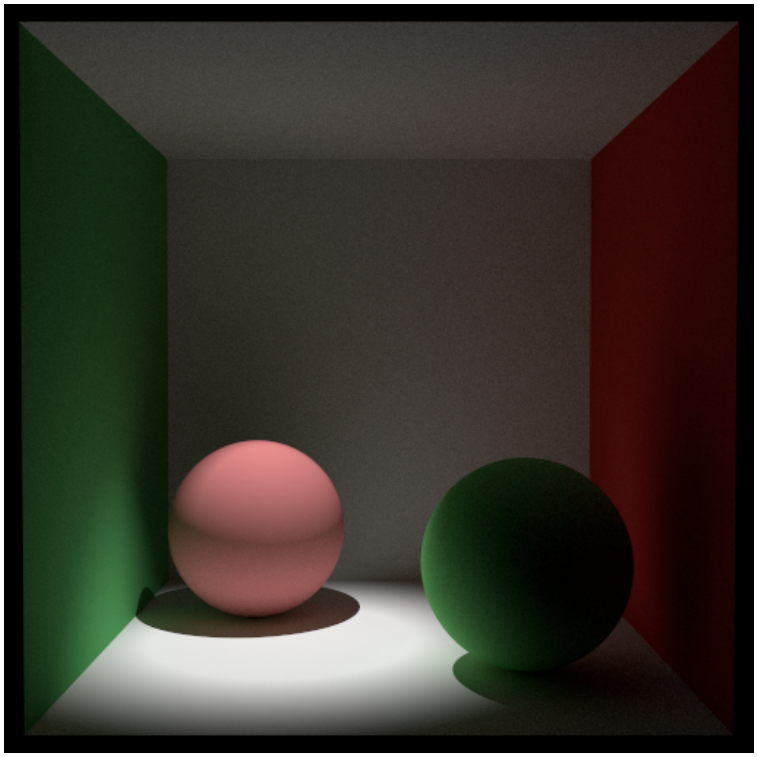


图 9.6: 有衰减聚光灯光源场景（PathTracerMIS 积分器，1024 采样点/像素，深度 128）

如上图所示，在考虑到边缘衰减之后，聚光灯在地面的光照效果，其边缘是柔和的，显示了聚光灯光线聚集在中心的特性。

9.4.3 方向光源

方向光源（directional light）描述的是来自于（概念上）无穷远[2]并且只有单一方向的平行光源。因此，方向光源通常通过方向 ω_d 和辐射亮度 L_d 描述。

方向光源也需要通过狄拉克函数来描述，因为除了方向光源的方向，其余方向都不应该产生光照。积分式

$$L_r(\mathbf{x}, \omega_r) = \int_{\mathcal{H}^2} f_r(\mathbf{x}, \omega_i, \omega_r) L_e(r(\mathbf{x}, \omega_i), -\omega_i) \cos\theta_i d\omega_i$$

中，狄拉克函数应该出现在 L_e 一项中，

$$L_e(\mathbf{y}, \omega) = V(\mathbf{y}, \omega_d) L_d \delta(\omega_d - \omega),$$

将其代入积分式，

$$L(\mathbf{x}, \omega_r) = f_r(\mathbf{x}, \omega_d, \omega_r) V(\mathbf{x}, \omega_d) L_d \cos\theta_d.$$

[2]实际上在很多时候我们也会把距离足够远的光源视作方向光源，例如阳光、月光等。

9.5 软阴影光源

区域光源（Area Light）是一种由某种形状定义的光源，该形状的整个表面都会向特定方向发光。从实现的角度来说，我们可以认为区域光源就是 `Light` 和 `Shape` 的结合，另外它需要提供一个表面上每一点的辐射亮度方向分布。通常来说，区域光源的辐射度量需要通过计算整个表面上的某一数值的积分得到，而且这个积分通常没有解析解（虽然这种性质很适合进行蒙特卡洛积分）。然而，这个复杂度换来的效果确实是很好的—有着更加真实的光照以及软阴影的效果。

9.5.1 四边形光源

四边形光源（Quad Light）顾名思义，就是发光区域为一个四边形的光源。由于是区域光源，我们不再把光源抽象为一个点，这也就意味着有光照到更多的地方，也有光照到相对更少的地方。

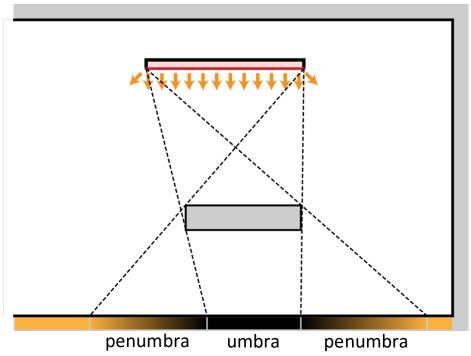


图 9.7: 四边形光源

对于一个被照到的物体，如图所示，其存在光源上任何光都照不到的区域。这个区域被我们称为**本影**（umbra）区域，而在本影的周围，有光源中部分光可以照到，且越往外照亮的程度也越高的范围，这个范围被称作**半影**（penumbra）区域。本影与环绕本影的半影共同构成了光源造成的**软阴影**（soft shadow）。

9.5.2 球光源

球光源（Sphere）就是发光区域是一个球的光源。球本身会通过球心 $\mathbf{o}$，半径 r 来定义，发光功率通过 Φ 或者出射辐射亮度 L_e 来定义。

一个半径为 r 的球的表面积为 $4\pi r^2$，因此球光源的发光面积也就是这么大。关于球光源本身并没有多少要说的，主要讨论一下在球光源上随机采样一点的方法。

在球光源上随机采样一点

第一种方法就是简单地在球的表面上均匀地采样一点。这个方法听起来可行，但是实际上我们从任何一个点看球，都只能看到球的某个半球面，而另外半球则是处于完全不能看到的范围。这就意味着我们随机采样的样本中，可能有接近一半的无效采样。

第二种方法是均匀地在可见半球（帽）上采样一点。这种思路肯定是更符合重要性采样的思想的，从着色点看不到的那个半球完全不会对着色点产生贡献。用我们上一章中提到的帽盒定理，这样的采样是很好实现的。

第三种方法基于第二种方法作出进一步的改进。我们不再通过对球面均匀采样，而是对可见球帽对应的立体角进行均匀采样。这样的采样方法能够更好地匹配光照与表面夹角的余弦项的重要性。

需要注意的是，以上的采样方法对应的坐标系并不相同，因此在计算 PDF 的时候要注意各自的测量。

9.5.3 网格光源

网格光源（Mesh Light）是形状可能更为复杂的区域光源。网格上的每个表面都有可能沿着某个方向发光。

假设一个网格有 k 个面，我们也需要一种重要性采样策略，来应对 k 个面。

预处理

我们可以先设计好一个离散的 PDF，p_Δ，用以通过面积大小来随机选择网格中的某个多边形。

$$p_\Delta(i) = \frac{A(i)}{\sum_k A(k)}$$

运行时

- 采样第 i 个多边形，并在第 i 个多边形上采样一个点 $\mathbf{x}$。
- 计算采样该点的 PDF。

$$p_A(\mathbf{x}) = p_\Delta(i) p_A(\mathbf{x}|i) = \frac{1}{\sum A(k)}$$

9.6 光源的实现

9.6.1 `Light` 类

在 DIRT 的基础框架中，我们并没有明确对光源进行定义，DIRT 提供了一个的基础的 `DiffuseLight` 类，这个类继承自 `Material` 类：

```
// material.h
class DiffuseLight : public Material {
  public:
    DiffuseLight(const json &j = json::object());

    // Returns a constant Color3f if the ray hits the surface on the front side.
    Color3f emitted(const Ray3f &ray, const HitInfo &hit) const override;

    bool isEmissive() const override { return true; }

    Color3f emit; ///< The emissive color of the light
};
```

实际上，根据我们的光线追踪的算法，当光线接触表面之后，光线就会检查表面是否会发光。如果是发光表面，则计算其光照。因此，发光的概念本身包括在 `Material` 中是合理的。在未来的一些场景中，我们可能会更希望对场景中不同种类的光源进行评估，将光源单独封装成一个 `Light` 类，不仅有助于增加代码的可读性，也使得我们在实现新的光源种类时能够简化流程。

```
// material.h
class Light : public Material {
  public:
    virtual ~Light() = default;

    // Returns a constant Color3f if the ray hits the surface on the front side.
    Color3f emitted(const Ray3f &ray, const HitInfo &hit) const override
    { return Color3f(0.f); }

    bool isEmissive() const override { return true; }

    Color3f emit; ///< The emissive color of the light
};

class DiffuseLight : public Light {
  public:
    DiffuseLight(const json &j = json::object());

    Color3f emitted(const Ray3f &ray, const HitInfo &hit) const override;

    float pdf(const Vec3f &dirIn, const Vec3f &scattered, const HitInfo &hit)
        const override { return 0.0f; }
};
```

9.6.2 DeltaPoint 类

为了加入 Delta 光源，我们必须要保证它们有响应的图形数据结构来表示它们。为了不更改我们的代码结构，我们可以像四边形和球形一样，继续使用 `Surface` 类。基本的框架可以与 `Quad` 保持一致。

```
// deltapoint.h
class DeltaPoint : public Surface {
  public:
    DeltaPoint(const Vec2f &size = Vec2f(0.f), shared_ptr<const Material> material = Material::
    defaultMaterial(),
         const Transform &xform = Transform());
    DeltaPoint(const Scene &scene, const json &j = json::object());

    Box3f localBBox() const override;
    Vec3f sample(const Vec3f &o, const Vec2f &sample) const override;
    Vec3f sampleOn(const Vec2f &sample) const override;
    float pdf(const Vec3f &o, const Vec3f &v) const override;
    float pdfOn(const Vec3f &v) const override;
    bool intersect(const Ray3f &ray, HitInfo &hit) const override;
    bool isEmissive() const override { return m_material && m_material->isEmissive(); }

  protected:
    Vec2f m_size = Vec2f(0.f);
```

```
    shared_ptr<const Material> m_material;
    shared_ptr<const MediumInterface> m_medium_interface;
};
```

注意到，`m_size` 已经被显式地记录为零向量。实际上，考虑到性能我们甚至可以不需要这个字段，实际运行时我们不会在任何场景下使用 `DeltaPoint` 的大小。这个类应该是 Delta 光源专用的类，其余的场景中的任何物品都不应该被保存为 `DeltaPoint` 对象。

关于 `DeltaPoint` 的重载方法，其实基本都非常简单。

intersect 函数

对于给定的射线 `ray`，我们要判断该射线是否和该点相交，就是求该点是否在该射线上。

```
// deltapoint.h
bool DeltaPoint::intersect(const Ray3f &ray, HitInfo &hit) const {
    auto tray = m_xform.inverse().ray(ray);
    float t = -tray.o.z / tray.d.z;
    auto p = tray(t);
    if(abs(p.x) >= Epsilon || abs(p.y) >= Epsilon) {
        return false;
    }
    Vec3f gn = normalize(m_xform.normal({0, 0, 1}));
    Vec2f uv = Vec2f(0.f);

    hit = HitInfo(t, m_xform.point(p), gn, gn, uv, m_material.get(), m_medium_interface.get(),
    this);
    return true;
}
```

注意浮点数的化整误差，在这里我们希望计算一个值是否是 0 时，我们总是比较其绝对值是否小于小常数 `Epsilon`。

采样函数以及概率密度

我们有两种采样场景：一是给定一个起点，然后返回一个从起点到光源上一点的方向；二是直接在光源上采样一个点。因为 Delta 光源没有大小，所以在光源上选择点时，总是选择光源所在的那个点。而对于密度函数，根据狄拉克 δ 函数的定义，只要方向不是穿过的方向，其概率密度就是 0。而如果就是穿过光源的方向，根据我们的实现，我们则应该返回概率密度为 1。根据这些，我们可以进行以下的实现了。

```
// deltapoint.cpp
/// sample a point on the light
Vec3f DeltaPoint::sampleOn(const Vec2f &sample) const {
    return m_xform.point(Vec3f(0.f));
}

// given an origin o, sample a direction towards the light
Vec3f DeltaPoint::sample(const Vec3f &o, const Vec2f &sample) const {
    Vec3f center = m_xform.point(Vec3f(0.f));
    return center - o;
}

```

```
// return the pdf of picking the point v on the light
float DeltaPoint::pdfOn(const Vec3f &v) const {
    return 1.f;
}

// given an origion o and a direction v, return the pdf of the direction
float DeltaPoint::pdf(const Vec3f &o, const Vec3f &v) const {
    auto ray = Ray3f(o, v);
    HitInfo hit;
    if(intersect(ray, hit)) return 1.0f;
    return 0.f;
}
```

9.6.3 PointLight 类

对于点光源而言，唯一重要的就是其位置，因此，点光源类需要一个 position 字段。

```
// light.h
class PointLight : public Light {
  public:
    PointLight(const json &j = json::object());

    Color3f emitted(const Ray3f &ray, const HitInfo &hit, bool forced = false) const override;

    Vec3f position;
};
```

emitted 函数

点光源的 emitted() 函数也很简单，我们也只需要判断求交结果中的交点 hit.p 是否就是点光源即可。如果是，则说明光线经过了光源，带上了光照。否则光线没有经过光源。考虑到化整误差，我们同样与 Epsilon 进行比较，而不是 0。

```
// light.h
Color3f PointLight::emitted(const Ray3f &ray, const HitInfo &hit, bool forced) const {
    // only emit from the normal-facing side
    if (forced) return emit;

    if (abs(hit.p.x-position.x)<Epsilon
     && abs(hit.p.y-position.y)<Epsilon
     && abs(hit.p.z-position.z)<Epsilon) {
        return emit;
    } else {
        return Color3f(0.f);
    }
}
```

9.6.4 SpotLight 类

对于聚光灯而言，定义它需要光源的位置、光源的光照方向、内角、外角、角度衰减效率以及距离衰减效率。因此，这些都需要作为聚光灯类的内置字段。

```
// light.h
class SpotLight : public Light {
  public:
    SpotLight(const json &j = json::object());

    Color3f emitted(const Ray3f &ray, const HitInfo &hit, bool forced = false) const override;

    float pdf(const Vec3f &o, const Vec3f &v) const override {
        return 0.f;
    }

    Vec3f sample(const Vec3f &o, const Vec2f &sample) const override {
        return normalize(position - o);
    }

    Vec3f position;                // < position of the light
    Vec3f direction;               // < direction of the light
    float cosInnerAngle;           // < cosine of the inner angle
    float cosOuterAngle;           // < cosine of the outer angle
    float falloff = 0.5f;          // < falloff power parameter for the angular attenuation
    float attenuation = 0.1f;      // < attenuation for distance
};
```

emitted 函数

聚光灯的 `emitted()` 函数可以根据光线的方向 `ray.d` 与光源方向 `direction` 的夹角余弦值进行对比。需要注意的是，cos 函数在 $[0, \pi/2]$ 的范围内是单调递减的，因此 $\cos\theta_1 < \cos\theta_2 \Leftrightarrow \theta_1 > \theta_2$。根据两个向量的夹角值 $\cos\theta =$ `dot(-normalize(ray.d), normalize(direction))`，我们可以分成三种情况讨论：

- **情况 1**：$\cos\theta \in [\cos\theta_{\text{inner}}, 1]$：光线处于光锥的内圈，在这里光线不会随角度衰减。
- **情况 2**：$\cos\theta \in [\cos\theta_{\text{outer}}, \cos\theta_{\text{inner}}]$: 光线处于内圈与外圈之间，在这里光线会随角度衰减。
- **情况 3**：$\cos\theta \in [0, \cos\theta_{\text{outer}}]$: 光线处于光锥的外圈之外，这里光线并没有击中聚光灯的有效范围。

根据这些分析，我们就可以写出聚光灯函数的 `emitted()` 函数了。

```
Color3f SpotLight::emitted(const Ray3f &ray, const HitInfo &hit, bool forced) const {
    // only emit from the normal-facing side
    if (forced) return emit;

    if (abs(hit.p.x-position.x)<Epsilon
     && abs(hit.p.y-position.y)<Epsilon
     && abs(hit.p.z-position.z)<Epsilon) {
        Vec3f lightDir = -normalize(ray.d);
        float cosTheta = dot(lightDir, normalize(direction));

        float angleAttenuation;
        // case 1
        if(cosTheta > cosInnerAngle) {
            angleAttenuation = 1.0f;
        } else if(cosTheta > cosOuterAngle) { // case 2
```

```
            float t = (cosTheta - cosOuterAngle) / (cosInnerAngle - cosOuterAngle);
            angleAttenuation = pow(t, falloff);
        } else { // case 3
            angleAttenuation = 0.0f;
        }

        float distanceAttenuation = 1.0f / (1.0f + attenuation * hit.t * hit.t);
        return emit * angleAttenuation * distanceAttenuation;
    }

    return Color3f(0.f);
}
```

Chapter 10

随机路径追踪

有了对光线追踪算法、辐射度量学以及蒙特卡洛积分的了解，我们下面就可以来讨论将蒙特卡洛积分应用到光线追踪中的**路径追踪**（path-tracing）算法了。一般地，我们都认为简单的*路径追踪*一词指的是基于蒙特卡洛积分的**随机路径追踪**（Stochastic Path Tracing, SPT），对于之后会讨论的光路径追踪、双向路径追踪和光子追踪，它们也都属于某一种形式的路径追踪。

10.1 光传输方程

光传输方程（Light Transportation Equation, LTE）描述了场景中辐射亮度的分布。它通过表面发光、BSDF 以及到达特定点 $\mathbf{x}$ 的入射光来描述该点反射出的辐射亮度。本章节中我们不讨论参与介质的作用，这个部分放在下一章节中讨论。

一个基本形式的光传输方程，也被称作*渲染方程*（rendering equation），形如

$$L_o(\mathbf{x},\omega_o) = L_e(\mathbf{x},\omega_o) + L_r(\mathbf{x},\omega_o)$$

其中，L_o 是出射方向亮度，L_e 是发光项，L_r 是反射向。

10.1.1 传递理论

在第五章中，我们曾经介绍过辐射亮度的一个性质——辐射亮度沿光线不变。也就是说，

$$L_i(\mathbf{x},\omega) = L_o(r(\mathbf{x},\omega),-\omega).$$

其中，$r(\mathbf{x},\omega)$ 也被叫做**射线投射函数**（raycasting function），它计算的是从 $\mathbf{x}$ 出发经过方向 ω，第一次击中某一表面时产生的交点 $\mathbf{p}$。

根据这个关系，我们可以改写渲染方程为

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + \int_{H^2} f_r(\mathbf{x},\omega',\omega)L(r(\mathbf{x},\omega'),-\omega')\cos\theta' d\omega'$$

在这里，因为两侧都只有出射辐射亮度，所以我们可以将两侧的出射下标“o”都省略掉。这里的优化思想的核心就是我们不再有出射辐射亮度和入射辐射亮度两种物理量需要计算，而是只需要计算出射辐射亮度即可。

10.2 路径追踪算法

路径追踪的思想就是将蒙特卡洛积分引入光线追踪算法。在光线击中物体后，我们用蒙特卡洛随机采样下一个方向。也就是说，对于 LTE，

$$L(\mathbf{x},\omega) = L_e(\mathbf{x},\omega) + \int_{H^2} f_r(\mathbf{x},\omega',\omega)L(r(\mathbf{x},\omega'),-\omega')\cos\theta' d\omega'$$

我们利用蒙特卡洛积分计算其估值，

$$L(\mathbf{x},\omega) \approx L_e(\mathbf{x},\omega) + \frac{f_r(\mathbf{x},\omega',\omega)L(r(\mathbf{x},\omega'),-\omega')\cos\theta'}{p(\omega')}$$

这样，我们就可以有路径追踪算法的基本雏形的伪代码。

```
color(Point p, Vec3f w, int moreBounces) {
    if (moreBounces == 0) return Le(p, -w);

    // sample recursive integral
    nextDir = sample from BRDF
    return Le(p, -w) + BRDF * color(trace(p, nextDir), moreBounces-1) * dot(n, nextDir) / pdf(
    nextDir);
}
```

10.2.1 改善质量

上述的方法中，我们在路径追踪的每一次迭代中采样了一个方向。如果想要改善我们的渲染质量，一个非常符合直觉的做法就是每次采样多个方向。然而，这是错误的。我们绝不能在一次迭代中渲染采样超过一根散射方向，否则随着迭代层数的增加，需要采样的光线数量将指数级增长，很快就会将计算机的运算能力消耗殆尽。

因此，我们只能从一开始采样更多的光线。每条光线开启一个新的路径。

10.2.2 俄罗斯轮盘赌

下一个问题是，我们什么时候终止迭代?

如果我们每次都是使用固定深度，然后超过深度就终止迭代的话，是有可能产生误差的。因此，我们提出**俄罗斯轮盘赌**的概念。在概念上这很简单，原意指的就是在左轮枪膛中塞入一颗子弹，然后任意转动轮盘，接下来对着自己开一枪。那么，假设枪膛一共可以装 n 发子弹，自己存活的概率就是 $\frac{n-1}{n}$，死掉的概率就是 $\frac{1}{n}$。

在路径追踪的迭代过程中，我们也可以这么做。每次路径追踪有 $1-p_{rr}$ 的概率暴毙。

定义（俄罗斯轮盘赌）对于估计值 X，定义新的估值 X_{rr} 如下：以 p_{rr} 的概率，我们正常计算当前估值的值，以 $1-p_{rr}$ 的概率直接将其值记作 0。那么 X_{rr} 就叫做 X 的**俄罗斯轮盘赌**（Russian Roulette），其中 p_{rr} 也被叫做**存活率**（survival rate）。

观察到，

$$E[X_{rr}] = p_{rr} \cdot E\left[\frac{X}{p_{rr}}\right] + (1 - p_{rr}) \cdot E[0] = E[X].$$

因此，从期望上来看，采用轮盘赌并不会影响正确性，这是一个无偏的估计值。

注意，俄罗斯轮盘赌实际上实会增加方差的。但是，如果选择一个合适的 p_{rr}，使得我们能跳过一些产生贡献很小但是计算很昂贵的追踪迭代，俄罗斯轮盘赌也实际上会提高效率。

10.3 划分积分

一些渲染算法被能够很好地求解在一定限定情况下的 LTE，但是对于其它的通用情况则处理得很糟糕，例如 Whitted 的最原始的光线追踪算法，它就只处理高光反射而忽略来自于漫反射和发光 BSDF 的众多散射光。因此，一个思想是我们可以将积分分开成多个部分，然后对每一个部分使用适合它的算法去计算。

一个常见的积分划分的方法为

$$L(\mathbf{p}_1 \to \mathbf{p}_0) = L_e + L_{\text{dir}} + L_{\text{indir}}$$

其中，L_e 是发光项，可以简单地通过计算 $\mathbf{p}_1$ 处的出射辐射亮度即可得到。第二项是直接光照项，我们可以通过精确的计算来获得直接光照。第三项才是间接光照项，这一项才是唯一一个涉及递归计算的。我们可以对间接光照使用更快速但是略微不准确的算法。这样，我们就可以在质量和速度之间取得一个很好的平衡。

10.3.1 下一事件估计

下一事件估计（Next Event Estimation, NEE）是路径追踪算法中的一种重要采样技术，用于加速收敛和减少噪点。它之所以叫这个名字，是因为这个算法在当前着色点（也就是当前事件）的基础上，显式地估计下一个事件（也就是直接光照）的贡献。具体来说，

- 在路径追踪过程中, 对于每个交点（也就是当前着色点处），根据材质 BSDF，预测可能的光线传输方向 (即下一事件)。
- 根据出射方向，计算当前着色点到光源的可见性。这通常是通过投射一条阴影射线（shadow ray）来实现的。从当前着色点向光源方向发射一条射线，检查是否与场景中的物体相交，如果相交，则说明光源被遮挡，不直接照亮当前着色点；否则，光源可见。
- 如果光源可见, 则计算光源对当前着色点的直接照明贡献。这需要结合光源的发光强度、光源与着色点之间的距离衰减、光源的表面积 (对于面光源) 或立体角 (对于点光源或方向光源) 等因素。
- 将直接照明的贡献乘以 BSDF 的值，再除以该方向被采样的概率，作为蒙特卡洛估计值。这是因为我们显式地采样了一个特定的方向 (光源方向)，而不是随机采样，所以需要除以该方向的概率密度来进行均衡。
- 将 NEE 的结果与常规路径追踪的结果相加, 作为最终的像素颜色。

注意，在 NEE 中，我们不要出现重复计数的问题。观察下面的伪代码。

```
NEE::color(Point p, Vec3f w, int moreBounces) {
    if (moreBounces == 0) return L_e;

    // NEE:  compute L_dir by sampling the light
    w1 = sample from light;
    L_dir = BRDF * color(trace(p, w1), 0) * dot(n, w1)/pdf(w1);

    w2 = sample from BSDF;
    L_ind = BSDF * color(trace(p, w2), moreBounces-1) * dot(n, w2) / pdf(w2);

    return L_e + L_dir + L_ind;
}
```

这上面的代码中，我们首先通过常规的路径追踪计算了间接光照。这一项包括多次弹射之后的的光照贡献。然后，我们又通过 NEE 显式计算了来自光源的直接光照。然而，这一部分直接光照已经在之前的间接光照中被考虑了一次。因为间接光照是递归的，在某一个弹射后光线可能直接击中光源。

因此，对于直接光照，我们实际上计算两次。为了避免这种重复计数，我们可以在递归步骤中直接排除对光源的计算，只考虑次级光线弹射后的间接光照。

```
NEE::color(Point p, Vec3f w, int moreBounces, bool includeLe) {
    L_e = includeLe ? Le(p, -w) : BLACK;
    if (moreBounces == 0) return L_e;

    // NEE:  compute L_dir by sampling the light
    w1 = sample from light;
    L_dir = BRDF * color(trace(p, w1), 0, true) * dot(n, w1)/pdf(w1);

    w2 = sample from BSDF;
    L_ind = BSDF * color(trace(p, w2), moreBounces-1, false) * dot(n, w2) / pdf(w2);

    return L_e + L_dir + L_ind;
}
```

10.3.2 多重重要性采样

工业界更普遍的方法是将多重重要性采样（MIS）与 NEE 相结合。在计算每次光照的时候，我们随机采样一个方向，可能是向光源采样，也有可能是从材料表面做 BSDF 采样。然后，我们以一定的权重计算这两者的平均 pdf。

```
MIS::color(Point p, Vec3f w, int moreBounces) {
    if (moreBounces == 0) return L_e;

    w1 = sample from mixture PDF;
    return L_e(p, -w) + BRDF * color(trace(x, w1), moreBounces - 1) * dot(n, w1) / pdf(w1);
}

MIS::trace(Point p, Vec3f w, int moreBounces, float LeWeight) {
    hit = intersect(scene, ray);
    L_e = LeWeight * Le(p, -w);

```

```
    if (moreBounces == 0) return L_e;

    // NEE
    w1 = sample from light;
    L_dir = BRDF * trace(p, w1, 0, MIS_w1) * dot(n, w1) / pdf(w1);

    // BSDF
    w2 = sample from BSDF;
    L_ind = BSDF * trace(p, w2, moreBounces -1, MIS_w2) * dot(n, w2) / pdf(w2);

    return L_e + L_dir + L_ind;
}
```

10.4 支持路径追踪的 Integrator 派生类

10.4.1 DirectMatIntegrator 类

```
// integrator.h
/// Integrates direct lighting by sampling the material
class DirectMats : public Integrator {
  public:
    DirectMats(const json &j = json::object());

    Color3f Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const override;
};
```

```
// integrator.cpp
Color3f DirectMats::Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const {
    HitInfo hit;
    if (!scene.intersect(ray, hit)) {
        return scene.background(ray);
    }

    ScatterRecord srec;
    Color3f emitted = hit.mat->emitted(ray, hit);

    // Compute illumination by sampling the material BSDF
    Vec2f sample = sampler.next2D();
    if (!hit.mat->sample(ray.d, hit, sample, srec)) {
        return emitted;
    }

    Ray3f scattered(hit.p, srec.scattered);

    // If the sampled direction is specular, don't use Monte Carlo
    if (srec.isSpecular) {
        if (scene.intersect(scattered, hit)) {
            return emitted + srec.attenuation * hit.mat->emitted(scattered, hit);
        } else {
            return emitted + srec.attenuation * scene.background(ray);
```

```
        }
    }

    float pdf = hit.mat->pdf(ray.d, scattered.d, hit);
    if (pdf == 0.f) {
        return emitted;
    }
    Color3f eval = hit.mat->eval(ray.d, scattered.d, hit);

    if (scene.intersect(scattered, hit)) {
        return emitted + eval * hit.mat->emitted(scattered, hit) / pdf;
    } else {
        return emitted + eval * scene.background(ray) / pdf;
    }
}
```

10.4.2 NEEIntegrator 类

```
// integrator.h
/// Integrates direct lighting and performs next-event estimation by sampling the light
class NEEIntegrator : public Integrator {
  public:
    DirectNEE(const json &j = json::object());

    Color3f Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const override;
};
```

```
// integrator.h
Color3f NEEIntegrator::Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const {
    HitInfo hit;
    if (!scene.intersect(ray, hit)) {
        return scene.background(ray);
    }

    Color3f emitted = hit.mat->emitted(ray, hit);

    // Compute illumination by sampling the lights
    Vec2f sample = sampler.next2D();
    Vec3f scatterDir = scene.emitters().sample(hit.p, sample);
    Ray3f scattered(hit.p, scatterDir);

    float pdf = scene.emitters().pdf(hit.p, normalize(scattered.d));
    if (pdf == 0.f) {
        return emitted;
    }
    Color3f eval = hit.mat->eval(ray.d, normalize(scattered.d), hit);

    if (scene.intersect(scattered, hit)) {
        return emitted + eval * hit.mat->emitted(scattered, hit) / pdf;
    } else {
        return emitted + eval * scene.background(ray) / pdf;
```

```
    }
}
```

10.4.3 MISIntegrator 类

```
/// Performs recursive path tracing with multiple importance lighting that combines
/// material sampling and light sampling
class MISIntegrator : public Integrator {
  public:
    PathTracerMIS(const json &j = json::object());

    Color3f Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const override;

    Color3f recursiveColor(const Scene &scene, Sampler &sampler, const Ray3f &ray, int depth,
                           float misWeight) const;

    int m_maxBounces = 64;
    float m_power = 2.f;
};
```

```
// integrator.cpp
Color3f MISIntegrator::recursiveColor(const Scene &scene, Sampler &sampler, const Ray3f &ray, int
     depth,
                                      float misWeight) const {
    HitInfo hit;
    if (!scene.intersect(ray, hit)) {
        return scene.background(ray);
    }

    ScatterRecord srec;
    Color3f emitted = misWeight * hit.mat->emitted(ray, hit);

    // If we have no more bounces, just return the emitted color
    if (depth == m_maxBounces) {
        return emitted;
    }

    // Compute direct illumination by sampling the lights
    Vec2f sampleLight = sampler.next2D();
    Vec3f scatterDirLight = scene.emitters().sample(hit.p, sampleLight);
    Ray3f scatteredLight(hit.p, scatterDirLight);

    float pdfLight1 = scene.emitters().pdf(hit.p, normalize(scatteredLight.d));

    Color3f direct(0.f);
    if (pdfLight1 > 0.f) {
        float pdfLight2 = hit.mat->pdf(ray.d, normalize(scatteredLight.d), hit);
        float misWeightLight = powerHeuristic(pdfLight1, pdfLight2, m_power);
        Color3f evalLight = hit.mat->eval(ray.d, normalize(scatteredLight.d), hit);
        Color3f recursiveLight = recursiveColor(scene, sampler, scatteredLight, m_maxBounces,
    misWeightLight);
```

```
30          direct = evalLight * recursiveLight / pdfLight1;
31      }
32
33      // Compute indirect illumination by sampling the material BSDF
34      Vec2f sampleMat = sampler.next2D();
35      if (!hit.mat->sample(ray.d, hit, sampleMat, srec)) {
36          return emitted + direct;
37      }
38      Vec3f scatterDirMat = srec.scattered;
39      Ray3f scatteredMat(hit.p, scatterDirMat);
40
41      if (srec.isSpecular) {
42          // If the sampled direction is specular, don't use Monte Carlo
43          // Set the MIS weight to 1 since we can't sample it any other way
44          return emitted + srec.attenuation * recursiveColor(scene, sampler, scatteredMat, depth +
    1, 1.f);
45      }
46
47      float pdfMat1 = hit.mat->pdf(ray.d, normalize(scatteredMat.d), hit);
48
49      Color3f indirect(0.f);
50      if (pdfMat1 > 0.f) {
51          float pdfMat2 = scene.emitters().pdf(hit.p, normalize(scatteredMat.d));
52          float misWeightMat = powerHeuristic(pdfMat1, pdfMat2, m_power);
53          Color3f evalMat = hit.mat->eval(ray.d, normalize(scatteredMat.d), hit);
54          Color3f recursiveMat = recursiveColor(scene, sampler, scatteredMat, depth + 1,
    misWeightMat);
55          indirect = evalMat * recursiveMat / pdfMat1;
56      }
57
58      return emitted + direct + indirect;
59  }
60
61  Color3f MISIntegrator::Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const {
62      return recursiveColor(scene, sampler, ray, 0, 1.f);
63  }
```

Chapter 11

参与介质

到目前为止，我们所进行的所有渲染算法都基于一个事实：真空中辐射亮度沿光线不变。但是，现实中这个描述实际上是不准确的。这是因为**参与介质** (Participating Media) 的存在。参与介质是指能够与光发生相互作用的介质, 如烟雾、尘埃、云、雾等。它们通常由众多的微小颗粒组成，如果暴力计算光线与它们的每个单元的交互，很快计算机的算力就会被消耗殆尽。本章中我们将会重点讨论如何描述以及理解各类参与介质。

11.1 体积散射

当光线穿过这些介质时，主要会发生以下三种现象:

- **吸收**（absorption）：介质吸收部分光能量，转换成其它形式的能量（比如热能），使辐射亮度衰减。不同波长的光可能被吸收程度不同。
- **发光**（emission）：辐射亮度通过环境中的发光颗粒增加。
- **散射**（scattering）：光线与介质中的粒子发生相互作用, 改变传播方向。

这些性质可能是*同质的* (homogeneous), 也有可能是*异质的* (heterogeneous) 或是*非同质的* (inhomogeneous)。同质属性指的是在一个空间区域中会保持不变，这个区域通常要么是无限大的，要么被一个简单的形状包裹，限定其范围。而非同质的属性则可能在范围内发生变化。另外，上面的这三种现象通常也是与波长有关系的。虽然我们可以跟表面发光体做类似的处理，不过之后我们会看到当我们引入蒙特卡洛积分的时候，吸收和散射需要特别的处理。

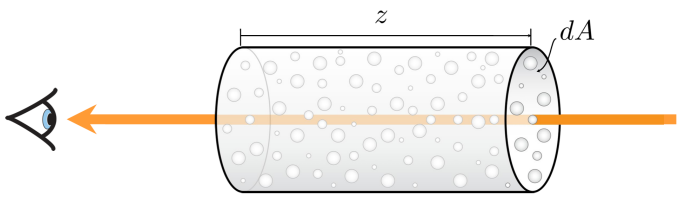
如开头所说，参与介质虽然概念上是由微小的颗粒组成，逻辑上可以通过显示地为每个颗粒建模然后暴力计算光照效果，但现实中这是不可能实现的。且不说计算的时间复杂度，光是要把所有颗粒的位置记住，就需要消耗大量的内存。因此，我们采用与微表面模型类似的想法——用统计学的语言来描述参与介质的各种现象。

11.1.1 微元描述

为了描述参与介质，我们也要用微积分的思想来讨论。首先，先关注通过参与介质的一条**光束**（beam）的行为。

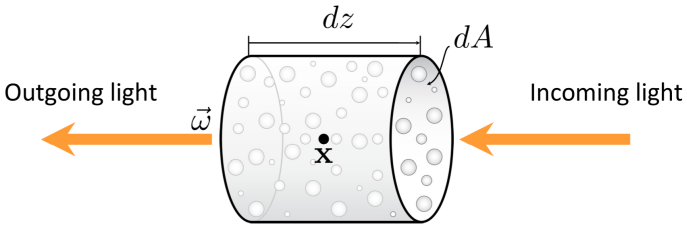
微分光束

微分光束（differential beam）描述的是一条长度为 z 的光束，我们认为它的横截面是一个很小的值，dA。如下图描述。



微分光束段

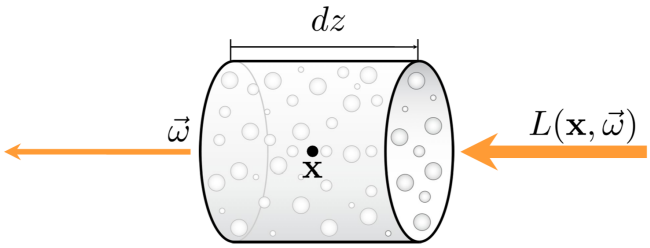
（differential beam segment）指的则是微分光束上的一个长度为 dz 的极小片段，如下图所示。



11.1.2 吸收

吸收通过介质的**吸收系数**（absorption coefficient）$\sigma_a(\mathbf{x})$ 来描述。吸收系数是光在介质中传播时单位距离被吸收的概率密度，因此其单位为 m^{-1}。由定义可得，它可以是任何的非负数，所以不一定需要在 [0,1] 之间。

严格来说，吸收系数与位置 $\mathbf{x}$ 和方向 ω 都有关系，但是在实现中我们通常会假设其与方向无关，所以我们会将吸收系数写成是 $\mathbf{x}$ 的函数。

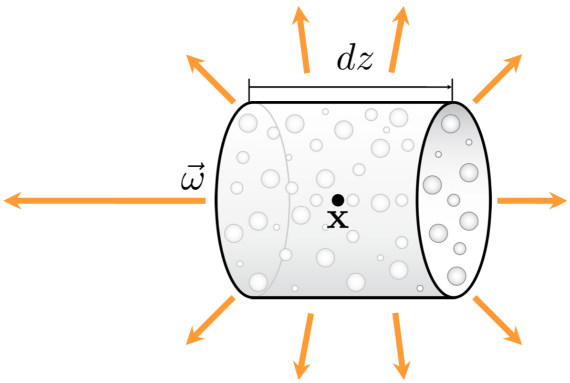


被吸收/损耗的辐射亮度 $dL(\mathbf{x},\omega)$ 满足如下微分方程：

$$dL(\mathbf{x},\omega) = -\sigma_a(\mathbf{x})L(\mathbf{x},\omega)dz$$

11.1.3 发光

吸收是由于光能转换成了其它形式的能量，而发光则是基于其它形式的能量转换为可见光。在这种情况下，辐射亮度会得到提高。



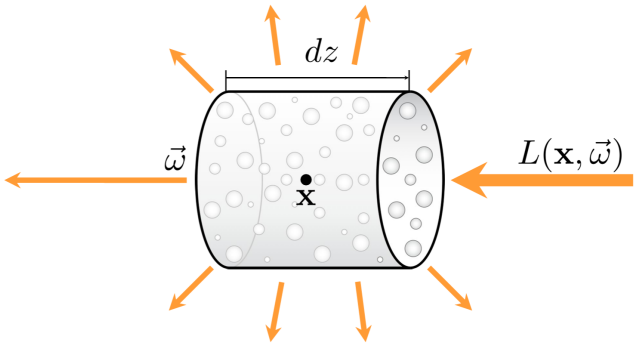
增加的辐射亮度 $dL(\mathbf{x}, \omega)$ 满足如下微分方程：

$$dL(\mathbf{x}, \omega) = \sigma_a(\mathbf{x}) L_e(\mathbf{x}, \omega) dz$$

其中，σ_a 依然是上一节中提及的吸收系数，$L_e(\mathbf{x}, \omega)$ 则是发出的辐射亮度。

11.1.4 外散射

散射是光与参与介质的另一种交互方式。光线击中参与介质中的微小颗粒时，会在微小颗粒的表面发生散射。因此，对于某一微分立体角（也就是某一方向）上辐射亮度，相比于真空会有部分损耗，因为这一部分的辐射亮度可能会被散射至其它方向，这种现象被称作**外散射**（out-scattering）。



描述单位距离内外散射事件发生的概率密度的函数被称为**散射系数**（scattering coefficient）$\sigma_s(\mathbf{x})$。外散射满足以下微分方程：

$$dL(\mathbf{x}, \omega) = -\sigma_s(\mathbf{x}) L(\mathbf{x}, \omega) dz$$

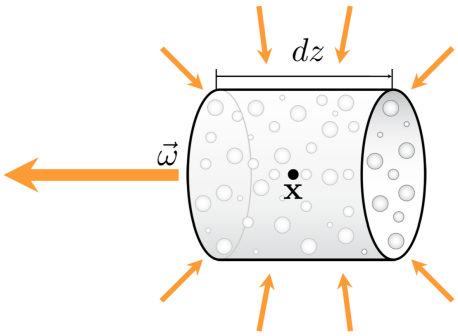
因此，在参与介质介入计算时，能够造成辐射亮度衰减的就是吸收系数和散射系数。这两个系数的和

$$\sigma_t(\mathbf{x}) = \sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x})$$

被称作**衰减系数**（attenuation），也叫做**消光系数**（extinction coefficient）。

11.1.5 内散射

虽然与微小颗粒之间的交互可能会导致辐射亮度衰减到其它的方向上，但这也意味着其它方向的辐射亮度也可能被散射到当前的方向上。这种现象就被称作为**内散射**（in-scattering）。



内散射满足以下微分方程：

$$dL(\mathbf{x},\omega) = \sigma_s(\mathbf{x})L_s(\mathbf{x},\omega)dz$$

其中，σ_s 就是上文提到的的散射系数，$L_s(\mathbf{x},\omega)$ 则是内散射的辐射亮度。

11.2 辐射传输方程

根据上面提到的光学现象，我们可以写出如下的**辐射传输方程**（radiative transfer equation, RTE）：

$$dL(\mathbf{x},\omega) = \text{衰减} + \text{增幅}$$

其中，

$$\begin{aligned}
\text{衰减} &= \text{吸收} + \text{外散射} \\
&= -\sigma_a(\mathbf{x})L(\mathbf{x},\omega)dz - \sigma_s(\mathbf{x})L(\mathbf{x},\omega)dz \\
\text{增幅} &= \text{发光} + \text{内散射} \\
&= \sigma_a(\mathbf{x})L_e(\mathbf{x},\omega)dz + \sigma_s(\mathbf{x})L_s(\mathbf{x},\omega)dz
\end{aligned}$$

11.2.1 衰减

在 RTE 中，观察到

$$\begin{aligned}
\text{衰减} &= \text{吸收} + \text{外散射} \\
&= -\sigma_a(\mathbf{x})L(\mathbf{x},\omega)dz - \sigma_s(\mathbf{x})L(\mathbf{x},\omega)dz \\
&= -(\sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x}))L(\mathbf{x},\omega)dz \\
&= -\sigma_t(\mathbf{x})L(\mathbf{x},\omega)dz
\end{aligned}$$

其中，σ_t 就是我们之前提到的消光系数。

11.2.2 透射率

在 11.1 部分中我们提到过的几种现象都还是在关注局部的光与参与介质交互的行为。在实际的实现中，我们其实更关心在一条光线上累积的对辐射亮度的影响。我们由其关注在参与介质存在的情况下，从一个点出发的光线到达另一个点后，其辐射亮度可能会减少多少。现在，我们考虑对于有限长度的光束的情况。假设 $\sigma_t(\mathbf{x})$ 是个常数，我们将其记作 σ_t。整理方程我们可以得到

$$\frac{dL(\mathbf{x},\omega)}{L(\mathbf{x},\omega)} = -\sigma_t dz \tag{11.1}$$

沿着光线从 0 到 z 作积分，

$$\ln(L_z) - \ln(L_0) = -\sigma_t z$$

根据对数函数的性质，

$$\ln\left(\frac{L_z}{L_0}\right) = -\sigma_t z$$

也就是说，

$$\frac{L_z}{L_0} = e^{-\sigma_t z}$$

这个定律就是**比尔-朗伯定律**（Beer-Lambert Law），等号左边的比值是光束距离 z 处的辐射亮度和光束初始位置的辐射亮度的比值，这个比值也被称作**透射率**（transmittance）。

注意我们也可以将等式 (11.1) 写作

$$\frac{dL(\mathbf{x},\omega)}{dz} = -\sigma_t(\mathbf{x})L(\mathbf{x},\omega) \tag{11.2}$$

我们会在接下来的章节中用到这个等式。

11.2.3 透射率的性质

透射率有两个有用且符合直觉的性质。

1. 从一个点到其自身的透射率为 1。
2. 透射率具有可乘的性质。

以上两点都可以通过定义直接获得。

首先，对于同质介质，因为衰减系数在各个位置不变，因此，对于起始点 $\mathbf{x}$ 和终点 $\mathbf{y}$，同质透射率满足

$$T_r(\mathbf{x},\mathbf{y}) = e^{-\sigma_t||\mathbf{x}-\mathbf{y}||}$$

对于非同质介质，衰减系数会随着位置变化，我们则需要使用积分，

$$T_r(\mathbf{x},\mathbf{y}) = e^{-\int_0^{||\mathbf{x}-\mathbf{y}||}\sigma_t(t)dt}$$

这就说明透射率具有可乘的性质，即

$$T_r(\mathbf{x},\mathbf{z}) = T_r(\mathbf{x},\mathbf{y})T_r(\mathbf{y},\mathbf{z})$$

且同时，如果 $\mathbf{x}=\mathbf{y}$ 的话，无论是同质或是异质介质中，T_r 都会简化为 1。

11.3 无效散射

在实现的过程中，我们会遇到两种让我们比较头疼的情况。首先，在非同质介质中，采样透射率的难度本身可能就比较高，在之后介绍采样时我们之后会了解到。另外，在一些介质非常稀薄的场景中，真正的散射事件可能相对来说非常稀少。这意味着光线可能会在没有多少实际物理影响的情况下穿过大量的参与介质，从而使得如果我们只计算真实的散射事件，会需要采样非常多的光线才能准确地估计光线传输，大大降低我们的计算效率。

为了解决这些问题，我们引入了一种被称为**无效散射**（Null Scattering）的技术。无效散射的核心想法在于：引入一种假设的散射事件，这些事件不改变光线的状态，但允许算法以统一的方式处理光线与参与介质的相互作用。在模拟中，当光线与介质相交时，计算一个散射事件。这个事件可能是无效散射，也有可能是真实散射。如果是无效散射，光线会继续前进，就好像没有发生散射一样。

11.3.1 主要系数

我们首先定义一个**无效散射系数**（null-scattering coefficient）σ_n。与其它系数类似，σ_n 计算的是在介质中单位距离内无效散射事件发生的概率。

我们再定义一个在介质内总是不小于 $\sigma_a + \sigma_s$ 的常数 σ_{maj}，这个常数被称作**主要系数**（majorant）。英文单词 majorant 在数学中的含义指的是一个函数或是序列，在整个定义域范围内都大于或等于另一个函数的序列。在体积渲染的上下文中，majorant 特指这个在介质内任何位置都不小于消光系数的常数，也就是说它至少等于介质内消光系数 $\sigma_t = \sigma_s + \sigma_a$ 的最大值。

根据定义，我们有

$$\sigma_n(\mathbf{x}, \omega) = \sigma_{\text{maj}} - \sigma_t(\mathbf{x}, \omega)$$

或者等价地，我们也可以认为在介质中

$$\sigma_{\text{maj}} = \sigma_n + \sigma_s + \sigma_a$$

是均匀的。

11.3.2 无效散射下的透射率

改写等式 (11.2) 中的 σ_t，我们得到，

$$\frac{dL(\mathbf{x}, \omega)}{dz} = -(\sigma_{\text{maj}} - \sigma_n(\mathbf{x}, \omega))(\mathbf{x})L(\mathbf{x}, \omega) \tag{11.3}$$

在这里我们省去推导的过程。通过积分上式，我们可以得到

$$T_r(\mathbf{p}, \mathbf{p}') = e^{-\sigma_{\text{maj}} z} + \int_0^z e^{-\sigma_{\text{maj}} t} \sigma_n(\mathbf{x} + t\omega) T_r(\mathbf{p} + t\omega, \mathbf{p}') dt. \tag{11.4}$$

注意到，如果无效散射系数为 0，那么 (11.4) 中的积分项就会完全消失，这个公式就会简化为比尔-朗伯定律，这是描述同质介质的方法。对于非同质介质，我们可以将 (11.4) 中的第一项视作在这里先计算一个最稀薄的同质介质的真实透射率，然后积分项描述的是除了这部分同质介质外还存在的多余颗粒对透射率的影响。

11.4 体积渲染方程

对于处于位置 $\mathbf{x}$ 的观察者，来自点 $\mathbf{x}_z$ 的辐射亮度 $L(\mathbf{x},\omega)$ 满足

$$\begin{aligned} L(\mathbf{x},\omega) &= T_r(\mathbf{x},\mathbf{x}_z)L(\mathbf{x}_z,\omega) && \text{（降低的背景表面辐射亮度）} \\ &+ \int_0^z T_r(\mathbf{x},\mathbf{x}_t)\sigma_a(\mathbf{x}_t)L_e(\mathbf{x}_t,\omega)dt && \text{（参与介质累积的发光辐射亮度）} \\ &+ \int_0^z T_r(\mathbf{x},\mathbf{x}_t)\sigma_s \int_{\mathcal{S}^2} f_p(\mathbf{x}_t,\omega',\omega)L_i(\mathbf{x}_t,\omega')d\omega' dt && \text{（参与介质累积的内散射辐射亮度）} \end{aligned}$$

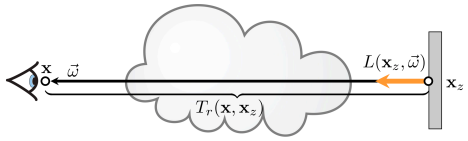
这个关系就被称作**体积渲染方程**（volumetric rendering equation, VRE），其中的 f_p 被我们称作相位函数，我们会在下面的小节中介绍。

11.4.1 透射项

在 VRE 中，第一项

$$T_r(\mathbf{x},\mathbf{x}_z)L(\mathbf{x}_z,\omega)$$

被称作透射项 (transmission term)。这部分代表了从光源到点 $\mathbf{x}$ 途径的直接光照。$T_r(\mathbf{x},\mathbf{x}_z)$ 是上文所体积的透射率，它表明了 $\mathbf{x}_z$ 到 $\mathbf{x}$ 之间的介质对光的衰减。

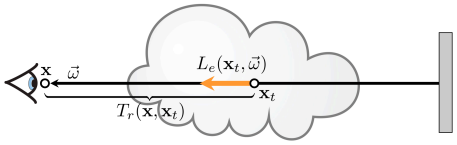


11.4.2 自发光项

第二项

$$\int_0^z T_r(\mathbf{x},\mathbf{x}_t)\sigma_a(\mathbf{x}_t)L_e(\mathbf{x}_t,\omega)dt$$

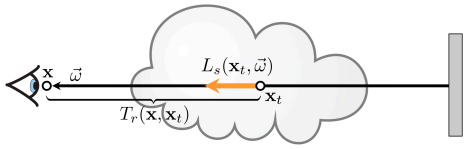
被称作自发光项（emission term）。这一项是对介质内部所有自发光的点的积分，表示了参与介质本身的发光特性（比如荧光、生物发光等），也就是 11.1.3 中提及的发光。积分内的被积函数也就对应了在某一点 $\mathbf{x}_t$ 的发光衰减情况。



11.4.3 散射项

第三项被称作散射项 (scattering term)，描述的是参与介质内部累积的内散射辐射亮度，类似于自发光项，我们可以写出

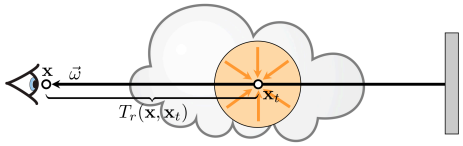
$$\int_0^z T_r(\mathbf{x},\mathbf{x}_t)\sigma_s(\mathbf{x}_t)L_s(\mathbf{x}_t,\omega)dt$$



其中，内散射的部分我们势必需要考虑来自于各方向的所有内散射。我们可以参照 BSDF 的思路，假设存在一个分布函数 $f_p(\omega, \omega')$ 告诉我们对于给定入射散射方向，在该点 $\mathbf{x}_t$ 上能够收到多少来自内散射的辐射亮度，这样的话我们就可以将散射项完整写出。

$$\int_0^z T_r(\mathbf{x}, \mathbf{x}_t)\sigma_s \int_{\mathcal{S}^2} f_p(\mathbf{x}_t, \omega', \omega) L_i(\mathbf{x}_t, \omega') d\omega' dt$$

这里的 f_p 就是我们在下一节中介绍的相位函数。



11.5 相位函数

就像不同的 BSDF 模型在描述不同表面的散射分布，**相位函数**（Phase Function）描述的是散射光的分布。对于给定入射方向 ω_i 以及出射方向 ω_o，相位函数 $f_p(\omega_i, \omega_o)$ 描述的就是沿着给定入射方向到达点 $\mathbf{x}$ 后再沿着给定出射方向离开的概率。

按照惯例，相位函数是归一化（normalized）的。也就是说其积分为 1，这一点与 BSDF 不同。也就是说

$$\int_{\mathcal{S}^2} f_p(\mathbf{x}, \omega', \omega) d\omega' = 1$$

注意到，这种性质使得相位函数本身可以被视为一种概率密度函数，它描述了光子在参与介质中经过散射事件后沿着不同方向散射的相对可能性的分布。

11.5.1 各向同性散射

在很多自然界的场景中，我们通常将相位函数写作是两个方向 ω_i 和 ω_o 的函数 $f_p(\omega_i, \omega_o)$，或者更简单地记作它们的夹角余弦值的函数 $f_p(\cos\theta)$。由于只有 θ 一个自由度，这种情况下的相位函数也就是一个 1D 函数。拥有这一类相位函数描述的介质就被称作是**各向同性**（isotropic）或是**对称**（symmetric）的。

各向同性相位函数是互易（reciprocal）的，也就是说

$$f_p(\omega_i, \omega_o) = f_p(\omega_o, \omega_i)$$

因此，在讨论各向同性相位函数时，通常我们也不指明入射和出射方向，也会匿名地将相位函数写作 $f_p(\omega, \omega')$。值得一提的是，$\cos\theta$ 本身就是对称函数，$\cos\theta = \cos(-\theta)$。

参与介质的**各向同性散射**（Isotropic Scattering）指的是会将光线向四周均匀地散射，我们可以将其类比成为参与介质中的朗伯材质的 BRDF。由于相位函数归一化的特征，各向同性散射只有一个满足此特征的解：

$$f_p(\omega', \omega) = \frac{1}{4\pi}.$$

对这个相位函数在单位球内求积分，

$$\int_{S^2} f_p(\omega', \omega) d\omega' = 1.$$

注意到，对于从球心采样到单位球面上的一个方向 ω，其概率密度函数也是 $\frac{1}{4\pi}$。这也符合我们对各向同性相位函数的理解：各向同性散射意味着光子从任何方向入射到散射介质中，被散射的光子有相同的概率沿任何方向散射出去，各向同性散射的相位函数确保了在单位球面上任意方向上散射概率是均匀的，与入射方向无关。

11.5.2 各向异性散射

与各向同性散射相对地，**各向异性散射**（Anisotropic Scattering）指的是不均匀地散射。各向异性散射的相位函数是两个方向的 4D 函数：入射方向和散射方向都可以通过两个角度来描述，因此我们一共需要四个参数。这个相位函数形如：

$$f_p(\omega, \omega') = f_p(\theta, \phi, \theta', \phi').$$

如果我们定义一个数值 g，

$$g = \int_{S^2} f_p(\omega', \omega) \cos\theta d\omega' = 2\pi \int_0^\pi f_p(\cos\theta) \cos\theta \sin\theta d\theta$$

其中，

$$\cos\theta = -\omega \cdot \omega'$$

那么 g 就是通过我们的相位函数根据散射角度 θ 的余弦加权积分得到的，因此，我们可以将 g 理解为*平均的余弦值*，它可以用来描述散射光线相对于入射光线方向的平均偏向程度，这个值被我们称作**各向异性因子**（anisotropic factor），一些文献上也会将其称作**非对称参数**（asymmetry parameter）。

如何理解 g

根据上式，我们可以得出以下结论：

- $g = 0$：表示各向同性散射。
- $g > 0$：表示散射主要向前方（入射光的方向）。
- $g < 0$：表示散射主要向后方。

通过这种方式，g 作为一个单一的参数，简化了复杂的相位函数，提供了一个量化介质中光散射各向异性程度的简单方法。我们很快就会看到一个利用了 g 来描述的相位函数。

11.5.3 Henyey-Greenstein 相位函数

计算机图形学中最常用的相位函数就是 **Henyey-Greenstein 相位函数**（以下简称 HG 函数）。它是一个参数化的相位函数，其中参数 g 就是我们之前提及的各向异性因子。其形式如下：

$$f_{p\mathrm{HG}}(\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2g\cos\theta)^{\frac{3}{2}}}$$

HG 函数是一个经验函数，它并不准确描述物理行为，而是基于实验和经验的，与 Blinn-Phong 在表面上的行为非常相似。然而，它可以很优秀地通过调整 g 值去模拟从完全各向同性散射到强各向异性散射的广泛情况。

11.6 米氏散射 *

米氏散射也被称作**洛伦兹-米散射**（Lorenz-Mie Scattering），是一种描述球形粒子（如水滴或气溶胶）对光波散射的物理理论。当发生散射的物质颗粒的直径与光的波长处于同一数量级时，我们就不能再忽略光的波动性质了。德国物理学家古斯塔夫 · 米（Gustav Mie）在 1908 年提出该理论，用于解释为什么颗粒物质可以散射光波。

在这里，我们不深入讨论米氏散射背后的数学原理—米氏散射是一种麦克斯韦方程组[1]的解。当微粒半径的大小接近于或大于入射光线波长 λ 的时候，大部分的入射光线会沿着前进的方向进行散射。这种大微粒包括灰尘、水滴、来自污染物的颗粒物质，例如烟雾、烟霾等，形成类似于体积光柱的视觉效果，这种现象也被称作**丁达尔效应**（Tyndall Effect）。

11.6.1 Lorenz-Mie 相位函数

米氏散射的相位函数——也就是 Lorenz-Mie 相位函数（以下简称 LM 函数）并没有简单的解析形式。因此，我们只能通过一些近似来计算 LM 函数值，这涉及到解决麦克斯韦方程组在球坐标下的边界条件问题。我们会使用以下的两种简化模型。

霾笼罩环境

霾笼罩环境（hazy atmosphere）对应的是大气中拥有烟霾颗粒物导致的能见度降低。霾（haze）可能由尘土、烟尘、水汽等多种因素造成。霾能散射阳光，使得远处的物体或景色变得模糊不清。这种笼罩环境下，我们认为 LM 函数的近似形式为：

$$f_{p\mathrm{haze}}(\theta) = \frac{1}{4\pi}\left(5+\left(\frac{1+\cos\theta}{2}\right)^8\right).$$

浓雾笼罩环境

浓雾笼罩环境（murky atmosphere）描述了更加浑浊的大气状态，由浓雾或是非常密集的烟雾笼罩。这种笼罩环境下，我们认为 LM 函数的近似形式为：

$$f_{p\mathrm{murky}}(\theta) = \frac{1}{4\pi}\left(17+\left(\frac{1+\cos\theta}{2}\right)^{32}\right).$$

[1] 关于麦克斯韦方程组，详情可以查阅麦克斯韦方程组。

图 11.1: 霾笼罩环境

图 11.2: 浓雾笼罩环境

11.7 瑞利散射 *

瑞利散射（Rayleigh Scattering）是米氏散射的一种特殊情况，其近似地描述了当微小颗粒的大小比可见光波长的十分之一小的时候的情况。

11.7.1 Rayleigh 相位函数

瑞利散射的相位函数就是 Rayleigh 相位函数（以下简称 R 函数）。其形式如下：

$$f_{p\text{Rayleigh}}(\theta) = \frac{3}{16\pi}(1+\cos^2\theta).$$

注意到，

- 在 $\theta = 0$ 或 $\theta = \pi$ 时，$1+\cos^2\theta$ 获得最大值。这意味着散射在前向和后向时是最强的。

- 在 $\theta = \frac{\pi}{2}$ 时，$1 + \cos^2\theta$ 获得最小值。这意味着散射在侧向时较弱。

11.7.2 瑞利散射的散射系数

瑞利散射发生外散射的概率密度函数，也就是瑞利散射的散射系数具有如下形式：

$$\sigma_{s\text{Rayleigh}}(\lambda, d, \eta, \rho) = \rho\frac{2\pi^5 d^6}{3\lambda^4}\left(\frac{\eta^2 - 1}{\eta^2 + 2}\right)^2$$

其中，

- λ 是光的波长。
- d 是散射物质微小颗粒的直径。
- η 是介质的折射率。
- ρ 是参与介质的密度。

观察散射系数，我们也可以得到如下的结论：

- σ_s 与波长的四次方成反比。这意味着短波长的光会比长波长的光散射得更多。
- σ_s 与粒子直径的六次方成正比。这意味着粒子大小的微小变化会对散射效率产生巨大影响。
- σ_s 也与粒子密度成正比。这意味着单位体积内的粒子总量对散射效果也是有影响的。

11.7.3 天空的颜色

瑞利散射解释了为什么白天天空呈蓝色和傍晚时变为红色的原因，这与大气对不同波长光的散射能力有关。以下是这一现象的详细解释。

白天的蓝色天空

上文提到，瑞利散射的强度与光的波长成四次方反比，这意味着较短的波长（例如蓝光）会比较长的波长（例如红光）散射得更强。太阳发出的光包含了所有颜色的光，当它进入大气层时，较短的蓝色波段受到更强的散射作用。当我们看向天空，不是直接看向太阳时，我们看到的是散射的光。由于蓝色波段的光被散射得最多，所以我们从各个方向都能看到散射的蓝光，因此天空显现为蓝色。

傍晚的红色天空

当太阳接近地平线时，太阳光必须穿过更多的大气层才能到达观察者。这增加了光通过大气的路径长度，因此增加了光的散射次数。随着路径长度增加，短波长的光（蓝色和紫色）更可能被散射出视线之外，因为它们在穿过大气层时被散射得更多。相对来说，长波长的光（红色、橙色和黄色）散射较少，因此这些波长的光更有可能在长路径传输中存留，并且到达观察者的眼睛。

另外，人类的眼睛对蓝色光的感光性随着光线强度的降低而减少，因此在光线较弱的情况下（如日落时），人眼对红色光的敏感性更高。

11.8 求解体积渲染方程

有了上面的理论基础，我们回到实现的角度。现在开始我们要考虑体积渲染方程的具体求解了。上文我们已经介绍过，VRE

$$\begin{aligned} L(\mathbf{x},\omega) &= T_r(\mathbf{x},\mathbf{x}_z)L(\mathbf{x}_z,\omega) && \text{（降低的背景表面辐射亮度）} \\ &+ \int_0^z T_r(\mathbf{x},\mathbf{x}_t)\sigma_a(\mathbf{x}_t)L_e(\mathbf{x}_t,\omega)dt && \text{（参与介质累积的发光辐射亮度）} \\ &+ \int_0^z T_r(\mathbf{x},\mathbf{x}_t)\sigma_s\int_{\mathcal{S}^2} f_p(\mathbf{x}_t,\omega',\omega)L_i(\mathbf{x}_t,\omega')d\omega' dt && \text{（参与介质累积的内散射辐射亮度）} \end{aligned}$$

第一项仅与背景有关，是最容易计算的，因此，我们先考虑后面两个带积分的项。

11.8.1 同质参与介质

我们首先考虑不发光的同质参与介质，例如雾（fog）。对于这种参与介质，因为介质的散射属性在各处相等，因此我们知道在 VRE 中，

$$\begin{aligned} L(\mathbf{x}+\omega) &= \text{背景透射项} + \text{散射项} \\ &= T_r(\mathbf{x}\leftrightarrow\mathbf{x}_z)L(\mathbf{x}_z,\omega) + \int_0^z T_r(\mathbf{x}\leftrightarrow\mathbf{x}_t)\sigma_s(\mathbf{x}_t)L_i(\mathbf{x}_t,\omega)dt \\ &= T_r(\mathbf{x}\leftrightarrow\mathbf{x}_z)L(\mathbf{x}_z,\omega) + \sigma_s\int_0^z T_r(\mathbf{x}\leftrightarrow\mathbf{x}_t)L_i(\mathbf{x}_t,\omega)dt && \text{（由于散射属性各处相同，可以将 } \sigma_s \text{ 提出）} \\ &= e^{-s\sigma_t}L(\mathbf{x}_z,\omega) + \sigma_s\int_0^z e^{-t\sigma_t}L_i(\mathbf{x}_t,\omega)dt \end{aligned}$$

常数内散射

如果我们假设内散射项 $L_i(\mathbf{x}_t,\omega)$ 也是各处均等的一个常数，那么我们进一步可以得到

$$\begin{aligned} L(\mathbf{x},\omega) &= e^{-s\sigma_t}L(\mathbf{x}_z,\omega) + \sigma_s L_i\int_0^z e^{-t\sigma_t}dt && \text{（根据假设，可以将 } L_i \text{ 提出）} \\ &= e^{-s\sigma_t}L(\mathbf{x}_z,\omega) + \sigma_s L_i\frac{1-e^{-s\sigma_t}}{\sigma_t} \\ &= \text{lerp}\left(\frac{\sigma_s}{\sigma_t}L_i, L(\mathbf{x}_s,\omega), e^{-s\sigma_t}\right) && \text{（根据线性插值函数 lerp 的定义）} \end{aligned}$$

单次散射

散射项原本的模样是

$$\int_0^z T_r(\mathbf{x},\mathbf{x}_t)\sigma_s\int_{\mathcal{S}^2} f_p(\mathbf{x}_t,\omega',\omega)L_i(\mathbf{x}_t,\omega')d\omega' dt$$

之前假设内散射为常数时，我们简化了过多的步骤。现在，我们假设 L_i 是仅来自光源的直接光照的，也就是说：

$$L_i(\mathbf{x},\omega) = T_r(\mathbf{x},r(\mathbf{x},\omega))L_e(r(\mathbf{x},\omega),-\omega)$$

类似于我们之前的 NEE 路径追踪，我们把只来自于直接光照的内散射称为**单次散射**（Single scattering）。在这样的简化下，

$$L(\mathbf{x},\omega) = \int_0^z T_r(\mathbf{x},\mathbf{x}_t)\sigma_s(\mathbf{x}_t)\int_{\mathcal{S}^2} f_p(\mathbf{x}_t,\omega',\omega)T_r(\mathbf{x}_t,\mathbf{x}_e)L_e(\mathbf{x}_e,\omega'),-\omega')d\omega' dt$$

如果我们假设介质为同质介质，发光的光源为点光源或是聚光灯，相位函数相对简单，也没有遮罩效果，我们可以得到一个比较简单的解析解。

$$L(\mathbf{x},\omega)=\frac{\Phi}{4\pi}\frac{1}{4\pi}\sigma_s\int_0^z e^{-\sigma_t||\mathbf{x},\mathbf{x}_t||}\frac{e^{-\sigma_t||\mathbf{x}_t,\mathbf{x}_p||}}{||\mathbf{x}_t,\mathbf{x}_p||^2}dt$$

在没有这些假设的情况下，解析解虽然理论上存在，但是在实现中却是过于复杂而不可能完成的目标。为了解决这个问题，我们会采用下一节介绍的光线行进的数值积分技术。

11.8.2 光线行进

在上一节中，我们在尝试计算散射项解析解的时候，会遇到过于复杂而无法求解的情况。下面，我们介绍**光线行进**（Ray-Marching）的思想，来解决这个问题。

对于散射项，

$$L(\mathbf{x},\omega)=\int_0^z T_r(\mathbf{x},\mathbf{x}_t)\sigma_s(\mathbf{x}_t)L_s(\mathbf{x}_t,\omega)dt$$

我们尝试用黎曼求和来近似计算其值，

$$L(\mathbf{x},\omega)\approx\sum_{i=1}^{N}T_r(\mathbf{x},\mathbf{x}_{t,i})\sigma_s(\mathbf{x}_{t,i})L_s(\mathbf{x}_{t,i},\omega)\Delta t$$

其中，Δt 就是我们的**步长**（stride）。对于每一条穿过体积的光线，在光线的传播路径上以一定的步长对体积数据进行采样。我们假设光线在每一步之间的衰减是均等的。通过采样点上的体积数据值，我们计算该点对最终渲染的贡献，如图 11.1 所示。

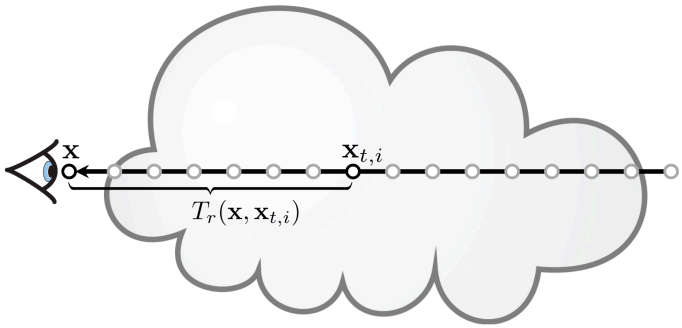


图 11.3: 同质介质光线行进

在这里，对于第 i 个采样点 $\mathbf{x}_{t,i}$，它到达人眼 $\mathbf{x}$ 时透射率可以通过 $T_r(\mathbf{x},\mathbf{x}_{t,i})$ 来计算。假设我们的体积是一个同质体积，我们可以通过下式计算透射率：

$$T_r(\mathbf{x},\mathbf{x}_{t,i})=e^{-\sigma_t||\mathbf{x},\mathbf{x}_{t,i}||}$$

假设我们的体积是一个异质体积，由于我们假设了光线在每一步之间的衰减是均等的，我们则可以通过递归的方式来计算这里的透射率，如图 11.2 所示：

$$T_r(\mathbf{x},\mathbf{x}_{t,i})=T_r(\mathbf{x},\mathbf{x}_{t,i-1})e^{-\sigma_t(\mathbf{x}_{t,i})\Delta t}$$

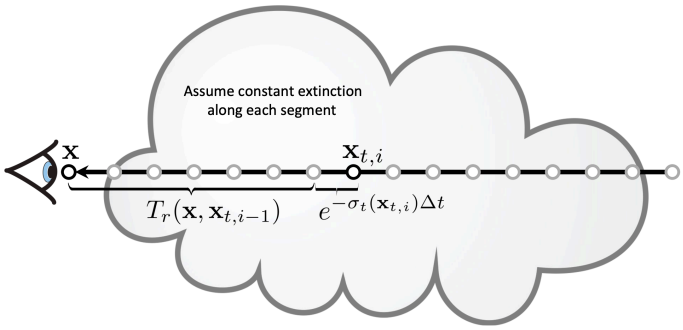
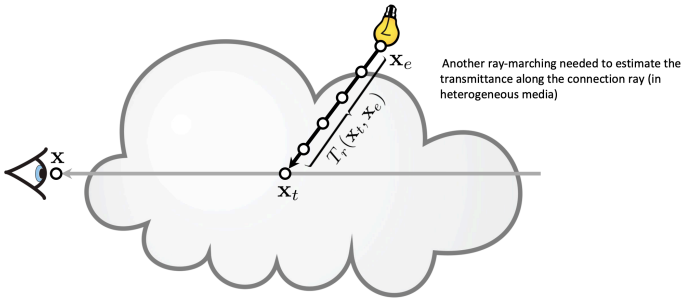


图 11.4: 异质介质光线行进

同质介质单次散射的光线行进

在上面介绍单次散射的时候，我们知道单次散射的过程就是从视点出发，经过一段距离进入介质后，随机选择光源上一点做连线。在光线行进的过程中，除了穿过介质的主光线纸外，我们还需要另一条光线从光源连接到主光线上的采样点 $\mathbf{x}_t$。在计算散射项的 $L_i(\mathbf{x}_t, \omega)$ 时，这个步骤是必要的。



异质介质单次散射的光线行进

在异质介质中，由于介质复杂，采样点之间与光源的连接往往都需要单独的计算，这样一来计算的效率就会变得非常低下。

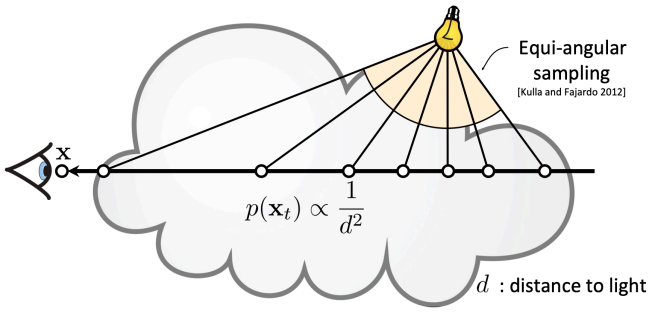
11.8.3 解耦透射率和内散射

一种思路是将透射率和内散射分开计算，就像我们在重要性采样中做的积分划分一样。

- 对于透射率的估计，我们选择合适的步长以捕捉介质的变化，但不需要过于密集的采样。
- 对于内散射的估计，我们选用一个合适的采样方式进行蒙特卡洛积分，例如 $p(\mathbf{x}_t) \propto \frac{1}{d^2}$，其中 d 是采样点与光源 $\mathbf{x}_e$ 的距离。选择在主光线上与 d 的平方成反比的方式采样可以达到光源的等角度采样的效果，如下图所示。

这种**解耦**（decouple）的策略的优势在于：

- 透射率和内散射的估计可以使用不同的采样率和策略，采样过程更灵活。
- 透射率的估计通常比内散射更光滑，需要的样本数也会更少。



- 内散射的估计可以在透射率采样点的基础上进行，也减少了对整条光线的采样。

在有限的采样预算下，这种解耦思想再一次地体现出蒙特卡洛方法中的把算力集中在重要的部分的思想。

11.9 体积路径追踪

当我们把参与介质加入路径追踪后，我们肯定要保持一些已有的惯例：与普通的路径追踪一样，我们也必须要避免渲染次数指数级增长。但是，与普通的路径追踪不同，现在我们的散射可能发生在体积中，但是同时，我们也依然要支持表面上的折射和反射。

在 VRE 中，自发光项和散射项均是从 0 到 z 的 dt 的积分，因此，我们可以将 VRE 简化表达为

$$L(\mathbf{x},\omega) = \int_0^z T_r(\mathbf{x},\mathbf{x}_t)[\sigma_a(\mathbf{x}_t)L_e(\mathbf{x}_t,\omega) + \sigma_s(\mathbf{x}_t)L_s(\mathbf{x}_t,\omega)]dt + T_r(\mathbf{x},\mathbf{x}_z)L(\mathbf{x}_z,\omega)$$

11.9.1 蒙特卡洛采样方法

我们简单地使用蒙特卡洛估值法，

$$\langle L(\mathbf{x},\omega)\rangle = \frac{T_r(\mathbf{x},\mathbf{x}_t)}{p(t)}[\sigma_a(\mathbf{x}_t)L_e(\mathbf{x}_t,\omega) + \sigma_s(\mathbf{x}_t)L_s(\mathbf{x}_t,\omega)] + \frac{T_r(\mathbf{x},\mathbf{x}_z)}{P(z)}L(\mathbf{x}_z,\omega)$$

注意，上式中 $p(t)$ 是距离 t 的概率密度，而 $P(z)$ 是超过距离 z 的概率。

对 $L_s(\mathbf{x}_t,\omega)$ 也进行蒙特卡洛积分，我们可以得到，

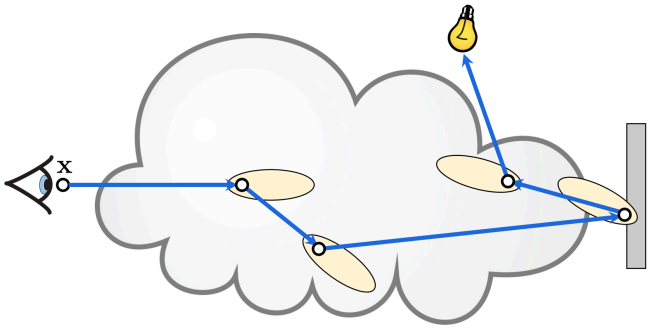
$$\langle L(\mathbf{x},\omega)\rangle = \frac{T_r(\mathbf{x},\mathbf{x}_t)}{p(t)}\left[\sigma_a(\mathbf{x}_t)L_e(\mathbf{x}_t,\omega) + \sigma_s(\mathbf{x}_t)\frac{f_p(\omega,\omega_i)L(\mathbf{x}_t,\omega_i)}{p(\omega_i)}\right] + \frac{T_r(\mathbf{x},\mathbf{x}_z)}{P(z)}L(\mathbf{x}_z,\omega)$$

其中，$p(\omega_i)$ 是方向 ω_i 的概率密度。

所以，当我们调用采样函数时，一共要采样的有两个随机变量，体积光线追踪的基本思路其实都可以模拟成如下图所示的此流程：

- 采样当前至下一次光线与场景的交互的距离 t。
- 采样在体积内的散射方向或是在表面上的散射方向 ω_i。

我们当然也可以像之前的路径追踪中进行下一事件估计，对直接光照进行一次采样，然后再进行多重重要性采样。在 11.11 章节中讨论具体的实现时，我们再来讨论这些机制的加入。



11.9.2 自由路径采样

其中一种采样路径的策略被称作**自由路径采样**（Free-path Sampling）。在讨论自由路径采样之前，我们先考虑一下如何去采样相位函数。

采样相位函数

相位函数可以被认为是体积渲染中与 BSDF 同地位的存在，因此，肯定也需要采样相位函数的方法。对于各向同性的介质，采样相位函数可以通过简单的均匀球面采样完成。对于 Henyey-Greenstein 相位函数，我们可以使用之前的反函数方法，这里不再具体重复流程。通过在 [0,1] 上均匀采样两个随机变量 ξ_1, ξ_2，我们可以得到

$$\cos\theta = \frac{1}{2g}\left(1 + g^2 - \left(\frac{1-g^2}{1-g+2g\xi_1}\right)^2\right)$$

$$\phi = 2\pi\xi_2$$

通过上面的流程我们可以完成采样的过程，在蒙特卡洛积分中我们还需要采样的概率密度分布。由之前对相位函数的介绍，我们注意到相位函数本身已经内置了散射概率的分布特征，且相位函数本身可以被视作一个概率密度函数，因此，我们可以直接将相位函数的值作为概率密度函数的值。

同质介质的自由路径采样

下面我们介绍自由路径采样的过程。

定义（自由路径长度）点 $\mathbf{x}$ 的**自由路径长度**（或者**自由飞行距离**）（free-path length/free-flight distance）指的是从该点出发到下一次散射事件发生前行进的距离。

自由路径采样主要是采样光线在参与介质中传播的自由路径长度。在模拟光线在散射介质中传播时，随机确定光线在遇到下一个散射事件或被吸收前可以行进的距离。在 11.3 中，我们介绍了无效散射。在体积光线追踪中，我们总是要采样当前至下一次光线与场景的交互事件发生的距离 t，且我们采样的分布应该是能够与我们的积分值成正比的。

我们首先需要采样的是透射率。对于同质介质，透射率

$$T_r(t) = e^{-\sigma_{\text{maj}} t}$$

是一个指数函数。我们可以使用指数分布来采样这个函数。其步骤如下：

- 生成随机数 $\xi \in [0,1]$。
- 获得采样距离 $t = -\frac{\ln(1-\xi)}{\sigma_{\text{maj}}}$。
- 计算 PDF $p(t) = \sigma_{\text{maj}} e^{-\sigma_{\text{maj}} t}$

在使用这个流程的时候，对应的 PDF

$$p(t) = \sigma_{\text{maj}} e^{-\sigma_{\text{maj}} t}$$

其值在任意 $t \in [0, \infty)$ 上都是非负的，因此这个采样算法有的时候会采样出 $t > z$ 的情况，其中 z 是路径追踪一次路径起点与终点的距离。这似乎不太可行，但实际上考虑到我们可以将等式 (11.4) 的第二项写作

$$\int_0^z e^{-\sigma_{\text{maj}} t} \sigma_n(\mathbf{x} + t\omega) T_r(\mathbf{p} + t\omega, \mathbf{p}') dt = \int_0^z e^{-\sigma_{\text{maj}} t} \sigma_n(\mathbf{x} + t\omega) T_r(\mathbf{p} + t\omega, \mathbf{p}') dt + \int_z^\infty 0 dt.$$

如果我们对等式右边的两个积分项进行蒙特卡洛估值，我们可以看到 t 的值可以决定哪一个积分被计算，所以当我们计算出 $t > z$ 时，也就可以退出递归了。这样，我们也就得到了自由路径采样中透射率的蒙特卡洛估值。

$$T_r(\mathbf{x}, \mathbf{y}) \approx e^{-\sigma_{\text{maj}} z} + \begin{cases} \frac{\sigma_n(\mathbf{x}+t\omega)}{\sigma_{\text{maj}}} T_r(\mathbf{x} + t\omega, \mathbf{y}) & \text{If } t < z \\ 0 & \text{Otherwise.} \end{cases} \tag{11.5}$$

11.10 Delta 追踪

上文中提到的自由路径采样的

11.11 参与介质数据结构 *

11.11.1 Medium 类

```
// medium.h
class Medium {
public:
    virtual ~Medium() = default;
    virtual float Tr(const Ray3f &ray, Sampler &sampler) const = 0;
    virtual float sample(const Ray3f &ray, Sampler &sampler, MediumInteraction &mi) const = 0;
    virtual float density(const Vec3f &p) const = 0;
    std::shared_ptr<PhaseFunction> phase;
};
```

11.11.2 PhaseFunction 类

```
// medium.h
class PhaseFunction {
public:
    virtual ~PhaseFunction() = default;
    virtual float p(const Vec3f &wo, const Vec3f &wi) const = 0;
```

```
    virtual float sample(const Vec3f &wo, Vec3f &wi, const Vec2f &sample) const = 0;
};
```

```
// medium.h
class HenyeyGreenstein : public PhaseFunction {
public:
    HenyeyGreenstein(const json &j = json::object());
    float p(const Vec3f &wo, const Vec3f &wi) const override;
    float sample(const Vec3f &wo, Vec3f &wi, const Vec2f &sample) const override;
private:
    float g = 0;
};
```

11.11.3 MediumInterface 结构体与 MediumInteraction 结构体

```
// medium.h
struct MediumInteraction {
    Vec3f p;
    Vec3f wo;
    const Medium *medium = nullptr;

    MediumInteraction(){};
    MediumInteraction(const Vec3f &p, const Vec3f &wo, const Medium *medium) : p(p), wo(wo),
    medium(medium){};
    bool isValid() { return medium != nullptr; };
};
```

```
// medium.h
struct MediumInterface {
    MediumInterface() : inside(nullptr), outside(nullptr) {}

    MediumInterface(const std::shared_ptr<const Medium> medium) : inside(medium), outside(medium)
     {}

    MediumInterface(const std::shared_ptr<const Medium> inside, const std::shared_ptr<const
    Medium> outside)
        : inside(inside), outside(outside) {}

    bool isMediumTransition() const { return inside != outside; }

    std::shared_ptr<const Medium> getMedium(const Ray3f &ray, const HitInfo &hit) const;

    std::shared_ptr<const Medium> inside = nullptr;
    std::shared_ptr<const Medium> outside = nullptr;
};
```

11.12 支持体积渲染的 Integrator 类 *

11.12.1 VolPathTracerUni 类

```cpp
// integrator.h
class VolpathTracerUni : public Integrator {
  public:
    VolpathTracerUni (const json &j = json::object());

    Color3f Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const override;

    Color3f recursiveColor(const Scene &scene, Sampler &sampler, const Ray3f &ray, int depth)
    const;

    int m_maxBounces = 64;
};
```

```cpp
// integrator.cpp
Color3f VolpathTracerUni::Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const {
    Color3f throughput(1.0f);
    Color3f result(0.0f);
    Ray3f currentRay = ray;

    std::shared_ptr<const Medium> currentMedium = ray.medium;

    int depth = 0;

    while (depth <= m_maxBounces) {
        HitInfo hit;
        if(!scene.intersect(currentRay, hit)) {
            result += throughput * scene.background(currentRay);
            break;
        }

        // limit the maxt of the ray
        currentRay.maxt = length(hit.p - currentRay.o) / length(currentRay.d) + Epsilon;

        // check if it hits the medium boundary
        if(hit.mat == nullptr && hit.mi != nullptr && hit.mi -> isMediumTransition()) {
            currentMedium = hit.mi -> getMedium(currentRay, hit);
            currentRay.medium = currentMedium;
        }
        MediumInteraction mi;
        if (currentRay.medium != nullptr) {
            throughput *= currentMedium -> sample(currentRay, sampler, mi);
        }

        if (mi.isValid()) {
            currentRay.maxt = length(mi.p - currentRay.o) / length(currentRay.d) + Epsilon;

            result += throughput * TrL(scene, sampler, currentRay);
            Vec3f wo = -currentRay.d;
            Vec3f wi;
            float volpdf = currentMedium -> phase -> sample(wo, wi, sampler.next2D());
            if(volpdf == 0.f) break;
            currentRay = Ray3f(mi.p, wi).withMedium(currentMedium);
        } else {
```

```
            ScatterRecord srec;

            Color3f emitted = hit.mat -> emitted(currentRay, hit);


            Vec2f sample = sampler.next2D();

            if(!hit.mat -> sample(currentRay.d, hit, sample, srec)) {
                result += throughput * emitted;
                break;
            }

            if(srec.isSpecular) {
                result += throughput * emitted;
                throughput *= srec.attenuation;
            }

            float pdf = hit.mat -> pdf(currentRay.d, srec.scattered, hit);
            if(pdf == 0.f) {
                result += throughput * emitted;
                break;
            }

            Color3f eval = hit.mat -> eval(currentRay.d, srec.scattered, hit);
            result += throughput * emitted;
            throughput *= eval / pdf;

            currentRay = Ray3f(hit.p, srec.scattered).withMedium(currentMedium);
        }


        float lum = luminance(throughput);
        float q = 1 - lum;
        if(depth >  0 && lum < 0.01f){
            float xi = sampler.next1D();
            if(xi < q) {
                break;
            }
            throughput /= (1-q);
        }

        depth++;
    }

    return result;
}
```

11.12.2 VolPathTracerMIS 类

```
// integrator.h
class VolpathTracerNEE : public Integrator {
```

```
  public:
    VolpathTracerNEE (const json &j = json::object());

    Color3f Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const override;

    Color3f recursiveColor(const Scene &scene, Sampler &sampler, const Ray3f &ray, int depth,
                           float misWeight) const;

    int m_maxBounces = 64;
    float m_power = 2.f;
};
```

```
// medium.cpp
Color3f TrL(const Scene &scene, Sampler &sampler, const Ray3f &ray_) {
    Ray3f ray = ray_.normalizeRay();
    float Tr = 1.0f;
    shared_ptr<const Medium> currentMedium = ray.medium;

    while(true) {
        HitInfo hit;
        if(scene.intersect(ray, hit)) {
            ray.maxt = length(hit.p - ray.o) / length(ray.d) + Epsilon;

            if (hit.mat != nullptr) {
                // hit a light, accumulate the light;
                Color3f emission = hit.mat->emitted(ray, hit);
                return Tr * emission;
            }

        } else {
            return Tr * scene.background(ray);
        }

        if (currentMedium != nullptr) {
            Tr *= currentMedium -> Tr(ray, sampler);
        }

        if(hit.mi != nullptr && hit.mi->isMediumTransition()) {
            ray.medium = hit.mi -> getMedium(ray, hit);
            currentMedium = ray.medium;

        }

        if(Tr < Epsilon) break;

        ray.o = hit.p;
        ray.maxt = INFINITY;
    }

    return Color3f(0.f);
}

// integrator.cpp
```

```
Color3f VolpathTracerNEE::Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const {
    Color3f throughput(1.0f);
    Color3f result(0.0f);

    Ray3f currentRay = ray;

    float misWeight = 1.0f;
    int depth = 0;

    while (depth <= m_maxBounces) {
        HitInfo hit;
        // Intersecting with the scene
        if (!scene.intersect(currentRay, hit)) {
            result += throughput * scene.background (currentRay);
            break;
        }

        currentRay.maxt = length(hit.p - currentRay.o) / length(currentRay.d) + Epsilon;

        if(hit.mat == nullptr && hit.mi != nullptr && hit.mi -> isMediumTransition()) {
            currentRay.medium = hit.mi -> getMedium(currentRay, hit);
        } if(hit.mat == nullptr && hit.mi == nullptr) {
            currentRay.medium = nullptr;
            continue;
        }

        Color3f emitted(0.0f);
        if(hit.mat != nullptr)
            emitted = misWeight * throughput * hit.mat -> emitted(currentRay, hit);
        result += emitted;

        MediumInteraction mi;
        if(currentRay.medium != nullptr) {
            throughput *= currentRay.medium -> sample(currentRay, sampler, mi);
        }


        if (mi.isValid() && currentRay.medium != nullptr) {
            Vec3f wi;
            float pdfMat1 = currentRay.medium -> phase -> sample(mi.wo, wi, sampler.next2D());
            if(pdfMat1 == 0.f) break;
            float pdfMat2 = scene.emitters().pdf(mi.p, wi);

            Vec3f scatterDirLight = scene.emitters().sample(mi.p, sampler.next2D());
            Ray3f scatteredLight = Ray3f(mi.p, scatterDirLight);
            HitInfo lightHit1;
            float pdfLight1 = scene.emitters().pdf(mi.p, normalize(scatterDirLight));
            float Tr = currentRay.medium -> Tr(scatteredLight, sampler);

            if(pdfLight1 > 0.f) {
                float pdfLight2 = mi.medium -> phase -> p(mi.wo, normalize(scatteredLight.d));
                float misWeightLight = powerHeuristic(pdfLight1, pdfLight2, m_power);
```

```
                float eval = mi.medium -> phase -> p(mi.wo, normalize(scatteredLight.d));

                if(!scene.intersect(scatteredLight, lightHit1)) {
                    result += throughput * scene.background(scatteredLight);
                }
                else {
                    result += eval * misWeight * throughput * misWeightLight
                    * TrL(scene, sampler, scatteredLight)/ pdfLight1;
                }

                misWeight = powerHeuristic(pdfMat1, pdfMat2, m_power);

                result += throughput * misWeight * eval * Tr / pdfMat1;
            }

            currentRay = Ray3f(mi.p, wi);

        } else {
            ScatterRecord srec;
            if(hit.mat != nullptr) {
                if(!hit.mat->sample(currentRay.d, hit, sampler.next2D(), srec)) {
                    break;
                }

                Vec3f scatterDirLight = scene.emitters().sample(hit.p, sampler.next2D());
                Ray3f scatteredLight(hit.p, scatterDirLight);
                HitInfo lightHit2;
                float pdfLight1 = scene.emitters().pdf(hit.p, normalize(scatterDirLight));

                if(pdfLight1 > 0.f) {
                    float pdfLight2 = hit.mat ->
                    pdf(currentRay.d, normalize(scatteredLight.d), hit);
                    float misWeightLight = powerHeuristic(pdfLight1, pdfLight2, m_power);
                    Color3f evalLight = hit.mat ->
                    eval(currentRay.d, normalize(scatteredLight.d), hit);
                    if(!scene.intersect(scatteredLight, lightHit2))
                        result += throughput * scene.background(scatteredLight);
                    else {
                        result += misWeight * throughput * misWeightLight * evalLight
                        * lightHit2.mat -> emitted(scatteredLight, lightHit2) / pdfLight1;
                    }
                }


                if(srec.isSpecular) {
                    misWeight = 1.0f;
                    throughput *= srec.attenuation;
                } else {

                    Vec3f scatterDirMat = srec.scattered;
                    Ray3f scatteredMat(hit.p, scatterDirMat);

```

```
                    float pdfMat1 = hit.mat->pdf(currentRay.d, normalize(scatteredMat.d), hit);


                    Color3f evalMat(1.0f);
                    if(pdfMat1 > 0.f) {
                        evalMat = hit.mat -> eval(currentRay.d, normalize(scatteredMat.d), hit);
                        throughput *=  evalMat / pdfMat1;
                    } else {
                        break;
                    }

                    float pdfMat2 = scene.emitters().pdf(hit.p, normalize(scatteredMat.d));
                    misWeight = powerHeuristic(pdfMat1, pdfMat2, m_power);
                }
            }

            currentRay = Ray3f(hit.p, srec.scattered);

            float lum = luminance(throughput);
            float q = 1 - lum;
            if(depth >  0 && lum < 0.01f){
                float xi = sampler.next1D();
                if(xi < q) {
                    break;
                }
                throughput /= (1-q);
            }
        }

        depth++;
    }

    return result;
}
```

Part II

前沿话题

Chapter 12

双向路径追踪

在使用普通路径追踪处理一些复杂光照场景时，我们有的时候不仅会存在效率低下的问题，更严重的是有可能会产生错误渲染。例如，在**焦散** (caustic)[1]场景中，普通的路径追踪可能根本无法处理。

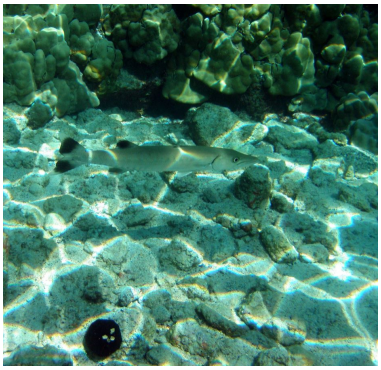


图 12.1: 焦散现象

在小孔光照、细缝光线传播中，正确的光路径可能只占所有可能路径中的一部分。甚至我们即将会看到，在一些特定情况下我们永远无法采样真实存在且重要的光路径。于是，Lafortune 和 Willems（1993）与 Veach 和 Guibas（1994）分别独立提出了**双向路径追踪**（bidirectional pathtracing, BDPT）方法。其核心思想在于通过从光源发出路径，并于摄像机发出的路径相结合，使得我们更容易找到这些复杂光照效果的正确路径。它虽然增加了计算上的复杂性，但是在很多情况下可以极大地提高渲染效率和质量，特别是在场景中直接和间接光照相互作用频繁的情况下。

12.1 光路径描述

首先，我们需要一种更准确的抽象方式来正确、直观地描述光路径。一种方法是使用 Heckbert 提出的节点描述法，其将光路径描述成为光与表面发生交互事件时，代表事件的种类的节点组成的链。

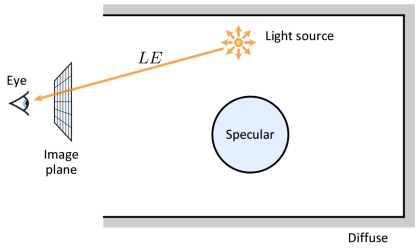
[1]Ling-qi Yan 教授曾经在 GAMES101 中说过焦散是一个失败的中文翻译。焦散实际上指的是光线通过透明物体或反射物体（例如水面、玻璃和其他光滑的反射材料）时发生的光线集中与聚焦，与“散”字表达的意义完全不同。

12.1.1 Heckbert 描述法

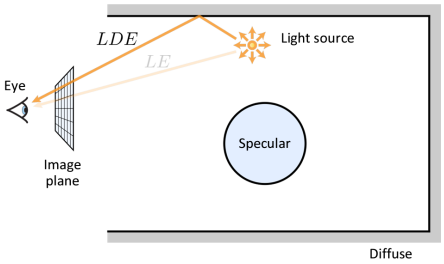
在 Heckbert 的分类中，有以下几种事件节点：

- L: 一种光源。
- E：人眼。
- S：一种镜面反射。
- D：一种漫反射。

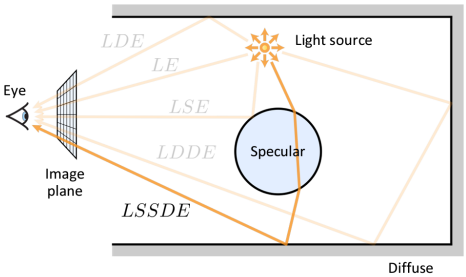
例如，一条从光源出发直接进入人眼的光路径可以被记录为 LE。



一条从光源出发，击中漫反射墙面后反射，再进入人眼的光路径可以被记录为 LDE。



一条从光源出发，击中镜面球体之后穿过球体，再从球体内折射出来击中漫反射墙面，之后反射再进入人眼的光路径可以被记录为 $LSSDE$。



12.1.2 光路径正则表达式

我们可以用一些正则表达式进一步拓展这个编码方式。

- k^+：在当前位置可能存在一个或更多的 k 类节点。
- k^*：在当前位置可能存在零个或更多的 k 类节点。
- $k?$：在当前位置可能存在一个或零个 k 类节点。
- $(k|h)$：在当前位置可能存在一个 k 类或 h 类节点。

在这种正则表达式下，直接光照就可以被表示为 $L(D|S)E$，间接光照就可以被表示为 $L(D|S)(D|S)^+E$，之前提及的焦散现象就可以被描述为 LS^+DE。

12.2 传统路径追踪的问题

传统的路径追踪固然有一些很好的优势，例如它能提供渲染方程的一个完整的解，也很容易被实现。但是，它也存在着一些问题。

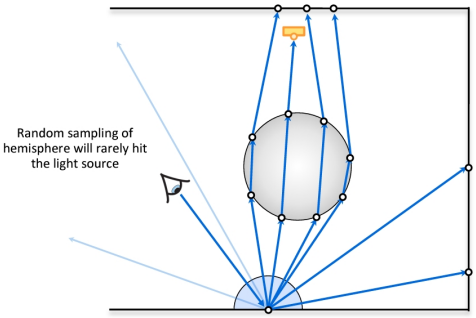
- 收敛速度很慢。在 7.3.4 中，我们曾经得出过结论：需要四倍的采样量才能使得误差减半。
- 健全性有问题。我们很快就会在下面分析，传统的光线追踪完全无法处理特定的光路径。
- 效率低下。没有健全的复用或是计算缓存的算法，使得每一轮迭代都必须重新计算。

这些问题应该有其特定的解决方案。但是首先，我们先对第二点中提到的健全性进行一些更详细的解释。

从焦散现象看路径追踪

焦散（caustic）现象指的是光线经过透明曲面（例如透镜、水面或是玻璃时）发生弯曲，然后再在另一个表面上聚焦而产生明亮区域的现象，其光路径为 LS^+DE。这意味着来自光源的直接光照在焦散现象中非常重要。

然而，观察传统的路径追踪中我们的采样逻辑。



当光源的面积减小时，我们从光线出发点采样一条经过光源的路径就会变得更加困难。在我们采样的多数路径中，可能它们都不经过光源。而且，注意到如果该光源是我们在第 9 章中讨论过的点光源或是聚光灯光源，那么实际上它们的大小为 0，因此随机采样将永远不会采样到一条经过光源的路径。在处理这类小面积光源时，虽然蒙特卡洛渲染本身是无偏的，但是因为光路径通常采样失败或是采样不到，也因此，焦散现象本身就会无法渲染出来，而得出了必定错误的渲染结果。

为了解决这个健全性的问题，我们引入重要性度量学。

12.3 重要性函数的直观感受

12.3.1 直观定义

到目前为止，我们已经将全局照明问题描述成：给定光源分布，计算点 $\mathbf{x}$ 和方向 ω_o 中的入射或出射辐射亮度问题。渲染方程描述的就是在点 $\mathbf{x}$ 处的辐射属性平衡：即，在属于 $\mathcal{A}\times\mathcal{H}^2$ 的所有点和方向上定义的特定函数 L_e 通过辐射亮度的传输，即可确定同样在 $\mathcal{A}\times\mathcal{H}^2$ 上定义的函数 L_o：

$$L_o(\mathbf{x},\omega_o)=L_e(\mathbf{x},\omega_o)+\int_{\mathcal{H}^2}f_r(\mathbf{x},\omega_i,\omega_o)L_i(\mathbf{x},\omega_i)\cos\theta_i d\omega_i.$$

而全局照明算法必须解决的问题是计算在图像中某一个特定像素 j 可见的“光能”。在这里，每个像素都充当了传感器，“光能”到达传感器上，而像素的概念则和如何响应或表示这些“光能”有关。在下式中，

$$I_j=\int_{\mathcal{A}_{\text{film}}}\int_{\mathcal{H}^2}W(\mathbf{x},\omega)L_i(\mathbf{x},\omega)\cos\theta d\omega d\mathbf{x}. \tag{12.1}$$

其中的 W 是在屏幕的位置 $\mathbf{x}$ 上的假想传感器对来自 ω 方向的辐射亮度的一种响应度量（response），这个值被称作**响应函数**（response function），在一些文献中，响应函数也被称作**潜力函数**（potential function）或是**重要性**（importance）。方程 (12.1) 也被称作**重要性方程**（Importance Equation）。

重要性在形式上与渲染方程类似：

$$W(\mathbf{x},\omega_o)=W_e(\mathbf{x},\omega_o)+\int_{\mathcal{H}^2}f_r(\mathbf{x},\omega_i,\omega_o)W(\mathbf{x},\omega_i)\cos\theta d\omega_i \tag{12.2}$$

值得一提的是，重要性的流向与辐射的方向是相反的。我们可以通过一个不正式的响应函数举例说明。假设有两个表面，i 和 j。如果表面 i 在特定图像中人眼可见，那么 $W_e(i)$ 将捕获该表面对图像的重要性（图像表面投影面积的某种度量）。再假设表面 j 也是可见的，并且表面 i 会将光反射到表面 j，那么，由于表面 j 的重要性，表面 i 会间接地更加重要。因此，当光能从表面 i 流向表面 j 时，重要性却从表面 j 流向表面 i。

12.3.2 重要性与辐射亮度的区别

这两者虽然有着很类似的形式，但是，对于辐射亮度而言，

- 辐射亮度发射自光源。
- 辐射亮度被描述为在一个微分光束中的光子的数量。

而对于重要性而言，

- 重要性“发射自”传感器（人眼、相机、辐射传感器）。
- 重要性被描述为传感器对某一微分光束中的辐射亮度的响应度量。

12.3.3 辐射亮度与重要性的二重性

对于公式（12.1），我们开始进行以下的解构。首先，我们尝试将重要性的公式转换至直角坐标系。

$$\begin{aligned}I_j&=\int_{\mathcal{A}_{\text{film}}}\int_{\mathcal{H}^2}W_e(\mathbf{x},\omega)L_i(\mathbf{x},\omega)\cos\theta d\omega d\mathbf{x}\\&=\int_{\mathcal{A}_{\text{film}}}\int_{\mathcal{A}}W_e(\mathbf{x},\mathbf{y})G(\mathbf{x},\mathbf{y})L_o(\mathbf{y},\mathbf{x})d\mathbf{y}d\mathbf{x}\end{aligned}$$

在这里，$\mathbf{x}$ 表示的是屏幕上的一个点，我们关注的是 $\mathbf{x}$ 到 $\mathbf{y}$ 的光路径。其中，$G(\mathbf{x},\mathbf{x})$ 是 $\mathbf{x}$ 到 $\mathbf{y}$ 的几何项，$L_o(\mathbf{y},\mathbf{x})$ 表示从 $\mathbf{y}$ 到 $\mathbf{x}$ 的出射辐射亮度。接下来，我们尝试展开 L_o，并只考虑直接光照。

$$\begin{aligned} I_j &= ... \\ &= \int_{\mathcal{A}_{\text{film}}} \int_{\mathcal{A}} \int_{\mathcal{A}_{\text{light}}} W_e(\mathbf{x},\mathbf{y}) G(\mathbf{x},\mathbf{y}) f(\mathbf{y},\mathbf{z},\mathbf{x}) G(\mathbf{y},\mathbf{z}) L_e(\mathbf{z},\mathbf{y}) d\mathbf{z} d\mathbf{y} d\mathbf{x} \end{aligned}$$

交换一下积分的顺序，

$$\begin{aligned} I_j &= ... \\ &= \int_{\mathcal{A}_{\text{light}}} \int_{\mathcal{A}} \int_{\mathcal{A}_{\text{film}}} W_e(\mathbf{x},\mathbf{y}) G(\mathbf{x},\mathbf{y}) f(\mathbf{y},\mathbf{z},\mathbf{x}) G(\mathbf{y},\mathbf{z}) L_e(\mathbf{z},\mathbf{y}) d\mathbf{x} d\mathbf{y} d\mathbf{z} \end{aligned}$$

由于 $G(\cdot)$ 和 $f(\mathbf{y},\cdot)$ 均为对称函数，我们可以交换 $\mathbf{x}$ 和 $\mathbf{y}$ 的顺序。

$$\begin{aligned} I_j &= ... \\ &= \int_{\mathcal{A}_{\text{light}}} \int_{\mathcal{A}} \int_{\mathcal{A}_{\text{film}}} W_e(\mathbf{x},\mathbf{y}) G(\mathbf{y},\mathbf{z}) f(\mathbf{y},\mathbf{x},\mathbf{z}) G(\mathbf{z},\mathbf{y}) L_e(\mathbf{z},\mathbf{y}) d\mathbf{x} d\mathbf{y} d\mathbf{z} \\ &= \int_{\mathcal{A}_{\text{light}}} \int_{\mathcal{A}} W_o(\mathbf{y},\mathbf{z}) G(\mathbf{z},\mathbf{y}) L_e(\mathbf{z},\mathbf{y}) d\mathbf{y} d\mathbf{z} \end{aligned}$$

换回球坐标系，

$$\begin{aligned} I_j &= \int_{\mathcal{A}_{\text{light}}} \int_{\mathcal{A}} W_o(\mathbf{y},\mathbf{z}) G(\mathbf{z},\mathbf{y}) L_e(\mathbf{z},\mathbf{y}) d\mathbf{y} d\mathbf{z} \\ &= \int_{\mathcal{A}_{\text{light}}} \int_{\mathcal{H}^2} W_i(\mathbf{z},\omega) L_e(\mathbf{z},\omega) \cos\theta d\omega d\mathbf{z} \end{aligned}$$

至此，我们就可以写出如下的关系式:

$$\begin{aligned} I_j &= \int_{\mathcal{A}_{\text{film}}} \int_{\mathcal{H}^2} W_e(\mathbf{x},\omega) L_i(\mathbf{x},\omega) \cos\theta d\omega d\mathbf{x} \\ &= \int_{\mathcal{A}_{\text{light}}} \int_{\mathcal{H}^2} W_i(\mathbf{z},\omega) L_e(\mathbf{z},\omega) \cos\theta d\omega d\mathbf{z} \end{aligned}$$

在这个等式中，第一个部分就是传统的路径追踪的思路：我们从屏幕出发，然后查找光源的辐射亮度。而第二个部分就是我们即将讨论的光追踪：从光源出发，然后查找其在人眼的重要性。

12.4 光追踪

光追踪（Light Tracing）的核心思想是从光源发射多条随机路径，追踪直至其离开场景或进入人眼。焦散现象在路径追踪（左）与光追踪（右）的区别就如图 12.2 所示。

12.4.1 测量方程

渲染方程表示场景中光能的稳态分布（steady-state distribution），重要性方程表示表面对图像的相对重要性。而**测量方程**（measurement function）则用于评估不同光线传输路径的重要性，并以此指导蒙特卡洛采样，提高渲染效率。该等式将两个基本量——辐射亮度和重要性统一在一个表达式中。在积分式

$$I_j = \int_{\mathcal{A}} \int_{\mathcal{A}} W_e(\mathbf{x}_0,\mathbf{x}_1) G(\mathbf{x}_0,\mathbf{x}_1) L_o(\mathbf{x}_1,\mathbf{x}_0) d\mathbf{x}_1 d\mathbf{x}_0$$

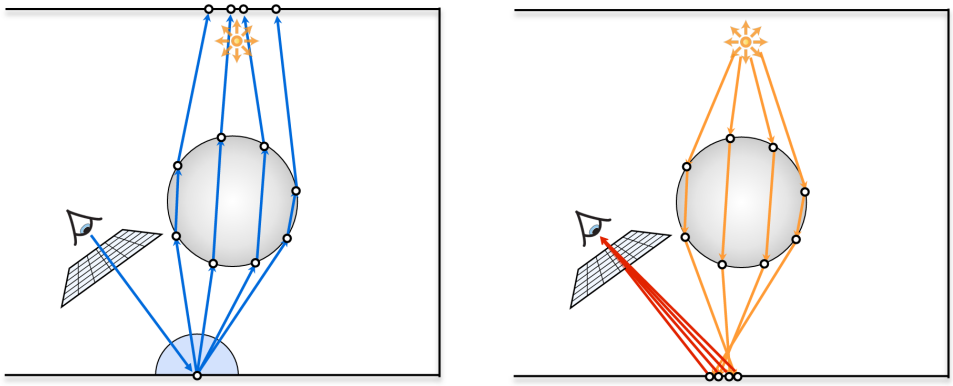


图 12.2: 路径追踪（左）与光追踪（右）的区别

中，我们可以将其拆分成发光项和间接光照项，

$$
\begin{aligned}
I_j &= \int_{\mathcal{A}}\int_{\mathcal{A}} W_e(\mathbf{x}_0,\mathbf{x}_1)G(\mathbf{x}_0,\mathbf{x}_1)L_o(\mathbf{x}_1,\mathbf{x}_0)d\mathbf{x}_1 d\mathbf{x}_0 \\
&= \int_{\mathcal{A}}\int_{\mathcal{A}} W_e(\mathbf{x}_0,\mathbf{x}_1)G(\mathbf{x}_0,\mathbf{x}_1)L_e(\mathbf{x}_1,\mathbf{x}_0) && (\text{发光项}E) \\
&+ \int_{\mathcal{A}} f(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_0)G(\mathbf{x}_1,\mathbf{x}_2)L_o(\mathbf{x}_2,\mathbf{x}_1)d\mathbf{x}_2 d\mathbf{x}_1 d\mathbf{x}_0 && (\text{间接光照项}KL)
\end{aligned}
$$

我们可以进一步将第二项的间接光照项 KL 再次展开，

$$
\begin{aligned}
I_j &= \int_{\mathcal{A}}\int_{\mathcal{A}} W_e(\mathbf{x}_0,\mathbf{x}_1)G(\mathbf{x}_0,\mathbf{x}_1)L_e(\mathbf{x}_1,\mathbf{x}_0) && (\text{发光项}E) \\
&+ \int_{\mathcal{A}} f(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_0)G(\mathbf{x}_1,\mathbf{x}_2)L_o(\mathbf{x}_2,\mathbf{x}_1)d\mathbf{x}_2 d\mathbf{x}_1 d\mathbf{x}_0 && (\text{间接光照项}KL) \\
&= \int_{\mathcal{A}}\int_{\mathcal{A}} W_e(\mathbf{x}_0,\mathbf{x}_1)G(\mathbf{x}_0,\mathbf{x}_1)L_e(\mathbf{x}_1,\mathbf{x}_0) && (\text{发光项}E) \\
&+ \int_{\mathcal{A}} f(\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_0)G(\mathbf{x}_1,\mathbf{x}_2)L_e(\mathbf{x}_2,\mathbf{x}_1) && (\text{发光项}KE) \\
&+ \int_{\mathcal{A}} f(\mathbf{x}_2,\mathbf{x}_3,\mathbf{x}_1)G(\mathbf{x}_2,\mathbf{x}_3)L_o(\mathbf{x}_3,\mathbf{x}_2)d\mathbf{x}_3 d\mathbf{x}_2 d\mathbf{x}_1 && (\text{间接光照项}K^2L)
\end{aligned}
$$

可以无穷尽地将这个式子继续展开下去，我们可以将这个式子简写为一个递归式，

$$L = E + KL = E + KE + K^2E + K^3E + ...$$

其中，E 代表的就是直接进入人眼的发光，K^nE 代表的就是经过 n 次弹射进入人眼的间接光照。

12.4.2 测量方程的积分式

将测量方程完全写成积分式，我们可以继续遵从上一小节的步骤展开，并找到规律，最终，我们会得到下面的结果。:

$$
\begin{aligned}
I_j =& (\text{测量方程原式}) \int_{\mathcal{A}} \int_{\mathcal{A}} W_e(\mathbf{x}_0, \mathbf{x}_1) G(\mathbf{x}_0, \mathbf{x}_1) L_o(\mathbf{x}_1, \mathbf{x}_0) d\mathbf{x}_1 d\mathbf{x}_0 \\
=& (\text{发光项}) \iint_{\mathcal{A}} W_e(\mathbf{x}_0, \mathbf{x}_1) G(\mathbf{x}_0, \mathbf{x}_1) L_e(\mathbf{x}_1, \mathbf{x}_0) \\
&+ (\text{直接光照}) \iiint_{\mathcal{A}} W_e(\mathbf{x}_0, \mathbf{x}_1) L_e(\mathbf{x}_2, \mathbf{x}_1) G(\mathbf{x}_0, \mathbf{x}_1) f(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_0) G(\mathbf{x}_1, \mathbf{x}_2) d\mathbf{x}_2 d\mathbf{x}_1 d\mathbf{x}_0 + ... \\
&+ ((k-2)\text{次弹射间接光照}) \int ... \int_{\mathcal{A}} W_e(\mathbf{x}_0, \mathbf{x}_1) L_e(\mathbf{x}_k, \mathbf{x}_{k-1}) G(\mathbf{x}_0, \mathbf{x}_1) \prod_{j=1}^{k-1} f(\mathbf{x}_j, \mathbf{x}_{j+1}, \mathbf{x}_{j-1}) G(\mathbf{x}_j, \mathbf{x}_{j+1}) d\mathbf{x}_k ... d\mathbf{x}_0 + ...
\end{aligned}
$$

这里，我们引入两个定义。

定义（路径空间）所有由 k 条线段组成的路径形成的**路径空间**（path space）被记作 $\mathcal{P}_k$，其定义为

$$\mathcal{P}_k = \{\bar{\mathbf{x}} = \mathbf{x}_0 ... \mathbf{x}_k ; \mathbf{x}_0 ... \mathbf{x}_k \in \mathcal{A}\} \tag{12.3}$$

定义（路径吞吐量）一条光线路径 $\bar{\mathbf{x}_k}$ 的**路径吞吐量**[2]（throughput）定义为

$$T(\bar{\mathbf{x}_k}) = G(\mathbf{x}_0, \mathbf{x}_1) \prod_{j=1}^{k-1} f(\mathbf{x}_j, \mathbf{x}_{j+1}, \mathbf{x}_{j-1}) G(\mathbf{x}_j, \mathbf{x}_{j+1}) \tag{12.4}$$

有了这两个定义，我们就可以将测量方程的积分式简化表达为

$$
\begin{aligned}
I_j =& (\text{测量方程原式}) \int_{\mathcal{A}} \int_{\mathcal{A}} W_e(\mathbf{x}_0, \mathbf{x}_1) G(\mathbf{x}_0, \mathbf{x}_1) L_o(\mathbf{x}_1, \mathbf{x}_0) d\mathbf{x}_1 d\mathbf{x}_0 \\
=& (\text{发光项}) \int_{\mathcal{P}_1} W_e(\mathbf{x}_0, \mathbf{x}_1) L_e(\mathbf{x}_1, \mathbf{x}_0) T(\bar{\mathbf{x}_1}) d\mathbf{x}_1 \\
&+ (\text{直接光照}) \int_{\mathcal{P}_2} W_e(\mathbf{x}_0, \mathbf{x}_1) L_e(\mathbf{x}_2, \mathbf{x}_1) T(\bar{\mathbf{x}_2}) d\mathbf{x}_2 + ... \\
&+ ((k-2)\text{次弹射间接光照}) \int_{\mathcal{P}_k} W_e(\mathbf{x}_0, \mathbf{x}_1) L_e(\mathbf{x}_k, \mathbf{x}_{k-1}) T(\bar{\mathbf{x}_k}) d\mathbf{x}_k + ...
\end{aligned}
$$

进一步地，如果我们定义

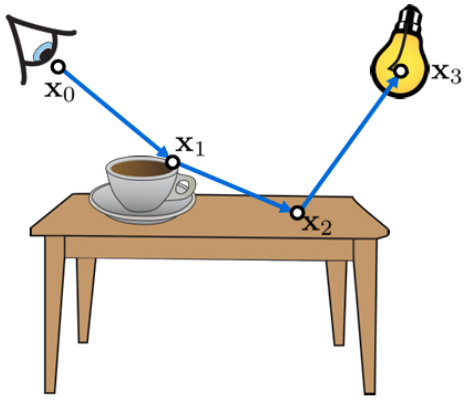
$$\mathcal{P} = \bigcup_{k=1}^{\infty} \mathcal{P}_k$$

为总路径空间，即所有长度的所有路径的空间，则测量方程的积分式可以进一步简化为

$$I_j = \int_{\mathcal{P}} W_e(\mathbf{x}_0, \mathbf{x}_1) L_e(\mathbf{x}_k, \mathbf{x}_{k-1}) T(\bar{\mathbf{x}}) d\mathbf{x}$$

12.4.3 路径构造

以长度为 $k = 3$ 的路径为例，即 $\bar{\mathbf{x}} = (\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) = \mathbf{x}_0 \mathbf{x}_1 \mathbf{x}_2 \mathbf{x}_3$.



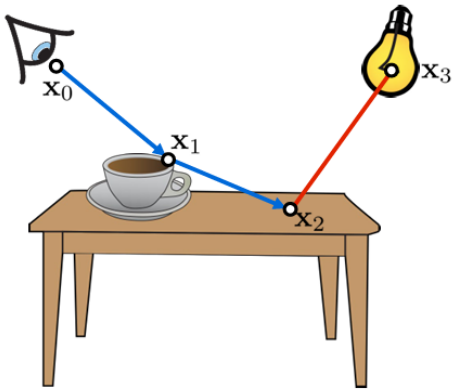
没有 NEE 的路径追踪

这是一个最基本的路径，几乎没有什么需要多做解释的地方。从人眼处开始采样，路径 $\bar{\mathbf{x}}$ 发生的概率可以被表示为

$$p(\bar{\mathbf{x}}) = p(\mathbf{x}_0)p(\mathbf{x}_1|\mathbf{x}_0)p(\mathbf{x}_2|\mathbf{x}_0\mathbf{x}_1)p(\mathbf{x}_3|\mathbf{x}_0\mathbf{x}_1\mathbf{x}_2)$$

有 NEE 的路径追踪

当我们加入下一事件估计之后，路径的概率在 $\mathbf{x}_3$ 处就不再是条件概率，而会是直接采样得到的概率 $p(\mathbf{x}_3)$ 了。



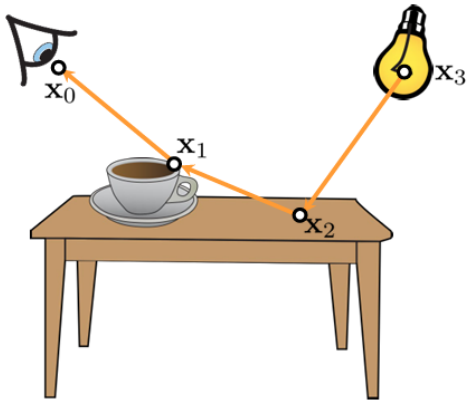
$$p(\bar{\mathbf{x}}) = p(\mathbf{x}_0)p(\mathbf{x}_1|\mathbf{x}_0)p(\mathbf{x}_2|\mathbf{x}_0\mathbf{x}_1)p(\mathbf{x}_3)$$

这是因为，当我们在 $\mathbf{x}_2$ 处引入 NEE 的时候，我们直接在光源上采样了一个新的顶点 $\mathbf{x}_3$。这个操作本身独立于 $\mathbf{x}_0, \mathbf{x}_1$ 与 $\mathbf{x}_2$ 的位置。注意，因为这种采样不依赖于路径上的其它点，这使得我们可以在任意位置引入 NEE，提高光路径的采样效率。引入 NEE 后的概率密度函数虽然形式上发生了变化，但仍然是一个有效的 PDF，可以用于计算渲染方程的蒙特卡洛估计。

[2]在一些文献中 throughput 一词也被翻译作路径通量。为了与辐射度量学中的通量产生区分，这里我们使用吞吐量一词。

没有 NEE 的光追踪

与路径追踪几乎没有什么区别，只是这次我们的光路径是从光源到人眼的。

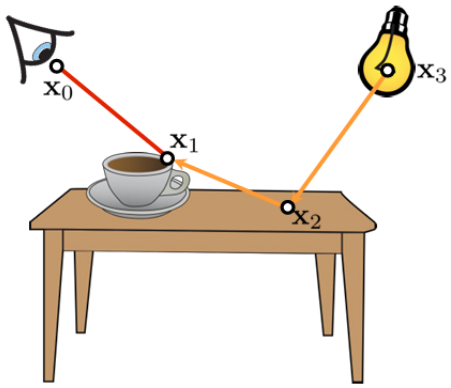


对于没有 NEE 的光追踪，我们也可以很轻松地写出路径 $\bar{\mathbf{x}}$ 的概率

$$p(\bar{\mathbf{x}}) = p(\mathbf{x}_3)p(\mathbf{x}_2|\mathbf{x}_3)p(\mathbf{x}_1|\mathbf{x}_3\mathbf{x}_2)p(\mathbf{x}_0|\mathbf{x}_3\mathbf{x}_2\mathbf{x}_1)$$

有 NEE 的光追踪

与有 NEE 的路径追踪相似，只不过光追踪的路径是进入人眼的路径，因此对于光追踪而言，下一事件的语义指代的是进入人眼的那条路径。



$$p(\bar{\mathbf{x}}) = p(\mathbf{x}_3)p(\mathbf{x}_2|\mathbf{x}_3)p(\mathbf{x}_1|\mathbf{x}_3\mathbf{x}_2)p(\mathbf{x}_0)$$

当我们在 $\mathbf{x}_1$ 处引入 NEE 的时候，我们直接在人眼处采样一个新的顶点 $\mathbf{x}_0$。这个操作本身也独立于其它三个顶点的位置，因此，点 $\mathbf{x}_0$ 的概率就不是条件概率了。

12.5 重要性函数的数学问题 *

从这里开始，我们将更严格地看待重要性函数，并使用更严格的记号[3]：$L(\mathbf{x} \to \omega)$ 指的是从点 $\mathbf{x}$ 沿着方向 ω 离开的辐射亮度，而 $L(\mathbf{x} \leftarrow \omega)$ 指的是从方向 ω 到达点 $\mathbf{x}$ 的辐射亮度。对于重要性 W 也依此类推。

12.5.1 回顾定义

形象地理解重要性的定义：假设有一个集合 S，它包括这些点周围的点和方向，现在要计算离开点 S 的辐射通量 $\Phi(S)$。现在我们不从固定的 L_e 分布开始，而是考虑计算每一对 $(\mathbf{x},\omega)$ 对计算 $\Phi(S)$ 可能产生的影响。更准确地说，如果只有单一的辐射亮度值 $L(\mathbf{x} \to \omega)$ 被放置于 $(\mathbf{x},\omega)$ 上（形象地说就是一个照射在微分面积、微分立体角上的光源），并且如果没有其他照明光源存在，那么结果的 $\Phi(S)$ 会有多大？为了获得 $\Phi(S)$，我们需要给 $L(\mathbf{x} \to \omega)$ 分配一个权重，这个权重就被称为关于 S 的 $(\mathbf{x},\omega)$ 的重要性 $W(\mathbf{x} \leftarrow \omega)$。

下一步就是要计算出 W。直觉上来考虑，重要性肯定不应该与 $L(\mathbf{x} \to \omega)$ 有关，因为任何光源产生的辐射通量都会由于关于入射辐射亮度的 BRDF 的线性变化而产生线性的缩放。$W(\mathbf{x} \leftarrow \omega)$ 仅应该取决于场景的几何形状以及材质的反射特性。我们可以从以下两个方面来考虑 $W(\mathbf{x} \leftarrow \omega)$ 的表达式：

- *自身贡献*。如果 $(\mathbf{x},\omega) \in S$，那么 $L(\mathbf{x} \to \omega)$ 将完全贡献给 $\Phi(S)$。这也被称作集合 S 的**自身重要性**（self-importance），并被记作 $W_e(\mathbf{x} \leftarrow \omega)$。

$$W_e(\mathbf{x} \leftarrow \omega) = \begin{cases} 1 & \text{如果}(\mathbf{x},\omega) \in S \\ 0 & \text{如果}(\mathbf{x},\omega) \notin S \end{cases}$$

- *通过一个或多个反射做出的贡献*。我们也必须考虑所有的间接贡献，因为有可能 $L(\mathbf{x} \to \omega)$ 的一部分通过在某些表面的一次或多次反射后为 $\Phi(S)$ 做出贡献。我们知道，$L(\mathbf{x} \to \omega)$ 会沿着直线路径行进到达某个点 $\mathbf{p}= r(\mathbf{x},\omega)$，然后在该点进行反射，根据 BRDF 确定半球分布。我们在 $\mathbf{p}$ 点会有分布于半球的所有方向，每个方向会因为反射发射出一个微分辐射亮度值。将这里所有方向的重要性积分，我们就会有 W 的另一项。

通过以上两个部分的贡献，我们也就可以写出从 ω 流向 $\mathbf{x}$ 的重要性方程：

$$W(\mathbf{x} \leftarrow \omega) = W_e(\mathbf{x} \leftarrow \omega) + \int_{\mathcal{H}^2} f_r(\mathbf{p},\omega_i,\omega) W(\mathbf{p} \leftarrow \omega_i) \cos\theta d\omega_i. \tag{12.5}$$

12.5.2 出入射重要性

结合上文，当重要性是从方向 ω 流向 $\mathbf{x}$ 时，光能实际上是从 $\mathbf{x}$ 朝着 ω 方向向外发射的。而事实上，回顾出射辐射亮度的光传输方程，

$$L(\mathbf{x} \to \omega_o) = L_e(\mathbf{x} \to \omega_o) + \int_{\mathcal{H}^2} f_r(\mathbf{x},\omega_i,\omega_o) L(\mathbf{x} \leftarrow \omega_i) \cos\theta d\omega_i \tag{12.6}$$

其确实具有和方程（12.3），即**入射重要性**（incident importance）类似的形式。因此，我们可以进一步引入**出射重要性**（exitant importance），使我们的类比更完善：

$$W(\mathbf{x} \to \omega) = W(r(\mathbf{x},\omega) \leftarrow -\omega).$$

[3]12.5 以及 12.6 节的内容大部分摘自 Philip Dutre 等人所编著的第二版 Advanced Global Illumination。其中的一些内容并不常见于大部分图形学文献。

其中，回顾一下，$r(\mathbf{x},\omega)$ 是**投射函数**（raycasting function），它是指沿着方向 ω，从点 $\mathbf{x}$ 发出一根射线，返回最近的可见物体上的点。另外，在第五章中，我们曾经通过辐射亮度沿直线不变的属性得出过下式：

$$L(\mathbf{x} \leftarrow \omega) = L(r(\mathbf{x},\omega) \rightarrow -\omega).$$

因此，我们使得辐射亮度在真空中沿直线不变的性质进一步拓展到重要性上。利用这个性质，我们可以很轻松地得出，

$$W(\mathbf{x} \rightarrow \omega) = W_e(\mathbf{x} \rightarrow \omega) + \int_{\mathcal{H}^2} f_r(\mathbf{x},\omega_i,\omega) W(\mathbf{x} \leftarrow \omega_i) \cos\theta d\omega_i. \tag{12.7}$$

这就是我们之前提及的方程 (12.2)。

再次强调，重要性的出入射性质和辐射强度的出入射性质是相反的。入射重要性代表别的表面使得这个表面更加重要，这对应的是当前表面的光经过散射抵达了别的表面。出射重要性则代表当前表面把自己受到关心的程度转移给了更应该受到关注的表面，也就意味着当前表面的光实际上是从别的表面来的。因此，入射重要性对应的是出射辐射亮度，而出射重要性对应的是入射辐射亮度。

考虑到这些，我们也许会觉得 $W_e(\mathbf{x} \rightarrow \omega)$，也就是自身出射重要性，会很违背直觉，就如同自身入射辐射亮度一般。我们可以通过引入一个光感设备（例如人眼）来形象理解这个问题。人眼对光能在场景中的传播没有影响，但是可以侦测到我们所定义的 S 的光通量。此时，自身出射重要性就可以被理解成从这个假想人眼流出的重要性。

12.5.3 通量

光源是场景中唯一提供光能的物品，因此在计算场景中的通量的时候我们也只需要考虑这些光源。给定一个点和方向的几何 S，通量可以通过以下的方程计算：

$$\Phi(S) = \int_{\mathcal{A}} \int_{\mathcal{H}^2} L_e(\mathbf{x} \rightarrow \omega) W(\mathbf{x} \leftarrow \omega) \cos\theta d\omega d\mathbf{x} \tag{12.8}$$

$$= \int_{\mathcal{A}} \int_{\mathcal{H}^2} L_e(\mathbf{x} \leftarrow \omega) W(\mathbf{x} \rightarrow \omega) \cos\theta d\omega d\mathbf{x} \tag{12.9}$$

$$= \int_{\mathcal{A}} \int_{\mathcal{H}^2} L(\mathbf{x} \rightarrow \omega) W_e(\mathbf{x} \leftarrow \omega) \cos\theta d\omega d\mathbf{x} \tag{12.10}$$

$$= \int_{\mathcal{A}} \int_{\mathcal{H}^2} L(\mathbf{x} \leftarrow \omega) W_e(\mathbf{x} \rightarrow \omega) \cos\theta d\omega d\mathbf{x} \tag{12.11}$$

我们可以对表面 $\mathcal{A}$ 上的所有点进行积分，因为在不发光（不处于光源上）的点而言，L_e 会被定义为 0。使用上述的方程和重要性方程，我们也可以找到一种求解全局光照的算法。

现在我们有两种不同的求解全局光照的思路了。

- 求解 S 上的点和方向的辐射亮度。通过求解 (描述辐射亮度的) 光传输方程来计算全局光照。在这个思路下，我们从集合 S 出发，通过递归使光路径向着光源前进。这种思路对应着路径追踪。

- 求解每个光源对 S 的重要性。通过递归地求解重要性方程计算场景的通量，我们从光源出发，通过递归使光路径向着 S 前进。这种思路对应着光追踪。

12.5.4 线性传输算子

使用**线性传输算子**（linear transport operators）可以更简洁地描述辐射亮度和重要性传输的递归积分方程。事实上，渲染方程的反射部分本身可以被视作一个算子，其将一个特定沿着表面点和方向的辐射亮度分布转换至另一个给出了反射辐射亮度值的另一个分布。而这另一个分布本身又是在整个场景中定义的辐射亮度值的函数。我们用 $\mathcal{T}$ 来定义这种运算：

$$\mathcal{T}L(\mathbf{x} \to \omega_o) = \int_{\mathcal{H}^2} f_r(\mathbf{x}, \omega_i, \omega_o) L(r(\mathbf{x}, \omega_o) \to -\omega_i) \cos\theta d\omega_i \tag{12.12}$$

函数 $\mathcal{T}L$ 也是一个定义在 $\mathcal{A} \times \mathcal{H}^2$ 上的函数。通过这个算子，我们可以简单地将渲染方程写作：

$$L(\mathbf{x} \to \omega) = L_e(\mathbf{x} \to \omega) + \mathcal{T}L(\mathbf{x} \to \omega).$$

通过类似的方式，我们也可以使用传输算子来描述入射重要性 W_i 的传输方程，我们用 $\mathcal{Q}$ 来表示重要性的传输算子：

$$\mathcal{Q}W(\mathbf{x} \leftarrow \omega_o) = \int_{\mathcal{H}^2} f_r(r(\mathbf{x}, \omega_o), \omega_i, -\omega_o) W(r(\mathbf{x}, \omega_o) \leftarrow \omega_i) \cos\theta d\omega_i \tag{12.13}$$

通过这个算子，我们可以简单地将重要性方程写作：

$$W(\mathbf{x} \leftarrow \omega) = W_e(\mathbf{x} \leftarrow \omega) + \mathcal{Q}W(\mathbf{x} \leftarrow \omega).$$

由于 L_i 和 W_o，L_o 和 W_i 分别具有相同的传输方程描述，因此，我们一共有四个传输方程可以用来描述三维环境中的辐射亮度和重要性传输：

$$\begin{aligned} L_o &= (L_e)_o + \mathcal{T}L_o & \qquad W_i &= (W_i)_e + \mathcal{Q}W_i \\ L_i &= (L_e)_i + \mathcal{Q}L_i & \qquad W_o &= (W_o)_e + \mathcal{T}W_o \end{aligned}$$

其中，$(L_e)_o$ 是自发光出射辐射亮度，$(L_e)_i$ 是自发光入射辐射亮度。$(W_e)_o$ 是自发光出射重要性，$(W_e)_i$ 是自发光入射重要性。

12.5.5 伴随算子

首先，对于在五维定义域 $\mathcal{A} \times \mathcal{H}^2$ 上定义的函数空间中，我们可以先定义出射函数 F 和入射函数 G 的**内积**（inner product）：

$$\langle F, G \rangle = \int_{\mathcal{A}} \int_{\mathcal{H}^2} FG \cos\theta d\omega d\mathbf{x}$$

这样，我们就可以通过内积的方式简写方程 (12.8)∼(12.11)。

$$\begin{aligned} \Phi(S) &= \langle L_o, (W_e)_i \rangle \\ &= \langle L_i, (W_e)_o \rangle \\ &= \langle (L_e)_o, W_i \rangle \\ &= \langle (L_e)_i, W_o \rangle \end{aligned}$$

另外，数学上，如果两个对相同向量空间 V 进行运算的算子 $\mathcal{O}_1$ 和 $\mathcal{O}_2$ 关于定义在该空间上的函数 F 和 G 的内积 $\langle F, G \rangle$ 满足

$$\forall F, G \in V, \langle \mathcal{O}_1 F, G \rangle = \langle F, \mathcal{O}_2 G \rangle$$

则称它们互为**伴随算子**（adjoint operators）。在这种情况下，我们也可以将 $\mathcal{O}_2$ 写作 $\mathcal{O}_1^*$。

根据这个定义，我们可以证明在 12.5.4 中提到的 $\mathcal{T}$ 和 $\mathcal{Q}$ 就互为伴随算子，换言之，$\mathcal{Q} = \mathcal{T}^*$。这样，我们就有

$$W_i = (W_i)_e + \mathcal{T}^* W_i$$
$$L_i = (L_e)_i + \mathcal{T}^* L_i$$

这样，我们就可以将使用出射辐射亮度和入射重要性的通量表达式，通过内积属性和传输算子的伴随性来相互转换。

$$\begin{aligned}\Phi(S) &= \langle L_o, (W_e)_i \rangle \\ &= \langle L_o, W_i - \mathcal{T}^* W_i \rangle \\ &= \langle L_o, W_i \rangle - \langle L_o, \mathcal{T}^* W_i \rangle \\ &= \langle L_o, W_i \rangle - \langle \mathcal{T} L_o, W_i \rangle \\ &= \langle L_o - \mathcal{T} L_o, W_i \rangle \\ &= \langle (L_e)_i, W_o \rangle\end{aligned}$$

因此，我们现在有四个数学上等价的全局光照的解。

1. $\Phi(S) = \langle L_o, (W_e)_i \rangle \Leftrightarrow L_o = (L_e)_o + \mathcal{T} L_o$
2. $\Phi(S) = \langle (L_e)_o, W_i \rangle \Leftrightarrow W_i = (W_e)_i + \mathcal{T}^* W_i$
3. $\Phi(S) = \langle L_i, (W_e)_o \rangle \Leftrightarrow L_i = (L_e)_i + \mathcal{T}^* L_i$
4. $\Phi(S) = \langle (L_e)_i, W_o \rangle \Leftrightarrow W_o = (W_e)_o + \mathcal{T} W_o$

12.6 全局反射分布函数 *

根据 L_o 的传输方程，很明显场景中的每一个辐射亮度值都依赖于 $(L_e)_o$ 的初始分布。对于一个集合 S 的通量，这个依赖性则由重要性方程 W_i 给出。现在我们引入一个新的函数，这个函数也是一种分布函数，它描述了任意位置单一辐射亮度值 $L(\mathbf{x} \to \omega)$ 与初始 $(L_e)_o$ 分布的关系，这个函数就称作**全局反射分布函数**（Global Reflectance Distribution Function, GRDF）。

12.6.1 定义

GRDF 是一个四维函数，该函数描述两对 $\mathcal{A} \times \mathcal{H}^2$ 点-方向数对之间在三维场景的光传输，形如

$$G_r(\mathbf{x} \leftarrow \omega_o, \mathbf{y} \to \omega_i).$$

直观上，GRDF 描述了两个点-方向数对的某种全局传输—如果将其中一个点-方向数对 A 当做另一对 B 的某种微元光照来源，GRDF 也就是在描述 A 对 B 的这种贡献。这也就使得 GRDF 拥有如下的定义式：

$$L(\mathbf{y} \to \omega_i) = \int_{\mathcal{A}} \int_{\mathcal{H}^2} L_e(\mathbf{x} \to \omega_o) G_r(\mathbf{x} \leftarrow \omega_o, \mathbf{y} \to \omega_i) \cos\theta d\omega_i d\mathbf{x} \tag{12.14}$$

所以，GRDF 表示通过立体角 $d\omega_i$ 离开 $d\mathbf{x}$ 的总辐射功率，对在点 $\mathbf{y}$ 上测量沿着方向 ω_i 的辐射亮度值的影响。类似地，我们也可以得到重要性的 GRDF 关系：

$$W(\mathbf{x} \leftarrow \omega_o) = \int_{\mathcal{A}} \int_{\mathcal{H}^2} W_e(\mathbf{y} \leftarrow \omega_i) G_r(\mathbf{x} \leftarrow \omega_o, \mathbf{y} \rightarrow \omega_i) \cos\theta d\omega_i d\mathbf{x} \tag{12.15}$$

对 (12.14) 和 (12.15) 求微分，我们得到

$$G_r(\mathbf{x} \leftarrow \omega_o, \mathbf{y} \rightarrow \omega_i) = \frac{d^2 L(\mathbf{y} \rightarrow \omega_i)}{L_e(\mathbf{x} \rightarrow \omega_o) \cos\theta d\omega_i d\mathbf{x}}$$
$$G_r(\mathbf{x} \leftarrow \omega_o, \mathbf{y} \rightarrow \omega_i) = \frac{d^2 W(\mathbf{x} \leftarrow \omega_o)}{W_e(\mathbf{y} \leftarrow \omega_i) \cos\theta d\omega_i d\mathbf{x}}$$

12.6.2 GRDF 的性质

传输方程

通过传输算子，下面的两个传输方程都可以描述 GRDF 的行为。

$$G_r(\mathbf{x} \leftarrow \omega_o, y \rightarrow \omega_i) = \delta(\mathbf{x} \leftarrow \omega_o, \mathbf{y} \rightarrow \omega_i) + \mathcal{T} G_r(\mathbf{x} \leftarrow \omega_o, y \rightarrow \omega_i)$$
$$G_r(\mathbf{x} \leftarrow \omega_o, y \rightarrow \omega_i) = \delta(\mathbf{x} \leftarrow \omega_o, \mathbf{y} \rightarrow \omega_i) + \mathcal{T}^* G_r(\mathbf{x} \leftarrow \omega_o, y \rightarrow \omega_i)$$

$\delta(\mathbf{x} \leftarrow \omega_o, \mathbf{y} \rightarrow \omega_i)$ 是一个定义合适的狄拉克 δ 函数。使用 $\mathcal{T}$ 和 $\mathcal{T}^*$ 算子时，需要记住 $\mathcal{T}$ 只能运算 G_r 的出射部分，$\mathcal{T}^*$ 只能运算 G_r 的入射部分。

转换参数

G_r 的另一个实用特性是可以切换参数，其方式与 BRDF 的方向可以逆转的方式是大致相同的：

$$G_r(\mathbf{x} \leftarrow \omega_o, y \rightarrow \omega_i) = G_r(r(\mathbf{y}, \omega_i) \leftarrow \omega_i, r(\mathbf{x}, \omega_o) \rightarrow -\omega_o)$$

通量

使用 GRDF，我们也就可以给出集合 S 的通量的另一个表达式。

$$\Phi(S) = \int_{\mathcal{A}} \int_{\mathcal{H}^2_{\mathbf{x}}} \int_{\mathcal{A}} \int_{\mathcal{H}^2_{\mathbf{y}}} L_e(\mathbf{x} \rightarrow \omega_o) G_r(\mathbf{x} \leftarrow \omega_o, \mathbf{y} \rightarrow \omega_i) W_e(\mathbf{y} \leftarrow \omega_i) \cos\theta_i \cos\theta_o d\omega_i d\mathbf{y} d\omega_o d\mathbf{x} \tag{12.16}$$

注意到，这个表达式不再是一个递归式了。

12.6.3 GRDF 的用途

GRDF 允许我们以非常简短而优雅的方法来描述全局光照问题，并且这是独立于任何的自发光辐射亮度以及重要性的初始分布的。GRDF 仅取决于场景的几何形状和表面的反射特性，并没有假设光源的定位，也没有假设我们已经知道需要计算通量值的重要性来源被放置的位置。

由于公式（12.16）不再是递归式，假设我们知道 GRDF，则公式（12.16）就会变得很好计算。不过，在实践中这还是难以实现的，因为 GRDF 需要两个点-方向数对进行输入参数，如果想要为大量参数计算和存

储 GRDF，则所需的内存足迹会变得非常巨大。

一个基于 GRDF 的蒙特卡洛算法就是我们很快要介绍的双向路径追踪。理解下面的 12.7 其实并不需要 12.5 和 12.6 的复杂数学推导（甚至不需要知道 GRDF 的概念）。在下面的 12.7 和 12.8 中，我们会以更直观的角度来思考双向路径追踪算法本身。

12.7 双向路径追踪

路径追踪从表面点开始追踪，通过场景中的路径，最终在光源处结束，无论是否采用 NEE。而光追踪则是方向完全相反的另一种路径追踪：路径从光源开始，最终到达任何相关像素中结束。我们现在提出一个问题：有没有什么办法把这里的路径构造方式结合一下？解决这个问题的技术就被我们称作**双向路径追踪**（bidirectional path tracing, BDPT）技术。

双向路径追踪是由我们在附加小节 12.6 中提及的 GRDF 的公式开始的少数算法之一。核心思想就是同时从光源和人眼出发，开始构造子路径（subpath），然后将其相连，构成完整光路径。

12.7.1 算法步骤

双向路径追踪的算法可以总结为以下的步骤。

1. 构造人眼子路径。从人眼出发，生成一条随机路径，直到路径终止或者达到预定的最大长度。
2. 构造光源子路径。从光源出发，生成一条随机路径，直到路径终止或者达到预定的最大长度。
3. 子路径连接。将人眼子路径的每个顶点与光源子路径的每个顶点连接，形成完整的光路径。
4. 路径贡献评估。对于每个连接出的光路，计算其对最终图像的贡献。
5. 累积样本。重复以上步骤，生成多个人眼-相机子路径。并将它们的贡献累积到最终图像中。

其伪代码如下。

```
Color3f estimate (Vec3f px) {
    lp = sample light subpath
    cp = sample camera subpath for image point px

    for each vertex s in lp {
        for each vertex t in cp {
            fullpath = join(lp[0...s], cp[0...t])
            splat(fullpath.screenPos, fullpath.contribution)
        }
    }
}
```

12.7.2 算法分析

关于双向路径追踪，观察到一些关键点：

1. 对于每一条人眼子路径生成的完整光路径，如果这是一条 k 个节点的光路径，那么它可以通过使用 $k+1$ 个采样策略来构造。
    - 这里的策略指的是生成完整光路的方法，即如何分配顶点到人眼子路径和光源子路径。
    - 例如，对于一个 4 个顶点的光路，我们可以有以下的 5 种策略。
        - 策略 1：人眼子路径包含 0 个顶点，光源子路径包含 4 个顶点。
        - 策略 2：人眼子路径包含 1 个顶点，光源子路径包含 3 个顶点。
        - 策略 3：人眼子路径包含 2 个顶点，光源子路径包含 2 个顶点。
        - 策略 4：人眼子路径包含 3 个顶点，光源子路径包含 1 个顶点。
        - 策略 5：人眼子路径包含 4 个顶点，光源子路径包含 0 个顶点。
2. 对于一个特定的路径长度，所有的策略都在估计同一个积分。
    - 虽然不同的策略构建光路径的方式不同，但最终估计的还是同一个积分。
    - 这个性质也确保了双向路径追踪本身是无偏的，因为所有策略的期望值都等于真实的光传输积分。
3. 每一个策略都会有不同的 PDF，也就是说每一个策略都可能会有其优势和劣势。

这些关键点几乎就是在不停地对我们提示—多重重要性采样是非常适合双向路径追踪的采样方法。

12.8 支持双向路径追踪的 `Integrator` 类 *

回顾我们一开始提出双向路径追踪的初衷——最需要解决的一个问题是传统的路径追踪的健全性。我们经常使用的康奈尔方盒顶部有一个面积巨大的四边形光源—这使得其对传统的路径追踪非常友好。图 12.3 是使用我们之前写过的 `PathTracerMIS` 积分器完成的渲染。

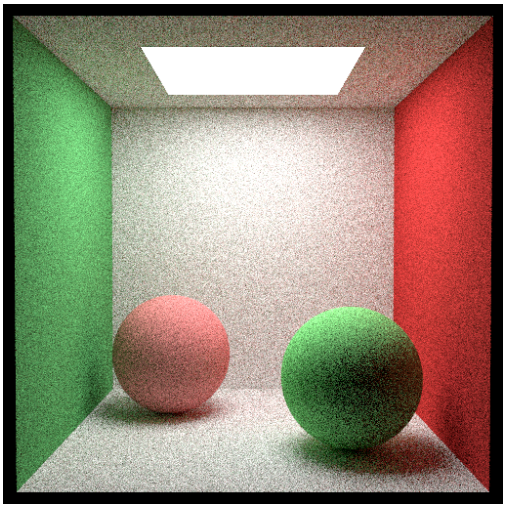


图 12.3: 路径追踪渲染的康奈尔方盒-大光源（128 次/像素采样）

之所以图中的设置对路径追踪很友好，是因为光源的面积很大，这也就使得路径追踪随机采样到的光路径是很容易碰到光源的。在图 12.3 中，光源的面积为 300×300。下面，我们仅将光源的大小调整为 10×10。再这样的设置下，光源面积很小，采样到光源的可能性已经降低很多，如同预期，路径追踪在其它配置完全

相同的情况下，如图 12.4，出现了大量的噪点。

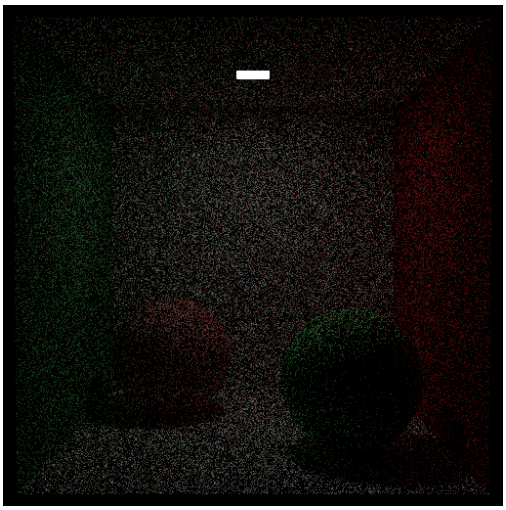


图 12.4: 路径追踪渲染的康奈尔方盒-小光源（128 次/像素采样）

之前我们解决这个噪点的方式是引入了下一事件估计（NEE），以及 NEE 和材质采样的双重重要性采样，这样的方式确实可以有效地减少噪点。现在我们使用双向路径追踪的思路。

12.8.1 BDPTIntegrator 类

回顾 12.7.1 的算法步骤，对于双向路径追踪类，与其它积分器类相同，它也需要一个最大深度以及评估函数 `Li()`。不同点在于，由于我们要使用显式构建的光路径，因此，我们不仅需要提供构造光路径的函数，也需要提供定义光路径节点的结构体 `Vertex`。

```
class BDPTIntegrator : public Integrator {
  public:
    BDPTIntegrator(const json &j = json::object());

    Color3f Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const override;

  private:
    struct Vertex {
      Vec3f position;
      Vec3f normal;
      Color3f throughput;
      float pdf;
      shared_ptr<const Material> material;
    };

    vector<Vertex> traceCameraPath(const Scene &scene, Sampler &sampler, const Ray3f &ray) const;
    vector<Vertex> traceLightPath(const Scene &scene, Sampler &sampler) const;

    Color3f connect(const Scene &scene, const Vertex &cameraVertex, const Vertex &lightVertex)
    const;

```

```
    int m_maxBounces = 64;
};
```

在这里，我们提供了 `Vertex` 结构体需要保存的数据。对于每个光路径节点，我们需要知道这个光路径节点的位置、法线、路径吞吐量以及被采样的概率密度函数。我们也记录被采样的材质，以便在之后的计算中使用。

同时，`BDPTIntegrator` 需要提供三个私有方法以实现 12.7.1 中的算法步骤。

- `traceLightPath()`：这个函数用以构建 `estimate[2]` 中的光源子路径采样。其返回一条光源子路径，也就是一个由 `Vertex` 组成的 C++ 标准库向量。
- `traceCameraPath()`：这个函数用以构建 `estimate[3]` 中的人眼子路径采样。其返回一条光源子路径，也就是一个由 `Vertex` 组成的 C++ 标准库向量。
- `connect()`：这个函数用以实现 `estimate[7-8]` 中的路径连接。

下面，开始逐一实现这里的每一个函数。

12.8.2 BDPTIntegrator::Li() 函数

这个函数应该与 12.7.1 中的 `estimate` 有完全相同的形式。所以这个函数的实现很简单。

```
Color3f BDPTIntegrator::Li(const Scene &scene, Sampler &sampler, const Ray3f &ray) const {
    Color3f L(0.f);

    // sample light path
    vector<Vertex> lightPath = traceLightPath(scene, sampler);
    // sample camera path
    vector<Vertex> cameraPath = traceCameraPath(scene, sampler, ray);

    // for each vertex s in light path
    for(int s = 0; s < lightPath.size(); ++s) {
        // for each vertex t in camera path
        for(int t = 0; t < cameraPath.size(); ++t) {
            if(s + t > m_maxBounces) break;
            // construct the full path
            L += connect(scene, lightPath[s], cameraPath[t]);
        }
    }

    return L;
}
```

12.8.3 采样光源子路径

12.8.4 采样相机子路径

12.8.5 构造完整光路径

Chapter 13

光子映射

在上一章的双向路径追踪中，我们解决了一个非常重要的问题—小光源（或者说无体积光源）无法被传统路径追踪有效采样的问题。通过从光源出发，我们能够使得光源对场景辐射亮度的重要性得以更多地体现在最终的图像中。然而，双向路径追踪依然也有其缺陷。依然是在焦散现象这样比较复杂的光路径场景中，即使双向路径追踪已经优于传统路径追踪很多，但是它依然不够健全。如下图所示[1]。通常来说，对于 LSDSE

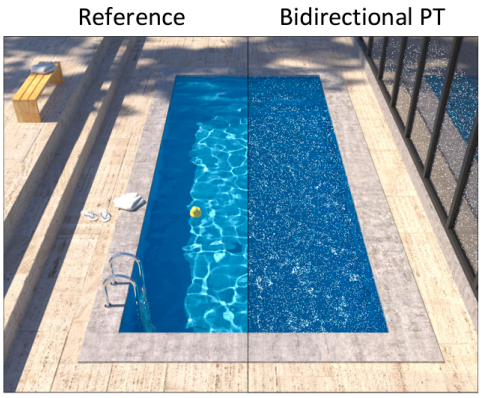


图 13.1: 双向路径追踪下渲染的泳池焦散图

路径，对于无偏方法很难做到一个高效而效果好的渲染。这就使得我们尝试在正确性上做出一些牺牲，以换来更好的渲染效果—这就是我们本章中要讨论的光子映射技术。

13.1 估计值的性质

在了解光子映射之前，我们先复习一下对于估计值无偏和一致性的定义。我们曾在第七章简单地介绍过这两个概念。

13.1.1 无偏性

首先回顾一下无偏性的概念。

[1]图片来自于 Jaroslav Křivánek

定义（无偏性）对于 $F = \int f(x)dx$ 的一个估计值 $\langle F \rangle$，如果其期望等于被估计的 F 的真实值，即

$$E[\langle F \rangle] = \int f(x)dx$$

则我们说 $\langle F \rangle$ 是**无偏**（unbiased）的。

对于无偏估计值，注意到它的误差仅来自于方差。因此，如果在渲染中我们使用的积分估计值是无偏的，那么我们在图中看到的噪点就会是方差的体现。

另外，注意到无偏估计值的一个性质：如果 $\langle F \rangle$ 是一个无偏估计值，则

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} \langle F^k \rangle = \int f(x)dx$$

也就是说，当我们对无数次估计（每次估计都有有限次数的采样）的结果取平均，那么它也能得到正确的结果。

对应无偏的概念，那自然也会有有偏（biased）的概念，因此，我们也应该要有偏差的定义。根据无偏的定义，偏差的定义应该非常符合直觉。

定义（偏差）估计值 $\langle F \rangle$ 的**偏差**（bias）被定义为估计值均值与实际值的差，即

$$\beta(\langle F \rangle) = E[\langle F \rangle] - \int f(x)dx$$

注意到，β 实际上也是有偏估计值与实际值之差的期望。如果我们对无数次估计的结果取平均，那么有偏估计值得到的平均结果应该是正确的结果加上该偏差，即

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} \langle F^k \rangle = \int f(x)dx + \beta$$

13.1.2 一致性

另一种估计值的特征被我们称作一致性。需要注意的是，一致性与无偏性并不矛盾，一个估计值可以是既无偏也一致的，也可能是一致但有偏的，也可能是无偏但不一致的，也有可能有偏也不一致。首先先让我们了解一致的定义。

定义（一致性）对于 $F = \int f(x)dx$ 的一个估计值 $\langle F \rangle$，如果随着采样次数增加，偏差会趋于 0，即

$$\lim_{N\to\infty} E[\langle F^k \rangle] = \int f(x)dx + 0$$

则我们说该估计值 $\langle F \rangle$ 是**一致**（consistent）的。

由无偏与一致性的定义，我们可以说一致的估计值和无偏的估计值在渐进意义上是等价的（asymptotically equivalent）。这个结论可以这么理解：

- 当样本量无穷大时，一致估计值会收敛到真实值，其偏差和方差都会减小到 0。同时，对于无偏估计量，由于其期望恒等于真实值，所以无穷多次重复采样的平均结果也必然等于真实值。因此，当 $N \to \infty$ 时，无论是一致估计值还是无偏估计值都能给出真实值。

- 要让估计误差减小到 0，无论是一致估计值还是无偏估计值都必须要求无穷多的样本。这是因为，虽然无偏估计值的期望等于真实值，但是方差并不会随着样本量增大而减小，所以即使是无偏估计值，也必须通过增加样本量来减少结果的不确定性，直到误差减小到可以忽略的程度。

需要注意的是，渐进意义的等价与一般意义等价并不是一个意思。在现实中，样本量只能是有限的，此时一致估计值和无偏估计值的表现可能会有所不同。对于无偏估计值，虽然期望是正确的，但是方差可能很大，导致样本小的时候单次估计结果不稳定，使得这样的结果很难用在需要保持多张渲染结果相对稳定的情形（例如动画）中。而对于一致估计值，即使其期望可能是有偏的，但是它能保持多张渲染结果“错得一致”，在连续渲染的情形中表现更为出色。总的来说，两种估计值各有优劣。

13.1.3 误差测量

既然偏差和方差都是会影响到渲染结果的，我们有理由使用一个单独的测量标准来表示**误差**（Error）。在这里我们介绍我们常用的两种误差测量。

定义（平均平方误差）对于一个估计值 $\langle F \rangle$，其**平均平方误差**（Mean Squared Error, MSE）是方差和偏差平方的和，即

$$\mathrm{MSE}[\langle F \rangle] = \mathrm{Var}[\langle F \rangle] + \mathrm{Bias}[\langle F \rangle]^2$$

定义（平均平方误差根）对于一个估计值 $\langle F \rangle$，其**平均平方误差根**（Root Mean Squared Error, RMSE）是平均平方误差的平方根，即

$$\mathrm{RMSE}[\langle F \rangle] = \sqrt{\mathrm{MSE}[\langle F \rangle]}$$

我们之所以要定义 RMSE，是因为 RMSE 拥有与被估计值相同的单位。另外，对于无偏估计值，RMSE 与标准差是相同的。

13.1.4 渲染技术的估计值属性

对于目前我们学过的以及即将要学习的一些技术，也可以按照无偏和一致进行一些分类。

典型的*无偏*方法包括：

- 传统的路径追踪
- 传统的光追踪
- 双向路径追踪

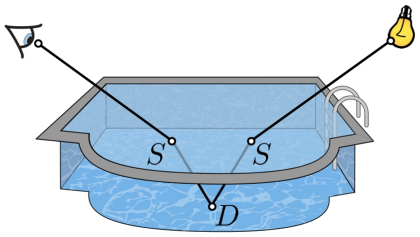
典型的*有偏但一致*的方法包括：

- **（渐进式的）光子映射**
- 多光源（Many-light）渲染方法

其中，光子映射就是我们今天要讨论的渲染技术。这是一个有偏但是一致的渲染技术，我们很快就会在算法分析中验证这件事。

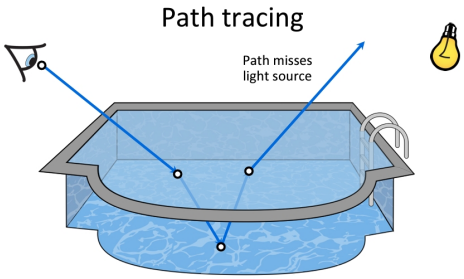
13.2 复杂光路径

在本章开头我们提到的泳池情境中，用上一章介绍过的 Heckbert 描述来说，这样的光路径就是 LSDSE 路径。在泳池中，光线从光源进入水面，折射至泳池底部发生漫反射至水面，再通过折射离开水面，这就是一个经典的 LSDSE 路径。我们来考虑一下使用无偏方法来渲染这条光路径可能会遇到什么样的问题。



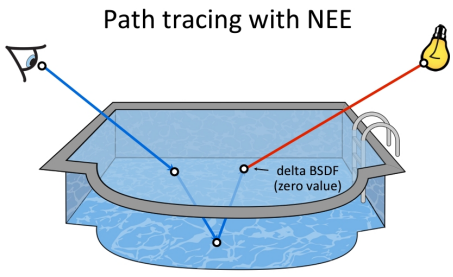
路径追踪

如果我们使用路径追踪，在完成 LSDS 的采样时，从水面离开时，可能会错过光源。在这种情况下，光源就不会被“看到”，因此这个路径贡献不到最终的光照中。对于包含镜面反射的路径，这个问题更加显著，因为镜面反射方向高度集中，导致正确的路径被采样到的概率非常低。这个问题我们在介绍双向路径追踪时已经说明过了。



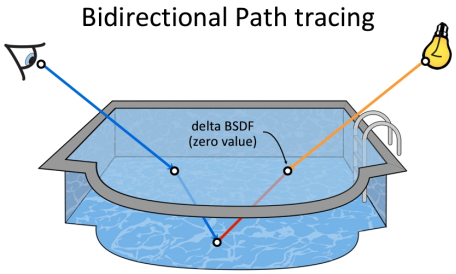
带有 NEE 的路径追踪

虽然 NEE 可以直接采样光源以改善路径追踪的问题，但是在处理 δ 分布的 BSDF 时，这个方法不起作用，因为在数学上，采样到指向光源的方向的概率几乎为 0。在这种情况下，即使使用了 NEE，路径也可能不会有贡献。这种情况被我们称为 δ−BSDF 的零值问题（zero-value）。

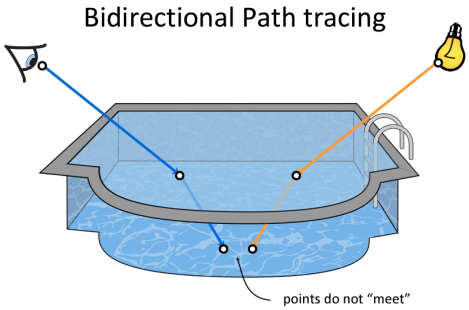


双向路径追踪

注意到，双向路径追踪实际上依然没法处理 $\delta-$BSDF 的零值问题。虽然在双向路径追踪中，我们能保证从光源出发的光源子路径经过光源（即使光源面积无限小），但是当光源子路径进入水面时，要使得光源子路径能与人眼子路径相连，依然只有一个无限小的方向才能建立连接。对于光源来说，采样到这个方向的概率也几乎为 0，因此，这条路径在现实中也不会存在。



在双向路径追踪中，绝大部分情况是光源子路径与人眼子路径无法相连，如下图所示。人眼子路径与光源子路径在池面上的采样点相距很近，但又没有近到可以被当做是同一个点来处理。



显然，我们需要一个方法来处理上述的情况。

13.3 后向光线追踪

我们先介绍在光子映射技术之前出现的一个提供了铺垫的技术—**后向光线追踪**（Backward Ray Tracing）。

后向光线追踪由 James Arvo 于 SIGGRAPH ’86 在 *Developments in Ray Tracing* 的授课笔记中提出。这个技术的核心思想为：从光源处发射光子，并将它们存储在照明图（illumination map）中。照明图是一种数据结构，它记录了场景中不同位置上累积的辐照度信息。我们可以将它理解为一种纹理，用来存储和累积场景中的辐照度。简单地说，它是场景中每一点被照亮情况的一种映射，通常用来存储和重用光照计算的结果。

13.3.1 算法描述

后向光线追踪可以分成以下的两个部分。

预处理

- 从光源处发射光线。
- 将光子能量信息存储在照明图中。

我们简单地回忆一下辐照度的定义–辐照度（irradiance）指的是每秒击中单位面积的光子数量。

着色

- 对于每个着色点（像素）：
  - 计算直接光照。
  - 从照明图中查询间接光照信息。

13.3.2 局限性

即使后向光线追踪是最早能模拟焦散效果这种复杂的 SDS 路径的技术之一，然而它需要能够参数化表面或是模型网格，非常难处理复杂的亦或是生成式的几何图形。我们也很难选择照明图的分辨率。如果选用高分辨率但少量光子的处理方式，就会产生高频噪点。如果采用低分辨率但光子数量够多的处理方式，也可能会产生模糊的光照。

13.4 光子映射概述

在后向光线追踪中，我们在本课程中首次看到了在渲染的过程中生成一张纹理图用以存储信息的技术。这个技术提供的空间换取时间的思路给光子映射的技术提供了很好的起步条件。**光子映射**（Photon Mapping）是一种有偏一致的渲染技术。它是一个拥有两个通道（Pass）流程的算法。

13.4.1 双通道算法

光子映射的双通道可以被概括为下面的流程：

第一通道：从光源开始追踪光子，并且将它们缓存在光子图（photon map）中。
第二通道：从人眼开始追踪，并利用光子图估计间接光照。

与后向光线追踪有相似之处，但是在存储光子和计算密度的方面有区别。下面我们就来详细介绍光子映射的两个通道。

13.5 第一通道

13.5.1 算法描述

第一个通道与后向光线追踪中的预处理步骤非常类似。可以简单地概括为以下三个步骤：

1. 发射光子。从所有的光源发射一定数量的光子。每个光子都携带着光源的能量和信息。

2. *散射光子*。每个光子沿着由光源的发散特性确定的路径穿过场景。在这个过程中，光子可能会被场景中的对象反射、折射或是吸收。

3. *存储光子*。交互后的光子信息（方向、位置、能量和颜色）会被存储到一个全局的数据结构中，这就是我们所说的光子图。光子图允许快速查找场景中的某一点附近的光子。

第一通道为场景中的全局光照效果提供了基础数据。这个过程通常需要大量的计算，

13.5.2 光子信息

在光子映射中，**光子**（Photon）携带的信息是通量而不是辐射亮度。需要注意的是，在这个技术中提到的光子并不是物理意义的光子。它代表了一定的通量，可以将其视作一个能量包。

每个发射的“光子”携带一小部分光源的总功率。在实际的实现中，每个光子也包含多种波长，这允许光子反映颜色的信息。

对于单个“光子”，我们可以定义以下的*初始信息*。

1. *位置* $\mathbf{x}_p$。光子的位置是在光源表面的某点随机采样得到的。

2. *方向* ω_p。光子的方向是基于光源发光特性的概率密度函数随机采样得到的。我们可以通过条件概率 $p(\omega_p|\mathbf{x}_p)$ 进行采样。

3. *光子功率* Φ_p。光子功率是根据发射的光子数量和光源总功率分配的。其可以通过

$$\Phi_p = \frac{1}{M}\frac{L_e(\mathbf{x}_p,\omega_p)\cos\theta_p}{p(\mathbf{x}_p)p(\omega_p|\mathbf{x}_p)}$$

计算得到，其中，M 是发射的光子的总数。

我们可以从这个定义中得到一些有趣的推导。如果采样位置与方向的 PDF 正比于光源的辐射发射，也就是理想状态下的重要性采样情况，那么，采样点 $\mathbf{x}_p$ 的概率密度就应该是光源辐照度与总通量的比值，这样就可以确保对面积积分时为 1，保证概率密度的归一化条件，即

$$\begin{aligned}
p(\mathbf{x}_p) &= \frac{\int_{\mathcal{H}}^2 L_e(\mathbf{x}_p,\omega)\cos\theta d\omega}{\int_{\mathcal{A}}\int_{\mathcal{H}^2} L_e(\mathbf{x},\omega)\cos\theta d\omega d\mathbf{x}} \\
p(\omega_p|\mathbf{x}_p) &= \frac{L_e(\mathbf{x}_p,\omega)\cos\theta}{\int_{\mathcal{H}^2} L_e(\mathbf{x},\omega)\cos\theta d\omega}
\end{aligned}$$

那么，将上述两个 PDF 代入 Φ_p，我们可以得到

$$\begin{aligned}
\Phi_p &= \frac{1}{M}\frac{L_e(\mathbf{x}_p,\omega_p)\cos\theta_p}{p(\mathbf{x}_p)p(\omega_p|\mathbf{x}_p)} \\
&= \frac{1}{M}\frac{L_e(\mathbf{x}_p,\omega_p)\cos\theta_p}{\frac{\int_{\mathcal{H}}^2 L_e(\mathbf{x}_p,\omega)\cos\theta d\omega}{\int_{\mathcal{A}}\int_{\mathcal{H}^2} L_e(\mathbf{x},\omega)\cos\theta d\omega d\mathbf{x}}\frac{L_e(\mathbf{x}_p,\omega)\cos\theta}{\int_{\mathcal{H}^2} L_e(\mathbf{x},\omega)\cos\theta d\omega}} \\
&= \frac{1}{M}\int_{\mathcal{A}}\int_{\mathcal{H}^2} L_e(\mathbf{x},\omega)\cos\theta d\omega d\mathbf{x} \\
&= \frac{\Phi}{M}
\end{aligned}$$

其中，Φ 是光源的总通量。也就是说，如果我们*完美地*对出射辐照度重要性采样，单个光子的光子功率就应该是总功率除以光子总数，这是非常符合直觉的。

13.5.3 生成光子图

当光子撞击非镜面表面时，它们存储在被称为**光子图** (Photon Map) 的全局数据结构中。为了便于对光子的有效搜索，我们通常会使用 KD 树来实现该数据结构。

生成光子图的伪代码如下。

```
void GeneratePhotonMap() {
    do {
        (l, prob_l) = ChooseRandomLight();
        (x, w, phi) = EmitPhotonFromLight(l);
        TracePhoton(x, w, phi / prob_l);
    } while (# of photons is not enough);

    Divide all photon powers by # of photons;
}
```

在这个过程中，我们需要一个类似于光线追踪的光子追踪流程，也就是伪代码中的 `TracePhoton()` 函数。本质上来说，光子追踪和光线追踪是有差异的，主要的不同在于，当光子经历折射时，光子携带的能量并不会发生改变。相反，光线的辐射亮度则必须使用相对折射率的平方进行加权处理。

`TracePhoton()` 函数的伪代码如下。

```
void TracePhoton(Vec3f x, Vec3f w, Vec3f phi) {
    (nextX, n) = NearestSurfaceHit(x, w);
    // Store photon information only on diffuse (or moderately glossy) surfaces
    StorePhotonWhenDiffuse(nextX, w, phi);
    (nextW, pdf) = sampleBSDF(nextX, -w);
    nextPhi = phi * absDot(n, nextW) * BSDF(nextX, -w, nextW) / pdf;
    tracePhoton(nextX, nextW, nextPhi);
}
```

目前的这个光子追踪算法显然还有很多问题，例如存储光子的方法不明确，以及没有终止条件。下面我们来解决这些问题。

13.5.4 Photon 结构

在光子映射中，通常只会在漫反射或是中等光泽的表面上存储光子。这是因为这些表面类型会在很宽的方向上散射光线，所以光子的存储和随后的重用能够有效地估计间接光照。

对于镜面表面，情况就截然不同，镜面表面会在非常特定的角度上反射光线，这导致光线几乎以相同的角度离开表面。由于这种高度定向的反射，光子映射算法中的常规密度估计方法在这些表面上表现不佳，因此，为了正确模拟镜面散射，通常会使用路径追踪从人眼出发。

那么，对于漫反射表面，我们知道存储单个光子需要的信息是位置、方向和功率。因此，我们可以定义如下的 `Photon` 数据结构。

```
// [36 Bytes] in total
struct Photon {
    Vec3f position; // 12 bytes
```

```
    Vec3f power; // 12 bytes
    Vec3f direction; // 12 bytes
};
```

显然，如果使用上述的数据结构，单个光子就需要 36B 的空间。对于一个存储一百万个光子的光子映射算法，一次渲染就需要使用掉 36MB 的空间。我们应该去寻找一些优化手段。

在下面的这个优化光子结构中，

```
// [36 Bytes] in total
struct Photon {
    Vec3f position; // 12 bytes
    char power[4]; // 4 bytes, packed RGBE format
    char phi, theta; // 2 bytes, packed direction
};
```

在位置部分我们没有什么优化，因为位置信息通常需要较高的精度。而功率部分我们则使用压缩的 RBGE 格式存储光子功率。RGBE 格式由一个三元素颜色向量和一个共享的指数构成，这允许在不丢失太多精度的情况下表示很宽范围的颜色值和亮度级别。方向通过存储两个角度值来表示，且因为方向都可以通过一个字符类型可以表达的数值表示，因此，也能够做出一些优化。

优化后的光子结构最少[2]可以达到 18B，省去了至多一半的空间，虽然有一些精度的下降，但是相比于两倍的性能提高，这种损失完全可以接受。

13.5.5 光子图数据结构

光子图需要满足的条件主要有两个：

- 足够紧凑。我们需要存储非常多的光子。
- 快速的最近邻查找。这里所说的最近邻指的就是空间上最近的 k 个相邻的光子。

显然，**KD 树**（K-Dimensional Tree）是一个满足以上两个条件的数据结构。我们曾在讨论包围盒层级结构时简单提及过 KD 树。KD 树是一种用于组织点在 k 维空间中的数据结构，它通过递归地将空间切分成两部分来存储数据。在光子映射中，KD 树应该存储用以检索光子位置的信息，以便快速执行最近邻搜索。

下面是一个非常基本的 C++ 中的光子图 KD 树节点类的实现。

```
/// KDNode implementation
class KDNode {
public:
    Photon photon;
    KDNode* left;
    KDNode* right;
    int axis;

    // KDNode constructor
    KDNode() : left(nullptr), right(nullptr), axis(0) {}

```

[2]在一些有对齐要求的设备上可能达不到这个最低值。

```
    // KDNode destructor
    ~KDNode() {
        delete left;
        delete right;
    }
};

/// Comparison functions for sorting the photons and use std::nth_element for sorting
bool cmpX(const Photon &a, const Photon &b) { return a.position.x < b.position.x; }
bool cmpY(const Photon &a, const Photon &b) { return a.position.y < b.position.y; }
bool cmpZ(const Photon &a, const Photon &b) { return a.position.z < b.position.z; }

/// Recursively constructing the KDTree
KDNode* BuildKDTree(std::vector<Photon>& photons, int depth = 0) {
    if(photons.empty()) return nullptr;

    // choose the axis
    int axis = depth % 3;
    int median = photons.size() / 2;
    if (axis == 0)
        std::nth_element(photons.begin(), photons.begin() + median, photons.end(), cmpX);
    else if (axis == 1)
        std::nth_element(photons.begin(), photons.begin() + median, photons.end(), cmpY);
    else if (axis == 2)
        std::nth_element(photons.begin(), photons.begin() + median, photons.end(), cmpZ);

    // create the node
    KDNode* node = new KDNode(photons[median], axis);

    // Recursively construct the child trees
    std::vector<Photon> leftPhotons (photons.begin(), photons.begin() + median);
    std:vector<Photon> rightPhotons(photons.begin() + median + 1, photons.end());

    node->left = BuildKDTree(leftPhotons, depth+1);
    node->right = BuildKDTree(rightPhotons, depth+1);

    return node;

}
```

需要注意的是，在上面的简单实现中，输入的光子向量 `photons` 会被修改，因为 C++ 标准库提供的函数 `std::nth_element` 函数会重排向量中的元素。实际的光子映射算法可能需要更复杂的数据结构和算法来优化搜索性能，例如支持 k 近邻搜索。

我们同样可以使用 C++ 标准库中提供的数据结构—优先队列（Priority Queue）来进行 k 近邻搜索，以下是一个简单的实现框架。

```
// Tructure for saving the nearest neighbor
struct KDNodePtr {
    KDNode* node;
    float squaredDistance; // to avoid square root calculation

```

```
    KDNodePtr(KDNode* node, float dist)
        : node(node), squaredDistance(dist) {}

    // Pop the farthest point first
    bool operator<(const KDNodePtr& other) const {
        return squaredDistance < other.squaredDistance;
    }
};

// a PQ for k nearest neighbors
typedef std::priority_queue<KDNodePtr> KDNodePtrQueue;

// calculate the squared distance between 2 points
inline float squaredDistance(const Vec3f& a, const Vec3f& b) {
    Vec3f diff = a - b;
    return dot(diff, diff);
}

// Recursively search the KDTree for the k nearest neighbors
void SearchKDTree(KDNode* node, const Vec3f& target, KDNodePtrQueue& kNearestNeighbors, int k) {
    if (node == nullptr) return;

    float dist = squaredDistance(target, node->photon.position);

    // if we have space or current point is further than the current furthest point
    if (kNearestNeighbors.size() < k || dist < kNearestNeighbors.top().squaredDistance) {
        // pop the furthest if the queue is full
        if (kNearestNeighbors.size() == k) kNearestNeighbors.pop();

        // push the current node to the PQ
        kNearestNeighbors.push(KDNodePtr(node, dist));
    }

    // determine which children to search
    float splitPlane = target[node->axis] - node->photon.position[node->axis];
    KDNode* nearChild = splitPlane < 0 ? node->left : node->right;
    KDNode* farChild = splitPlane < 0 ? node->right : node->left;

    // first search the closer child tree
    SearchKDTree(nearChild, target, kNearestNeighbors, k);

    // if the other child tree has a closer point, search as well
    if (kNearestNeighbors.size() < k || std::pow(splitPlane, 2) < kNearestNeighbors.top().
    squaredDistance) {
        SearchKDTree(farChild, target, kNearestNeighbors, k);
    }
}

KDNodePtrQueue FindKNearestNeighbors(KDNode* root, const Vec3f& target, int k) {
    KDNodePtrQueue kNearestNeighbors;
    SearchKDTree(root, target, kNearestNeighbors, k);
    return kNearestNeighbors;
```

```
}

// use the above function to find k nearest neighbors.
// Vec3f target = ...; // query point
// int k = ...;
// KDNodePtrQueue nearestNeighbors = FindKNearestNeighbors(root, target, k);
```

上面的代码还是有可优化的空间，我们并没有在这段代码中处理特殊的情况，例如 KD 树节点的选择和分割轴的处理。实际的项目中，还可以更进一步优化，推荐阅读 H.W.Jensen 的著作 *Realistic Image Synthesis Using Photon Mapping*。

13.5.6 光子散射

我们知道我们要进行光子追踪的原因在于光子在场景中是会被吸收或者被散射的。BSDF 可以决定其被反射还是被散射，光子功率的下降也是为了模拟被吸收的现象。

目前也还存在一些问题：

- 随着光子的弹射，它携带的功率会越来越少。
- 理想状态下，所有的光子都会有着相同的功率。
- 光子追踪的递归无法终止。

解决递归终止的问题可以通过简单地加入俄罗斯轮盘赌即可。

```
void TracePhoton(Vec3f x, Vec3f w, Vec3f phi) {
    (nextX, n) = NearestSurfaceHit(x, w);
    // Store photon information only on diffuse (or moderately glossy) surfaces
    StorePhotonWhenDiffuse(nextX, w, phi);
    (nextW, pdf) = sampleBSDF(nextX, -w);
    nextPhi = phi * absDot(n, nextW) * BSDF(nextX, -w, nextW) / pdf;
    if(SurvivedRR(phi, nextPhi))
        tracePhoton(nextX, nextW, nextPhi);
}
```

对于俄罗斯轮盘赌而言，

$$E[F'] = (1-p)\cdot 0 + p\cdot \frac{E[F]}{p} = E[F]$$

我们需要选择一个合适的存活几率 p。应该如何选择俄罗斯轮盘赌的存活几率呢?

方法 1：局部的终止几率

局部的终止几率方法下，

$$p = \min\left(1, \frac{\Phi'}{\Phi}\right)$$

其中的 Φ' 就是我们伪代码中的 `nextPhi`。

方法 2：根据历史选择的终止几率

在这个方法下，调整每个光子的存活概率，以确保即使在经过多次反射和可能的能量损失之后，保持每个光子携带大致相同的功率。具体地，当一个光子的能量在每次弹射下降后，响应地增加它的权重，以便在统计意义上保持整体能量的不变。

无论选用哪种方法，俄罗斯轮盘赌本身的伪代码与之前蒙特卡洛路径追踪中的俄罗斯轮盘赌并没有什么大的区别。

```
bool SurvivedRR(Phi, nextPhi) {
    survival rate p;
    if(randf() > p) return false; // terminate
    else {
        nextPhi /= p;
        return true;
    }
}
```

13.6 第二通道

在第一通道中，我们已经做好了预处理的步骤—光子图已经被生成了。在第二通道中，我们的目标就是要从人眼出发，然后利用我们预制的光子图去估计间接光照。

13.6.1 算法描述

第二通道的步骤可以简单地概括为：

- 对于每一个着色像素：
  - 找到 k 个最近的光子。对于图像中的每一个着色点，算法会在光子图中查找距离该点最近的 k 个光子。这通常通过 KD 树来高效地完成。
  - 利用光子的密度估计辐射亮度值。基本思想是，如果许多光子聚集在一小块区域内，那么这个区域很可能接收到较强的光照。光子的功率和颜色会被累加起来，并除以区域的大小，以估算这个点的辐射亮度。

13.6.2 内核密度估计

根据上面的算法描述，辐射亮度是根据光子密度来估算的。这个密度也被我们称为**内核密度**（kernel density）。

内核密度估计（Kernal Density Estimation, KDE）是一种用于估计概率密度函数的非参数方法。在光子映射的上下文中，它被用来估计场景中某一点的辐射亮度值 L_r。辐射亮度 $L_r(\mathbf{x},\omega$ 是场景中的点 $\mathbf{x}$ 沿着方向 ω 接收到的光的度量，其值可以通过

$$L_r(\mathbf{x},\omega) = \int_{\mathcal{H}^2} f_r(\mathbf{x},\omega,\omega')L_i(\mathbf{x},\omega')\cos\theta' d\omega'$$

计算。我们简单地整理一下上式：

$$\begin{aligned} L_r(\mathbf{x},\omega) &= \int_{\mathcal{H}^2} f_r(\mathbf{x},\omega,\omega') L_i(\mathbf{x},\omega') \cos\theta' d\omega' \\ &= \int_{\mathcal{H}^2} f_r(\mathbf{x},\omega,\omega') \frac{d\Phi^2(\mathbf{x},\omega')}{\cos\theta' d\omega' dA} \cos\theta' d\omega' \\ &= \int_{\mathcal{H}^2} f_r(\mathbf{x},\omega,\omega') \frac{d\Phi^2(\mathbf{x},\omega')}{dA} \\ &\approx \sum_{p=1}^{n} f_r(\mathbf{x},\omega_p,\omega) \frac{\Delta\Phi_p(\mathbf{x},\omega_p)}{\Delta A} \end{aligned}$$

方法 1：最近的 k 个光子

在 13.6.1 中提到的第一步就是寻找所求点最近的 k 个光子，然后通过这 k 个光子定义一个包围它们的面积，然后计算 $\Delta\Phi_p/\Delta A$。

用 Garcia 在 2012 年提出的方法，我们可以无视 k 个光子中最远的那个（假设它就是第 k 个光子），然后利用与它的距离确定一个圆盘，用该圆盘的面积作为这 k 个光子的总覆盖面积，也就是说

$$L_r(\mathbf{x},\omega) \approx \sum_{p=1}^{k-1} f_r(\mathbf{x},\omega_p,\omega) \frac{\Phi_p}{\pi r_k^2}$$

方法 2：确定面积数光子

我们当然也可以换一个思路。找到所求点附近的一个固定面积，然后数里面光子的数量，这个方法也可以用来估计光子密度。假设这个固定面积是一个半径为 r 的圆盘，并且假设这个圆盘里面有 k 个光子，那么很符合直觉地，我们应该有

$$L_r(\mathbf{x},\omega) \approx \sum_{p=1}^{k} f_r(\mathbf{x},\omega_p,\omega) \frac{\Phi_p}{\pi r^2}$$

13.6.3 误差分析与改进

在辐射亮度估计的步骤中，会存在误差。

- 除非使用的光子数量非常庞大，否则这种算法会生成模糊的图像，导致图片质量不佳。
- 有时也可能会产生更亮或是更暗的区域。

当光子映射与光线追踪方法结合在一起的时候，光子图才会更加有效。我们可以将光照的计算分成几个部分。

- 直接光照。这一部分交给光线追踪，从人眼开始追踪，通过场景的一次漫反射或者光泽反射。
- 焦散光子图（caustic photon map）。由于焦散仅发生在场景中的一小部分，因此它们以更高质量的分辨率进行计算，允许直接高质量显示。
- 剩余的间接照明。使用全局光子图（global photon map）进行计算。

焦散光子图

由于焦散仅发生在场景中的一小部分，因此我们可以单独为焦散渲染高质量的焦散光子图，其包括遍历路径 LS^+D 的光子。与之对应地，全局光子图则包含所有的路径，即 $L(S|D)^*D$。

焦散光子图之所以能够有效地计算，是焦散是因为光的聚焦而产生的—这意味着在这些区域，即使是少量的光子也可以形成清晰的焦散图案。因此，每个光子携带的信息在视觉上会在焦散区域更加集中和明显，所以需要的光子数目比模拟整个场景的间接光照时需要的光子要更少。

另外，在典型场景中，导致焦散的表面（如水面、玻璃等）的数量通常都相对比较少，因此，可以针对性地仅向这一小部分的镜面表面发射光子，而不必处理场景中的每一个表面。因为这些表面能够有效地聚焦光线，光子在视觉上也就可以因为聚焦而产生明显的焦散效果。这种选择性发射光子的策略减少了需要计算和存储的总光子数，提高了效率。

全局光子图

原来的光子映射会沿着光子路径计算直接密度估计。这涉及到在全局光子图中查找给定点周围的光子，并基于这些光子来估计所求点的照明情况。

这里我们介绍**最终收集**（Final Gathering）方法来改进光子映射的过程。最终收集法中，我们会从相机出发进行路径追踪，直到我们抵达第一个非镜面表面上的点 $\mathbf{x}$。然后，在该点执行以下步骤。

- *自发光*。计算该点的自发光。
- *直接光照*。使用阴影光线计算该点的直接照明。这与我们在第九章中提到的直接光照相同。
- *焦散效果*。仅使用光子图中的焦散光子（即那些打在高光表面并被强烈偏折的光子）来估计在点 $\mathbf{x}$ 的焦散。
- *剩余的间接光照*。继续进行路径追踪直到下一个非镜面顶点 $\mathbf{y}$，并在该处使用全局光子图进行密度估计。

通过这个方式，最终收集法结合了传统路径追踪中精确处理直接光照的优势，以及使用焦散光子图估计焦散、全局光子图估计更复杂的间接光照的优势。

13.7 优劣势总结

最后我们总结一下目前为止光子映射的优势和劣势。

优势

光子映射的优势包括：

- *能够比无偏算法更健全地处理复杂光路径*。光子映射在处理诸如焦散和某些类型的阴影等复杂的光线路径时，比无偏差算法更加稳健，因为它通过在场景中预存储光子的路径信息来处理这些复杂的光效。
- *拥有一个一致的估计值*。在统计上，光子映射提供了一个一致的估计器，这意味着随着光子数量的增加，估计的准确性会逐渐提高，趋于真实解。

- 保存了光子图，能够复用计算结果。一旦建立了光子图，就可以用于多次渲染，这对于动画序列或交互式场景浏览尤其有用，因为不需要在每一帧都重新计算整个光线传播过程。

劣势

光子映射的劣势在于：

- 偏差有着不同形式的显现。例如光子的不均匀分布导致的亮斑、估计的平滑度不足等问题。
- 需要一个额外的较为复杂的数据结构用以保存光子图。光子映射需要构建和维护一个光子图，通常是 KD 树这样的数据结构，这需要额外的计算和内存。
- 没有渐进式的渲染。光子映射通常不支持渐进式渲染（progressive rendering）。这是一种渲染技术，它允许图像逐步提升其质量，直到达到所需的细节水平或者直到渲染过程被停止。渐进式渲染的一个典型例子是路径追踪。在路径追踪中，初始的几次迭代可能会展示一个非常嘈杂的图像，但随着计算的继续，图像会变得越来越清晰。相比之下，光子映射通常不是渐进式的。在光子映射算法中，首先需要建立光子图，这个过程涉及将光子发射到场景中并记录它们的路径和相遇表面的信息。一旦光子图构建完成，就使用它来估计光照。这种方法不能立即提供反馈，因为必须先完成光子图的构建。此外，图像的质量并不是随着时间线性提高的，而是在光子图被认为足够好用于渲染之后，最终的图像才开始生成。这意味着整个图像的渲染需要在最终查看之前完成，而不能逐渐提高其质量。
- 有相当大的内存占用。由于光子图的存储需求，光子映射可能会占用大量内存，尤其是在处理复杂场景或高分辨率渲染时。
- 超参数调整不直观。光子映射的性能和质量可能对算法的各种超参数非常敏感，比如光子图中的光子数量、搜索半径等。找到最优的超参数设置可能需要大量的实验和经验，这对初学者来说并不直观。

13.8 改进：渐进式光子映射 *

在上面分析光子映射的劣势时，我们提到光子映射并不是渐进式的。2011 年，Knaus 和 Zwicker 提出**渐进式光子映射**（Progressive Photon Mapping, PPM）的想法。核心的思想就是：渲染出多张独立的、光子查找半径越来越小的图，然后求平均。

13.8.1 一致性分析

首先，在之前的光子映射中，我们知道它的估计值是一致的。这意味着它的结果最终会收敛于正确的结果。收敛自然是有条件的：

1. 光子的查找半径需要是无穷小。
2. 查询无穷多的最近邻光子。

因此，这实际上是不可能达到的结果，因为光是存储无穷多的光子数据就需要无穷大的存储容量。

13.8.2 概述

与传统的光子映射相比，PPM 顾名思义，提供了一种渐进式的方法，可以在每个迭代步骤中逐渐收敛到正确的解。渐进式光子映射的大致流程如下：

1. 发射光子。与传统光子映射相同，PPM 从光源处发射光子，并将它们存储在全局光子图中。这些光子会在场景中的表面之间弹跳，直至它们被吸收或者达到预设的弹跳次数上限。
2. 查找估计。算法会在场景中的点上收集光子，并估计这些点的光照情况，与传统光子映射中的密度估计相同。
3. 渐进式细化。不同于传统的光子映射一次性完成渲染，PPM 通过多个迭代的过程来逐步改善这些估计。在每一个迭代中，会发射新的光子，并更新光照估计。每一轮迭代都在前一轮的基础上提升渲染质量。
4. 光照传输。PPM 会执行光照传输的步骤，通常涉及路径追踪。
5. 结果输出。随着迭代次数增加，渲染质量逐渐提升，最终输出高质量的图像。

13.8.3 半径缩减

我们重点讨论 PPM 的第三步—渐进式细化。渐进式细化的具体流程大致如下：

1. 第一步：
   - 光子追踪。发射、散射、存储光子。
2. 第二步：
   - 追踪人眼路径。
   - 使用半径 r_i 估计辐射亮度。
3. 第三步：
   - 运行平均。在每个迭代后，使用当前和之前迭代得到的结果计算运行平均值（running average），以稳步提升渲染质量。
4. 第四步：
   - 重复迭代。计算并调整下一轮迭代的估计范围半径。
   - 半径的收缩公式为
   $$r_{i+1}^2 = \frac{i+\alpha}{i+1} r_i^2$$
   其中，$\alpha \in (0,1)$ 是控制半径缩减的参数。

13.8.4 渐进式光子映射的优势

相比与传统光子映射，PPM 的优势主要体现在以下几点。

1. 减少内存占用。PPM 算法通过渐进式地处理光子，减少了传统光子映射算法所需要的大量内存，它不需要无限的内存来收敛至无偏估计。

2. 渐进式渲染。提供了一个对用户更友好的渲染方式，能够渐进地显示渲染结果。

3. 简化数据结构。与传统光子映射需要复杂数据结构（KD 树）来维护光子数据不同，PPM 简化了这个需求。

4. 单一光子图。PPM 不再需要焦散光子图，用一张光子图就可以处理所有的情况了。

13.9 支持光追踪的 `Integrator` 类 *

Chapter 14

马尔科夫链蒙特卡洛渲染

在目前我们的随机路径追踪中，每一次采样都是基于前一次采样已经完成的条件概率事件。**马尔科夫链**（Markov Chain）是另一种随机系统，在链中的下一个状态的概率仅依赖于当前的状态，而与以前的状态无关。使用这个技术的随机渲染被称为**马尔科夫链蒙特卡洛渲染** (Markov Chain Monte Carlo Rendering, MCMC Rendering)。

14.1 马尔科夫链蒙特卡洛方法

马尔科夫链蒙特卡洛方法是一种统计学方法，它用于从复杂的概率分布中生成样本。

14.1.1 马尔科夫链

马尔科夫链是一种随机过程，它的最大的特点是**无记忆性** (memorylessness)：下一状态的概率分布只能由当前状态决定，在时间序列中在它前面的事件均与之无关。这种性质也叫做**马尔科夫性质**（Markov property）。

定义（马尔科夫链）**马尔科夫链**（Markov Chain）是满足马尔科夫性质的随机变量序列 $X_1, X_2, X_3, ...$，即如果 $\Pr(X_1 = x_1, ..., X_n = x_n) > 0$，则

$$\Pr(X_{n+1} = x | X_1 = x_1, ..., X_n = x_n) = \Pr(X_{n+1} = x | X_n = x_n)$$

14.1.2 马尔科夫链蒙特卡洛算法

蒙特卡洛方法（Monte Carlo Method）是一种利用随机抽样来估计数值解的方法。而**马尔科夫链蒙特卡洛方法**（Markov Chain Monte Carlo, MCMC）指的是一类算法，这类算法通过构建马尔科夫链来生成随机样本，这些样本的分布最终会收敛到目标概率分布。在 MCMC 中，蒙特卡洛方法用于通过马尔科夫链生成的样本来估计目标分布的特性。

14.1.3 Metropolis-Hastings 方法

米特罗波利斯-黑斯廷斯方法（Metropolis-Hastings）[1]是一种具体实现 MCMC 的随机模拟算法，这个方法用来在概率分布上生成一系列的随机样本。这些样本可以用来估计分布的特性，比如均值或者方差。这种方法可以用于直接采样那些直接抽样很困难的分布，因此非常重要。

14.1.4 Metropolis-Hastings 算法描述

Metropolis-Hastings 方法也是一种马尔科夫链蒙特卡洛方法。以下是这个算法的简单描述。

- **输入**：
  - 一个非负的函数 f。这个函数不需要是一个归一化的概率密度函数，在物理模拟中，这个函数通常与系统的某些性质成比例。
  - 一个建议概率密度 $g(y \to x)$。这个函数在给定前一个样本值 y 的前提下，生成一个候选样本（candidate sample）x。
- **步骤**：给定当前的样本 X_i，
  1. 使用建议 PDF $g(X_i \to X')$ 生成新的样本 X'。
  2. 计算接受比率
  $$a = \min\left(1, \frac{f(X')}{f(X_i)} \frac{g(X' \to X_i)}{g(X_i \to X')}\right)$$
  这是当前样本和新样本的函数值与建议分布的比值。
  3. 随机生成一个 $\xi \sim \texttt{unif}(0, 1]$。
     - 如果 $\xi \leq a$，则接受 X' 作为下一个样本 X_{i+1}。
     - 否则，保持当前样本不变做为 X_{i+1}。
- **过程**：
  - 算法开始时，从任意初始状态 X_0 开始。
  - 随着时间的推移，算法会生成越来越多的样本，这些样本会趋于目标分布，也就是与函数 f 成比例的分布：f 越大的地方样本数量越多，反之亦然。

需要注意的是，Metropolis-Hastings 方法是在不了解 f 或其概率分布的情况下实现的，它的强大之处就在于可以评估每个已生成样本的函数 f。

需要注意的是，如果算法正确执行且迭代次数足够多，通过 Metropolis-Hastings 算法构造的马尔科夫链应该就是服从目标 f 分布的。然而，实际应用中有几个重要的注意事项：

- 烧入期（Burn-in Period）：在实际抽样开始之前，算法需要经过一定的烧入期。这是因为最初的样本可能受到初始状态选择的影响，并不一定符合目标分布。经过足够的迭代后，这种影响会减少，最终更好地反映目标分布。

[1]很有趣的是无论 Metropolis 还是 Hastings 都有实际的英文意义，但是它们都只是发明方法的人名，因此不能翻译成大都市-极速方法，需要音译为米特罗波利斯-黑斯廷斯。在之后的文本中，我们都会使用英文原文来叙述，而不再使用中文的音译名。

- 收敛性检验。需要有一种方法来检查马尔科夫链是否已经收敛。如果没有收敛，则生成的样本并不会准确代表目标分布。
- 相关性。由于 MCMC 生成的样本是马尔科夫链的一部分，因此连续的样本之间是相关的。在某些情况下，可能要对样本进行筛选—即只选择序列中的每第 n 个样本，以减少样本之间的相关性。

14.1.5 Metropolis-Hastings 算法使用示例

下面我们来看一个使用本方法的例子。

Example 14.1. *采样目标是一个以 0 为中心的正态分布，其概率密度函数为*

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}},$$

使用 Metropolis-Hastings 方法来生成这个分布的样本。

Solution. 我们使用以下的步骤来生成这个分布的样本。

1. **初始化**。
   - 选择一个任意起点 x_0 作为马尔科夫链的初始状态。
2. **迭代过程**。对于每一次迭代 i（从 $i = 1$ 开始）：
   - 提出新的状态。从一个简单的建议分布中抽取样本 x'，例如以 x_i 为中心的正态分布。
   - 计算接受概率。根据公式，
     $$a = \min\left(1, \frac{f(x')}{f(x_i)} \frac{g(x_i|x')}{g(x'|x_i)}\right)$$
     其中，$g(x'|x_i)$ 是从状态 x_i 转移到状态 x' 的概率，$g(x_i|x')$ 以此类推。
   - 接受或拒绝。生成一个 $[0,1]$ 区间内均匀分布的随机数 ξ。如果 $\xi \leq a$，则接受 x' 作为新的状态，即 $x_{i+1} = x'$。否则，拒绝 x'，保持当前状态不变，即 $x_{i+1} = x_i$。
3. **收集样本**。重复 2. 多次之后，我们会得到一系列样本，它们理论上服从 f 的分布。

■

14.2 Metropolis 光线传输

14.2.1 估计像素值

现在我们考虑一个问题：如何精确估计计算机生成图像中每个像素的光强度?

在全局光照（global illumination）模型中，场景中的每个像素的颜色和亮度都是由许多光线路径的贡献共同决定的，这些光线路径代表了光源发出的光如何在场景中多次反弹后到达人眼。我们可以将这个光线传输的积分抽象为：

$$\begin{aligned} I^{(j)} &= \int_{\Omega} f^{(j)}(\bar{x}) d\bar{x} \\ &= \int_{\Omega} h^{(j)}(\bar{x}) f(\bar{x}) d\bar{x} \end{aligned}$$

其中，

$$h^{(j)}(\bar{x}) = W_e^{(j)}(\mathbf{x}_1 \to \mathbf{x}_0)$$
$$f(\bar{x}) = L_e(\mathbf{x}_k \to \mathbf{x}_{k-1}) \left[\prod_{j=0}^{k-1} G(\mathbf{x}_{j+1} \leftrightarrow \mathbf{x}_j)\right] \left[\prod_{j=1}^{k-1} f_r(\mathbf{x}_{j+1} \to \mathbf{x}_j \to \mathbf{x}_{j-1})\right]$$

像素 j 的**强度**（intensity）$I^{(j)}$ 是我们在第五章讨论辐射度量学物理量中被跳过的一个，因为这个概念现在已经绝迹于大部分的文献，但是在讨论 MCMC 渲染时也许更容易理解。在这里，$f(\bar{x})$ 是场景中光线路径的某一种辐射度量函数，它是光源发射项 L_e，几何项 G 和 BRDF 项 f_r 的乘积。而函数 $h^{(j)}(\bar{x})$ 是一个**滤波函数**（filter function），它代表了像素 j 如何根据其视锥体对路径贡献进行加权。

注意：每一个像素都有自己的滤波函数，但是每一个像素使用的 f 是相同的。我们之前已经看到如果我们可以根据某一种概率密度函数 p 去采样 N 条路径，那么

$$I^{(j)} = E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{h^{(j)}(\bar{x}_i)f(\bar{x}_i)}{p(\bar{x}_i)}\right]$$

特别地，如果我们采用一个与 f 成正比的 p，即 $p \propto f$，或者说 $p(\bar{x}) = f(\bar{x})/b$，其中 b 是归一化因数，那么

$$I^{(j)} = E\left[\frac{1}{N}\sum_{i=1}^{N} b h^{(j)}(\bar{x}_i)\right]$$

到了这里，我们就会遇到两个挑战：

- 我们如何获得 $b = \int_\Omega f(\bar{x})d\bar{x}$?
- 我们如何从 $p(\bar{x}) = f(\bar{x})/b$ 中采样呢?

第一个挑战是积分的估算，所以可以用蒙特卡洛积分解决。而第二个问题涉及复杂分布的采样，显然，这就可以使用我们上面提到的 Metropolis-Hastings 方法了。

14.2.2 MLT 算法描述

米特罗波利斯光传输（Metropolis Light Transport, MLT）算法以一组 n 个随机路径开始，使用双向路径追踪的方法构建，从灯光到图像平面。然后，它会使用一些**突变策略**（Mutate strategies）使得路径产生突变。

概述

MLT 的大致步骤如下。

- **第一阶段：**初始化。
  1. 使用双向路径追踪方法，通过已知的密度函数 p_0 采样 N' 条种子路径（seed path）$\bar{x}_1^{\text{seed}}, ..., \bar{x}_{N'}^{\text{seed}}$。
  2. 估计归一化常数 $\langle b \rangle$ 作为目标函数 f 和概率密度函数 p_0 之间的比值。

$$\langle b \rangle = \frac{1}{N'}\sum_{i=1}^{N'}\frac{f(\bar{x}_i^{\text{seed}})}{p_0(\bar{x}_i^{\text{seed}})}$$

3. 从生成的路径中选择一小部分（比如一条）作为代表，用于第二阶段。

- **第二阶段：** Metropolis-Hastings 方法。

1. 从选取的种子路径开始，应用 Metropolis-Hastings 方法生成新的样本。根据接受准则接受或拒绝基于当前路径的随机**突变**（mutation）。这些随机变化产生了新的候选路径，它们被用来探索路径空间并生成符合目标分布 f 的样本。
2. 不断重复上述步骤，从而生成一系列路径，这些路径代表了图像中像素的光强度分布。每个新接受的路径用来更新图像，最终得到渲染结果。

伪代码

MLT 的伪代码如下。

```
MLT() {
    clear pixels in image to 0;
    x = InitialSeedPath(); // sample n paths here
    for i = 1 to N {
        y = Mutate(x);
        a = AcceptanceProbability(x, y);
        if (randf() < a) x = y; // accept the mutation
        RecordSample(image, x);
    }
}
```

14.2.3 MLT 中的路径突变

对于 MLT 的 Metropolis-Hastings 阶段，最核心的部分就是突变部分。对于一个给定的路径 $\bar{x}$，我们需要去定义一个**转化概率密度函数**（transition probability density function）$g(\bar{x} \to \bar{y})$，以允许基于 $\bar{x}$ 来采样突变路径 $\bar{y}$。通过转换概率，我们可以计算接受概率

$$a(\bar{x} \to \bar{y}) = \min\left\{1, \frac{f(\bar{y})}{f(\bar{x})}\frac{g(\bar{y} \to \bar{x})}{g(\bar{x} \to \bar{y})}\right\}$$

我们希望突变拥有一些性质。

- 高接受率。$a(\bar{x} \to \bar{y})$ 对于高概率而言应该较大。
- 需要既有大也有小的路径突变。
- 遍历性（ergodicity）。这些突变不应该卡在路径空间中的部分区域。也就是说，$g(\bar{x} \to \bar{y}) \neq 0, \forall \bar{x}, \bar{y}$，其中，$f(\bar{x}) > 0, f(\bar{y}) > 0$。
- 低计算成本。

在接下来的小节中，我们将着重讨论 Veach 和 Guibas 在 1997 年提出的三种突变—双向突变、路径突变和镜头子路径突变。

14.3 双向突变

14.4 镜头摄动与子路径突变

14.5 路径采样器

14.5.1 逆路径采样器

14.5.2 突变融合

14.6 渲染中的梯度

14.6.1 随机梯度下降（SGD）

14.6.2 Langevin 蒙特卡洛方法（LMC）

14.6.3 SGD 与 LMC 的比较

14.6.4 偏差分析

Chapter 15

反向渲染和可微分渲染

15.5 重参数化

15.5.1 散度定理

15.5.2 示例：速度

15.6 路径-空间可微分渲染

15.6.1 前向路径积分

15.6.2 微分路径积分

15.7 纹理参数化

15.7.1 重参数化下的微分路径积分

15.7.2 估计边界积分

15.8 可微分渲染的局限性

Chapter 16

科研图像渲染应用

Part III

附录

附录 A

多元微积分

为了降低阅读的难度，建立更多的直觉，本笔记中提及的多元微积分并不会采用非常严谨的探讨方式。在不明确讨论边缘情况时，我们都认为我们讨论的函数可导或可积。我们也不会对例如向量、邻域、极限、定义域等非常基础的概念咬文嚼字。这里的附录完全是为了能够理解本笔记中的数学推导而存在的。

A.1 偏导数与微分几何应用

A.1.1 偏导数

计算多元函数 $f(x,y,z,...)$ 对其中一个自变量的**偏导数** (partial derivative)，例如 x，就是将 x 以外的所有变量都视作常数求导，记作 $\frac{\partial f}{\partial x}$，读作 “partial f 比 partial x”，或者在一些不产生歧义的情况下，也可以读作 “df 比 dx”。f 对 x 的偏导函数也可以写作 $f_x(x,y,z...)$。

Example A.1. 计算函数 $z=x^2+3xy+y^2$ 对 x 的偏导数。

Solution. 把 y 视作常量，我们就有

$$\frac{\partial z}{\partial x}=2x+3y.$$

■

A.1.2 复合函数求导

一元函数与多元函数的复合

假设函数 $u=\varphi(t)$，$v=\psi(t)$，... 都在 t 处可导，函数 $z=f(u,v,...)$ 在对应点 $(u,v,...)$ 具有连续偏导数，那么复合函数 $z=f(\varphi,\psi,...)$ 在点 t 处可导，并且，

$$\frac{dz}{dt}=\frac{\partial z}{\partial u}\frac{du}{dt}+\frac{\partial z}{\partial v}\frac{dv}{dt}+....$$

上式中省略的部分请通过找规律自行补充。在这里 $\frac{dz}{dt}$ 也叫做**全导数**（total derivative）。

多元函数与多元函数的复合

以二元函数为例，假设函数 $u = \varphi(x,y)$，$v = \psi(x,y)$，... 都在 (x,y) 处具有对 x,y 的偏导数，函数 $z = f(u,v)$ 在对应点 (u,v) 具有连续偏导数，那么复合函数 $z = f(\varphi,\psi,...)$ 在点 (x,y) 处的两个偏导数都存在，并且，

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial z}{\partial v}\frac{\partial v}{\partial x},$$

对 y 的偏导数也类似。事实上，在求对 x 的偏导数的时候，我们都是将 y 当做常量的，因此 u,v 依然可以视作一元函数应用上一小节的结论。

A.1.3 方向导数

偏导数反映函数沿坐标轴方向的变化率，而**方向导数**（directional derivative）研究的则是某一指定方向的变化率。当函数在点 $\mathbf{p} = (x_0, y_0, ...)$ 沿着方向单位向量 $\mathbf{v} = (v_1, v_2, ...)$ 的方向导数存在时，我们有

$$D_{\mathbf{v}} f = \frac{\partial f}{\partial \mathbf{l}}|_{(x_0,y_0,...)} = f_x(x_0,y_0,...)v_1 + f_y(x_0,y_0,...)v_2 + ...$$

A.1.4 梯度

梯度是一个表征多元函数最快增长方向的向量。

定义（梯度）多元函数 $f(x,y,...)$ 在点 $\mathbf{p}$ 的**梯度**（gradiant）是由函数在 $\mathbf{p}$ 的偏导数组成的向量，其形式为

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, ...\right)$$

其中，$\nabla = \left(\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, ...\right)$ 被称为 **Nabla 算子**（Nabla Operator）。

根据定义，方向导数的形式也可以写作

$$D_{\mathbf{v}} f = \nabla f \cdot \mathbf{v}$$

梯度的性质

从直观上来理解梯度，首先我们需要记住梯度是一个向量，而这个向量具有以下的性质。

- 梯度的方向指向函数增长最快的方向。在一个点上沿着梯度的方向移动，函数值以最快的速度增加。
- 梯度的反方向指向函数减少最快的方向。
- 梯度的大小表示函数在这个方向上增长的速率。
- 如果在某个点上的梯度为零向量，则意味着在该点附近函数没有增长或减少，在这种情况下，这个点可能是一个局部最大或最小值，也有可能是鞍点。

Example A.2. *设 $f(x,y) = \frac{1}{2}(x^2+y^2)$，点 $\boldsymbol{P}_0 = (1,1)$。求:*
(1) $f(x,y)$ 在点 $\boldsymbol{P}_0$ 处增加最快的方向以及 $f(x,y)$ 沿这个方向的方向导数。
(2) $f(x,y)$ 在点 $\boldsymbol{P}_0$ 处减少最快的方向以及 $f(x,y)$ 沿这个方向的方向导数。
(3) $f(x,y)$ 在点 $\boldsymbol{P}_0$ 处变化率为 0 的方向。

Solution. (1) $f(x,y)$ 在点 $\mathbf{P}_0$ 处沿着 $\nabla f(1,1)$ 的方向增长最快。

$$\nabla f(1,1) = \left(\frac{\partial f}{\partial x}\mathbf{i} + \frac{\partial f}{\partial y}\mathbf{j}\right)|_{(1,1)} = (x\mathbf{i} + y\mathbf{j})|_{1,1} = \mathbf{i} + \mathbf{j}.$$

故所求方向为

$$\mathbf{v} = \frac{\nabla f(1,1)}{|\nabla f(1,1)|} = \frac{1}{\sqrt{2}}\mathbf{i} + \frac{1}{\sqrt{2}}\mathbf{j}.$$

方向导数为

$$D_{\mathbf{v}} f = \nabla f(1,1) \cdot \mathbf{v} = \sqrt{2}.$$

(2) 减少最快方向为 $-\nabla f(1,1)$，因此由 (1) 取负号可得所求方向为 $-\frac{1}{\sqrt{2}}\mathbf{i} - \frac{1}{\sqrt{2}}\mathbf{j}.$，方向导数为 $-\sqrt{2}$。

(3) 在点 $\mathbf{P}_0$ 处沿垂直于 $\nabla f(1,1)$ 的方向变化率为 0。这个方向是 $\mathbf{n}_2 = -\frac{1}{\sqrt{2}}\mathbf{i} + \frac{1}{\sqrt{2}}\mathbf{j}$ 或 $\mathbf{n}_3 = \frac{1}{\sqrt{2}}\mathbf{i} - \frac{1}{\sqrt{2}}\mathbf{j}$。 ■

A.2 隐函数求导

A.2.1 隐函数求导公式

一元隐函数/二元方程的求导公式

如果函数 $F(x,y)$ 在点 $\mathbf{p}$=(x_0,y_0) 的某一个邻域内具有连续偏导数，且 $F(x_0,y_0)=0, F_y(x_0,y_0)\neq 0$，则方程 $F(x,y)=0$ 在点 $\mathbf{p}$ 的某一邻域内能确定唯一一个连续且具有连续导数的函数 $y=f(x)$，它满足 $y_0=f(x_0)$，并且有

$$\frac{dy}{dx} = -\frac{F_x}{F_y}.$$

这个公式就是一元隐函数的求导公式。

二元隐函数/三元方程的求导公式

如果函数 $F(x,y,z)$ 在点 $\mathbf{p}$=(x_0,y_0,z_0) 的某一个邻域内具有连续偏导数，且 $F(x_0,y_0,z_0)=0, F_y(x_0,y_0,z_0)\neq 0$，则方程 $F(x,y,z)=0$ 在点 $\mathbf{p}$ 的某一邻域内能确定唯一一个具有连续偏导数的函数 $z=f(x,y)$，它满足 $z_0=f(x_0,y_0)$，并且有

$$\frac{\partial z}{\partial x} = -\frac{F_x}{F_z}, \frac{\partial z}{\partial y} = -\frac{F_y}{F_z},$$

这就是二元隐函数的求导公式。

A.3 向量值函数

A.4 二重积分

对于以封闭区域 D 为底，曲面 $z=f(x,y)$ 为顶的曲顶柱体，其体积可以通过将底面 D 划分为 n 个极小的封闭区域，再以极小区域的高计算平顶柱体的体积并累加所有平顶柱体而得。也就是说，柱体体积为

$$V = \lim_{\lambda\to 0}\sum_{i=1}^{n} f(\xi_i,\eta_i)\Delta\sigma_i$$

其中，λ 为小封闭区域 $\Delta\sigma_i$ 的直径，点 (ξ_i, η_i) 为每个平顶柱体底面的中心，对应函数值 $f(\xi_i, \eta_i)$ 则为该平顶柱体的高。我们将二元函数 $f(x, y)$ 在闭区域的**二重积分**（double integral）记作

$$\iint_D f(x, y)d\sigma = V.$$

A.4.1 直角坐标计算二重积分

对于曲顶柱体的底面封闭区域 D，若其可以通过直角坐标 x, y 用不等式

$$\varphi_1(x) \leq y \leq \varphi_2(x), a \leq x \leq b$$

来表示，其中 φ_1, φ_2 在区间 $[a, b]$ 是连续的，则二重积分可以通过

$$\iint_D f(x, y)d\sigma = \int_a^b \left[\int_{\varphi_1(x)}^{\varphi_2(x)} f(x, y)dy \right] dx.$$

进行计算。这样的计算方式也叫做先对 y 后对 x 的二次积分。一开始先将 x 看做常数，然后再把所得结果对 x 进行区间 $[a, b]$ 的定积分即可。类似地，我们也可以先对 x 后对 y 进行二次积分，取决于封闭区域适合用什么样的方式来定义。

A.4.2 极坐标计算二重积分

直角坐标转换

直角坐标中的积分可以通过坐标转换至极坐标进行积分计算，其公式为

$$\iint_D f(x, y)dxdy = \iint_D f(r\cos\theta, r\sin\theta)rdrd\theta.$$

极坐标的二次积分

极坐标也可以通过类似于直角坐标的二次积分进行计算。对于曲顶柱体的底面封闭区域 D，若其可以通过极坐标 r, θ 用不等式

$$\varphi_1(\theta) \leq r \leq \varphi_2(\theta), \alpha \leq \theta \leq \beta$$

来表示，其中 φ_1, φ_2 在区间 $[\alpha, \beta]$ 是连续的，则二重积分可以通过

$$\iint_D f(r\cos\theta, r\sin\theta)rdrd\theta = \int_\alpha^\beta \left[\int_{\varphi_1(\theta)}^{\varphi_2(\theta)} f(r\cos\theta, r\sin\theta)rdr \right] d\theta.$$

A.4.3 换元法

如果存在转换 T 将点从 (u, v) 坐标系转换至 (x, y) 坐标系，即 $x = x(u, v), y = y(u, v)$，且 $x(u, v), y(u, v)$ 均在 (u, v) 坐标系的封闭区域 D' 具有偏导数，则它们的**雅可比矩阵**（Jacobian Matrix）定义为

$$J(u, v) = \begin{bmatrix} \frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \end{bmatrix}$$

我们也叫它雅可比项。其**雅可比行列式**（Jacobian determinant）为

$$|J(u, v)| = |\det(J)| = \left| \frac{\partial x}{\partial u}\frac{\partial y}{\partial v} - \frac{\partial x}{\partial v}\frac{\partial y}{\partial u} \right|$$

对于二元函数 $f(x,y)$，给定其二重积分 $\iint_D f(x,y)dxdy$。如果我们想要通过新的变量 u,v 来表达这个积分，则积分的换元公式写作

$$\iint_D f(x,y)dxdy = \iint_{D'} f[x(u,v),y(u,v)] \cdot |J(u,v)|dudv$$

其中，D' 是在 (u,v) 坐标系下的积分区域。因此，在进行积分换元的时候，不能忘记雅可比行列式的存在！

Example A.3. 计算函数 $e^{\frac{y-x}{y+x}}$ 在区域 D 上的二重积分，其中 D 是由 x 轴、y 轴以及直线 $x+y=2$ 所围成的封闭区域。

Solution. 令 $u=y-x$, $v=y+x$，则 $x=\frac{v-u}{2}, y=\frac{v+u}{2}$。
则雅可比行列式为

$$\left|\frac{\partial x}{\partial u}\frac{\partial y}{\partial v} - \frac{\partial x}{\partial v}\frac{\partial y}{\partial u}\right| = |-1/2| = 1/2.$$

利用换元公式可得，

$$\iint_D e^{\frac{y-x}{y+x}}dxdy = \iint_D e^{u/v}\frac{1}{2}dudv = \frac{1}{2}\int_0^2 dv\int_{-v}^{v} e^{u/v}du = \frac{1}{2}\int_0^2 (e-e^{-1})vdv = e-e^{-1}.$$

■

A.5 雅可比矩阵

雅可比矩阵在图形学中的地位非常重要，最根本的原因就在于变换是图形学的基础。因此，在一个坐标系下的积分操作变换到另一个坐标系时，就必须要有雅可比矩阵的参与。我们要确保对雅可比矩阵拥有正确的理解。

A.5.1 二元情况

在 A.4.3 的定义中，我们提到的是二元函数求二重积分时的雅可比矩阵。在这里，雅可比矩阵的行列式指的实际上就是 xy 平面上的面积微元与 uv 平面上的面积微元的比值。在变换后的 xy 平面上的区域 D 中，面积微元的大小为

$$dA = dxdy$$

而在变换前的 uv 平面上的区域 D' 中，其对应的面积微元的大小则应该是

$$dA = |J(u,v)|dudv$$

因此，二重积分的换元法的公式为

$$\iint_D f(x,y)dxdy = \iint_{D'} f[x(u,v),y(u,v)] \cdot |J(u,v)|dudv$$

A.5.2 局部仿射逼近

回顾一下仿射变换的定义。对于 $\mathbb{R}^n$ 空间上的仿射变换，其具有形式

$$T(\mathbf{x}) = \mathbf{A}\mathbf{x} + \mathbf{b}$$

其中，$\mathbf{A}$ 是一个矩阵，$\mathbf{b}$ 是一个 n 维向量。

对于向量值函数，在所求点 $\mathbf{x}$ 上产生微小扰动 $\Delta\mathbf{x}$，使得向量值发生如下变化，

$$\mathbf{f}(\mathbf{x}_0 + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}_0) + \mathbf{A}\Delta\mathbf{x} + O(||\Delta\mathbf{x}||^2).$$

其中，$\mathbf{A}$ 是某个矩阵，忽略高阶项 $O(||\Delta\mathbf{x}||^2)$，得到向量值函数的仿射逼近，

$$\mathbf{f}(\mathbf{x}_0 + \Delta\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + \mathbf{A}\Delta\mathbf{x}.$$

这正是一个仿射变换，我们也可以写作

$$\Delta\mathbf{f} = \mathbf{A}\Delta\mathbf{x}.$$

换言之，对于向量值函数，我们可以用一个矩阵去逼近表示函数的微小局部。当然，每一点的矩阵可能是不同的，这个矩阵的具体形式，就是我们所说的雅可比矩阵，即

$$\mathbf{A} = J_{\mathbf{f}} = \begin{bmatrix} \frac{\partial \mathbf{f}}{\partial x_1} & \dots & \frac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix}$$

回顾一下导数的概念。在一元函数中，导数

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{\Delta f}{\Delta x} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} = f'(x)$$

在这里，导数是一个标量，它的几何意义是斜率。我们也有下面的关系，

$$df(x) = f'(x)dx$$

在多元标量值函数中，函数的导数是一个向量，它的几何意义是梯度。我们有下面的关系

$$df(\mathbf{x}) = \nabla f \cdot d\mathbf{r} = (\frac{\partial f}{\partial x}dx, \frac{\partial f}{\partial y}dy)$$

而在多元向量值函数中，函数的导数则是一个矩阵，这就是雅可比矩阵的几何意义。

$$d\mathbf{f}(\mathbf{x}) = J_{\mathbf{f}}d\mathbf{x}$$

详细地说，如果多元向量值函数 $\mathbf{f}(\mathbf{x})$ 在 $\mathbf{x}$ 处可微，$\mathbf{h}$ 是一个位移向量，代表在定义域内的一个微小移动，则雅可比矩阵和该位移向量的乘积 $J_{\mathbf{f}}\mathbf{h}$ 也是一个位移向量，用于近似表示 $\mathbf{h}$ 引起的微小移动在值域中的位移向量 $\mathbf{f}(\mathbf{x} + \mathbf{h}) - \mathbf{f}(\mathbf{x})$。

A.6 三重积分

A.6.1 直角坐标计算三重积分

A.6.2 柱面坐标计算三重积分

A.6.3 球面坐标计算三重积分

A.6.4 重积分应用

曲面面积

质心

转动惯量

引力

A.7 曲线积分

A.7.1 对弧长曲线积分

A.7.2 对坐标曲线积分

A.7.3 格林公式

A.8 曲面积分

A.8.1 对面积曲面积分

A.8.2 对坐标曲面积分

A.8.3 高斯公式

A.9 级数

附录 B

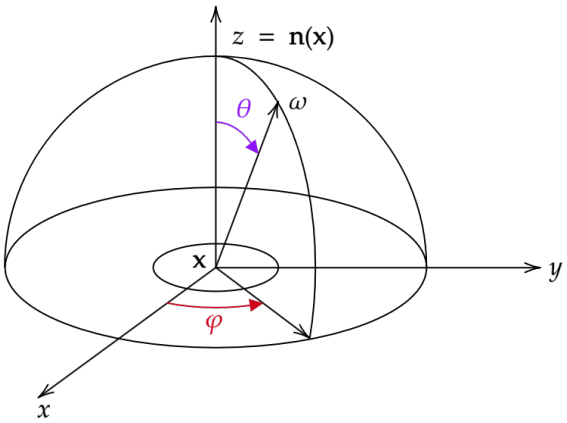
半球坐标

在图形学中，我们经常研究的是半球内的光线行为，因为在表面之下的部分通常不能对光照产生贡献。因此，用球面坐标系表达半球是我们需要理解的一个部分。

B.1 球坐标系下的半球坐标

B.1.1 半球上一点

以点 $\mathbf{x}$ 为球心的球坐标系下的半球是一个二维空间，从 $\mathbf{x}$ 出发指向半球上一点的方向 ω 可以用 (θ, φ) 坐标表达。如图所示。



在这里，$\omega = (\theta, \varphi)$。我们通常称 φ 为**方位角**（azimuth angle），称 θ 为**天顶角**（zenith angle）。方位角指的是正 x 轴到 ω 在 xy 平面上的投影线之间的夹角。天顶角指的是正 z 轴，也就是改点处的表面法向量，到该方向的夹角。

根据定义，我们知道 θ 和 φ 各自的取值范围：

$$\theta \in [0, \frac{\pi}{2}],$$

$$\varphi \in [0, 2\pi].$$

B.1.2 三维空间中任意一点

在三维空间中的任意一点 $\mathbf{p}$ 通常并不一定在半球上，但是我们可以通过三维数组 (θ, φ, r) 来表示，其中，r 是沿着半球上方向 $\omega = (\theta, \varphi)$ 时，$\mathbf{p}$ 与 $\mathbf{x}$ 的距离。那么，根据简单的画图，我们就能得出从 (θ, φ, r) 变换至 (x, y, z) 坐标的转换公式。

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

反过来，我们也可以从 (x, y, z) 坐标转换成 (θ, φ, r) 坐标。

$$
\begin{aligned}
r &= \sqrt{x^2 + y^2 + z^2} \\
\theta &= \arctan\left(\frac{\sqrt{x^2 + y^2}}{z}\right) \\
\varphi &= \arctan\left(\frac{y}{x}\right)
\end{aligned}
$$

B.2 立体角

在第五章中我们提及过立体角的定义和概念，在此不再赘述。需要明确的是：立体角是球面上面积对应的角度。如果半球的半径为 1，则立体角的数值就是其面积的大小。

对于很小的表面，可以使用余弦近似值来计算表面对应的立体角大小，

$$
\Omega = \frac{A \cos\alpha}{r^2}
$$

其中，α 是表面法线与表面中心点与球心的连线的夹角。

B.3 半球积分

在第五章中，我们曾经介绍过微分立体角的表达式，

$$
d\omega = \sin\theta d\theta d\varphi
$$

在半球上建立函数 $f(\omega = (\theta, \varphi))$ 的积分，其表达式则为

$$
\int_{\Omega} f(\omega) d\omega
$$

将其转换为 $\theta - \varphi$ 变量的积分，则为

$$
\int_0^{2\pi} \int_0^{\pi/2} f(\theta, \varphi) \sin\theta d\theta d\varphi
$$

Example B.1. 对以 N_x 为中心，功率为 N 的余弦波函数 $\cos^N(\omega, N_x)$ 进行半球内积分。

Solution.

$$\begin{aligned}\int_\Omega \cos^N(\omega, N_x)d\omega &= \int_0^{2\pi}\int_0^{\pi/2} \cos^N(\theta,\varphi,N_x)\sin\theta d\theta d\varphi \\ &= \int_0^{2\pi} d\varphi \int_0^{\pi/2} \cos^N\theta \sin\theta d\theta \\ &= \int_0^{2\pi} d\varphi \left[-\frac{\cos^{N+1}\theta}{N+1}\right]_0^{\pi/2} \\ &= \int_0^{2\pi} \frac{1}{N+1}\cdot d\varphi \\ &= \frac{2\pi}{N+1}.\end{aligned}$$

■

B.4 半球区域转换

在渲染算法中，有的时候将半球积分表达成从点 $\mathbf{x}$ 看到的表面的积分会更方便。例如，如果想要计算远距离光源在某个点处产生的所有入射光，可以在光源所对应的立体角的所有方向上积分，也可以在光源的实际表面上进行积分。这就涉及到半球积分与区域积分的转换。

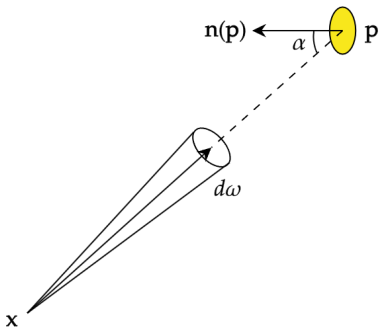
在 B.2 中我们提及，

$$\Omega = \frac{A\cos\alpha}{r^2}$$

对于微分立体角，这一关系保持不变，因此

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

其中，r 为表示光源的点 $\mathbf{p}$ 与当前点 $\mathbf{x}$ 的距离，α 为光源的法线与从 $\mathbf{p}$ 到 $\mathbf{x}$ 的方向的夹角。



通过半球区域转换，我们可以将围绕方向 ω 的微分立体角转换至 $\mathbf{p}$ 点处的微分表面 dA。对于半球上的函数 $f(\omega)$，它可以被改写成点 $\mathbf{p}$ 的函数，

$$\int_\Omega f(\omega)d\omega = \int_A f(\mathbf{p})\frac{\cos\alpha}{r^2}dA$$

附录 C

C++

由于 C++ 面向对象的特性，以及 C 语言系对于程序员理解底层提供的重要帮助，C++ 通常作为各类图形引擎的编写首选。C++11 版本提供了一些新特性，也对我们编写提供了更多的帮助。因此，我们留出一章单独解释 C++ 各版本常用的共通特性以及 C++11 的一些常用的特性。

C.1 面向对象编程

简单复习一下**类**（class）和**对象**（object）的关系。例如，一只叫吠仔的狗（对象）是“狗”类的实例（instance）。其实用人话说“A 是 B 类的实例”就是“A 是个 B”。比如上文就是“吠仔是条狗”。**面向对象编程**（object-oriented programing, oop）最重要的三个特性就是**封装**、**继承**和**多态**。

C.1.1 封装

封装（encapsulation）从概念上来理解就是对象向外提供有限接口，隐藏内部状态和实现细节。

接口（interface）一词可以直观地理解：就像电脑上的 USB 接口一样，你知道那里可以插入一个 USB，但你不需要理解这个 USB 接口后面的走线是什么样的，你知道电脑有读取 USB 的功能就行。对于程序员而言，很多时候拿到一个实例或者一个类，只需要知道这个类提供哪些接口、哪些功能就行，并不需要去关注背后的实现原理。

封装的特性体现在以下的几点：

- **访问修饰符**（access modifiers）：3 个访问修饰符。`public`, `private`, `protected`。其中，
  - `public`: 类内外都可使用。
  - `private`: 仅当前类内可使用。
  - `protected`: 类与其继承类内可使用。
- **成员函数**（member functions）：定义在类内部的函数，用于实现类的行为。一般包含在类的定义中，但也可以在类的外部定义。
- **构造函数**（constructor）：用于初始化新对象的状态。

- **析构函数**（destructor）：用于清理内存资源。

C.1.2 继承

继承（inheritance）是指延伸现有类的定义，去定义新的类的行为。在这里，被继承的类叫做**基类**（base class），继承的类叫做**派生类**（derived class）或**子类**（subclass）。

被继承的类与派生的类形成“是个”（is-a）关系。例如，狗显然是动物类的子类。所以狗是个动物。

从定义上来说，继承并不禁止同时继承多个类，也被称为**多重继承**（multiple inheritance）。但是实际编写中，多重继承经常会产生问题，如致命的**钻石继承**（diamond problem），所以我们应该尽量减少或完全不适用多重继承。

钻石继承

假设有基类 A，从其中派生两个子类 B, C，它们都继承 A。现在有一个 D 同时继承 B, C。将它们的关系图画出来，就会看起来像个菱形/钻石，因此得名**钻石继承问题**。在这种情况下，D 中会包含两份 A 的数据，一份从 B 继承得来，一份从 C 继承得来。这不仅会导致数据冗余，还可能引发严重的问题，比如数据不一致性以及不确定的访问路径。

C++ 也提供了解决钻石继承的方案，就是使用**虚继承**（virtual inheritance）。利用 `virtual` 关键字，使用虚继承时，无论一个类被继承了多少次，都会只存一个基类的实例。

```
class A { };
class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {
    // Now D has only one copy of A in the memory.
}
```

C.1.3 多态

多态（polymorphism）指的就是同一个接口，但可以得到不同的实现。多态有静态动态之分。

静态多态也叫做编译时多态，是在编译时实现的。这种多态主要通过函数**重载**（override）和运算符重载体现。其中，函数重载指的是在同一个类中存在多个同名函数，但它们的参数类型、数量可能不同。运算符重载则是为现有的部分运算符提供新的实现。

动态多态是在程序运行时实现的，主要通过**虚函数**（virtual function）以及继承体现。我们重点了解一下虚函数。

虚函数

在一个类中声明函数时，我们给它加上一个 `virtual` 关键字，就可以在派生类中重写这个函数。虚函数也可以允许在派生类中定义一个与基类虚函数具有相同名称和签名[1]的函数。

纯虚函数（pure virtual function）是指在基类中声明但是不定义的函数，需要用 `=0` 作为其赋值（作用实际上是语法标记而不是赋值）。拥有纯虚函数的类称为**抽象类**（abstract class），抽象类不能被实例化，只能作为基类被继承。

在下例中，我们定义抽象类 `Shape`，然后定义这个抽象类的纯虚函数 `Draw()`。我们将这个函数的实现留给各个派生类。

```
class Shape {
public:
    virtual void Draw() = 0;
}

// derived class circle
class Circle : public Shape {
    virtual void Draw () {
        // code for drawing the circle
    }
}

class Rectangle : public Shape {
    virtual void Draw() {
        // code for drawing the rectangle
    }
}
```

C.2 链接

C++ 程序是由翻译单元组成的，编译器每次翻译一个.cpp 文件，并输出相应的对象文件.o 或者.obj。编译器操作的最小单位就是.cpp，因此其被称作翻译单元。这个对象文件不仅会含有.cpp 文件内定义的所有函数（编译后的机器码），还会包含.cpp 文件内定义的所有全局变量和静态变量。对象文件也可能含有**未解决引用**（unresolved reference），这些未解决引用是其他.cpp 文件内定义的函数和全局变量。

链接器（linker）的作用就是把所有对象文件组合成最终**可执行映像**（executable image）。在这个过程中，链接器会尝试通过链接解决对象文件之间的交叉引用。如果链接成功，生成的可执行映像将包含所有函数、全局和静态变量，并正确解决.cpp 文件之间的交叉引用。链接器不允许以下两种错误：

- 找不到 `extern` 引用的目标。链接器会报告*无法解决的外部符号*（unresolved external symbol）的错误。

[1]签名（signature）指的是函数的名字以及参数类型、顺序、数量，并不包含返回类型。如果两个不同返回类型的函数拥有同样的名字以及同样的参数列表，它们会被认为是相同签名的。

- 函数在多个翻译单元间被定义，或在同一单元内被定义多次。链接器会报告符号被多重定义（multiply defined symbol）错误。

C.2.1 定义声明

声明（declaration）指的是函数的描述。而**定义**（definition）指的是个别内存区域的描述。一个定义必然是一个声明，但一个声明并不一定会是定义。

函数声明的形式包括

```
extern int someFunction(int a, double b);
int someOtherFunction(int a, double b);
```

函数定义的形式如同

```
int someFunction(int a, double b) {
    return (a>b) ? 0 : 1;
}
```

C.2.2 内联函数

通常，我们不会把函数的定义放在头文件.h 中，这是因为如果有多个.cpp 文件 `#include` 了该头文件，这个函数就会被多重定义。但是，有一种函数例外，就是带着 `inline` 关键字的函数，这些函数被称作**内联函数**（inline function），因为它们的本质是将函数的机器码复制到调用方的函数中。如果内联函数被多个.cpp 文件调用，则其必须定义在.h 文件里。

例如，我们在 `foo.h` 的头文件中，声明并定义

```
inline int max(int a, int b) {
    return (a > b) ? a : b;
}
```

这个函数可以被正确内联。但是如果我们不给定义

```
inline int max(int a, int b);
```

则不会被正确内联。

C.2.3 链接规范

外部链接（external linkage）指的是在定义处.cpp 文件之外的.cpp 文件中也能看见并引用的定义。相对地，**内部链接**（internal linkage）则只能在该定义所处的.cpp 看见，其它.cpp 文件无法引用。有点类似于 `public, private` 的作用，只不过作用对象是.cpp 文件而不是类。

对于任何定义，我们都可以使用 `static` 关键字使其强制成为内部链接。其余情况下，我们默认定义预设为外部链接。值得一提的是，被 `static` 修饰过的同名变量可以在多个.cpp 中出现，但并不会被报多重定义。

C.3 内存布局

下面回顾一下 C++ 中各类变量、函数、结构、类在内存中的布局。

C.3.1 基础数据类型

回顾一下以下数据类型的大小。另外记得位与字节的换算是 1 字节（byte）=8 位（bit）。

- `char`：1 字节。
- `short`：通常为 2 字节。
- `int`：通常为 4 或 8 字节。32 位机器上通常为 4 字节，而 64 位机器上则为 8 字节。本笔记中不做特殊说明时，均认为 `int` 为 4 字节。
- `long`：通常为 4 或 8 字节。其要求是不小于 `int`，其它取决于 CPU 架构和操作系统。
- `float`: 4 字节。
- `double`：8 字节。
- `bool`：1 字节或 4 字节，取决于硬件架构。

C.3.2 内存栈

当可执行程序被载入内存并运行时，操作系统会保留一块称为**内存栈**（memory stack）的内存空间。调用函数时，一块连续的内存会被压入栈，这个内存块被称作**栈帧**（stack frame）。若函数 `a()` 调用函数 `b()`，则函数 `b()` 的栈帧会被压入栈，位于 `a()` 的栈帧之上。当 `b()` 返回（return）时，栈帧会被弹出，然后继续执行 `a()`。

我们观察到，后进入栈的角色会被栈优先弹出，这种结构被称作 LIFO（Last In First Out，后进先出）结构。

内存栈存储三类数据。

- 调用方函数的返回地址（return address）。这样当被调用的函数返回时，程序知道要返回到哪里。
- CPU 寄存器（register）相关内容。把寄存器上原有的内容先存好，这样被调用的函数就可以随便用寄存器了。等返回的时候再把值恢复回去。
- 局部变量。

C.3.3 内存堆

有时，程序需要在运行时动态地请求内存块。为了提供动态分配的功能，操作系统会在**内存堆**（memory heap）上申请内存块。在 C 语言中，我们调用 `malloc()` 函数申请内存块，而在 C++ 语言中，我们使用 `new` 关键字申请内存块。

在堆上申请的内存块必须手动地释放，否则就会造成**内存泄漏**（memory leak）。在 C 语言中，我们使用 `free()` 函数释放之前由 `malloc()` 函数调用的内存块。在 C++ 中我们则使用关键字 delete。为了不出

现问题，这两套必须成对使用。在 C++ 中，个别类可能会重载 `new` 操作符自定义内存分配方式，因此也不能假设所有的 `new` 都一定会申请内存块。

C.3.4 对象的内存布局

对齐

为了使得 CPU 能够更高效地读取数据[2]，编译器会将结构体内的数据类型自动**对齐**（alignment）。对齐的基本要求是：一个大小为 s 的数据类型，其地址总需要是 s 的倍数。例如，一个 2 字节的数据类型，它的数据地址的（十六进制）最后一位总需要是 0x0, 0x2, 0x4, 0x8, 0xA, 0xC, 或 0xE。因此，编译器会自动在小的数据类型后面补上填充（padding），使得跟在其后面的大的数据类型能够满足其对齐要求。因此，观察下列两个结构体的对齐方式。

```
struct Inefficient {
    int a; // 4 bytes
    float b; // 4 bytes
    char c; // 1 byte + 3 bytes padding
    int d; // 4 byte
    bool e; // 1 byte + 3 bytes padding
    char* f; // 4 byte
}

struct Efficient {
    int a; // 4 bytes
    float b; // 4 bytes
    int d; // 4 byte
    char c; // 1 byte
    bool e; // 1 byte + 2 bytes padding
    char* f; // 4 byte
}
```

这两个结构体拥有完全一样的内部变量，但是只是因为排列顺序的不同，导致在第二个结构中我们可以减少减少 4 字节的容量。这就要求我们在排列结构体的大小时总是想好它们排列的顺序，以确保节省空间。

再观察这两个例子。

```
struct Efficient {
    int a; // 4 bytes
    float b; // 4 bytes
    int d; // 4 byte
    char c; // 1 byte
    bool e; // 1 byte + 2 bytes padding
    char* f; // 4 byte
}

struct AnotherEfficient {
    int a; // 4 bytes
    float b; // 4 bytes
    int d; // 4 byte
    char* f; // 4 byte
```

[2]事实上很多 CPU 只能读取特定位置的数据。

```
    char c; // 1 byte
    bool e; // 1 byte + 2 bytes padding
}
```

在结构体 `AnotherEfficient` 中，尽管我们最后不加两个填充，对于结构体而言内部的所有变量都能满足对齐要求，但是编译器会自动地预测如果在内存中添加数个该结构体，对齐是否能够满足要求。在这种情况下，编译器会认为结尾增加两个填充能有助于整个结构体的对齐，因此就会增加这两个填充。

对象的大小

有了上面的对齐的知识基础，计算结构体的大小就很简单。内部成员的数据大小加上填充即可。需要注意，由于指针的实际值是地址，所以在 32 位系统上，指针的大小为 4 字节。在 64 位系统上，指针的大小为 8 字节。另外，无论是类还是结构体，仅在拥有实例之后才会占据内存，它们的声明是不占据内存的。

C.3.5 C++ 类的内存布局

由于有继承和虚函数的存在，C++ 类的内存布局与 C 的结构体有一些差别。

继承

类 B 在继承类 A 之后，类的实例在内存中就会体现为：首先是 A 中的变量布局，然后是类 A 与类 B 之间的对齐填充，接着类 B 中新增的变量紧随其后。在多重继承时，父类代码可能会有钻石继承导致被多次布局；但我们现实中尽量避免使用多重继承，所以这里我们就不讨论多重继承的布局了。

需要注意的是，虽然类 A 中的 `private` 变量在类 B 中无法访问，类 B 却依然要在自己的内存布局中为这些私有变量划出空间。这是为了避免在类 A 提供的 `public` 或者 `protected` 函数中访问类 A 的私有变量出现问题的情况。

虚函数

通常，在 C++ 中，成员函数（包括析构函数、构造函数和其它方法）不会占据类实例的内存，因为这些成员函数的代码对于同一个类的所有对象来说是共享的。但是有一个例外——虚函数。如果一个类拥有或者继承了虚函数，就会消费额外的一个指针空间（4 或 8 字节，取决于机器是 32 位还是 64 位）。这个指针被称为**虚表指针**（virtual table pointer），通常用 `vptr` 来指代，通常会被加在这个类布局的开头。它由其功能得名——它指向**虚函数表**（virtual function table），通常用 `vtable` 来指代。虚表中包含指向类的所有虚函数的指针的数组。

需要说明的是，虚表是占据内存的，但是与类的实例中的变量不同，它既不在内存栈上，也不在内存堆上，而是在程序的**数据段**（Data Segment）中，其位于可执行映像中，在这里不深入探究。对于同一个类的所有实例，它们共用一个虚表。另外，在发生类继承时，派生类也不会再次为基类的虚表指针划分空间。如果类 B 没有引入新的虚函数，并且没有覆盖类 A 的任何虚函数，那么类 B 的实例将使用类 A 的虚表。内存中的布局形如：（类 B 的）虚表指针 - 类 A 变量 - 可能的填充 - 类 B 变量 - 可能的填充。

Example C.1. *观察下面的代码，判断继承自 `Shape` 类的派生类 `Circle` 的实例在 64 位机器的内存中占据的紧凑型空间。其中 `Vec3` 是一个大小为 16 字节的结构体。*

```
class Shape {
public:
    virtual void SetId(int id) { m_id = id; }
    int          GetId() const { return m_id; }

    virtual void Draw() = 0;

private:
    int m_id;
};

class Circle : public Shape {
public:
    void  SetCenter(const Vec3& c) { m_center=c; }
    Vec3  GetCenter() const { return m_center; }
    void  SetRadius(float r) { m_radius = r; }
    float GetRadius() const { return m_radius; }

    virtual void Draw(){
        // code to draw the circle
    }

private:
    Vec3  m_center;
    float m_radius;
}
```

Solution. 考虑到成员中空间占用最大的是 `Circle::m_center`，我们采用 16 字节对齐。`Circle` 类的布局如下：

- 虚表指针：8 字节。
- `Shape::m_int`：4 字节；
- 类 A 填充：4 字节；
- `Circle::m_center`：16 字节；
- `Circle::m_radius`：4 字节；
- 类 B 填充：12 字节。

总计 48 字节。 ■

C.4 C++11 特性

尽管 2017 年 7 月 31 日 ISO 发布了最新的 C++ 版本 C++17，但是在版本标准间切换的代价对于项目而言可能是巨大的。因此，我们在这里还是讨论目前业界普遍使用的、也提供了最多重要功能的、2011 年发布的 C++11。

C.4.1 智能指针

C++11 提供的**智能指针**（smart pointers）我们主要使用三种。`std::unique_ptr`，`std::shared_ptr` 以及 `std::weak_ptr`。传统 C++ 指针操作可能会出现一些问题，例如内存管理和循环引用，而这三种指针能够帮我们很好地解决这类问题。

std::unique_ptr

`std::unique_ptr` 是一种独占所有权的智能指针。当你需要确保资源只被一个指针所拥有，并且需要该指针销毁时自动释放资源，那就可以使用 `std::unique_ptr`。

声明 `std::unique_ptr` 的方式有以下几种：

```
#include <memory> // have to include this head file
std::unique_ptr<MyClassA> myPtr1; // empty pointer
std::unique_ptr<MyClassB> myPtr2(new MyClassB()); // using new
std::unique_ptr<MyClassC> myPtr3 = std::make_unique<MyClassC>(); // only works after C++14
```

尽管 `std::unique_ptr` 声明类似于指针，但其不需要 `delete` 来回收，如果使用 `delete` 反而可能会报错。如果需要提前释放，可以使用 `std::unique_ptr` 的 `reset` 方法。

std::shared_ptr

与之前的独占指针不同，`std::shared_ptr` 的作用就是提供一个共享的（但是指向同一个对象）的指针。`std::shared_ptr` 的一个强大功能是它内部使用*引用计数*(reference count)来跟踪有多少个 `std::shared_ptr` 指向同一个资源。当最后一个 `std::shared_ptr` 被销毁时，资源将会被释放。

`std::shared_ptr` 的声明与 `std::unique_ptr` 类似。在 C++14 中可以使用更加安全的 `std::make_share` 方法，在更早的版本中就使用 `new` 即可。

`std::shared_ptr` 的一个问题是它可能会导致*循环引用*（reference cycle）。循环引用是指两个或多个对象通过指针或引用相互持有对方，形成一个闭环的情况。在使用智能指针，特别是 `std::shared_ptr` 的环境下，循环引用可能导致内存泄漏。例如下面的例子。

```cpp
class B;  // forward declaration

class A {
public:
    std::shared_ptr<B> bPtr;
    ~A() { std::cout << "A:destroyed\n"; }
};

class B {
public:
    std::shared_ptr<A> aPtr;
    ~B() { std::cout << "B:destroyed\n"; }
};

int main() {
    auto a = std::make_shared<A>();
    auto b = std::make_shared<B>();
    a->bPtr = b;
    b->aPtr = a;
    return 0;
}
```

在这个例子中，两个对象 a 和 b 通过 std::shared_ptr 互相“持有”对方。当函数 main() 结束时，尽管 a 和 b 的作用域已经结束，但是它们管理的对象不会被销毁，因为都还被对方持有，引用计数没有归零，它们的析构函数永远不会被调用。

在这种情况下，我们就要借用下面的 std::weak_ptr 来解决。

std::weak_ptr

std::weak_ptr 不拥有对象的所有权，仅用来观察由 std::shared_ptr 管理的对象，它不计入 std::shared_ptr 的引用计数。

std::weak_ptr 不能像前两个智能指针一样直接创建，而是要从一个 std::shared_ptr 或是另一个 std::weak_ptr 中创建出来。

```cpp
#include <memory> // have to include this head file
std::shared_ptr<MyClassC> myPtr = std::make_unique<MyClassC>(); // only works after C++14
std::weak_ptr<MyClass> weakPtr (sharedPtr);
```

如果要使用 std::weak_ptr 所指向的对象，必须要先临时将 std::weak_ptr 升格为 std::shared_ptr。这通常是通过 std::weak_ptr.lock() 函数来实现的。

```cpp
if (std::shared_ptr<MyClass> tempPtr = weakPtr.lock()) {
    // now we can safely use tempPtr
} else {
    // the object pointed to has been destroyed, no shared_ptr created from if
}
```

这样我们就比较完整地回顾了 C++ 中智能指针的使用方式了。

C.4.2 auto 关键字

尽管在 C++03 中就已经有 `auto` 关键字，但它的语义已经完全改变。在 C++11 中 `auto` 用于让编译器自动判断等号右边的值的类型。

C.4.3 nullptr 关键字

在 C 与 C++ 的早期版本中，空指针通常由 `0` 指代，有的时候被强制转化为 `(void*)` 或者 `(char*)`。由于 C/C++ 的隐式整型转换，这可能会导致数据类型的安全问题。所以，在 C++11 中，增加了 `nullptr` 关键字，用于专门指代空指针。

C.4.4 移动语义和右值引用

C++11 中引用的**移动语义**（move semantics）和**右值引用**（rvalue reference）是对语言能力的显著扩展。它们允许更高效的资源管理，特别是在涉及临时对象和重资源对象（如大型容器或者独占资源）的情况下。

右值引用

在 C/C++ 中，左值（lvalue）对应的是实际的 CPU 寄存器或者内存的存储位置，右值（rvalue）对应的是一个临时数据，逻辑上存在但并不一定需要占据任何内存。例如简单的 `int a = 7;` 中，`a` 指的是其在栈上的实际位置，而 `7` 则没有占据任何的内存空间。

移动语义

在 C++11 中，我们可以通过 `&&` 来表示一个变量是右值引用，标识其为可以被移动的临时对象。传统的复制操作中，如果我们需要复制的对象是一个临时变量，那么我们需要创建一个副本，先将这个临时变量复制进内存，再从内存上把复制出来的值复制到目标上。这样做的原因是因为没有右值引用。在右值引用可用后，我们可以标记右值，然后直接将临时变量的值移动到目标上，这就是**移动**（move）语义。

下面给一个案例理解这两个概念。考虑一个包含大型的缓冲区（Buffer）的类。

```
class Buffer {
public:
    Buffer(size_t size) : data(new int[size]), size(size) {}
    ~Buffer() { delete[] data; }

    // move constructor
    Buffer(Buffer&& other) : data(other.data), size(other.size) {
        other.data = nullptr;
        other.size = 0;
    }

    // move operator
    Buffer& operator=(Buffer&& other) {
        if (this != &other) {
            delete[] data;
            data = other.data;
            size = other.size;
            other.data = nullptr;
```

```
            other.size = 0;
        }
        return *this;
    }

    // prohibit copy constructor and copy operator
    Buffer(const Buffer&) = delete;
    Buffer& operator=(const Buffer&) = delete;

private:
    int* data;
    size_t size;
};
```

在这个例子中，Buffer 类包含一个动态分配的数组。它的移动构造函数和移动赋值运算符“窃取”了源对象的资源（在这里是指针和大小），然后将源对象置于空状态（指针为 nullptr，大小为 0）。这比复制整个数据数组要高效得多。为了防止深拷贝和强制移动语义，我们也禁用了复制构造函数和复制赋值运算符。

C.4.5 迭代器及基于范围的 for 循环

迭代器

尽管**迭代器**（iterator）并不是 C++11 的新特性，但是也是 C++ 中非常常用且重要的一个功能。实际上，功能这个词描述地并不准确，迭代器是一种面向对象设计模式，用来提供对容器（比如数组、列表、映射等）中元素的访问，不需要暴露容器内部的表示，它与 C++ 语言本身其实没有强关联，但这也是 C++ 封装特性的体现。

通过 begin() 函数，我们可以获得一个容器的第一个元素（的迭代器），而 end() 则返回最后一个。最常见的使用场景就是 for 循环中的迭代。

```
std::vector<int> vec = {1,2,3,4,5};
for(std::vector<int>::iterator it = vec.begin(); it != vec.end(); it++) {
    std::cout << *it << endl;
}
```

在这个例子中，我们也看到了迭代器的另外两个特性。迭代器可以通过运算符 ++ 或--来访问上一个或者下一个元素，也可以通过解引用运算符 * 来访问迭代器当前指向的元素。

基于范围的 for 循环

C++11 提供了类似于“foreach”语义的循环方式。如果容器拥有有效的 begin() 和 end() 函数，则我们可以通过: 运算符来进行循环的遍历。

```
for (auto element : container) {
    // do something with element
}

for (auto it = container.begin(); it != container.end(); ++it) {
    auto element = *it;
    // do something with element
}
```

可以对比一下上述两种循环方式。

C.5 STL 库数据结构

C++ 提供了一个 STL 库，在这个库中也已经实现了一些基础的数据结构。它们有着各自的优势以及应用场景，在这里我们回顾一下最重要的几个。

C.5.1 `std::vector<T>`

首先是向量类型。

应用场景：适用于需要快速随机访问元素且元素数量频繁变动的场景，如存储元素的列表或数组。
实现原理：动态数组。
性能：

随机访问：$O(1)$。

插入/删除：尾部元素均摊 $O(1)$，中间元素 $O(n)$。

扩容：

机制：当向量的当前容量不足以容纳新元素时，它通常会重新分配一个更大的内存块，大小通常是当前容量的两倍（这可能因实现而异）。

性能：扩容涉及复制或移动现有元素到新的内存位置，因此是一个 $O(n)$ 的操作。但由于扩容不频繁发生（因为每次扩容都增加相当多的额外空间），所以向量的 `push_back` 操作的均摊（amortized）时间复杂度仍然是 $O(1)$。

C.5.2 `std::list<T>`

双向链表类型。

应用场景：适用于需要频繁在序列中间插入或删除元素，而不关心随机访问性能的场景。
实现原理：双向链表。
性能：

随机访问：$O(n)$。

插入/删除：$O(1)$。

扩容：

机制：由于是链表实现的，在扩容时不需要像 `std::vector` 那样重新分配整个数据块。

性能：因为不需要复制整个容器的内容。插入新元素的时间复杂度通常是 $O(1)$。

C.5.3 `std::deque<T>`

双端队列类型。

应用场景：适用于需要频繁在序列中间插入或删除元素，而不关心随机访问性能的场景。
实现原理：双端队列，通常实现为一系列动态数组。
性能：

随机访问：$O(n)$。

插入/删除：头部尾部均摊 $O(1)$。

扩容:

机制：由于是由多个小块连续内存组成的，在扩容时不需要像 `std::vector` 那样重新分配整个数据块。

性能：因为不需要复制整个容器的内容。插入新元素的时间复杂度通常是 $O(1)$。

C.5.4 `std::map<Key, Value>`

映射类型。

应用场景：适合于需要有序数据结构并且经常进行查找、插入和删除操作的场景。

实现原理：平衡二叉树（通常是红黑树）。

性能:

查找/插入/删除：$O(\log n)$。

扩容:

机制：基于树（通常是红黑树）的结构在添加新元素时，通过树的调整来保持平衡。没有所谓的”重新分配整个容器”这一说。

性能：插入新元素的时间复杂度通常是 $O(\log n)$。

C.5.5 `std::unordered_map<Key, Value>`

无序映射类型。

应用场景：适用于需要快速访问（平均情况下）且数据顺序不重要的场景。

实现原理：哈希表。

性能:

查找/插入/删除：平均 $O(1)$，最坏 $O(n)$。

扩容:

机制：这些基于哈希表的结构会在负载因子（即元素数量与底层数组大小的比率）超过某个阈值时进行扩容，通常是通过创建一个更大的底层数组，并重新散列所有元素。

性能：扩容的时间复杂度是 $O(n)$，但由于它发生的频率较低，所以平摊时间复杂度仍然是 $O(1)$。

C.5.6 `std::set<T>`

有序集合类型。

应用场景：适用于需要存储不重复元素且保持元素有序的场景。

实现原理：红黑树。

性能:

查找/插入/删除：$O(\log n)$。

扩容:

机制：基于树（通常是红黑树）的结构在添加新元素时，通过树的调整来保持平衡。没有所谓的”重新分配整个容器”这一说。

性能：插入新元素的时间复杂度通常是 $O(\log n)$。

C.5.7 `std::stack<T, Container>`

后进先出（LIFO）的栈类型。

应用场景：适用于需要后进先出操作的场景，如在算法（如递归算法）中存储临时数据。。
性能扩容：性能及扩容依赖于它所使用的底层容器（如 `std::deque` 或 `std::list`）。

C.5.8 `std::queue<T, Container>`

先进先出（FIFO）的队列类型。

应用场景：适用于需要后进先出操作的场景，如在算法（如递归算法）中存储临时数据。。
性能扩容：性能及扩容依赖于它所使用的底层容器（如 `std::deque` 或 `std::list`）。

附录 D

99 行光线追踪

以下是来自 Kevin Beason[1]的 99 行光线追踪的代码。

```
#include <math.h>   // smallpt, a Path Tracer by Kevin Beason, 2008
 #include <stdlib.h> // Make : g++ -O3 -fopenmp smallpt.cpp -o smallpt
 #include <stdio.h>  //        Remove "-fopenmp" for g++ version < 4.2
 struct Vec {        // Usage: time ./smallpt 5000 && xv image.ppm
   double x, y, z;                  // position, also color (r,g,b)
   Vec(double x_=0, double y_=0, double z_=0){ x=x_; y=y_; z=z_; }
   Vec operator+(const Vec &b) const { return Vec(x+b.x,y+b.y,z+b.z); }
   Vec operator-(const Vec &b) const { return Vec(x-b.x,y-b.y,z-b.z); }
   Vec operator*(double b) const { return Vec(x*b,y*b,z*b); }
   Vec mult(const Vec &b) const { return Vec(x*b.x,y*b.y,z*b.z); }
   Vec& norm(){ return *this = *this * (1/sqrt(x*x+y*y+z*z)); }
   double dot(const Vec &b) const { return x*b.x+y*b.y+z*b.z; } // cross:
   Vec operator%(Vec&b){return Vec(y*b.z-z*b.y,z*b.x-x*b.z,x*b.y-y*b.x);}
 };
 struct Ray { Vec o, d; Ray(Vec o_, Vec d_) : o(o_), d(d_) {} };
 enum Refl_t { DIFF, SPEC, REFR };  // material types, used in radiance()
 struct Sphere {
   double rad;       // radius
   Vec p, e, c;      // position, emission, color
   Refl_t refl;      // reflection type (DIFFuse, SPECular, REFRactive)
   Sphere(double rad_, Vec p_, Vec e_, Vec c_, Refl_t refl_):
     rad(rad_), p(p_), e(e_), c(c_), refl(refl_) {}
   double intersect(const Ray &r) const { // returns distance, 0 if nohit
     Vec op = p-r.o; // Solve t^2*d.d + 2*t*(o-p).d + (o-p).(o-p)-R^2 = 0
     double t, eps=1e-4, b=op.dot(r.d), det=b*b-op.dot(op)+rad*rad;
     if (det<0) return 0; else det=sqrt(det);
     return (t=b-det)>eps ? t : ((t=b+det)>eps ? t : 0);
   }
 };
 Sphere spheres[] = {//Scene: radius, position, emission, color, material
   Sphere(1e5, Vec( 1e5+1,40.8,81.6), Vec(),Vec(.75,.25,.25),DIFF),//Left
   Sphere(1e5, Vec(-1e5+99,40.8,81.6),Vec(),Vec(.25,.25,.75),DIFF),//Rght
   Sphere(1e5, Vec(50,40.8, 1e5),     Vec(),Vec(.75,.75,.75),DIFF),//Back
```

[1]Kevin Beason - smallpt: Global Illumination in 99 lines of C++.

```
  Sphere(1e5, Vec(50,40.8,-1e5+170), Vec(),Vec(),           DIFF),//Frnt
  Sphere(1e5, Vec(50, 1e5, 81.6),    Vec(),Vec(.75,.75,.75),DIFF),//Botm
  Sphere(1e5, Vec(50,-1e5+81.6,81.6),Vec(),Vec(.75,.75,.75),DIFF),//Top
  Sphere(16.5,Vec(27,16.5,47),       Vec(),Vec(1,1,1)*.999, SPEC),//Mirr
  Sphere(16.5,Vec(73,16.5,78),       Vec(),Vec(1,1,1)*.999, REFR),//Glas
  Sphere(600, Vec(50,681.6-.27,81.6),Vec(12,12,12),  Vec(), DIFF) //Lite
};
inline double clamp(double x){ return x<0 ? 0 : x>1 ? 1 : x; }
inline int toInt(double x){ return int(pow(clamp(x),1/2.2)*255+.5); }
inline bool intersect(const Ray &r, double &t, int &id){
  double n=sizeof(spheres)/sizeof(Sphere), d, inf=t=1e20;
  for(int i=int(n);i--;) if((d=spheres[i].intersect(r))&&d<t){t=d;id=i;}
  return t<inf;
}
Vec radiance(const Ray &r, int depth, unsigned short *Xi){
  double t;                               // distance to intersection
  int id=0;                               // id of intersected object
  if (!intersect(r, t, id)) return Vec(); // if miss, return black
  const Sphere &obj = spheres[id];        // the hit object
  Vec x=r.o+r.d*t, n=(x-obj.p).norm(), nl=n.dot(r.d)<0?n:n*-1, f=obj.c;
  double p = f.x>f.y && f.x>f.z ? f.x : f.y>f.z ? f.y : f.z; // max refl
  if (++depth>5) if (erand48(Xi)<p) f=f*(1/p); else return obj.e; //R.R.
  if (obj.refl == DIFF){                  // Ideal DIFFUSE reflection
    double r1=2*M_PI*erand48(Xi), r2=erand48(Xi), r2s=sqrt(r2);
    Vec w=nl, u=((fabs(w.x)>.1?Vec(0,1):Vec(1))%w).norm(), v=w%u;
    Vec d = (u*cos(r1)*r2s + v*sin(r1)*r2s + w*sqrt(1-r2)).norm();
    return obj.e + f.mult(radiance(Ray(x,d),depth,Xi));
  } else if (obj.refl == SPEC)            // Ideal SPECULAR reflection
    return obj.e + f.mult(radiance(Ray(x,r.d-n*2*n.dot(r.d)),depth,Xi));
  Ray reflRay(x, r.d-n*2*n.dot(r.d));     // Ideal dielectric REFRACTION
  bool into = n.dot(nl)>0;                // Ray from outside going in?
  double nc=1, nt=1.5, nnt=into?nc/nt:nt/nc, ddn=r.d.dot(nl), cos2t;
  if ((cos2t=1-nnt*nnt*(1-ddn*ddn))<0)    // Total internal reflection
    return obj.e + f.mult(radiance(reflRay,depth,Xi));
  Vec tdir = (r.d*nnt - n*((into?1:-1)*(ddn*nnt+sqrt(cos2t)))).norm();
  double a=nt-nc, b=nt+nc, R0=a*a/(b*b), c = 1-(into?-ddn:tdir.dot(n));
  double Re=R0+(1-R0)*c*c*c*c*c,Tr=1-Re,P=.25+.5*Re,RP=Re/P,TP=Tr/(1-P);
  return obj.e + f.mult(depth>2 ? (erand48(Xi)<P ?   // Russian roulette
    radiance(reflRay,depth,Xi)*RP:radiance(Ray(x,tdir),depth,Xi)*TP) :
    radiance(reflRay,depth,Xi)*Re+radiance(Ray(x,tdir),depth,Xi)*Tr);
}
int main(int argc, char *argv[]){
  int w=1024, h=768, samps = argc==2 ? atoi(argv[1])/4 : 1; // # samples
  Ray cam(Vec(50,52,295.6), Vec(0,-0.042612,-1).norm()); // cam pos, dir
  Vec cx=Vec(w*.5135/h), cy=(cx%cam.d).norm()*.5135, r, *c=new Vec[w*h];
#pragma omp parallel for schedule(dynamic, 1) private(r)       // OpenMP
  for (int y=0; y<h; y++){                       // Loop over image rows
    fprintf(stderr,"\rRendering (%d spp) %5.2f%%",samps*4,100.*y/(h-1));
    for (unsigned short x=0, Xi[3]={0,0,y*y*y}; x<w; x++)   // Loop cols
      for (int sy=0, i=(h-y-1)*w+x; sy<2; sy++)     // 2x2 subpixel rows
        for (int sx=0; sx<2; sx++, r=Vec()){        // 2x2 subpixel cols
          for (int s=0; s<samps; s++){
```

```
            double r1=2*erand48(Xi), dx=r1<1 ? sqrt(r1)-1: 1-sqrt(2-r1);
            double r2=2*erand48(Xi), dy=r2<1 ? sqrt(r2)-1: 1-sqrt(2-r2);
            Vec d = cx*( ( (sx+.5 + dx)/2 + x)/w - .5) +
                    cy*( ( (sy+.5 + dy)/2 + y)/h - .5) + cam.d;
            r = r + radiance(Ray(cam.o+d*140,d.norm()),0,Xi)*(1./samps);
          } // Camera rays are pushed ^^^^^ forward to start in interior
          c[i] = c[i] + Vec(clamp(r.x),clamp(r.y),clamp(r.z))*.25;
        }
  }
  FILE *f = fopen("image.ppm", "w");         // Write image to PPM file.
  fprintf(f, "P3\n%d %d\n%d\n", w, h, 255);
  for (int i=0; i<w*h; i++)
    fprintf(f,"%d %d %d ", toInt(c[i].x), toInt(c[i].y), toInt(c[i].z));
}
```

附录 E

表面形状的实现

E.1 Surface 与 SurfaceBase 类

E.2 Sphere 类

E.3 Quad 类

E.4 Triangle 类

附录 F

基于物理渲染的类库

本附录将提出一个基于达特茅斯的 DIRT(Dartmouth Introductory Ray Tracer) 渲染器[1]的软件类库，以帮助在计算机程序中生成这样的路径。这篇附录不提供具体的实现，旨在展示类之间的关系以及宏观视角下的构建思路。

F.1 概要

整个 DIRT 需要实现以下的基本功能。

- 数据结构。这里包括向量、矩阵、变换等数据结构的存储、使用以及对应计算的实现。
- 几何表达。这里包括射线、球、四边形、网格等不同类型基础几何的表达。
- 数据读取。这里包括 (.json) 文件的读取、进程进度的显示、纹理解析等。
- 数据采样。这里包括各类采样器、采样函数的实现。
- 材质贴图。这里包括各类材质以及纹理贴图的定义。
- 物理渲染。这里通过积分器提供光线追踪的各类实现方式。
- 通用功能。这里包括一些不便于归类的杂项函数实现以及一些通用的功能，例如计时器、质数表等。

其中的所有类的分布以及继承关系可以见图 E.1。该图中，虚线框表示该内容是一个结构体而不是类。实心框表示该内容是一个头文件内定义的多个函数的集合。

下面的几个小节里我们详细介绍一下每个类的任务、支持的关键函数以及实现建议。再次声明，本笔记不提供具体的实现，正文中提到的实现也是参考实现方式，并不代表正确答案或是最优解。

[1]Copyright © 2019, Wojciech Jarosz

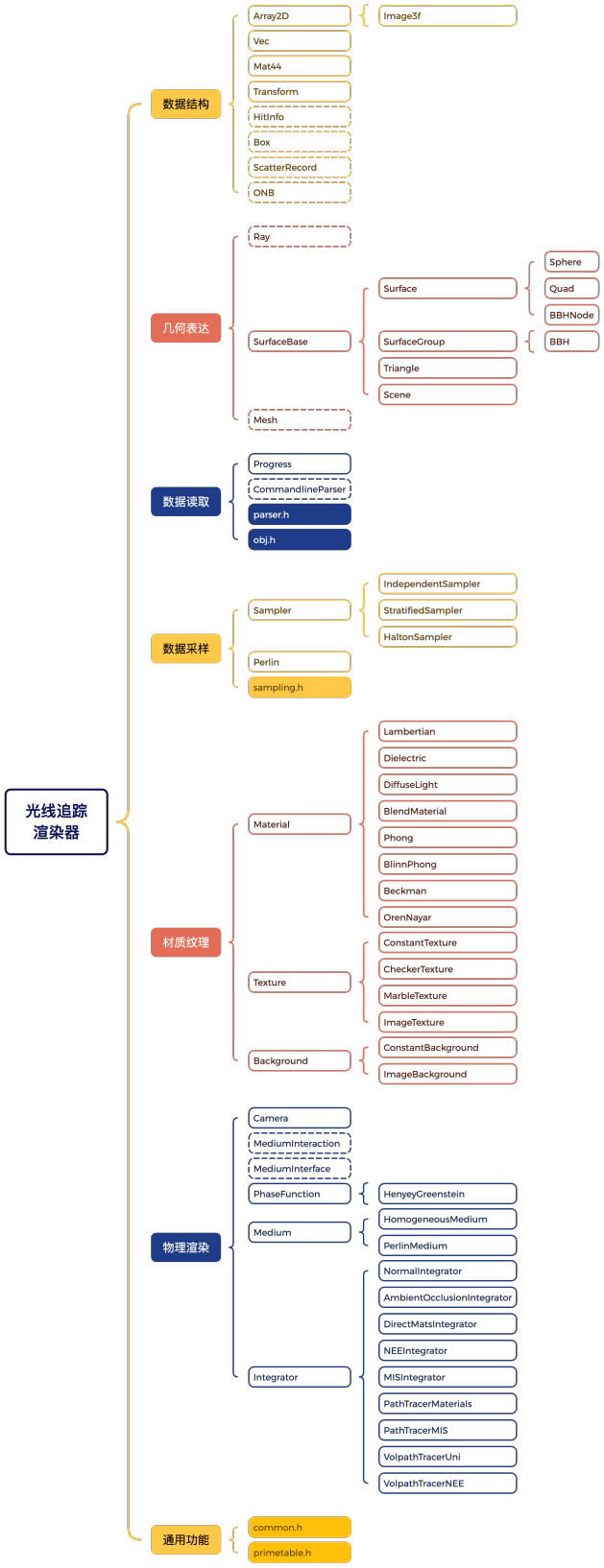
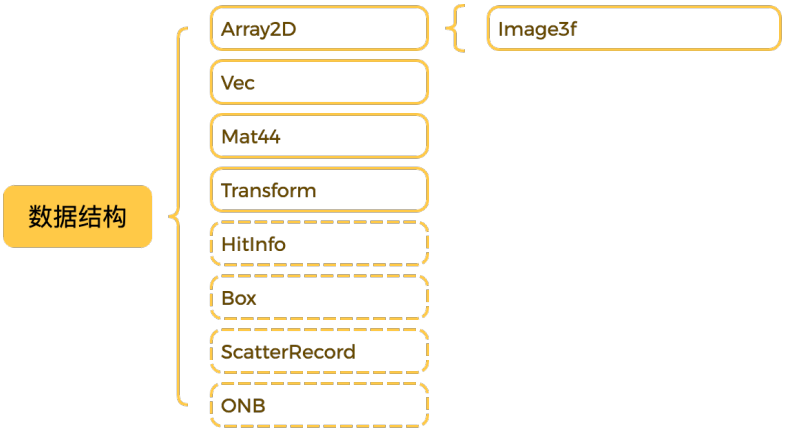


图 F.1: DIRT 的类与继承关系图

F.2 数据结构

数据结构不仅需要支持保存必要的数据，也需要有相应的必要功能支持。



F.2.1 Vec 类

Vec 类是用来保存向量相关类型的。我们建议向量使用模板泛型处理，因为无论向量的单个维度是什么类型，它们的方法实现原理是相同的，具体实现可以参考 2.2。

在 PBR 的语境下，向量与 C++ 标准库提供的向量有所不同。PBR 中的向量并没有动态保存数组的意义，它在绝大部分的使用场景下都是二、三、四维向量中的一种，而不是一种需要支持无限长度拓展和自动扩容的数据结构。因此，给予特定长度、类型的向量一个单独的别名有助于提高代码的可读性。

```
template <typename T> using Vec2 = Vec<2, T>;
template <typename T> using Vec3 = Vec<3, T>;
template <typename T> using Vec4 = Vec<4, T>;

template <typename T> using Color3 = Vec<3, T>;
template <typename T> using Color4 = Vec<4, T>;

using Vec2f = Vec2<float>;
using Vec2d = Vec2<double>;
using Vec2i = Vec2<std::int32_t>;
using Vec2u = Vec2<std::uint32_t>;
using Vec2c = Vec2<std::uint8_t>;

using Vec3f = Vec3<float>;
using Vec3d = Vec3<double>;
using Vec3i = Vec3<std::int32_t>;
```

```
using Vec3u = Vec3<std::uint32_t>;
using Vec3c = Vec3<std::uint8_t>;
using Color3f = Vec3<float>;
using Color3d = Vec3<double>;
using Color3u = Vec3<std::uint32_t>;
using Color3c = Vec3<std::uint8_t>;

using Vec4f = Vec4<float>;
using Vec4d = Vec4<double>;
using Vec4i = Vec4<std::int32_t>;
using Vec4u = Vec4<std::uint32_t>;
using Vec4c = Vec4<std::uint8_t>;
using Color4f = Vec4<float>;
using Color4d = Vec4<double>;
using Color4u = Vec4<std::uint32_t>;
using Color4c = Vec4<std::uint8_t>;
```

注意到我们不仅将各长度的向量用 `Vec` 实现，`Color` 我们也是用 `Vec` 实现的。这是很合理的，毕竟颜色本来就是一个三维（或四维，如果包含透明度的话）的向量。因此，对于 `Vec` 类，我们应该提供基于 $xyzw$ 语境的 `swizzle` 方法，也应该提供基于 $rgba$ 语境的 `swizzle` 方法。

以四维向量、`xyz` 和 `rgba` 方法为例。

```
template <typename T> struct Vec<4, T> {
    union {
        std::array<T, 4> e;
        struct {
            T x, y, z, w;
        };
        struct {
            T r, g, b, a;
        };
        Vec<3, T> xyz;
        Vec<3, T> rgb;
        Vec<2, T> xy;
    };

    constexpr Vec() = default;
    constexpr explicit Vec(T e0) : x(e0), y(e0), z(e0), w(e0) {}
    constexpr Vec(T e0, T e1, T e2, T e3) : x(e0), y(e1), z(e2), w(e3) {}
    constexpr Vec(const Vec<3, T> xyz, T _w) : x(xyz.x), y(xyz.y), z(xyz.z), w(_w) {}

    T operator[](size_t i) const { return e[i]; }
    T &operator[](size_t i) { return e[i]; }

    static inline Vec Zero() { return Vec(T(0)); }
    static inline Vec UnitX() { return Vec(T(1), T(0), T(0), T(0)); }
    static inline Vec UnitY() { return Vec(T(0), T(1), T(0), T(0)); }
    static inline Vec UnitZ() { return Vec(T(0), T(0), T(1), T(0)); }
    static inline Vec UnitW() { return Vec(T(0), T(0), T(0), T(1)); }
};
```

向量类需要支持的方法包括但不局限于：

- *算数*。包括基本的加减、标量乘法、点乘、叉乘。
- *规模*。包括向量的长度、长度的平方。特别地，对于颜色类向量，我们还应该提供**亮度**（luminance）功能。

```
inline float luminance(const Color3f &c) {
    return c[0] * 0.212671f + c[1] * 0.715160f + c[2] * 0.072169f;
}
```

- *索引*。返回向量的第 i 个位置的值，或是根据 `swizzle` 操作返回由向量的部分维度组成的更短的、或是维度顺序不同向量。

F.2.2 Mat44 类

与 `Vec` 类相似，对于矩阵的数据类型，在 PBR 的语境下，也只有存储不同数据类型的 4×4 矩阵的意义而已。因此，我们也可以给 `Mat44` 提供一些类别名。

```
using Mat44f = Mat44<float>;
using Mat44d = Mat44<double>;
```

4×4 矩阵类需要支持的方法包括但不局限于：

- *算数*。包括基本的加减、矩阵标量乘法、4×4 矩阵与 4×4 矩阵的乘法、转置、求逆、行列式。
- *索引*。返回矩阵的第 (i, j) 个位置的值，或是以向量形式返回矩阵第 i 行，或是第 j 列。

F.2.3 Transform 类

我们知道在图形学中，变换（transform）和变换矩阵实际上是通常互相指代的两个词。变换的本质也就是左乘变换矩阵。第二章中我们也已经对 `Transform` 类做过详细的介绍。

变换类需要提供的方法包括但不限于：

- *求逆*。这个方法通常要求 `Transform` 类维护一个变换矩阵的逆矩阵，然后在使用逆变换函数 `inverse()` 时通过构造函数返回一个以逆矩阵为变换矩阵，当前矩阵为新的逆矩阵的 `Transform` 对象。
- *对基本几何表达形式的变换*。几何表达我们会放在下一个小节展开，在这里我们提供几个函数。

```
/// Apply the homogeneous transformation to a 3D vector
Vec3f vector(const Vec3f &v) const {
    // Implementation OMIT
}

/// Apply the homogeneous transformation to a 3D normal
Vec3f normal(const Vec3f &n) const {
    // Implementation OMIT
}

/// Transform a point by an arbitrary matrix in homogeneous coordinates
Vec3f point(const Vec3f &p) const {
    // Implementation OMIT
}
```

```
15
16 /// Apply the homogeneous transformation to a ray
17 Ray3f ray(const Ray3f &r) const {
18     // Implementation OMIT
19 }
20
21 /// Transform the axis-aligned Box and return the bounding box of the result
22 Box3f box(const Box3f &box) const {
23     // Implementation OMIT
24 }
```

F.2.4 Array2D 类与 Image3f 类

Array2D 类是一个通用的、可以扩容的二维数组类型，我们应该将其视作为一个 $w \times h$ 的表格，其中 w 是宽度，h 是高度。基于这样的想法，二维数组类型也应该用泛型模板实现，且应该提供以下的基本功能：

- 返回维度。维度本身最好作为私有变量，防止暴露在外被篡改。可以使用 `width()` 和 `height()` 函数返回宽高。
- 调整容量。应该要支持将二维数组本身进行容量的调整。我们可以用 C++ 标准库的向量类型来实现一个 Array2D 的数据库，因为 C++ 标准库的向量本身提供了高效的容量调整功能。
- 索引。通过某种方式范围处于第 (i, j) 格上的信息。也许也可以支持以向量形式返回第 i 行的信息，亦或是第 j 列的信息。
- 重置。提供 `reset()` 函数以便于快速清空或是快速将该二维数组填充成默认数值。

显然，在 PBR 的上下文中，Array2D 最重要的作用就是用来实现图像的类型。图像本身应该是一种 Color3f 的二维数组，但是图像本身应该支持更多的一些其它功能，例如读取来自某路径文件的、特定格式的图像。

```
1 class Image3f : public Array2d<Color3f> {
2     using Base = Array2d<Color3f>;
3     // OMIT Image3f functions
4 }
```

F.2.5 HitInfo 和 ScatterRecord 结构体

这两个数据结构以结构体（struct）的形式出现，因为它们经常需要在计算的过程中以参数传入，接受函数过程的修改。在这种情况下，我们通常将它们放在内存栈，使得它们的分配会更快，销毁过程更简单，以提高渲染的性能。

HitInfo 结构体的作用是记录光线与物体表面求交的结果。因此，其至少应该维护以下的信息：

- 相交情况。虽然可以简单地通过一个布尔值来返回是否相交，但是我们建议记录相交时刻射线的参数 t。这样可以帮助我们判断射线与物体相交的位置。
- 交点。如果相交的话，我们显然经常会用到交点。
- 法线。相交处的法线。根据实现的功能，也许不仅需要提供几何法线（geometric normal），也需要提供插值后的着色用法线（shading normal）。

- 纹理坐标。相交处的纹理坐标。
- 材质。相交处物体表面的材质，用以调用采样或是取值函数。
- 介质。相交处所处的参与介质，用以调用采样或是相位函数。
- 物体表面对象。相交物体本身的表面对象。

`ScatterRecord` 结构体的作用是记录一次散射操作的结果。这次结果包括散射方向、散射结果所带的颜色，以及散射是否是镜面的（如果是镜面的，则可能无法调用 `pdf()` 函数）。

F.2.6 `Box` 结构体

F.2.7 `ONB` 结构体

F.3 几何表达

F.4 数据解析

F.5 数据采样

F.6 材质纹理

F.7 通用功能