Computer Graphics Notes 计算机图形学笔记

Lingheng Tony Tao

2024 年 3 月 1 日

目录

前言

这份笔记旨在帮助复习**计算机图形学**的相关知识点，主要参考对象为 CMU 的课程 15-462 (Computer Graphics, Spring 2020) 以及 GAMES 公开课 GAMES 101（现代计算机图形学入门）。

在这份笔记中，参考书目如下，推荐各位购买及阅读：

- York, S. U. M. I. N., & Shirley, P. (2022). *Fundamentals of Computer Graphics: International Student Edition.*
- Pharr, M., Jakob, W., & Humphreys, G. (2023). *Physically Based Rendering, fourth edition: From Theory to Implementation.* MIT Press.
- Gregory, J. (2018). *Game Engine Architecture.* A K PETERS.
- Hearn, D., Baker, M. P., & Carithers, W. R. (2011). *Computer Graphics with OpenGL. Pearson.*
- Prata, S. (2011). *C++ Primer plus.* Addison-Wesley Professional.
- 高等数学（下册），第 8 版，同济大学出版社
- 孙家广，胡事民，(2009)．计算机图形学基础教程，第二版，清华大学出版社

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，并反馈至邮箱 linghent @ andrew.cmu.edu。本笔记目前仅供 CMU ETC 同学内部使用，请勿外传。

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

Chapter 1

基本变换

对于计算机图形学而言，最基础的部分就是各类的变换。这一章节主要去研究一些最基本的变换以及它们的操作方式。

1.1 线性变换

在图形学中，当我们说到**变换**（Transform）这个词的时候，我们基本上都可以认为这是一个通过矩阵相关数学完成的操作。变换一词本身指的是一种映射关系，将一个向量映射到另一个向量上的操作就可以称为变换。在没有上下文的情况下，我们都将一个 n 维向量看做一个*列向量*，也就是一个 $n \times 1$ 的一个矩阵。下面，我们给线性变换和仿射变换下一个不严格的定义。

定义（线性变换）一个 n 维列向量通过左乘一个 $n \times n$ 的矩阵得到另一个 n 维列向量的变换过程叫做**线性变换**（linear transformation）。

定义（仿射变换）一个 n 维列向量通过左乘一个 $n \times n$ 的矩阵，并再加上另一个 n 维列向量，得到另一个 n 维列向量的变换过程叫做**仿射变换**（affine transformation）。

我们很快将看到线性变换和仿射变换在计算机图形中的基本应用。

1.1.1 2D 缩放

我们仅在最基础的 **2D 缩放**（2D Scale）中提供变换矩阵的推导过程。实际上，另外的线性变换都可以仿照类似的步骤，因此此处也不再赘述。

对于二维平面上的点 (x, y)，我们将其沿着 x 轴缩放 s_x 倍，将其沿着 y 轴缩放 s_y 倍，就能得到一个新的点 $(s_x x, s_y y) = (x', y')$。

$$\begin{cases} x' = s_x x \\ y' = s_y y \end{cases}$$

注意到，我们可以通过一个矩阵来完成这个操作。

令

$$S_a = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$$

我们就有

$$S_a \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x \\ s_y y \end{bmatrix} = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

由此可见，二维缩放操作可以写成一个二维列向量左乘一个 2×2 的矩阵，所以二维缩放操作是一个线性变换。

下面我们将快速地浏览其它的线性变换。推导的过程同上，请读者自行推导。

1.1.2 2D 错切

沿着 x 轴的 **2D 错切** (2D Shear) 矩阵为

$$H_x = \begin{bmatrix} 1 & Sh_x \\ 0 & 1 \end{bmatrix}$$

而沿着 y 轴的 2D 错切矩阵为

$$H_y = \begin{bmatrix} 1 & 0 \\ Sh_y & 1 \end{bmatrix}$$

其中，Sh_x, Sh_y 为最大偏移量。

1.1.3 2D 旋转

围绕原点旋转 θ 度的 **2D 旋转** (2D Rotation) 矩阵为

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

1.1.4 3D 缩放

与 2D 缩放几乎没有区别。

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

1.1.5 3D 错切

这里提供一个沿着 x 方向的 3D 错切矩阵。其它两个方向的矩阵举一反三即可。

$$H_x(dy, dz) = \begin{bmatrix} 1 & dy & dz \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

1.1.6 3D 旋转

3D 旋转比起 2D 旋转会复杂很多，之后也会有专门的一个章节来讨论 3D 旋转。但是在这里，如果只考虑绕着一根轴旋转的情况，我们也就可以简单地改写 2D 的旋转矩阵。

$$R_{x,\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_{y,\theta} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_{z,\theta} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

注意，仅在 y 轴上的旋转上，第一行的 $\sin\theta$ 是不带负号的，这是由于右手坐标系旋向性。

1.2 仿射变换

首先注意到 **2D 平移**（2D Translate）的变换逻辑。

$$x' = x + x_t$$
$$y' = y + y_t$$

虽然这个变换逻辑非常简单直接，但是它并没有办法被写作一个矩阵乘以一个向量的形式。我们只能将其写作一个矩阵加一个列向量的形式。

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_t \\ y_t \end{bmatrix}$$

根据定义，2D 平移是一个仿射变换。类似地，3D 平移也是一个仿射变换。如此简单的变换却不能通过矩阵相乘的方式来处理，这就会出现一个很大的问题。为了解决这个问题，我们提出了**齐次坐标**（homogeneous coordinates）的概念。

定义（齐次坐标）对于一个 d 维向量 $\mathbf{p}$，对其增加一个维度 $w \in \mathbb{R}^+$ 获得 $\mathbf{p}_H$，并满足：

- 如果 $\mathbf{p}= (x_1, x_2, ..., x_d)$ 是一个点，$\mathbf{p}_H = (x_1, x_2, ..., x_d, w)$，其中 $w \neq 0$。
- 如果 $\mathbf{p}= (x_1, x_2, ..., x_d)$ 是一个向量，$\mathbf{p}_H = (x_1, x_2, ..., x_d, 0)$.

那么，这里 $\mathbf{p}_H$ 就被叫做 $\mathbf{p}$ 的**齐次坐标**。另外，我们认为 $\forall w \neq 0$,

$$(x_1, x_2, ..., x_d, w) \equiv (x_1/w, x_2/w, ..., x_d/w, 1)$$

它们对应的都是点 $\mathbf{p} = (x_1/w, x_2/w, ..., x_d/w)$.

这里，我们观察到齐次坐标的几个很好的特性。

- 在齐次坐标下，我们可以区分点和向量。
- 使用齐次坐标时，点 + 向量得到的结果其 $w = 1$，因此依然是一个点。
- 使用齐次坐标时，向量 + 向量得到的结果其 $w = 0$，因此依然是一个向量。
- 使用齐次坐标时，点 + 点得到的结果其 $w = 2$，归一化后得到两点的中点。

利用齐次坐标的这些性质，我们就可以将仿射变换转换为线性变换了。

1.2.1 2D 平移-齐次坐标

注意到，将点 $\mathbf{p}=(x,y)$ 改变成为齐次坐标之后，2D 平移就可以成为一个线性变换了。

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_t \\ y + y_t \\ 1 \end{bmatrix}$$

那么，对向量 $\mathbf{p}=(x,y)$ 用同样的操作，会得到同样的结果吗？答案是否定的。

$$\begin{bmatrix} x' \\ y' \\ 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix}$$

但是是正确的！向量不应该因为平移而改变，这是向量的性质。

1.2.2 其它变换在齐次坐标下

非常类似地，我们也可以将 **3D 平移**改为在 4D 齐次坐标下的线性变换。

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + x_t \\ y + y_t \\ z + z_t 1 \end{bmatrix}$$

我们可以去如法炮制其它的变换的齐次坐标形态。要注意到，齐次坐标形式下，变换矩阵的最后一行总是 $\begin{bmatrix} 0 & 0 & ... & 1 \end{bmatrix}$，因此在代码中实现的时候，我们可以将变换矩阵存储为 $n \times n-1$ 的矩阵，在参与计算的时候再考虑把这一行加回去，以节省空间资源。

1.3 组合变换

当我们对一个 n 维列向量施加变换之后，得到的结果依然是一个 n 维列向量——这就意味着我们可以再一次对结果进行变换矩阵的左乘，去施加另一个变换。对于 n 维列向量 $\mathbf{p}$，假设我们对其使用了变换矩阵 $\mathbf{A}$ 进行第一次变换，再对结果使用了变换矩阵 $\mathbf{B}$ 进行了第二次变换，最终所得到的结果 $\mathbf{p'}$ 应该满足

$$\mathbf{p}' = \mathbf{B}(\mathbf{A}\mathbf{p})$$

根据矩阵乘法的结合律，我们知道

$$\begin{aligned}\mathbf{p}' &= \mathbf{B}(\mathbf{A}\mathbf{p}) \\ &= (\mathbf{B}\mathbf{A})\mathbf{p}\end{aligned}$$

因此，只要我们将所有的变换矩阵按顺序从右到左依次相乘，我们就可以将最终的结果直接与 $\mathbf{p}$ 相乘，其结果与依次进行这些变换是等效的。这个过程就是**组合变换**（Composition of transformations）。

1.4 逆变换

注意到，如果一个变换矩阵可逆，那么我们就可以通过再乘这个逆矩阵将点恢复为变换前的样子。

$$\mathbf{p} = \mathbf{I}\mathbf{p} = \mathbf{M}^{-1}\mathbf{M}\mathbf{p}$$

所以，如果我们要逆转一个变换，就可以求这个变换的变换矩阵的逆矩阵。

回顾一下，在线性代数中，方阵可逆的充要条件是行列式不为 0。在图形学中，我们常见的 4×4 的矩阵有很多都是可逆的。但也依然会存在不可逆的矩阵，例如将 3D 坐标投射到 2D 坐标的变换。

1.5 法线变换

虽然变换可以保存很多的几何性质，例如*切线*——如果一个向量 $\mathbf{t}$ 在变换前与一个表面相切，在变换后依然会是变换后的表面的切线。但是另一个性质却不一定能保证——*法线*。法线在变换前与面垂直，在变换后却不一定可以保持这个性质。

假设这个变换为 $\mathbf{M}$。我们想寻求变换 $\mathbf{N}$，使得对法线使用 $\mathbf{N}$ 之后，结果可以依然与 $\mathbf{M}$ 变换后的表面相垂直。我们可以通过*纯代数*的方法来解决这个问题。

首先，我们知道，对于法线 $\mathbf{n}$，其与表面的任何切线都是相垂直的，即点积为 0。

$$\mathbf{n}^{\mathrm{T}}\mathbf{t} = \mathbf{0}$$

然后，记变换后的切向量 $\mathbf{t}_{\mathbf{M}} = \mathbf{M}\mathbf{t}$，记变换后的法向量 $\mathbf{n}_{\mathbf{N}} = \mathbf{N}\mathbf{n}$。我们实际上要做的就是找到能使得 $\mathbf{n}_{\mathbf{N}}^{\mathrm{T}}\mathbf{t}_{\mathbf{M}} = \mathbf{0}$ 的矩阵 $\mathbf{N}$。注意到，

$$\begin{aligned}\mathbf{n}^{\mathrm{T}}\mathbf{t} &= \mathbf{n}^{\mathrm{T}}\mathbf{I}\mathbf{t} \\ &= \mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}\mathbf{M}\mathbf{t} \\ &= \mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}\mathbf{t}_{\mathbf{M}} \\ &= \mathbf{0}\end{aligned}$$

因此，我们可以令 $\mathbf{n}_{\mathbf{N}} = \mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1}$。这样我们就有

$$\mathbf{n}_{\mathbf{N}} = (\mathbf{n}^{\mathrm{T}}\mathbf{M}^{-1})^{\mathrm{T}} = (\mathbf{M}^{-1})^{\mathrm{T}}\mathbf{n}$$

因此，一个能够将法向量保持垂直的变换就是原变换*逆矩阵的转置矩阵*，即上述的 $(\mathbf{M}^{-1})^{\mathrm{T}}$。

1.6 3D 旋转

3D 旋转也是线性变换的一种。在 1.1.6 中我们提到围绕三个轴的旋转，由于旋转是线性变换，我们也可以直接将 $R_{x,\theta_x}, R_{y,\theta_y}, R_{z,\theta_z}$ 相乘获得一个通用形式。这个通用形式过于冗长，并没有什么记住的必要。需要使用的时候调用刚才提到的三个矩阵相乘即可。

注意到 3D 旋转的两个重要的特征。首先，3D 旋转**并不是**可交换的。例如，

- 在 2D 旋转中，旋转 20° 再旋转 40° $\Leftrightarrow$ 旋转 40° 再旋转 20°。
- 在 3D 旋转中，绕着 x 轴旋转 20° 再绕着 y 轴旋转 40° $\nLeftrightarrow$ 绕着 y 轴旋转 40° 再绕着 x 轴旋转 20°。

这是合理的。矩阵本身也并不符合交换律，上面交换顺序则意味着同时也置换了变换矩阵的顺序，自然是可能会出问题的。

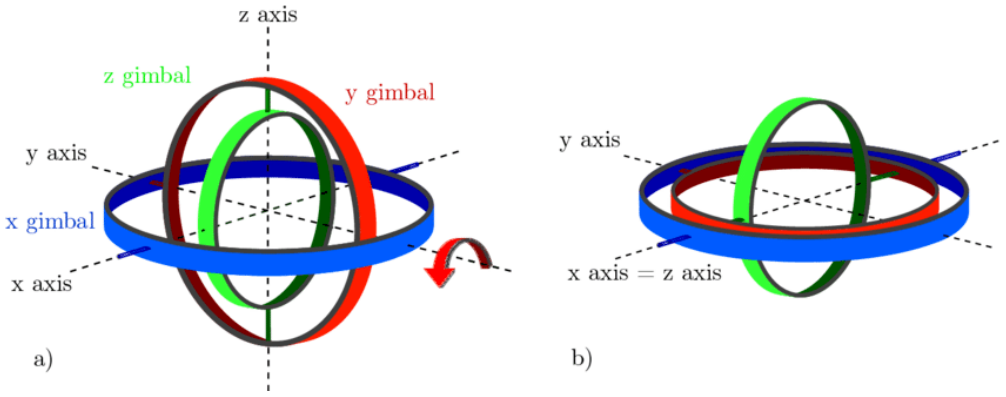
第二个问题是著名的**万向节死锁**（Gimbal Lock）的问题。

我们现在使用的利用 $(\theta_x, \theta_y, \theta_z)$ 来确定旋转的方法被称作**欧拉角**（Euler angle）。然而，如同我们上面看到的，这三个数值必须明确指明哪个顺序在最先，哪个其次，哪个是最后。实际上不同的应用场景中，同样的数对可能会表示不同旋转。例如，在航空中，我们常用的顺序是 ZYX 顺序，也就是先围绕 z 轴旋转——也叫做**偏航**（yaw），然后是围绕着 y 轴旋转——也叫做**俯仰**（pitch），最后才是围绕着 x 轴旋转——也叫做**翻滚**（roll）。

现在，假设我们在 y 轴上的旋转是 $\theta_y = \pi/2$，也就是说 $\cos\theta_y = 0, \sin\theta_y = 1$。此时，我们按照航空顺序，进行三维旋转。

$$
\begin{aligned}
R_x R_y R_z|_{\theta_y=\pi/2} &= \begin{bmatrix} \cos\theta_y\cos\theta_z & -\cos\theta_y\sin\theta_z & \sin\theta_y \\ \cos\theta_z\sin\theta_x\sin\theta_y + \cos\theta_x\sin\theta_z & \cos\theta_x\cos\theta_z - \sin\theta_x\sin\theta_y\sin\theta_z & -\cos\theta_y\sin\theta_x \\ -\cos\theta_x\cos\theta_z\sin\theta_y + \sin\theta_x\sin\theta_z & \cos\theta_z\sin\theta_x + \cos\theta_x\sin\theta_y\sin\theta_z & \cos\theta_x\cos\theta_y \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & 1 \\ \cos\theta_z\sin\theta_x + \cos\theta_x\sin\theta_z & \cos\theta_x\cos\theta_z - \sin\theta_x\sin\theta_z & 0 \\ -\cos\theta_x\cos\theta_z + \sin\theta_x\sin\theta_z & \cos\theta_z\sin\theta_x + \cos\theta_x\sin\theta_z & 0 \end{bmatrix} \\
&= \begin{bmatrix} 0 & 0 & 1 \\ \sin(\theta_x+\theta_z) & \cos(\theta_x+\theta_z) & 0 \\ -\cos(\theta_x+\theta_z) & \sin(\theta_x+\theta_z) & 0 \end{bmatrix}
\end{aligned}
$$

此时，无论如何调节 θ_x 和 θ_z，最终的旋转总是在同一个平面内。下图[1]可以帮助一个形象的理解。



在这里，我们绕着 y 轴旋转 $\pi/2 = 90°$ 后，红色的环和蓝色的环就处在同一个平面内了。此时，无论是绕着 x 轴旋转，还是绕着 z 轴旋转，都只能实际上围绕着 x 轴转动。实际上，此时 x 轴和 z 轴已经是同一根轴了。这也叫做**自由度**（DoF, Degree of Freedom）的损失。这在动画领域和航空领域都容易产生致命问题，也因此，我们提出了四元数的概念。

1.6.1 四元数

四元数（Quaternion）在外形上看起来像一个四维向量，但实际上是完全不同的东西。它作为复数的延伸，最早用以解决力学的问题。一个四元数由 4 个部分组成，其中有一个实部与三个虚部。对于一个四元数 q（四元数通常用非斜非粗的字体表示），它可以表示为

$$\mathrm{q} = a + bi + cj + dk$$

其中，

- $a, b, c, d \in \mathbb{R}$;
- $i^2 = j^2 = k^2 = ijk = -1$;
- $ij = k, ji = -k$;
- $jk = i, kj = -i$;
- $ki = j, ik = -j$.

对于上述的四元数 q, 它的模长 $|\mathrm{q}| = \sqrt{a^2 + b^2 + c^2 + d^2}$。如果模长为 1，它也被称作**单位四元数**。另外，如果将 q 写作向量形式，我们将其写作 (b, c, d, a)。

定义（三维旋转）绕旋转轴为 $\mathbf{a}$ 的 θ 角度旋转可以表示为唯一四元数 $\mathrm{q} = (\mathbf{q}_v = (q_x, q_y, q_z), q_r)$，其中

- $q_r = \cos(\theta/2)$;
- $\mathbf{q}_v = (q_x, q_y, q_z) = \mathbf{a}\sin(\theta/2)$

这里的 q 可以表示为 $(\mathbf{a}\sin\frac{\theta}{2}, \cos\frac{\theta}{2})$，即一个向量和一个标量的数对。旋转的方向使用**右手定则**，即右手大拇指指向旋转轴的方向，四个手指的方向为旋转的方向。

[1] https://www.researchgate.net/figure/llustrates-the-principle-of-gimbal-lock-The-outer-blue-frame-represents-the-x-axis-the_fig4_338835648

四元数乘法

对于给定的两个四元数 p 和 q，假设它们分别代表了旋转 $\mathbf{P}$ 和 $\mathbf{Q}$，那么 pq，即两个四元数的乘积，就代表了了它们的合成旋转 $\mathbf{PQ}$。虽然有很多种不同的四元数乘法，这边我们主要考虑与三维旋转有关的**格拉斯曼积**。

定义（格拉斯曼积）对于四元数 p=($\mathbf{p}_v$, p_s) 和四元数 q=($\mathbf{q}_v, q_s$)，它们的**格拉斯曼积**（Grassman Product）定义为

$$(p_s\mathbf{q}_v + q_s\mathbf{p}_v + \mathbf{p}_v \times \mathbf{q}_v, p_s q_s - \mathbf{p}_v \cdot \mathbf{q}_v)$$

共轭与逆

四元数另外一个重要的概念就是它的逆四元数。为了了解逆四元数的概念，首先得先了解共轭四元数的概念。

定义（共轭）对于四元数 p=($\mathbf{q}_v$, q_s)，其**共轭**（conjugate）四元数为 $\bar{\mathrm{q}}$=(-$\mathbf{q}_v$, q_s)。

定义（逆）对于四元数 q=($\mathbf{q}_v$, q_s)，其**逆**（inverse）四元数为 $\mathrm{q}^{-1} = \frac{\bar{\mathrm{q}}}{|\mathrm{q}|^2}$。

由于在三维旋转中，通常都是单位四元数，因此 $|\mathrm{q}|^2 = 1$。也就是说 $\bar{\mathrm{q}} = \mathrm{q}^{-1}$。

积的逆与共轭有相似的性质。

$$\overline{(\mathrm{pq})} = \overline{\mathrm{qp}}$$

$$(\mathrm{pq})^{-1} = \mathrm{q}^{-1}\mathrm{p}^{-1}$$

四元数旋转向量 v

以四元数 q 旋转向量 $\mathbf{v}$，先用 q 前乘向量 $\mathbf{v}$，再后乘逆四元数 q^{-1}。即

$$\mathbf{v}' = \mathrm{rotate}(\mathrm{q}, \mathbf{v}) = \mathrm{q}\mathbf{v}\mathrm{q}^{-1}$$

综合以上的特点，我们看到四元数提供了一种不依赖于特定旋转顺序的方式来表示三维旋转，也因此四元数可以避免万向节死锁的问题。另外，四元数还有一个很好的特性，就是便于**插值**。关于插值的定义和算法，我们会在第四章、第五章以及第十章中深入。

1.6.2 任意旋转

注意到，无论是 2D 还是 3D 旋转，它们的矩阵都是正交归一（orthonormal）矩阵，也就是每个行向量（或列向量）之间互相垂直，且每行（或每列）的模长都为 1。正交归一矩阵 $\mathbf{M}$ 的一个很好的特性就是 $\mathbf{M}^{\mathrm{t}} = \mathbf{M}^{-1}$，即逆为转置。

观察正交归一矩阵 $\mathbf{R}_{uvw} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix} = \begin{bmatrix} -\mathbf{u}- \\ -\mathbf{v}- \\ -\mathbf{w}- \end{bmatrix}$，这里，$\mathbf{u} = x_u\mathbf{x} + y_u\mathbf{y} + z_u\mathbf{z}$，$\mathbf{v}$ 和 $\mathbf{w}$ 类似。注意到，

$$\mathbf{R}_{uvw}\mathbf{u} = \begin{bmatrix} \mathbf{u}\cdot\mathbf{u} \\ \mathbf{v}\cdot\mathbf{v} \\ \mathbf{w}\cdot\mathbf{w} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \mathbf{x}$$

类似地，$\mathbf{R}_{uvw}\mathbf{v} = \mathbf{y}, \mathbf{R}_{uvw}\mathbf{w} = \mathbf{z}$。换句话说，我们可以认为 $\mathbf{R}_{uvw}$ 将轴 **uvw** 旋转到了轴 **xyz** 上。由于 $\mathbf{R}_{uvw}$ 是正交归一矩阵，我们也可以认为 $\mathbf{R}_{uvw}^{\mathrm{T}}$ 将轴 **xyz** 旋转到了轴 **uvw** 上。

因此，如果我们要绕着一个任意轴 **a** 旋转，我们就可以以 **w=a** 建立一个正交坐标系，然后将其旋转至标准坐标系 **xyz** 轴下，围绕 z 轴旋转，再将其转回 **uvw** 坐标系。

Chapter 2

观测变换

在上一章中，我们已经学习了关于变换的基本知识。本章中，我们将应用变换的相关知识，完成将三维物体变换至二维屏幕上的准备工作。

2.1 视图变换

所谓的渲染就是将某个东西“看到的”内容显示在屏幕上。在不同的语境中，我们默认相机与人眼是同一个东西。为了确定一个相机能看到的内容，需要以下几个参数。

- 位置：$\mathbf{e}$
- 看向：$\hat{\mathbf{g}}$
- 相机朝上的方向：$\hat{\mathbf{t}}$

我们假设有一个绝对的坐标系，世界有一个原点，以及三个轴 $\mathbf{x}$ =(1,0,0), $\mathbf{y}$ = (0,1,0), $\mathbf{z}$ = (0,0,1)，这样的坐标系就是**世界空间**。上面的 $\mathbf{e}$，$\hat{\mathbf{g}}$，$\hat{\mathbf{t}}$ 都是针对世界空间来说的。

为了方便描述相机看到的内容，我们先将所有物体变换到**相机空间**中。这是一个对于世界空间内的所有物体都适用的变换，因此我们也只需要考虑一个变换矩阵即可。在这个空间中，相机位于原点，看向**-z**，朝上方向为 $\mathbf{y}$。我们可以只考虑将相机变换到这个位置的矩阵。

第 1 步 平移**-e**。很简单，构建一个齐次平移矩阵就行。

$$\mathbf{T}_{-\mathbf{e}} = \begin{bmatrix} 1 & 0 & 0 & -x_{\mathbf{e}} \\ 0 & 1 & 0 & -y_{\mathbf{e}} \\ 0 & 0 & 1 & -z_{\mathbf{e}} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{-e} \\ \mathbf{0} & 1 \end{bmatrix}$$

第 2 步 将 $\hat{\mathbf{t}}$ 旋转至 $\mathbf{y}$，将 $\hat{\mathbf{g}}$ 旋转至**-z**。

这一步就是一个任意旋转的问题。我们将原来的 $\{\hat{\mathbf{g}} \times \hat{\mathbf{t}}, \hat{\mathbf{t}}, -\hat{\mathbf{g}}\}$ 坐标轴转移至 $\{\mathbf{x}, \mathbf{y}, \mathbf{z}\}$ 坐标轴上即可。这是我们在 1.6.2 中提及过的任意旋转问题。考虑这个变换的逆变换，即将 $\mathbf{x}$ 旋转至 $\hat{\mathbf{g}} \times \hat{\mathbf{t}}$，将 $\mathbf{y}$ 旋转至 $\hat{\mathbf{t}}$，

将 $\mathbf{z}$ 旋转至 $-\hat{\mathbf{g}}$。也就是，

$$\mathbf{R}^{-1} = \begin{bmatrix} x_{\mathbf{g}\times\mathbf{t}} & x_{\mathbf{t}} & x_{\text{-}\mathbf{g}} & 0 \\ y_{\mathbf{g}\times\mathbf{t}} & y_{\mathbf{t}} & y_{\text{-}\mathbf{g}} & 0 \\ z_{\mathbf{g}\times\mathbf{t}} & z_{\mathbf{t}} & z_{\text{-}\mathbf{g}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

由于 $\mathbf{R} = (\mathbf{R}^{-1})^{-1} = (\mathbf{R}^{-1})^{\mathrm{T}}$，因此

$$\mathbf{R} = \begin{bmatrix} x_{\mathbf{g}\times\mathbf{t}} & y_{\mathbf{g}\times\mathbf{t}} & z_{\mathbf{g}\times\mathbf{t}} & 0 \\ x_{\mathbf{t}} & y_{\mathbf{t}} & z_{\mathbf{t}} & 0 \\ x_{\text{-}\mathbf{g}} & z_{\mathbf{t}} & z_{\text{-}\mathbf{g}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{g}\times\mathbf{t} & \mathbf{t} & \text{-}\mathbf{g} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1}$$

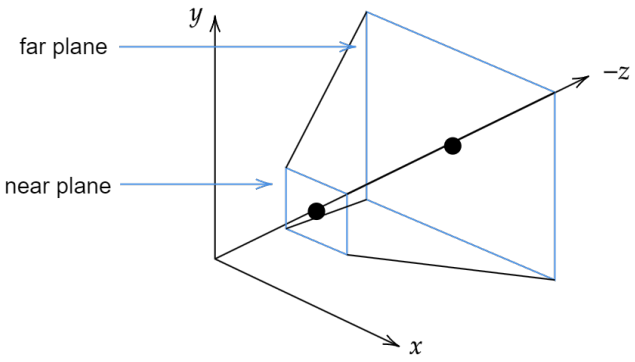
所以，将物体从世界空间转换至相机空间的矩阵 $\mathbf{M}_{cam}$ 为

$$\mathbf{M}_{cam} = \mathbf{R} \cdot \begin{bmatrix} \mathbf{I} & \text{-}\mathbf{e} \\ \mathbf{0} & 1 \end{bmatrix}$$

这个矩阵也叫做**视图矩阵**（View Matrix），在很多 API 中，这个矩阵被记作 $\mathbf{V}$。

2.2 投影变换

现实生活中我们常常说近大远小，这是一种透视观点。所谓的近大远小，形象上来说是说越靠近相机，能看见的物体越少。所以，相机可以被抽象成一个被称为**视锥体**（Viewing Frustum）的概念。



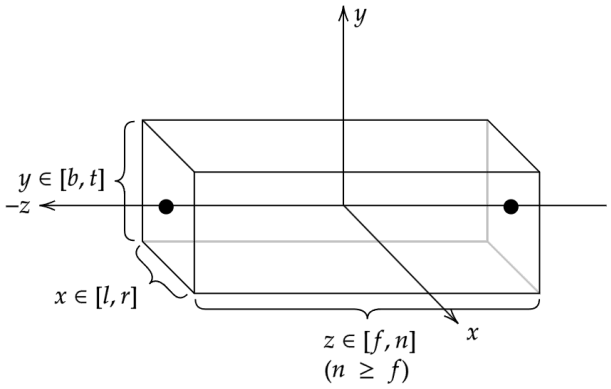
在这个锥体中，离相机更近的那一个平面叫做**近平面**（near plane），远的那个则叫**远平面**（far plane）。这样的相机抽象是比较符合人眼实际看到的世界的。将三维世界以透视的方式投射到纸面上的过程就叫做**透视投影**（Perspective Projection）。但是，在视锥体中，平行的属性并不能被保存——平行线在透视的情况下看起来是会相交的。

在工程制图中，我们也会经常使用正交视图。在这种视图下，平行线依然是平行的，也并没有近大远小的特征（也可以认为是近平面与远平面大小相同的视锥体）。这种投影方式就被叫做**正交投影**（Orthogonal Projection）。

显然，无论是正交投影还是透视投影，不处于视锥体内的物体都要被**剔除**（Cull），因为相机根本看不到它们，我们不应该浪费算力去计算这些不处于视锥体内的物体。因此，在投影变换这个过程中，我们就应该重点关注两个部分：投影和剔除。

2.2.1 正交投影

我们先从概念上更简单的正交投影开始，也就是视锥体是个四方体的情况。我们规定，在**裁剪空间**（Clipping Space）中，视锥体的中心处于原点。对于任何一个视锥体内的点 $\mathbf{P} = (x, y, z)$，$x \in [l, r], y \in [b, t], z \in [f, n]$。为了方便后续计算，我们甚至可以进一步规定我们的视锥体是一个棱长为 2 的立方体，也就是 $x, y, z \in [-1, 1]$。转换至这个标准化视锥体内的坐标也被叫做**归一化设备坐标**（Normalized Device Coordinate，NDC）。



需要注意的点是因为相机是看向**-z** 的，因此实际上近平面比远平面拥有更大的 z 值。我们要将一个处于相机空间内的点转换到裁剪空间上，只需要服从下面的两步。

第一步 将中心移动到原点。
第二步 整个盒子压缩至 $[-1, 1]^3$。

这个过程其实就是两个矩阵，一个平移，一个缩放。

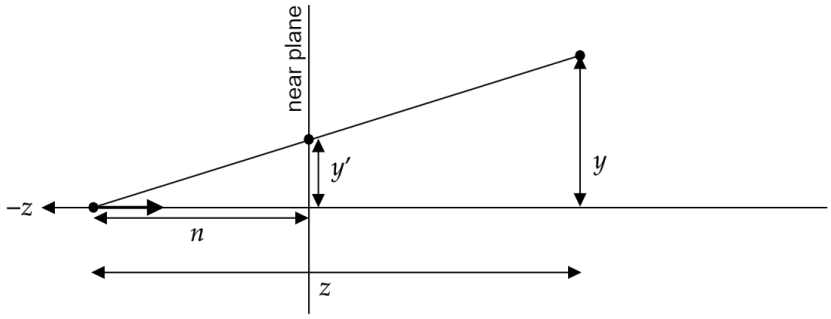
$$\begin{aligned}\mathbf{M}_{orth} &= \mathbf{S}\mathbf{T}_{-\mathbf{c}} \\ &= \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}\end{aligned}$$

这个矩阵被称作**正交投影矩阵**（Orthogonal Projection Matrix），我们通常也用 $\mathbf{P}_{orth}$ 来指代。

2.2.2 透视投影

直观来说，透视投影中唯一不同就是视锥体的不同。所以，如果有一个变换能够将视锥体的盒子“压缩”成正交透视的标准化视锥体，那透视也就没有什么难点了。

在开始讨论具体的透视之前，我们先回顾齐次坐标的定义。我们记得，在齐次坐标中，点 $\mathbf{p}=(x,y,z,1)$ 与点 $\mathbf{q}=(xz,yz,z^2,z)$ 表示的是同一个点。



从侧面观察透视，视锥体呈现的是上图这样的情况。处于视锥体内的任何一点 (x,y,z) 与相机连接后，其在近平面上的交点设为 (x',y',n)，其中 n 是近平面的 z 坐标。根据三角形的相似的，我们有

$$\frac{y'}{y}=\frac{n}{z}\Longrightarrow y'=\frac{ny}{z}$$

类似地，我们也可以得到 $x'=\frac{nx}{z}$ 的结论。这两个点在压缩以后，它们应该有着相同的 xy 坐标。因此，对于任何一点 $(x,y,z,1)$，它在视锥体压缩后的坐标应该都是 $(nx/z,ny/z,X,1)$，其中 X 未知。假设我们有一个变换矩阵 $\mathbf{M}_{pers}$，我们应该有

$$\mathbf{M}_{pers}\begin{bmatrix}x\\y\\z\\1\end{bmatrix}=\begin{bmatrix}nx/z\\ny/z\\X\\1\end{bmatrix}$$

另外，在齐次坐标下，$(nx/z,ny/z,X,1)\equiv(nx,ny,nX,z)$，所以，我们也可以认为矩阵 $\mathbf{M}_{pers}$ 将任意点 $(x,y,z,1)$ 变换至 (nx,ny,nX,z)。

$$\mathbf{M}_{pers}\begin{bmatrix}x\\y\\z\\1\end{bmatrix}=\begin{bmatrix}nx\\ny\\zX\\z\end{bmatrix}$$

首先，无疑，矩阵 $\mathbf{M}_{pers}$ 拥有如下形式：

$$\mathbf{M}_{pers}=\begin{bmatrix}n&0&0&0\\0&n&0&0\\A&B&C&D\\0&0&0&1\end{bmatrix}$$

否则，无法完成上述的线性变换。那么接下来，我们只需要代入特殊值求解即可。首先观察在近平面上的点，它依然要被映射到近平面上同一点。即 $(x,y,n,1)$ 在变换后应该成为 (nx,ny,n^2,n)。

$$Ax+By+Cn+D=n^2\Rightarrow A=0,B=0,Cn+D=n^2$$

另外观察远平面上一点 $(0,0,f,1)$，它在变换后应该成为 $(0,0,f^2,f)$。

$$A\cdot 0+B\cdot 0+Cf+D=f^2\Rightarrow Cf+D=f^2$$

根据上述两个方程求解，我们就可以得出 $C=n+f,D=-nf$ 了。因此，透视正交化矩阵的形式为

$$\mathbf{M}_{pers}=\begin{bmatrix}n&0&0&0\\0&n&0&0\\0&0&n+f&-nf\\0&0&0&1\end{bmatrix}$$

完成这一步操作之后，视锥体就被变换成了经典正交盒子。之后，我们再左乘 $\mathbf{P}_{orth}$，就可以得到**透视投影矩阵**（perspective projection matrix）了。这个矩阵通常也被指代为 $\mathbf{P}_{pers}=\mathbf{P}_{orth}\mathbf{M}_{pers}$，或者在上下文清晰时，直接简写为 $\mathbf{P}$。

2.3 模型变换

目前，我们已经完成了将物体从世界空间转换至相机空间，再从相机空间转换至裁剪空间的操作。然而，很多时候，我们拿到的物体是定义在自己的局部坐标中的。如果是这样，在开始之前应该还有一个步骤，就是将物体从局部空间转换到世界空间。这个转换被我们称为**模型变换**（Model Transform），通常用 $\mathbf{M}$ 指代。

假设物体在世界坐标中位置在 $\mathbf{t}=(t_x,t_y,t_z)$，物体的三个轴分别为 $\mathbf{u}$，$\mathbf{v}$，$\mathbf{w}$，物体坐标系相对于世界坐标系的缩放为 $\mathbf{s}=(s_x,s_y,s_z)$，我们遵循缩放——旋转——平移的顺序进行变换，因此，我们得到 $\mathbf{M}$ 的形式如下：

$$\mathbf{x}=\mathbf{M}\mathbf{u}$$

$$\begin{bmatrix}x\\y\\z\\1\end{bmatrix}=\begin{bmatrix}1&0&0&t_x\\0&1&0&t_y\\0&0&1&t_z\\0&0&0&1\end{bmatrix}\begin{bmatrix}x_u&x_v&x_w&0\\y_u&y_v&y_w&0\\z_u&z_v&z_w&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}s_x&0&0&0\\0&s_y&0&0\\0&0&s_z&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}u\\v\\w\\1\end{bmatrix}$$

其中 $\mathbf{x}$ 和 $\mathbf{u}$ 分别是改点在 **uvw** 坐标系和 **xyz** 坐标系下的齐次坐标。因此，$\mathbf{M}$ 矩阵有时也被称为**参照系标准化矩阵**（frame-to-canonical transform）。

现实中，通常 **uvw** 坐标系和 **xyz** 坐标系缩放一致，在这种情况下我们也可以省去缩放矩阵，然后将 $\mathbf{M}$ 简写为

$$\mathbf{M}=\begin{bmatrix}\mathbf{u}&\mathbf{v}&\mathbf{w}&\mathbf{e}\\0&0&0&1\end{bmatrix}$$

至此，我们就完成了将物体从模型空间转移至裁剪空间的所有操作。一个在模型空间中的点 $\mathbf{p}_M$ 可以通过

$$\mathbf{p}=\mathbf{PVM}\mathbf{p}_M$$

获得。在很多 API 下，我们将 $\mathbf{PVM}$ 矩阵的乘积保存为单独的一个矩阵，称其为 **MVP 矩阵**。

2.4 视场角

对于我们的视锥体，虽然我们可以指定任何的 (l, r, b, t)，但是通常来说我们都会将这个系统简化为我们看向屏幕的中心，也就是说我们总是有 $l = -r, b = -t$。另外，如果我们假设我们的像素都是正方形的，那么我们还会有

$$\frac{n_x}{n_y} = \frac{r}{t}$$

其中 n_x, n_y 分别是水平方向的像素数量和竖直方向的像素数量。一旦 n_x, n_y 确定了，我们就只剩一个自由度了（因为确定了 r 也就确定了唯一的 t）。我们通常通过**视场角**来确定这个自由度。

定义（垂直视场角）**垂直视场角**（Vertical Field-of-view Angle, FOV）θ 可以通过近平面距离 n 和视锥体高度的一半 t 来定义，它满足

$$\tan\frac{\theta}{2} = \frac{t}{|n|}$$

在这种情况下，如果 n 和 θ 确定，那么 t 也就确定了。

视场角固定后，我们可以回到投影矩阵，用视场角来调整其中的一些变量。n 与 f 依然是我们可以调整的数值。我们将 r 也称为 $aspect$，将 t 称为 $size$。因此，在假设我们看向屏幕中心后，正交投影矩阵 $\mathbf{P}_{orth}$ 也可以改写为

$$\begin{bmatrix} \frac{1}{aspect} & 0 & 0 & 0 \\ 0 & \frac{1}{size} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & \frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

而透视投影矩阵 $\mathbf{P}_{pers}$ 就可以改写为

$$\begin{bmatrix} \frac{\cot\frac{\theta}{2}}{aspect} & 0 & 0 & 0 \\ 0 & \cot\frac{\theta}{2} & 0 & 0 \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2nf}{n-f} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Chapter 3

光栅化数学基础

经过了前面对变换的了解，我们成功地将一个物体从物体空间转移到的裁剪空间上。现在，终于到了激动人心的将图像显示在 2D 屏幕上的阶段了，也就是将裁剪空间内的物体进一步地变换到屏幕空间上。这个过程也就是我们所说的*光栅化*。要完全了解光栅化，并且掌握它的优化过程，我们也需要做一些分析。

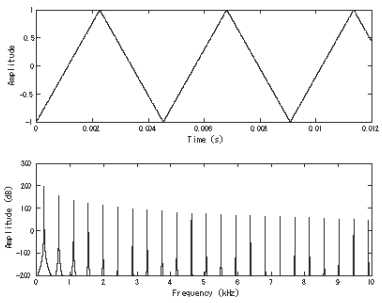
3.1 信号处理

这一章开始的部分乍一看与我们讨论的图像光栅化似乎没有直接的联系。其实不然，我们需要很好地理解*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。

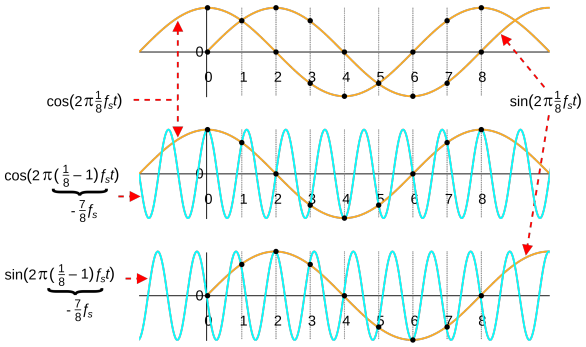
定义（频域）描述物理信号对频率的关系，就是**频域**（Frequency Domain）。在频域分析中，自变量为频率 f，因变量为待分析数据 S。下面的图像[1]中，上为时域信号，下为频域信号。



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

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（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）。



例如，在上图[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。

有了这些基础概念，我们就可以更好地去讨论其中的一些具体的问题了。在信号处理中，我们最常用、最重要的两个工具就是卷积和傅里叶变换。下面我们就详细地关于这两个概念进行讨论。

3.2 卷积

卷积是一种函数之间的操作，通过两个函数，组合它们并获得第三个函数的操作的就称为**卷积**（convolution）。在这里，我们只通过两个案例来解释卷积，一个用来研究连续函数的卷积，另一个用来研究离散函数的卷积。更深入的理解可以自行去搜索相关资料。

3.2.1 卷积示例

移动平均

对于连续函数 g，我们定义函数 $h(x)$ 为 g 的值在 $[x-r, x+r]$ 范围内的平均值。其中，r 被称为**平均半径**，也就是：

$$h(x) = \frac{1}{2r}\int_{x-r}^{x+r} g(t)dt.$$

这里就是一种基本的卷积操作。在更复杂的卷积中，我们可能不是简单的求平均，而更可能是去求一个加权平均值。

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

数列卷积

假设我们有两个离散数列 $a[i]$ 和 $b[i]$。定义它们的卷积 $(a*b)[i]$ 为：

$$(a*b)[i] = \sum_j a[j]b[i-j].$$

这里我们并没有明确给 j 一个取值范围，意思是 j 可以取任何整数。$b[i]$ 定义了 $a[i]$ 在这个累加中的权重。在很多场景下，序列 b 中只会有一小段取值不为 0，且有时就会像上面的移动平均一样，是有一个半径 r 的。这取决于实际的需求。

3.2.2 卷积滤波器

卷积可以做到筛选一部分信号值（如同上面的数列卷积）。因此，我们就可以用它来过滤信号，也就是人们所说的**滤波器**（filter）。滤波器与上面数列卷积中的 b 中提到的类似，通常会只有一小段定义为非零值（这样才能做到“抵消/滤过”不需要的信号数据），同样，滤波器本质也是加权平均的权重。这里我们也不会去深究各种滤波器，选择两个常用的作为案例分析，感兴趣的话可以去翻阅 Fundamentals of Computer Graphics (S Marschner & P. Shirley, 5th Edition)，10.3.1 的部分。

方框滤波器

方框滤波器（Box Filter）是一个片段连续函数，它的积分为 1。它的离散滤波器的形式为

$$a_{\text{box},r}[i] = \begin{cases} 1/(2r+1) & |i| \leq r, \\ 0 & \text{Otherwise.} \end{cases}$$

方框滤波器对当前值以及周围 r 个元素进行简单的平均。这是在图像处理领域中最简单的滤波技术，可以用来平滑数据和图像。

高斯滤波器

高斯滤波器（Gaussian Filter）是另一种用来平滑处理的滤波器。它的滤波器函数为

$$f_g,\sigma(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-x^2/2\sigma^2}.$$

高通滤波器虽然没有有限的半径 r，但是离中心远了之后，值会变得足够小，小到可以忽略。在实际的应用中，尤其是对离散信号使用时，通常也不考虑过小的权重。高斯滤波器非常平滑，最重要的应用的就是我们常说的**高斯模糊**。模糊的本质也是滤波，这件事我们很快就会看到。

3.3 傅里叶变换

[3] 对于上面提到的信号处理的各个问题，我们都是在尝试解决采样和重建中可能出现的情况。如果要严格地从数学层面理解采样中为什么会出现这些问题，我们就得从更基本的内容出发。

上文中我们曾经提及过采样理论中一个很重要的话题就是如何将信号从时域转向频域。在日常生活中，我们大部分直观感受到的都是时域的体现，时域是描述数学函数或物理信号对时间的关系。例如速度（位移

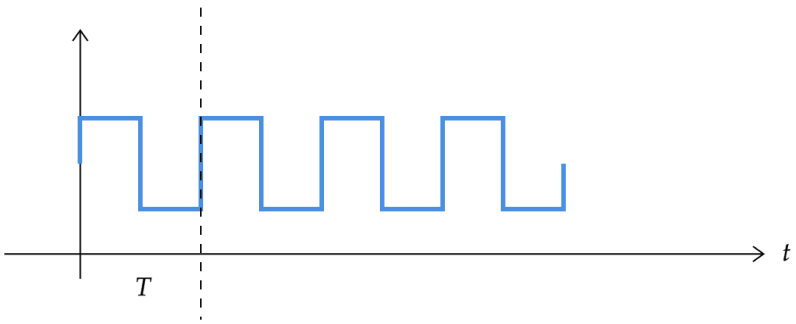
[3]这一章节在各类计算机图形学课程和教材中都被粗略跳过，仅推荐对傅里叶变换原理感兴趣的读者阅读

与时间之比）、通量（通过的物理量与时间之比）、功率（做功与时间的比）等。但是，例如如果我们想要调整一首歌中低音的音量，单纯给我们一首歌从头到尾的波形图是没法快速调整的，我们必须一秒一秒地去找低音所在的位置。如果我们想要快速地调整其中低音（也就是低频段）的音量，我们更希望能拿到声音信号的频率分布图，而不是时间分布图。在现实生活中，我们经常会遇到需要在频域做调整的情况，而非在时域。

3.3.1 正弦函数的理解

高中的时候我们曾经学习过，描述简谐运动的三角函数 $y = A\sin(\omega x + \varphi)$ 中，φ 叫做**初始相位**（Initial phase），A 叫做**振幅**（Amplitude），ω 叫做**角频率**（Angular Frequency）。这个函数也可以写作 $y = A\sin(2\pi f x + \varphi)$，其中 $f = \omega/2\pi$ 叫做**频率**（Frequency）。根据频率的定义，这个函数的**周期**（period）就是 $T = \frac{1}{f} = \frac{2\pi}{\omega}$。

在实际问题中，我们也会常常见到非正弦函数的周期函数，比如信号处理中常说的周期为 T 的*矩形波*。



我们的目标就是用三角函数组成的*级数*（可以理解为无穷长的数列的元素和）来描述非正弦周期函数。*傅里叶变换*[4]的核心思想就是任何信号 $f(t)$ 都可以拆分成全频率正弦或余弦函数波的和（无论这个函数是不是周期函数）。为了便于理解，我们从周期函数出发，那么我们就会有

$$f(t) = A_0 + \sum_{i=1}^{\infty} A_i \sin(i\omega t + \varphi_i). \tag{3.1}$$

其中，A_0, A_i, φ_i 都是常数。傅里叶变换将等号左边的函数转换为等号右边的函数，而后者描述了原始函数中各个频率成分的振幅和相位。因此我们也说，傅里叶变换将信号从*时域*转换到*频域*。

将周期函数按照上述的式子展开，物理意义是非常明确的——我们可以把复杂的周期运动看成是很多不同频率的简谐运动的叠加。在电工学上，这样的展开也被叫做**谐波分析**（harmonic analysis）。在上面的表达式中，A_0 也被叫做**直流分量**（Direct Current component，或简称 DC component），$A_1\sin(\omega t + \varphi_1)$ 也叫做**一次谐波**（first harmonic）或者**基波**（fundamental harmonic）。一般地，$\forall i \in \mathbb{Z}^+, A_i \sin(i\omega t + \varphi_i)$ 就叫做 n 次谐波。

[4]推荐阅读：https://www.zhihu.com/question/19714540/answer/1119070975
[5]推荐阅读：https://www.cnblogs.com/Fish0403/p/16938691.html

3.3.2 三角函数正交性

将正弦函数 $A_i \sin(i\omega t + \varphi_i)$ 按照高中学过的三角公式变形，可以得到

$$A_i \sin(i\omega t + \varphi_i) = A_i \sin\varphi_i \cos(\omega t) + A_i \cos\varphi_i \sin(i\omega t)$$

如果我们令 $A_0 = \frac{a_0}{2}$，$a_i = A_i \sin\varphi_i$，$b_i = A_i \cos\varphi_i$，$\omega = \frac{\pi}{l}$，那么对于 (3.1) 中的 $f(t)$ 的表达式中，等号右边的级数就可以被改写为

$$f(t) = \frac{a_0}{2} + \sum_{i=1}^{\infty} \left(a_i \cos\frac{i\pi t}{l} + b_i \sin\frac{i\pi t}{l} \right)$$

进一步令 $x = \frac{\pi t}{l}$，则可以将上述级数进一步简化为

$$f(x) = \frac{a_0}{2} + \sum_{i=1}^{\infty} (a_i \cos(ix) + b_i \sin(ix)) \tag{3.2}$$

替换 x 之前，$f(t)$ 的周期是 $2\pi/w = 2l$，替换之后，周期就成为了 2π。形如上式的级数就被叫做**三角级数**。

将 2π 周期函数展开成傅里叶级数

假设 $f(x)$ 是周期为 2π 的周期函数，且能展开成为三角级数

$$f(x) = \frac{a_0}{2} + \sum_{i=1}^{\infty} (a_i \cos(ix) + b_i \sin(ix))$$

我们接下俩应该思考的问题就是如何利用 $f(x)$ 将系数 a_0, a_i, b_i 表达出来。我们进一步假设等号右边的级数可以逐项积分。对上式从 $-\pi$ 到 π 积分，由于假设右端都可以逐项积分，我们有

$$\int_{-\pi}^{\pi} f(x)dx = \int_{-\pi}^{\pi} \frac{a_0}{2} dx + \sum_{i=1}^{\infty} \left[(a_i \int_{-\pi}^{\pi} \cos(ix)dx + b_i \int_{-\pi}^{\pi} \sin(ix)dx) \right]$$

为了能解出 a_0，我们必须要先了解三角函数系的一个特征，即正交性。

正交性

所谓的三角函数系，指的就是类似于

$$1, \cos x, \sin x, \cos 2x, \sin 2x, ..., \cos nx, \sin nx, ...$$

这一系列的函数。

定义（正交性）三角函数系的**正交性**（Orthogonality）指的是在区间 $[-\pi, \pi]$ 上，三角函数系的任何两个不同函数正交，即任何两个不同的函数的乘积在区间 $[-\pi, \pi]$ 上的积分等于 0。任意两个相同的函数的乘积在区间 $[-\pi, \pi]$ 上的积分不等于 0。

因此，对于 (3.2) 中的级数展开，我们关注到等号右边除了第一项以外，其余各项全部都是 0。这样，a_0 就很好求了。

$$a_0 = \frac{1}{\pi} \int_{-\pi}^{\pi} f(x)dx.$$

接下来求 a_i，将等式两端的同时乘 $\cos(ix)$，在区间 $[-\pi, \pi]$ 上积分。

$$\int_{-\pi}^{\pi} f(x)\cos(ix)dx = \int_{-\pi}^{\pi} \frac{a_0}{2}\cos(ix)dx + \sum_{k=1}^{\infty}\left[(a_k \int_{-\pi}^{\pi} \cos(ix)\cos(kx)dx + b_k \int_{-\pi}^{\pi} \sin(kx)dx)\right]$$

同样根据正交性，等式右端除了 $k = n$ 的一项外，其他的各项都是 0。因此我们有

$$a_i = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\cos(ix)dx.$$

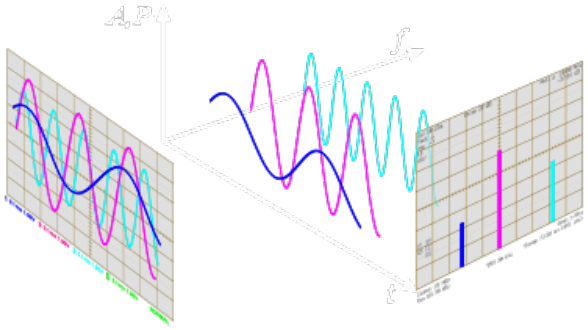
类似地，用 $\sin(ix)$ 去乘等式的两端，我们得到

$$b_i = \frac{1}{\pi}\int_{-\pi}^{\pi} f(x)\sin(ix)dx.$$

将 a_i, b_i, a_0 代入 (3.2) 中等号右端得到的三角级数，就叫做**傅里叶级数**（Fourier Series）。

3.3.3 傅里叶变换

对于周期函数而言，在上下文清晰时，我们可以认为将周期函数展开成傅里叶级数的过程就是周期函数的傅里叶变换。我们已经看到，傅里叶级数展开将一个原来以 t 为自变量的函数拆成了很多简谐运动正弦波的叠加，就如同下图所示[6]。



从时域的视角来看，我们最后看到的是叠加成为 $f(t)$ 的函数图象的一个信号，也就是信号随时间的分布（上图中左边的图象）。从频域的视角来看，我们可以看出这里使用了多少不同频率的正弦波，也就是信号随频率的分布（上图中右边的图象）。所以，这就是为什么我们说傅里叶变换将函数从时域转换到了频域。

然而，现实中更广泛使用的傅里叶变换的强大之处就是在于它能够将非周期性函数从时域转换到频域，维度也不限于一维的函数曲线。简单来说，一个非周期函数也可以看做是周期无限大的函数，之后就是微积分大展拳脚的情境了。这里受篇幅限制[7]，不再具体描述转换的过程和原理，对于图形学而言，我们目前更应该先建立对于频域的直观理解。在二维图像中，我们所说的高频部分指的就是颜色变化较大的部分，也就是边缘和细节；低频部分指的就是图像灰度平稳的区域。

3.4 采样理论

在重新回到渲染之前我们还需要做的最后一步理论基础建设就是**采样理论**（Sampling Theory）了。理解采样和走样在图形学中的语义就能方便我们理解渲染、抗锯齿等一系列操作的本质了。

[6] https://www.radartutorial.eu/10.processing/sp53.en.html

[7] 未来有时间再详细地补充非周期傅里叶变换和二维傅里叶变换的相关内容

3.4.1 奈奎斯特–香农采样定理

奈奎斯特-香农采样定理是信号处理中一个特别重要的结论，它解决了一个问题——我们应该采用什么样的周期/频率去采样信号，才能保证将信号正确地重建出来。

定义（奈奎斯特-香农采样定理）对于一个带限的信号（即一个没有频率超过阈值 ω_0 的信号），**奈奎斯特-香农定理**（Nyquist-Shannon Theorem）认为，在采样周期为 $T = \frac{\omega_0}{2}$ 的情况下，它可以被完美重建。

现实中，我们会看到各种各样的因为采样不完美导致的走样问题，我们就会看到出现一些**走样失真**（artifact）。例如

- 摩尔纹（Moire Pattern）
- 锯齿（Jaggies）
- 车轮效应（Wagon Wheel Illusion）

有兴趣的读者可以去自行查阅这些失真。

3.4.2 图像处理

上文提及，在二维图像中，我们所说的高频部分指的是颜色变化较大的边缘以及细节，低频部分指的是的图像灰度平稳的区域。因此，如果我们将图像从时域转换到频域上，我们也可以对其使用滤波器来过滤其中的一些信号。在这之中，如果只允许低频信号通过，那么我们就叫它**低通滤波器**（Low-pass filter）；如果只允许高频信号通过，则叫它**高通滤波器**（High-pass filter）。

在图像处理中，低通滤波器的作用通常就是模糊图像，因为此时图像已经没有锐利的边缘和小区域的细节了；而高通滤波器的作用相对地就是提取边缘。

Chapter 4

光栅化渲染管线

显然，之前提及的所有这些步骤都还只是停留在理论上。实际上，我们都需要硬件的配合。中央处理器（CPU）和显卡（GPU）一起提供了一套完整的流程，而我们的渲染都是通过流水线一般的操作迅速地完成的。这套流水线就被称为**光栅化渲染管线**（Rasterization Rendering Pipeline）。

光栅化的渲染管线可以高度抽象化为下面的几个步骤。

第一步（CPU） *应用阶段*。从模型文件（.obj/.fbx/...）中拿到各类数据，例如顶点位置信息、边连接情况、法线等。在这个阶段，CPU 也会负责剔除一部分不在视锥体内的物体。
第二步（GPU） *顶点着色器*。将物体的各类信息从模型空间转换至裁剪空间。
第三步（GPU） *剔除*。对不处于视锥体内的图元剔除，部分处于视锥体内的图元裁剪。
第四步（GPU） *屏幕变换*。将裁剪空间的坐标变换至 2D 的屏幕坐标。
第五步（GPU） *插值*。每个像素根据附近的顶点进行插值。
第六步（GPU） *片元着色器*。每个像素根据第五步插值的结果，进行颜色的计算。
第七步（GPU） *测试混合*。进行模板测试、深度测试和颜色混合。

下面就详细描述上述步骤中的一些技术细节。

4.1 屏幕变换

当我们把几何物体从模型空间转换到齐次裁剪空间的标准化 $[-1,1]^3$ 立方体后，接下来我们要做的也就是将它画在一个 $w \times h$ 的屏幕上。概念上步骤非常简单：

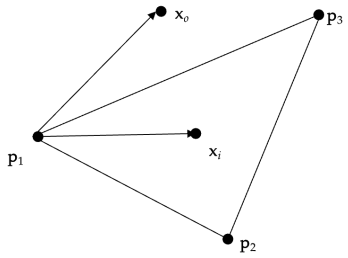
- 去掉 z 坐标。
- 将坐标缩放到 $[0,w] \times [0,h]$ 上。

由于标准立方体（在二维下）的中心是 $(0,0)$，我们要先进行一个 $(1,1)$ 的平移，然后将其缩放为原来的 $(w/2,h/2)$ 倍。

4.2 点是否处于三角形内

现在，我们已经将我们的几何体从三维转换到了一个二维 $[w \times h]$ 的平面上了。但是，现实中的屏幕并不是连续的——屏幕是由一个一个像素组成的，所以实际上是一个 $n_x \times n_y$ 的方阵，其中 $n_x = \frac{w}{p_x}, n_y = \frac{h}{p_y}$，其中 p_x, p_y 为像素的宽度和高度。把三角形画在方阵屏幕上的过程就叫做**光栅化**（rasterization）。

对于一个顶点，它一定只会出现在一个像素内。但是一个像素却不一定在三角形内——它可以完全不在三角形内（在三角形外），也可以完全在三角形内（在三角形内），也可以部分在三角形内（在三角形边上）。如果我们用像素的中心点来代替像素的话，我们就可以通过一个简单的**点是否处于三角形内**的求解来判断这个点属于上述的哪种情况。虽然这个方法无法区分像素是在三角形内还是在三角形边上，但我们先解决基本的问题。



叉乘法

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 $\mathbf{x}_o$。

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(Vector3 P, Vector3[] triangle) {
    Vector3 AP, BP, CP;
    AP = P - triangle[0];
    BP = P - triangle[1];
    CP = P - triangle[2];

    Vector3 AB, BC, CA;
    AB = triangle[1] - triangle[0];
    BC = triangle[2] - triangle[1];
    CA = triangle[0] - triangle[2];
```

```
        Vector3 n1, n2, n3;
        n1 = Vector3.Cross(AB, AP);
        n2 = Vector3.Cross(BC, BP);
        n3 = Vector3.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，我们很快就会在下个小节中提到。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。

4.3 插值

现在我们已经可以判断一个点在不在三角形内了。对于三角形的顶点覆盖到的像素，我们知道这个像素可以使用该顶点的数据，但是如果这个像素没有顶点覆盖，例如在三角形内，或者在三角形边上，这个时候，我们就需要使用**插值**（interpolation）来计算这些未覆盖区域的值。

4.3.1 线性插值

首先是最简单的一**维线性插值**。

定义（线性插值）对于函数 $f(x)$，若 $f(i) = A, f(j) = B, i < j$，则 $\forall t \in [i, j]$，其一**维线性插值**（linear interpolation, LERP）定义为

$$\hat{f}(t) = A + (t - i)\frac{B - A}{j - i}.$$

在图形学中，我们也常用另一种写法，将 t 定义为一个取值为 [0,1] 的值。

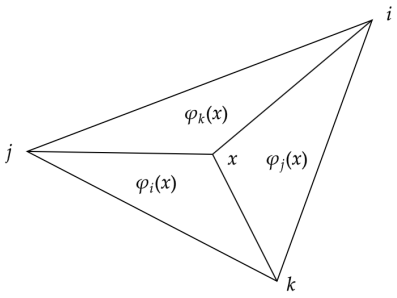
```
float lerp (float A, float B, float t) {
    return (1-t) * A + t * B;
}
```

上面的代码定义了关于 `float` 类型的线性插值。实际上，也完全可以对其他类型做类似的插值，例如 `Vec3` 或是 `Color`。因此，我们也常常使用 C++ 的泛型特性，用 `template` 关键字来写泛型 LERP。

```
template <typename T>
T lerp(const T& A, const T& B, const T& t) {
    return (1-t) * A + t * B;
}
```

4.3.2 三角插值

在上文中我们曾经提及用重心坐标来判断点是否在三角形内。可能读者已经注意到，重心坐标提供了天然的三角形内坐标系统，以重心坐标作为权重，用三角形的三个顶点的值作为加权平均的对象。因此，首先我们需要知道重心坐标的计算方法。



定义（重心坐标）三角形 Δijk 内一点 x 的重心坐标为 $(\varphi_i(x), \varphi_j(x), \varphi_k(x))$，其中

$$\varphi_i(x) = A(\Delta xjk)/A(\Delta ijk),$$
$$\varphi_j(x) = A(\Delta xik)/A(\Delta ijk),$$
$$\varphi_k(x) = A(\Delta xij)/A(\Delta ijk).$$

4.4 深度测试

每个像素的颜色被成功渲染之后，又会出现下一个问题。现在我们的 2D 屏幕是没有 z 坐标的，所有的物体都被压缩到了同一个 z 坐标上。所以，我们需要有办法去衡量它们的深度。

4.4.1 画家算法

第一种很简单的算法被我们称为**画家算法**（painter's algorithm），其内容如下：将所有图元排序，然后从后往前画，让位于前面的物体覆盖位于后面的物体。

这是很符合直觉的算法，但是它有一个很大的问题，它无法处理相交的两个三角形。在这种情况下，任何一个三角形都不能被定义为在另一个三角形的前面。而且，就算所有的三角形都不相交，也可能会陷入遮挡关系环（occlusion cycle）。在现实中，三角形 A 覆盖在三角形 B 上，三角形 B 覆盖在 C 上，然后三角形 C 又覆盖在 A 上是完全可能的情况。在这种情况下，它们就无法被排序，画家算法也就无法处理了。

4.4.2 深度缓冲区

画家算法失效的本质原因在于对于一个三角形而言，内部各个点的深度并不是一定相同的。处理这个的另一种方式是我们让每个像素去维护最先看到的东西，让它将看到的图元的深度自己记住，这样的方法被我们称作**深度缓冲**（Depth Buffer）算法，也叫 **Z 缓冲** (Z-Buffer) 算法。

Z 缓冲算法的描述如下：对于每个像素，让其记住自己最先看到的图形，并且自动排除那些比目前最近的图形更远的图形。只要像素总是知道自己覆盖到的图形的 z 值，这个算法就不会出现问题。当然，实现的方式就如同上面插值中提及的一样，我们除了在没有顶点的像素中插值顶点坐标、颜色等信息，我们也会插值深度信息。

```
初始化:
    对于屏幕上的每个像素，将深度缓冲设置为最大深度值（比如无穷大）

对每个要渲染的三角形执行:
    对三角形的每个像素执行:
        计算像素在三维空间中的深度 z
        如果 z < 深度缓冲中该像素位置的值:
            更新该像素的颜色
            将深度缓冲中该像素位置的值更新为z
```

4.5 颜色混合

在现实世界中，很多物体都并不是完全不透明的。对于透明物体，看到它们之后也依然可以看到下面一层的物体的颜色。因此，这个时候我们就需要做**颜色混合**（Color blending）。

对于 RBGA 颜色编码，这并不难，A 通道对应的数值就是透明度。我们需要定义一个**覆盖**（over）操作。

假设我们有颜色 A, B，其各自的不透明度分别是 α_A, α_B。对于它们的叠加颜色 $C = A$ over B，我们有以下两种计算方式。

- 非预乘（Non-premultiplied）：$C_{rgb} = \alpha_B B + (1-\alpha_B)\alpha_A A$, $\alpha_C = \alpha_B + (1-\alpha_B)\alpha_A$.
- 预乘（Premultiplied）：令 $A' = (\alpha_A A_r, \alpha_A A_g, \alpha_A A_b, \alpha_A), B' = (\alpha_B B_r, \alpha_B B_g, \alpha_B B_b, \alpha_B)$。令 $C' = B' + (1-\alpha_B)A'$，$\alpha_C = \alpha_B + (1-\alpha_B)\alpha_A$。则 $C = (C'_r, C'_g, C'_b, \alpha_C)$。

可以自行对比两种颜色混合的方式产生的差别。

4.6 简单的画线算法

对于一条直线，我们有它的隐式方程，

$$f(x,y) \equiv (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0 = 0.$$

在这里，我们会假设 $x_0 < x_1$。如果情况相反，我们就把两个点换过来，总之，我们总是把 x 坐标小的那个点的 x 坐标指代为 x_0。对于这条直线，其斜率为

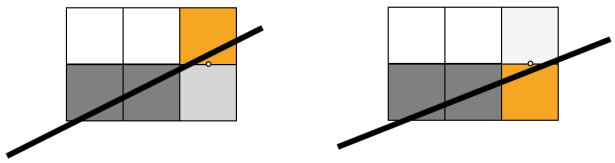
$$k = \frac{y_1 - y_0}{x_1 - x_0}.$$

在这里，我们讨论 $k \in (0,1]$ 的情况，对于其他情况，$k \in (-\infty,-1]$, $k \in (-1,0]$, $k \in (1,\infty)$ 都可以采用类似的方法进行计算。

4.6.1 中点画线法

用自然语言描述**中点画线法**（midpoint algorithm）就是“从左到右画直线，在 [___] 时向 y 轴上方画一格并继续向右。”这里的“在 [___] 时”显然就是我们算法中 `if` 条件句的部分了。

一个方法在直线经过的两个像素中，我们观察这两个像素公共边的中点。如果直线从这个中点的左边穿过，那我们就画位于上方的像素。如果从这个中点的右边穿过，我们就画位于下方的像素。如图所示。



那么我们就可以有简单的伪代码：

```
// midpoint algorithm
float y = y0;
for(x = x0 to x1) {
    draw(x,y);
    // f is the equation of the line f(x,y)
    if(f(x+1, y+0.5) < 0) {
        y = y+1;
    }
}
```

4.6.2 数值微分法

根据数学定义，给定直线方程并计算其斜率 k 后，对于沿着直线给定的 x 方向的增量 δx，可以计算出对应的 y 方向的增量 δy 为

$$\delta y = k \cdot \delta x.$$

而**数值微分法**（digital differential analyzer, DDA）就是通过在一个坐标轴上以单位时间间隔对线段采样，从而确定另一个坐标轴上最靠近线路径的对应整数值。

这个算法可以概括为下面的步骤：

- 输入线段两个端点的像素位置。
- 计算线段水平差值 δx 和垂直差值 δy。
- 通过上述二者中较大的那个决定参数 `steps`。这个值也是我们即将在这条线段上画出的像素数目。
- 绘制起始位置像素 `(x0, y0)`。
- 如果 $|\delta x| > |\delta y|$，且
  - 如果 `x0` $<$ `xEnd`，那么 x 和 y 方向的增量则分别为 1 和 k。
  - 如果 `x0` $\geq$ `xEnd`，那么采用减量-1 和 $-k$ 来生成线段上的点。
- 否则，y 方向使用单位增量（或减量），x 方向使用 $1/m$ 的增量（或减量）。

根据上面的步骤，我们可以写出 DDA 算法的代码。

```
inline int round (const float a) { return (int) (a+0.5); }

// Digital Differential Analyzer
void DrawLineDDA(int x0, int y0, int xEnd, int yEnd) {
    int dx = xEnd - x0, dy = yEnd - y0;

    float steps = (abs(dx) > abs(dy)) ? abs(dx) : abs(dy);
    float xIncrement = (float)dx / (float)steps;
    float yIncrement = (float)dy / (float)steps;

    float x = x0, y = y0;
    DrawPixel( round(x), round(y) );

    for(int i = 0; i < steps; i++) {
        x += xIncrement;
        y += yIncrement;
        DrawPixel( round(x), round(y) );
    }
}
```

DDA 算法计算像素位置要比中点法计算的速度更快。但是，在浮点增量的连续累加中，取整误差的积累使得对于较长线段所计算的像素位置偏离实际线段。而且，该过程中的取整操作和浮点运算仍然十分耗时。

4.6.3 Bresenham 算法

最后我们介绍工业界常用的另一种画线算法——Bresenham 算法。这是一种精确而高效的算法，该算法的特点是仅仅使用增量整数进行计算。

以斜率小于 1 的正斜率直线为例，假设我们已经决定要在像素点 (x_k, y_k) 上着色，那么下一步我们就需要考虑是在像素点 (x_k+1, y_k) 上着色，还是在 (x_k+1, y_k+1) 上着色。在取样位置 x_k+1，我们使用 d_{lower}

和 d_{upper} 来表示两个像素点取值与数学上的直线的垂直误差。根据定义我们有，

$$d_{\text{lower}} = y - y_k = k(x_k + 1) + b - y_k$$
$$d_{\text{upper}} = y_k + 1 - y = y_k + 1 - k(x_k + 1) - b$$

要确定在这两个像素中哪一个更接近路径，只需要计算 d_{lower} 和 d_{upper} 的差即可。因此，以斜率 $0 < k < 1$ 为例，Bresenham 算法的步骤大致可以描述为下面的过程：

- 输入两侧端点，左侧端点记录为 `(x0,y0)`。
- 计算常量 $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x$。计算决策参数 $p_0 = 2\Delta y - \Delta x$。
- 从 $i = 0$ 开始，在沿线段路径的每个 x_i 处，进行下列检测：
  - 如果第 i 个决策参数 $p_i < 0$，下一个绘制的点是 $(x_k + 1, y_k)$，并且
    $$p_{k+1} = p_k + 2\Delta y$$
  - 否则，下一个绘制的点是 $(x_k + 1, y_k + 1)$，并且
    $$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$
- 重复上一步共 $\Delta x - 1$ 次。

根据上面描述的步骤，我们可以给出 Bresenham 算法的实现。

```
// Bresenham Algorithm
void DrawLineBresenham(int x0, int y0, int xEnd, int yEnd) {
    int dx = abs(xEnd - x0), dy = (yEnd - y0);
    int twoDy = 2 * dy, twoDyMinusDx = 2 * (dy - dx);
    int p =2 * dy - dx;
    int x = x0 > xEnd ? xEnd : x0;
    int y = x0 > xEnd ? yEnd : y0;
    if (x0 > xEnd) xEnd = x0;

    DrawPixel(x, y);

    while (x < xEnd) {
        x++;
        if(p<0) p += twoDy;
        else {
            y++;
            p += twoDyMinusDx;
        }
        DrawPixe(x, y);
    }
}
```

4.7 反走样

对于图像处理而言，**反走样**（anti-aliasing）一词在很多地方也被叫做抗锯齿。顾名思义，反走样技术的作用就是让图像看起来更平滑。根据我们上一章的采样理论，我们已经能够理解为什么会出现走样了，本节我们重点讨论减少走样的方式。

4.7.1 多重采样抗锯齿

第一种方法被叫做**多重采样抗锯齿**（Multisampling Anti-Aliasing, MSAA）。MSAA 是**超采样抗锯齿**（Supersampling Anti-Aliasing，SSAA）的改良版，它们在概念上是相似的。它们的原理很简单：提高采样率。对图像中的像素进行多次采样，在渲染过程中，每个像素会被分成 4 个甚至更多个像素，然后我们也去对这些划分出来的小像素再次进行采样。在所有子样本的的颜色和深度信息采样完成后，MSAA 会对这些子样本的值进行平均，作为该像素的最终颜色。

优点

MSAA 能够有效平滑图像，减少锯齿，它在概念非常简单。

缺点

它需要消费大量的计算资源，尤其是在原本采样数量就很高的情况下，它需要使用大量的存储空间来存储子样本的信息，因此会带来很多性能开销。它与某些图形特效不兼容，例如深度场景与透明纹理。

4.7.2 快速近似抗锯齿

第二种常见的方法叫做**快速近似抗锯齿**（Fast Approximate Anti-Aliasing, FXAA）。FXAA 是一种屏幕空间后处理效果（screen-space post processing），可以理解为是在渲染管线的最后阶段应用。FXAA 找到屏幕上出现锯齿的地方，然后对这些区域进行平滑处理，本质上是一种智能模糊。

优点

FXAA 顾名思义，它拥有快的特点。它对性能的影响非常小，而且易于实现，不需要对渲染管线进行大的改动，并且，因为是屏幕后处理效果，它不影响其他图形特效。

缺点

因为 FXAA 的本质就是模糊，所以有可能会使得图像细节变得模糊。与 MSAA 相比，FXAA 在处理锐利边缘时可能不够精细。因为 FXAA 也是低通滤波器，因此可能会对图像中的高频信息产生负面影响，如小细节和高频纹理等。最后，FXAA 不擅长处理高速运动的动态场景。

4.7.3 时间混叠抗锯齿

最后我们再介绍**时间混叠抗锯齿**（Temporal Anti-aliasing, TAA）。这是 NVIDIA 开发的抗锯齿技术，TAA 采用不同时间帧的像素进行采样，跟前面几种抗锯齿比起来，TAA 尤其擅长减少移动中的破碎影像。

优点

TAA 能够在动态场景中提供高质量的抗锯齿效果，对于带有大量高频细节的场景，TAA 也有效减少了闪烁现象。它还拥有高效的性能。

缺点

TAA 有时会导致快速运动的画面变得过于模糊，而且由于 TAA 依赖前一帧的数据，因此快速移动的对象可能会产生鬼影。TAA 对于静态场景的细节处理不够好，也可能会导致细节的损失。最后，TAA 在实现上相比 FXAA 也更困难。

Chapter 5

着色

之前的章节中，我们已经完整地完成了将 3D 物体正确显示在 2D 屏幕上的操作了。但是现在我们显示出来的依然还是一个白模。现实世界中的物体是有颜色的，我们在此先不考虑颜色如何定义，先用直观的思维去思考——拿到白模以后，我们下一步操作就应该是为其着色。

5.1 纹理映射

在 3D 建模中，我们常常说给一个模型上纹理贴图。最简单的纹理贴图——固有色贴图，就是告诉模型要在哪个地方显示哪个颜色（当然，它最终显示的颜色会受到光照、反射等各种各样的因素影响）。因此，我们先讨论**纹理**是什么。

5.1.1 纹理

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。

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。

现实的工业界中确实也有各种各样的贴图，例如固有色贴图（diffuse/albedo）、法线贴图 (normal)、位移贴图（displacement）、凹凸贴图（bump）、环境光遮罩（AO）、金属度（metallic）、粗糙度（roughness）……因此，我们在直觉上并不可以将纹理和颜色画上等号。为了方便说明，下文中会主要使用固有色贴图来讨论，但是务必记住纹理可以记录各种各样的数据，纹理相关的各种操作也适用于各类贴图。

5.1.2 逐像素采样算法

下面的伪代码简单地介绍了逐像素（per-pixel）采样的算法逻辑。

```
// image is the image plane we will texture
SamplePerPixel(Image image, Texture mainTex) {
    for(Pixel p in image) {
        (u,v) = interpolate(p.pos);
        p.value = SampleTexture(mainTex, u,v);
    }
}
```

5.1.3 纹理寻址

对于着色器而言，需要知道如何在纹理上采样，也就是上面伪代码中的 `SampleTexture()` 的部分。这个过程被称为**纹理寻址**（texture lookup）。简单描述这个过程，就是着色器拿到顶点（或插值后）对应的纹理的位置，然后读取纹理上的值，然后记录成为采样点的值。

```
// image is the image plane we will texture
SampleTexture(Texture mainTex, float u, float v) {
    int i = round(u * mainTex.width() - 0.5);
    int j = round(v * mainTex.height() - 0.5);
    return mainTex.get_texel(i,j);
}
```

5.2 放大

在现实中，纹理采样会遇到两个问题，纹理放大（magnification）和纹理缩小（minification）的问题。我们知道，纹理依然也不是连续定义的一张贴图。它也是一个 $t_m \times t_n$ 的一张方阵，其中 t_m, t_n 分别是纹理的宽度和高度。

定义（纹素）在一个 $t_m \times t_n$ 的纹理上，一个 1×1 的区域就是一个**纹素**（texel）。

首先，当相机离场景中的物体非常近的时候，一个像素可能就只会覆盖纹理一个极小的区域，根本覆盖不到一个纹素。我们解决放大问题主要有几个思路。

5.2.1 最近邻插值

第一种思路叫做**最近邻插值**（Nearest Point Interpolation）。直接取像素所对应的纹素的值。但是在放大的情况下会导致图像像素化。

5.2.2 双线性插值

第二种思路为**双线性插值**（Bilinear Interpolation）。双线性插值考虑最近的四个纹素，并基于距离对它们的颜色进行加权平均。这种方法可以产生比最近最近邻插值更平滑的图像效果。

假设这四个点分别为 $Q_{11}(x_1, y_1), Q_{12}(x_1, y_2), Q_{21}(x_2, y_1), Q_{22}(x_2, y_2)$。我们认为 Q_{ij} 就是点 (x_i, y_j) 的某种取值（颜色、法线、大小等）。对于插值点 (x, y):

- 先计算**水平方向**的插值：这里，我们通过线性插值的公式计算即可。

  1. $R_1 = Q_{11} \cdot \frac{x_2 - x}{x_2 - x_1} + Q_{21} \cdot \frac{x - x_1}{x_2 - x_1}$.
  2. $R_2 = Q_{12} \cdot \frac{x_2 - x}{x_2 - x_1} + Q_{22} \cdot \frac{x - x_1}{x_2 - x_1}$.

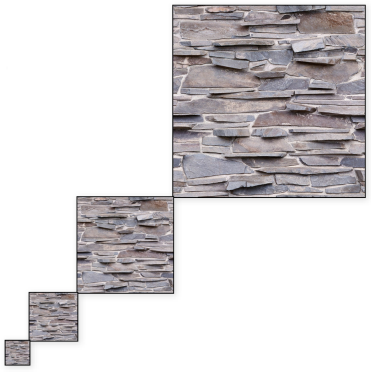
- 然后再计算**垂直方向**的插值，对 R_1 和 R_2 进行一次线性插值即可。

$$P(x, y) = R_1 \cdot \frac{y_2 - y}{y_2 - y_1} + R_2 \cdot \frac{y - y_1}{y_2 - y_1}.$$

5.3 缩小

5.3.1 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

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

5.3.2 三线性插值

三线性插值（Trilinear Interpolation）是双线性插值的一种扩展，它考虑两个层级 MIPMap 上最近的四个纹素，并基于距离对它们的颜色进行加权平均。这种方法主要是为了解决 MIPMap 在层级变化的边界会产生明显差别的走样。

三线性插值的计算步骤如下：

- 选择两个 MIPMap 级别。通常是相邻的两个级别 L_1 和 L_2。
- 在两个级别上分别进行一次双线性插值。假设插值结果分别为 P_1 和 P_2。
- 对两个级别进行一次线性插值：

$$P = P_1 \cdot (1 - d) + P_2 \cdot d.$$

这里的 d 的值通常是根据纹理在屏幕上的大小与其在两个 MIPMap 级别之间的大小进行计算的，我们显然可以直接通过上一小节计算 MIPMap 层级的时候计算出的 D 值用来进行线性插值。

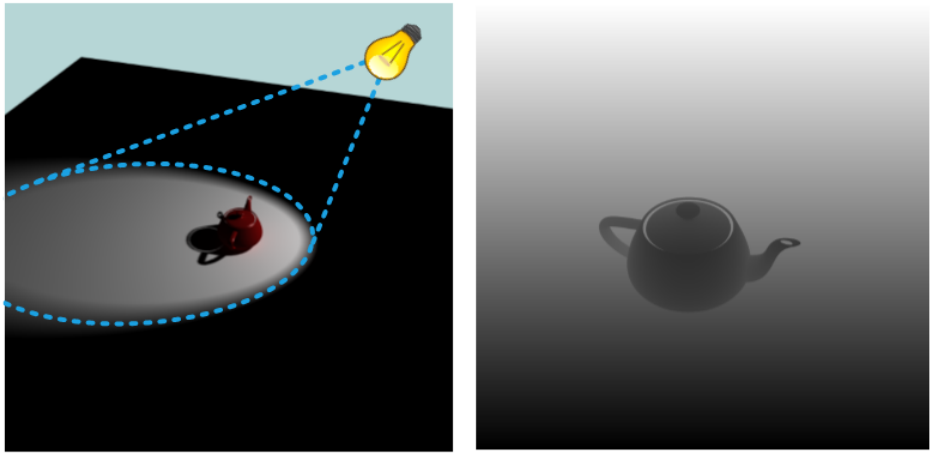
5.4 阴影映射

阴影在之后提到的光线追踪中很容易生成，但是在光栅化中却很难，因为面都是一个个被考虑的。于是，我们就考虑使用纹理映射的方式记录阴影信息。

阴影映射（shadow mapping）的核心思想：一个点会被光源照射到，当且仅当从光源能看到这个点。工作过程大概可以分成两个阶段：

5.4.1 深度图生成

首先，从光源的视角对场景进行渲染，但是不记录颜色信息，仅记录各个像素点距离光源的深度值（光源到这个点的距离）。这个渲染的结果被存储在一张被称为**阴影图**（shadow map）的特殊纹理中，如下图[1]所示。



深度图可以视为光源看到的一个二维表示，其中记录了场景中各个物体的深度信息。

[1] https://www.downloads.redway3d.com/downloads/public/documentation/bk_re_shadow_mapping_detailed.html

5.4.2 阴影渲染

接下来，将光源看到的深度值 d_{map}——即阴影图与相机（人眼）视角计算出的深度值 d——Z 缓冲进行比较。如果 Z 缓冲区中的值大于阴影图中的对应深度值，这意味着从光源的视角看，该点被其它物体挡住了，则说明该像素位于阴影中；反之，如果 Z 缓冲区的值等于阴影图中的值，则说明该像素直接被光源照射。

总结一下就是，光源能看到、相机看不到，说明处于光中；光源看不到，相机能看到，说明处于影中。

上面的**等于**一词在计算机图形中应该立刻引起你的注意。这个深度值肯定需要是浮点数，浮点数比较相等就可定会出现精度问题。因此，在实际使用中，我们常常会认为如果 $d - d_{\text{map}} < \epsilon$，我们就认为是相等的。这个 ϵ 就被成为**阴影偏移**（shadow bias）。

5.5 环境贴图

环境贴图（Environmental Maps）的核心思想是：将周围来的光定义成一个函数，而对各个方向皆有定义的函数是一个位于单位球上的函数。因此，我们可以将球上覆盖一张贴图，然后纹理寻址用其作为环境光的取值。

对于后面会提到的光线追踪，环境贴图极其容易使用，仅需在光线没有打到任何物体时使用环境贴图的值即可，我们在后文会提及。对于光栅化，我们也可以通过在着色环节增加一个镜面反射的步骤，以达成类似的效果。这样的操作方式也被称为**反射探针**（Reflection Probes）。环境贴图的使用与实现在本课程中不做要求。

5.6 Blinn-Phong 模型

Blinn-Phong 着色模型是计算机图形学中广泛使用的一种光照模型，主要用于模拟物体表面与光源相互作用的方式，从而产生逼真的视觉效果。这是一种经验着色模型，所以它并不是准确地描述光照的行为。

Blinn-Phong 着色模型认为光照分为三个主要的组成部分：环境光（ambient）L_a，漫反射（diffuse）L_d 和镜面反射/高光（specular）L_s。即

$$L = L_a + L_d + L_s.$$

在经验模型中，我们暂时不讨论 L 的物理意义或者单位，我们暂时可以将其理解为一种直觉上反应光强的物理量即可。

5.6.1 漫反射光

Blinn-Phong 模型认为，光打在物体表面上后，会有一部分向各个方向均匀地反射。漫反射光的描述为

$$L_d = k_d \frac{I}{r^2} \max(0, \mathbf{n} \cdot \mathbf{l})$$

其中，$\mathbf{n}$ 是表面法线，$\mathbf{l}$ 是光源方向（根据惯例指的是被光照亮的点指向光源的向量），这两者都是单位向量。因此，$\mathbf{n} \cdot \mathbf{l}$ 就是它们的夹角余弦。k_d 是颜色的编码（一个系数），I 是光的强度，r 是被照亮的点与光源的距离。

5.6.2 镜面反射光

Blinn-Phong 模型与 Phong 模型的一个核心区别就是简化了镜面反射光的计算。Blinn-Phong 模型借助半角向量的概念，大大简化了计算。

定义（半角向量）**半角向量**（half vector）定义为**观测方向向量**（View Direction）$\mathbf{v}$ 与光源方向向量（Light Direction）的角平分线的归一化向量，即

$$\mathbf{h} = \frac{\mathbf{v}+\mathbf{l}}{||\mathbf{v}+\mathbf{l}||}.$$

借用半角向量，我们定义高光项为

$$L_s = k_s \frac{I}{r^2} \max(0, \mathbf{n}\cdot\mathbf{h})^p$$

其中，k_s 为系数，p 是用来缩小高光区域的指数。用指数来调整视觉效果也是渲染中常用的一种手段，但基本上局限于经验模型。

5.6.3 环境光

环境光项很简单，我们认为环境光是来自环境的间接光照，因此直接给一个参数用来调整环境光强度，给一个参数用来调整颜色即可。

$$L_a = k_a I_a$$

其中 k_a 为系数，I_a 为环境光强度。

综上，Blinn-Phong 模型中，光照被计算为

$$L = L_a + L_d + L_s = k_a I_a + k_d \frac{I}{r^2} \max(0, \mathbf{n}\cdot\mathbf{l}) + k_s \frac{I}{r^2} \max(0, \mathbf{n}\cdot\mathbf{h})^p.$$

5.7 着色频率

最后我们回到管线上，通常着色有三种思路，它们有着不同的取值单位，所以也称为**着色频率**（Shading Frequency）。

5.7.1 逐三角形

逐三角形着色（Per-triangle）也成称作**扁平着色**（Flat Shading）。在这种着色频率下，我们假设所有的三角形都是平整的，并且只有一个法线 $\mathbf{n}$。对于每个三角形我们只上一种颜色。这种着色频率虽然很直观，但是在现实中很少使用，这会使得颜色看起来很不平整，并且一旦出现个别大块三角面的时候，颜色也不会有变化。

5.7.2 逐顶点

逐顶点着色（Per-vertex）也称作**高洛德着色**（Gouraud Shading）。在这样的着色频率下，我们对每个三角面的每个顶点进行着色，对三角面内部根据顶点使用三角插值。

5.7.3 逐像素

逐像素着色（Per-pixel）的着色频率下，我们根据每个像素对屏幕上色。这是现在最常见的光栅化着色频率。

Chapter 6

颜色

渲染的结果最终总还是要落实到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。每个光子会携带一定的能量，而这个能量就是波长的反映。然而，波长是光的物理属性，颜色却是人眼的主观反应。因此，研究颜色的性质以及如何通过编码反应人眼能够观察到的波长是一个很重要的主题。

6.1 色度测量学

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，以下简称光敏函数。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，

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

这个数值被我们称作三**刺激值**（tristimulus value）。其中，$\Phi(\lambda)$ 是对于特定波长的光的光通量。这个物理量计算了单位时间内通过的光子的数量，我们会在第九章中详细介绍。

6.1.1 视觉基础

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敏函数。

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 $m_i(\lambda)$ 来描述三刺激值，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）。

6.1.2 格拉斯曼法则

现在我们知道人眼中有三种可以对红、绿、蓝光的刺激产生反应的生理结构。而赫尔曼 · 格拉斯曼（Hermann Grassmann）指出，任何颜色产生的刺激都可以由红绿蓝三种光通过一定的比例叠加产生相同的效果。特别地，他提出了**格拉斯曼法则**（Grassmann's Law），指出这样的叠加具有：

- 对称性。如果颜色 A 产生的效果与颜色 B 产生的效果匹配，则颜色 B 产生的效果也与颜色 A 产生的效果相同。
- 传递性。如果颜色 A 产生的效果与 B 匹配，B 与 C 匹配，则 A 与 C 也是匹配的。
- 成比例的性质。如果颜色 A 产生的效果与颜色 B 匹配，则颜色 αA 与颜色 αB 也是匹配的。
- 叠加的性质。如果颜色 A 与颜色 B 匹配，颜色 C 与颜色 D 匹配，则 $A+C$ 与 $B+D$ 匹配，且 $A+D$ 与 $B+C$ 也是匹配的。

这些性质的存在才决定了颜色是符合我们关于叠加（additive）的直觉的。

6.1.3 同色异谱与颜色匹配

由于光敏函数是通过两个函数的积求积分获得的，我们的眼睛很明显就不是一个简单的波长检测器。事实上，我们可以找出两个不同的通量波长分布函数 $\Phi_1(\lambda)$ 和 $\Phi_2(\lambda)$，使得在积分后，它们能够产出相同的 $(L, M.S)$ 响应。这种现象就被称作**同色异谱**（metamerism）。

颜色匹配（color matching）也是基于同色异谱的原则进行的。假设我们有三种不同颜色的灯，我们叫它们原色灯。通过调节三盏灯的亮度，将它们叠加在一起来匹配某一个我们看到的颜色。当颜色匹配时，三盏原色灯各自的当前亮度就可以做为这个待测颜色的表征。经过一系列的测试，国际照明委员会 CIE（the Commission Internationale d'Eclairage）最终确定这三盏原色灯的波长为 438.5, 546.1 以及 700 nm。所有颜色通过这三盏灯互相叠加做到一个颜色匹配。其中，每种波长 λ 在标准化颜色匹配实验中对应的红绿蓝灯的通量分别为 $r(\lambda), g(\lambda)$ 以及 $b(\lambda)$，积分求得的三刺激值分别记录为 R, G 和 B。

在 RGB 的颜色匹配实验中，一部分的颜色需要通过“减去”一部分原色光得到。这个操作并没有物理意义，因为现实物理世界中并没有负光的概念。因此，为了匹配这个颜色，我们可以通过格拉斯曼法则，往匹配的两边同时叠加同一个颜色。对于 $r(\lambda), g(\lambda)$ 以及 $b(\lambda)$ 三个通量函数，其中 g 和 b 是一直不为负的，而 r 则在 438.5nm 到 546.1mm 之间为负值。

6.1.4 色度坐标

由格拉斯曼法则所指出的颜色的性质来看，我们可以通过一个三维数对 (X, Y, Z) 来表示三种锥状细胞受到的刺激的程度，换句话说颜色可以通过一个三维数对表示。一个符合直觉的方式是直接建立以 X, Y, Z 为轴的正交坐标系，并且把每个颜色对应的 (X, Y, Z) 值在坐标系中画出。而这样的三维空间，就被我们称

作**颜色空间**（color space）。

在上一节中，我们提到的 R, G, B 三刺激值表征肯定是一种可行的方法。但是，在 CIE 创建这个标准的二十世纪三十年代，含有负值的数值积分还并不好做，而 CIE 当时已经开发出了**明视觉亮度响应函数**（photopic luminance response function）$V(\lambda)$。因此，CIE 想出了一个方法——通过概念中的、现实中无法创造的原色光源，配合它们已经开发出的 $V(\lambda)$ 来创建叠加光，使得三个通量函数永不为负。在这种模式下，三盏灯对应的通量函数分别为 $x(\lambda), y(\lambda)$ 以及 $z(\lambda)$，其中 $y = V$。而它们对应的三刺激就被记作 X, Y 和 Z。

6.1.5 动态范围

6.2 颜色空间

6.2.1 颜色空间变换矩阵

6.2.2 常见编码形式

6.2.3 CIE 1976 $L^*a^*b^*$

6.3 伽马校正

6.3.1 伽马空间与线性空间

Chapter 7

几何查询

在图形学中，有一系列的问题与几何体有关，我们需要在几何体上找到一个满足我们需求的点，例如交点、最近点等。这一类问题被我们统称为**几何查询**（Geometry queries）。在本章中，我们将会从一些简单的问题出发，然后介绍之后在光线追踪中最重要的射线求交查询。

7.1 几何体上最近点

首先先从简单的问题开始出发：对于某一个点 $\mathbf{p}$，在给定几何体上返回与其直线距离最短的点 $\mathbf{q}$。我们按照不同的几何体来分类讨论这个问题。

7.1.1 点上的最近一点

这个问题是非常基本的。如果给定几何体是一个点，那么与点 $\mathbf{p}$ 最近的点就是它本身。

7.1.2 直线上最近一点

数学常识告诉我们，过一点做直线垂线，那么该垂线就是直线上与该点最近的点。

但是，首先我们需要先检查，该点是否已经在直线上。如果该点已经在直线上，那么直接返回该点即可。

如果，该点不在直线上，那么根据我们已有的数学常识，假设直线的点法式方程为

$$\hat{\mathbf{n}}^{\mathrm{T}}\mathbf{x} = c$$

其中，$\hat{\mathbf{n}}$ 是该直线的法线。我们知道，过点 $\mathbf{p}$ 的垂线必定沿着 $\hat{\mathbf{n}}$ 的方向，并且其垂足 $\mathbf{q}$ 必定在直线上，因此我们有

$$\begin{aligned}\hat{\mathbf{n}}^{\mathrm{T}}\mathbf{q} = c &\Leftrightarrow \hat{\mathbf{n}}^{\mathrm{T}}(\mathbf{p} + t\hat{\mathbf{n}}) = c \\ &\Leftrightarrow \hat{\mathbf{n}}^{\mathrm{T}}\mathbf{p} + t\hat{\mathbf{n}}^{\mathrm{T}}\hat{\mathbf{n}} = c \\ &\Leftrightarrow t = c - \hat{\mathbf{n}}^{\mathrm{T}}\mathbf{p}\end{aligned}$$

因此，所求点 $\mathbf{q} = p + (c - \hat{\mathbf{n}}^{\mathrm{T}}\mathbf{p})\hat{\mathbf{n}}$。

7.1.3 线段/射线上最近一点

这个问题看似和上一个问题是一样的，但实际上并非如此，因为垂足可能在线段/射线的非有效部分（即端点之外）。因此，在用直线上最近一点的方法求最近点之后，我们先判断它处不处在线段/射线的有效范围。如果在，则直接返回该点。如果不在，则直接测试端点。对于线段，返回两个端点中距离更短的那个点即可。

7.1.4 三角形上最近一点

在第四章中我们曾经介绍过如何判断一个点是否处于三角形内。如果点 **p** 在三角形内，那么直接返回点 **p** 即可。否则，假设三角形的三个顶点分别为 **abc**，那么我们去测量向量 **p-a**, **p-b** 和 **p-c** 的长度，返回长度最小的那个顶点即可。

7.1.5 3D 三角形上的最近一点

和上一个问题比较类似，只不过我们需要先把点投影到三角形的平面上。然后再用投影点去做 7.1.4 的查询即可。

7.1.6 三角网格上的最近一点

三**角网格**（triangular mesh）是由三角形组成的几何体，我们会在第八章详细介绍。对于三角网格而言，我们可以用最暴力的方法，用这个点与三角网格中的所有三角形做 7.1.5 的查询，然后返回最小的值，但这显然会浪费很多的性能。在本章的末尾，我们会介绍使用空间加速结构来加速这一类问题的方法。

7.2 射线-几何体求交

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 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。

在这里，**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。

7.2.1 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

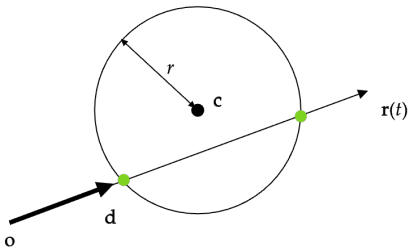
$$||\mathbf{x}-\mathbf{c}||^2 - r^2 = 0$$

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，这里就不再赘述求根方法了。总之，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。

7.2.2 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 $\mathbf{x}$。

$$\begin{aligned}(\mathbf{o}+t\mathbf{d}-\mathbf{p})\cdot n &= 0\\ t\mathbf{d}\cdot\mathbf{n}+(\mathbf{o}-\mathbf{p})\cdot\mathbf{n} &= 0\\ t &= -\frac{(\mathbf{o}-\mathbf{p}\cdot\mathbf{n})}{\mathbf{d}\cdot\mathbf{n}}\end{aligned}$$

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。

7.2.3 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。

以上的三个条件我们皆在之前的小节或章节中覆盖，因此在这里就没有什么可以多说的了。

7.3 空间加速结构

7.3.1 轴对齐包围盒

在之前的部分中，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，我们会去使用**轴对齐包围盒**（axis-aligned bounding box, AABB）。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 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**。

7.3.2 射线-轴对齐包围盒求交

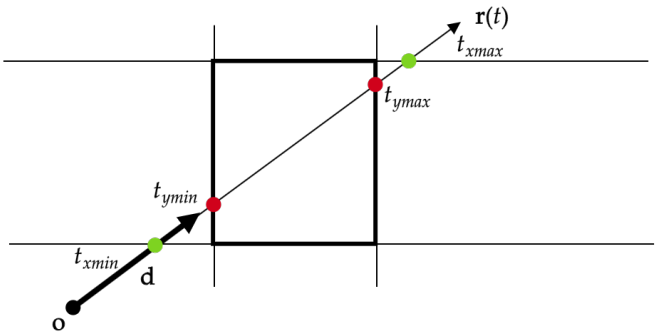
我们先从更简单的 2D 情况开始讨论。3D 情况实际上只要对 xy, xz, yz 三个平面分别做一次 2D 求交即可。以 xy 平面为例。

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;
        }
    }
```

7.4 层级包围盒优化

7.4.1 包围盒层级结构

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）。

7.4.2 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。

7.4.3 包围盒层级划分

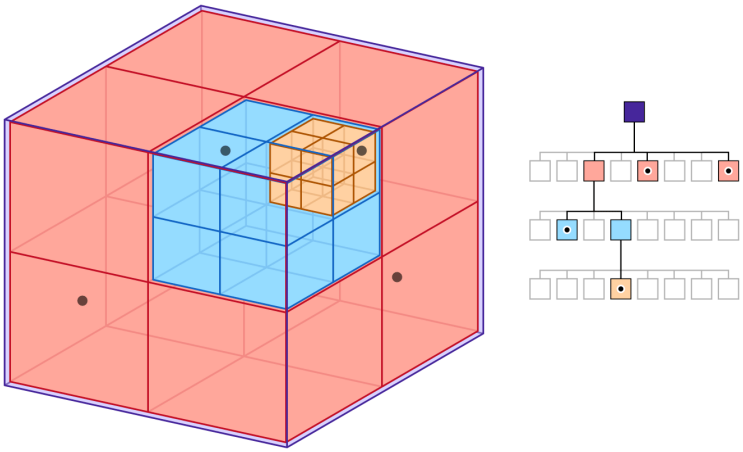
如果包围盒的层级划分得好，那么使用 BVH 来加速射线求交的问题就会取得更好的效果。诚然，现实中的几何体肯定不可能总是非常乖巧地长在各个包围盒的中间，无论我们采取怎样的策略，都一定会遇到一些较慢的情况。我们一般有这么两种策略:

- **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。我们接下来具体讨论。

7.4.4 八叉树

八叉树（Octree）是一种层级式的树状数据结构。它将三维空间分割成八个较小的立方体区域，父节点存储存储包裹这八个小立方体区域的总区域，子节点存储每一个小立方体区域，然后以此类推，直至一个立方体区域中仅包含一个对象。如下图所示[1]。



八叉树总是均匀地将空间分割成 8 个子区域，因此，它更适用于物体分布相对均匀地状况。一个八叉树节点的数据结构可以参考下面的方式实现。

```
class OctreeNode {
public:
    // Constructors and Destructors omitted
    OctreeNode *children[8];
    Vec3f points; // The data that the current node contains
    BBox region; // the bounding box that the current node contains
}
```

然后，我们就可以利用八叉树节点去声明一个八叉树类了。

```
class Octree {
public:
    // Constructors and Destructors omitted
    OctreeNode *root; // the root node

    void insert(Vec3f point);
    bool contains(Vec3f point);

private:
    // some supplementary methods
    void splitNode(OctreeNode *node);
}
```

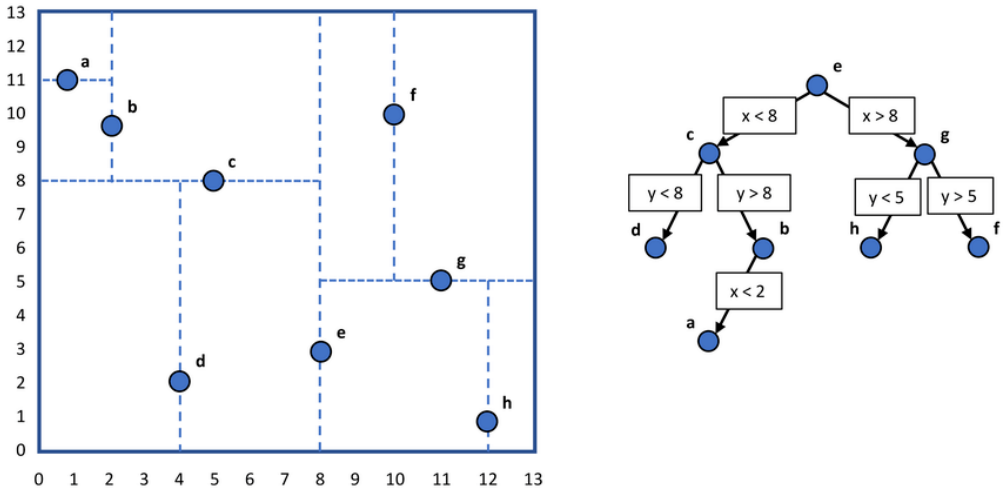
[1] https://developer.apple.com/documentation/gameplaykit/gkoctree

7.4.5 KD 树

KD 树（k-dimensional tree）是一种用于组织 k 维空间数据结构的二叉树结构，主要用于多维空间的查询任务，如最近邻搜索、范围搜索和光线追踪中的加速结构。

对于 KD 树而言，它递归地将 k 维空间划分为两个子空间来组织数据。我们在可用维度中选择一个维度作为分割维度，通常是轮询选择或选择方差最大的维度。在选定的维度上选择一个点作为分割点，这个点可以是中值点、可以是平均值点或是其它启发式方法选择的点。基于分割点，我们将数据集分为两个部分，并在每个部分上继续递归构建子树。

一个二维的 KD 树如下图所示[2]。



KD 树的优势在于其高效的 k 维空间数据查询以及灵活的空间划分策略，劣势在于对于高维数据而言，其效率显著下降。另外，动态数据集的频繁插入和删除可能导致树的结构不平衡。

一个 KD 树节点的数据结构可以参考下面的方式实现。

```
class KDTreeNode {
public:
    KDTreeNode *left, *right; // children
    Vec3f point; // some data

    KDTreeNode(const Vec3f &pt)
        : point(pt), left(nullptr), right(nullptr) {}
}
```

然后，我们就可以利用 KD 树节点去声明一个 KD 树类了。

```
class KDTree {
public:
    KDTreeNode *root;
    KDTree() : root(nullptr) {}

    void insert(const Vec3f &pt) {
```

[2]https://www.researchgate.net/figure/An-example-two-dimensional-k-d-tree-k-2-built-from-nodes-a-through-h-Dividing-planes_fig2_314298746

```
            root = insertRec(root, pt, 0);
        }
        // other methods like search, etc.

    private:
        // insert recursively
        KDTreeNode *insertRec(KDTreeNode *node, const Vec3f &pt, unsigned depth) {
            if(node == nullptr) {
                return new KDTreeNode(pt);
            }

            unsigned cd = depth % k; //k is the dimension for k-d

            if (pt[cd] < (node→pt[cd])) {
                node→left = insertRec (node→left, pt, depth+1);
            } else {
                node→right = insertRec (node→right, pt, depth+1);
            }

            return node;
        }

        // the dimension for the KDtree, for example 2
        static const int k = 2;
    }
```

在这里，函数 `insertRec()` 递归地比较点的坐标和当前节点的坐标来决定将点插入到左子树还是右子树。在每一层递归中，我们使用深度来选择分割维度，这里通过`%` 运算符实现。

Chapter 8

基础几何表示法

在光栅化中我们曾经讨论过直线的画法。但显然几何图形一次中包括的元素远远多于直线这样的单一形式。在本章中我们讨论更多不同的几何形式，并且也讨论如何高效地、正确地将它们的信息记录下来。

8.1 曲线与曲面

直观上来说，**曲线**（curve）一词指代的是可以用笔画出的一些东西。在二维纸面上笔在纸上画出的轨迹就是一条二维曲线，在空间中手一挥笔头经过的轨迹就是三维曲线。严格来说，曲线是大小为无穷大的点集，其中除了**端点**（endpoint）以外的任何点都有两个相邻点。曲线可能没有端点，例如一个封闭曲线或者一条无限长曲线。曲线也可能只有一个端点，例如射线。需要注意的是，曲线一词在中英文语境下都给人带来它是弯曲的直观印象，但实际上直线也是一种曲线。

类似地，我们也可以认为**曲面**（surface）是点在三维空间中的行动轨迹，只是这次它不局限在一个平面上了。对于曲线和曲面这样可以被测量维度、长度、角度和面积的几何，几何上每个点都可以用一个数字来代表它们所处于某一个特定维度的位置，我们就认为它是处于**欧几里得空间**（Euclidean Space）的几何，简称欧式几何。

对于欧式几何更进一步的拓展是**流形**（manifold）的概念。流形是局部看起来像欧几里得空间的结构，但是在全局上可能拥有更复杂的结构。例如地球，从局部上看起来都可以用二维的经纬度来表示，但是整体是个三维球体。常见的流形还有莫比乌斯带、克莱因瓶等。

8.1.1 几何的表示

对于几何体，我们有几种不同的表达方式。

隐式

几何的**隐式表达**（implicit expression）是通过把一个已知点代入方程式，检测其是否左右相等。我们把这个方程叫做几何的**隐函数**（implicit function）。以平面曲线为例，它的隐函数为

$$f(x, y) = 0$$

隐式表达的特点是我们很容易判断一个点在不在这个几何上——只需要代入方程计算就行。但是我们不容易从隐式表达中导出几何上的一个点。这需要我们提供一个方程的解，但通常隐函数求解的难度要远大于验证的难度。

一些常见的隐式图形学类型有代数表面、构造性实体几何、融球、软体表面、分型等。建议有兴趣的读者一一查询维基百科了解细节，在此不予赘述。

显式表达

几何的**显式表达**（explicit expression）与隐式表达在特征上相反——我们很容易导出一个几何上的点，但不容易验证一个点是否在几何上。显式表达中，**参数化表达**（parametric）是一种常见的形式。通过将几何的 $x, y, ...$ 坐标写成 t 的函数，然后通过带入一个 t 值就能求出一个在几何上的点了。例如

$$(x, y) = \mathbf{f}(t) = (\cos t, \sin t)$$

一些常见的显式表达有点集、多边形网格（Mesh）、三角网格、贝塞尔曲线等。我们在之后会重点讨论贝塞尔曲线。

8.1.2 连续性

设计一条复杂曲线的时候，出于设计和制造上的考虑，常常会通过多段曲线组合而成。所以，我们需要解决曲线段之间如何实现光滑连接的问题。我们从直观上先介绍 C^n **连续性**（continuity）：

- C^0 连续性。一条没有断点的曲线就符合 C^0 连续的要求。
- C^1 连续性。曲线的一阶导数也是连续的，因此，曲线不会有突变的角度（比如一个角），看起来会比较光滑。
- C^2 连续性。曲线的二阶导数也是连续的。因此，曲线在连接点的转弯也不会突然发生改变。

一般地，C^n 连续性指的就是曲线的 n 阶导数没有断点的性质。虽然高于 C^2 的连续性当然存在，且也能保证更光滑，但是实际使用中不多见。

除了 C^n 连续性，我们还会常常使用 G^n 连续性，与 C^n 连续性有概念上的区分。

- G^0 连续性。这是最基本的几何连续性，它仅仅要求曲线或曲面在连接点是连续的，也就是说，没有断开。G^0 连续性与 C^0 连续性是等价的。
- G^1 连续性。它不仅要求曲线在连接点连续，还要求曲线在这些点的切线方向连续。这意味着曲线的方向在连接点不会突然改变，但切线的长度（斜率的大小）可以改变。这与 C^1 连续性不同，后者要求一阶导数完全相同。
- G^2 连续性。进一步要求曲线在连接点不仅切线方向连续，而且曲率的变化也是平滑的。这意味着曲线在连接点的弯曲程度会平滑过渡，但曲率的实际值可以不同。这与 C^2 连续性不同，后者要求二阶导数完全相同。

G^n 连续性的概念对于计算机辅助设计和计算机图形学尤为重要，因为在许多情况下，设计师和工程师更关心的是曲线或曲面的几何形状和视觉外观，而不是它们的精确数学表达式。通过使用 G^n 连续性，可以创建看起来光滑和自然的曲线和曲面，即使它们的数学定义可能在技术上不是完全连续的。

8.2 贝塞尔曲线与曲面

在显式几何中，**贝塞尔曲线**（Bézier curve）是一类工业上广泛使用的曲线。我们重点研究贝塞尔曲线的性质和特性。

8.2.1 定义与性质

贝塞尔曲线是描述其**控制点**（control points）的近似曲线 (approximating curve)。这个曲线的形式可以是任意次数的多项式。一个度数为 d 的贝塞尔曲线拥有 $d+1$ 个控制点。首先，先了解伯恩斯坦基多项式的概念。

定义（伯恩斯坦基多项式）n 次的**伯恩斯坦基多项式** (Bernstein Basis Polynomial) 为

$$B_k^n(x) = \binom{n}{k} x^k (1-x)^{n-k}$$

其中，$k = 0, ..., n$。因此，n 次的伯恩斯坦基多项式共有 $n+1$ 个。

Example 8.1. *求出 3 次的所有伯恩斯坦基多项式。*

Solution.

$$\begin{aligned} B_0^1(x) &= (1-x)^3, \\ B_1^3(x) &= 3x(1-x)^2, \\ B_2^3(x) &= 3x^2(1-x), \\ B_3^3(x) &= x^3. \end{aligned}$$

■

这样我们就可以定义贝塞尔曲线了。

定义（贝塞尔曲线）n 次**贝塞尔曲线**（Bézier curve）是通过 n 次伯恩斯坦基多项式和 n 个控制点 $\mathbf{p}_1, ..., \mathbf{p}_n$ 表达的一条曲线。

$$\gamma_n(u) = \sum_{k=0}^{n} B_k^n(u) \mathbf{p}_k$$

其中，$\mathbf{p}_k$ 是控制点。

贝塞尔曲线有一些很好的性质。

- 曲线被控制点组成的凸包包围。
- 如果我们用折线将贝塞尔曲线的控制点相连，那么给定任何一条直线，这条直线与贝塞尔曲线的交点数量一定不多于这条直线与控制点的折线的交点数量。这个性质被叫做*变差递减性*（variation diminishing）。
- 如果我们将控制点的顺序逆置，我们会得出同一条贝塞尔曲线，只是参数化方程是相反的，这个性质叫做*对称性*（symmetry）。
- 贝塞尔曲线拥有*仿射不变性*（affine invariant）——对全体控制点一起进行仿射变换（平移、缩放、旋转）等于对贝塞尔曲线自己进行这个仿射变换。

8.2.2 de Casteljau 递推算法

上述基于伯恩斯坦基多项式的贝塞尔曲线描述显然是贝塞尔曲线的显式方程。因此，要计算在贝塞尔曲线上的点，可以直接使用参数化方程。但是使用 de Casteljau 提出的递推算法会大大简化这个步骤。

定义（de Casteljau 递推算法）由 $n+1$ 个控制点 $\mathbf{p}_i, i = 0, ..., n$ 定义的 n 次贝塞尔曲线 γ_n 可被定义成分别由前后 n 个控制点组成的两条 $n-1$ 次贝塞尔曲线的线性组合，即

$$\gamma_n = (1-t)\gamma_{[0,n-1]} + t\gamma_{[1,n]}, t \in [0,1].$$

这样我们就可以得到贝塞尔曲线的递推公式。在这里，t 就是贝塞尔曲线的参数。

基本情况：如果 $n = 0$，那么只有一个控制点 $\mathbf{p}_0$，则 $\gamma_0(t) = \mathbf{p}_0$。
递归步骤：对于 $n > 0$，使用 de Casteljau 算法递归计算线性插值，然后使用这些插值点构造更低次的贝塞尔曲线，最终都会到达基本情况。

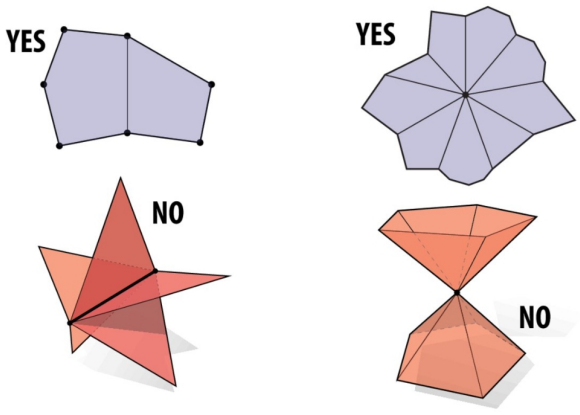
8.3 图形数据结构

上文中我们说明了一些常用的曲线以及它们的表示。图形学中的另一个主题就是如何把象征着这些表示方式的图形记录在计算机中。我们需要一些特定的数据结构来记录我们所需要的这些形状。

8.3.1 三角网格

工业中，我们常用三角形的面的集合来近似地表示各类集合。用呈网络装的三角面组成的图形被我们称为**三角网格**（triangular mesh）。我们在第一节中曾经提到过流形（manifold）的概念。一般地，我们都希望三角网格也组成流形。检查一个三角网格是不是流形，需要确认以下几点：

- 每条边存在于至少一个三角形中，存在于至多两个三角形中。
- 每个顶点围绕着的三角形形成扇状（fan），而不能形成鳍状（fin）。



在上图[1]中，YES 中的两个就是扇状流形，而 NO 中的两个就是鳍状非流形。

[1] http://15462.courses.cs.cmu.edu/fall2018/lecture/meshes/slide_013

最后，对于单个三角形而言，有一个明确的朝向也是很重要的。按照约定，三角形的前面（front）指的是当它的三个顶点被以逆时针的顺序排列时的方向。一个三角网格中，当且仅当任意一对相邻三角形的朝向一致时，我们说这个网格为**统一朝向**（consistently oriented）的。

8.3.2 存储三角网格

一个最简单的思路就是将三角形作为一个单独的数据结构保存，每个三角形独立记录它的三个顶点的位置。

```
struct Triangle {
    Vec3 vertexPosition[3];
}
```

但是，这样做会出现几个问题。

- 对于一个同时处于 n 个三角形的顶点，这个顶点会被记录 n 次。
- 由于顶点的坐标是浮点数，难以确保不同三角形中共同的顶点会被作为同一个点记录。

这里提及的第二个问题可以通过声明 `Vertex` 结构来解决，让三角形记录 `Vertex` 数组，而不是点的数组即可。

```
struct Vertex{
    Vec3 position;
}

struct Triangle {
    Vertex vertexPosition[3];
}
```

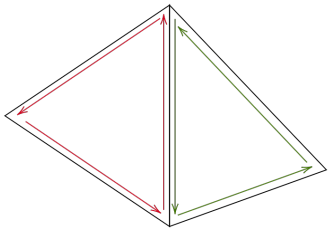
但是，这样的数据结构难以支持我们对三角网格进行编辑。我们需要一个能解决以下问题的数据结构。

- 能够返回与当前三角形相邻的三角形。
- 给定一条边，能够返回这条边两侧的三角形。
- 给定一个顶点，能够返回其处于的所有面。
- 给定一个顶点，能够返回其处于的所有边。

最暴力的方法就是像上面一样，单独地存储点、边、面、三角形的数据结构，然后每次查询都暴力地查询所有点、边、面、三角形。但显然我们可以做的更好。有一些常见的数据结构，例如**相邻三角形结构**（triangle-neighbor structure）、**翼边结构** (winged edge structure)，在这里，我们介绍一个比较高效的数据结构——**半边数据结构**（half-edge structure）。

8.3.3 半边数据结构

在半边数据结构中，我们认为每一条边都由两条方向相反的半边构成。这样做的好处是，观察到对于一条边，它的两个半边分别处于一个三角形中，且这两个三角形可以统一朝向。我们让每一条半边同时记录两个指针，`twin` 和 `next`，分别指向在同一个三角面内，它在这条边上对侧的那条半边，以及这条半边走完之后的下一个半边。注意，只要三角网格是流形，每个半边就只有可能出现在一个三角形内。每个半边只需要记录一个顶点——它的起点，因为它的终点也就会是 `next` 的起点。



```
struct HalfEdge {
    HalfEdge *twin;
    Halfedge *next;
    Vertex *v;
    Face *f;
}
```

有趣的是，对于一个面而言，它只需要知道其中的一个半边就行，因为半边可以通过遍历，自动地捕获这个面内的所有半边。类似地，对于一个顶点，我们也只要存储任何一条指向它的半边即可。

```
struct Vertex {
    HalfEdge *h;
    // ...per vertex data
}

struct Face {
    HalfEdge *h;
    // ...per face data
}
```

因为可以很容易地获得与边相邻三角形、与点相邻三角的信息，这就能让我们很快速地处理一些网格操作。

8.4 网格编辑操作

网格编辑操作（Mesh Edit）指的是通过网格中的数据结构实现基于顶点、边、面的操作。我们这里重点讨论两种操作：细分和减面。

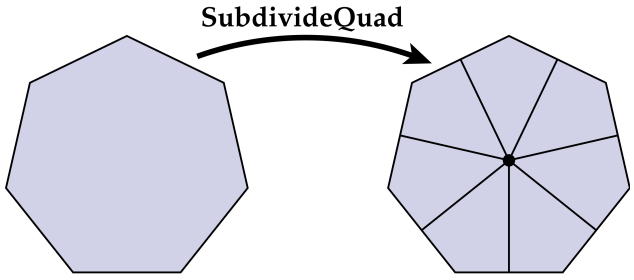
8.4.1 细分

在之前的反采样中，我们曾经提出过将一个像素划分成多个像素，然后进行采样以提高精确度的做法。在几何编辑中，我们也可以通过将一个面划成多个面，然后提高网格编辑操作的精度。这种做法就叫做**细分**（subdivision）。我们在这里讨论三种常见的细分算法。

线性细分

线性细分（linear subdivision）支持将一个 n 边形（$n \geq 4$）划分成为 n 个四边形。
线性细分规则很简单。其实基本上和上图表示的内容完全一样。

- 首先，计算原始 n 边形顶点的算术平均（直接全部加起来除以 n），记为新的顶点 $\mathbf{c}$。
- 然后，计算原始 n 边形每条边的中点（可以通过原始多边形的端点求算术平均）。
- 最后，将 $\mathbf{c}$ 与每边中点相连。



Catmull-Clark 细分

Catmull-Clark 细分与线性细分的规则几乎一样，除了新顶点的位置不同。Catmull-Clark 细分会对原多边形顶点进行加权平均，以确保在多次细分过程中面的分布依然合理平滑。

- 首先，计算原始 n 边形顶点的算术平均（直接全部加起来除以 n），记为面顶点 $\mathbf{c}$。（与线性细分相同）
- 然后，对于原始 n 边形的每条边，设置一个边顶点 $\mathbf{m}$：位置边的两侧的面顶点 $\mathbf{c}_1, \mathbf{c}_2$，以及边的两个端点 $\mathbf{E}_1, \mathbf{E}_2$ 的平均值 $\frac{\mathbf{c}_1+\mathbf{c}_2+\mathbf{E}_1+\mathbf{E}_2}{4}$。
- 最后，对于原始 n 边形的每个顶点 $\mathbf{p}$ 设置新位置，计算这个点所在的所有面的面顶点的算术平均 $\mathbf{F}$ 以及这个点所在的所有边的中点的算数平均 $\mathbf{R}$，然后将其移动到 $\frac{\mathbf{F}+2\mathbf{R}+(n-3)\mathbf{p}}{n}$。

Loop 细分

Loop 细分[2]是一种三角面网格的细分方式。简单地概括 Loop 细分的规则就是：

- 首先，将一个三角形通过边的中点两两相连，划分成四个三角形。
- 然后，通过周围顶点的加权平均，调整各个顶点的位置。

8.4.2 减面

8.5 点云

特别地，我们在这里单独讨论一下**点云**（point cloud）这种图形数据结构。显然，点云是一种显式的图形数据结构，它由大量空间中的点组成的数据集合组成，这些点通常代表了三维空间中物体的表面，因此，这个点和数学上的点通常并不指代同一个东西。

对于点云中的每个点，它们通常包含位置信息，也可能包含其它的数据，例如颜色、强度和法线。点云的数据可以通过各种方法获得，而我们要单独讨论点云也是因为现在的很多 3D 扫描设备都依然在使用点云结构，例如 iPhone 和 iPad 的 Pro 系列上携带的激光扫描仪（LiDAR）、立体摄像等。

8.5.1 数据结构

点云通常使用下列数据结构来存储：

[2]得名于提出者 Charles Loop，因此 Loop 不能翻译成循环。

- 数组或列表：最简答的方式就是使用动态数组，每个元素代表一个“点”，表示该点的属性（位置、颜色等）。
- 空间分割结构：为了提高处理效率，点云也可能被存储在八叉树、k-d 树中，这些结构可以加速空间搜索、邻近点搜索等操作。

8.5.2 优劣势

点云拥有以下的优势：

1. 高度详细：点云能够以非常高的精度去描述物体的表面和形状。
2. 现实交互：点云是直接从物理世界通过扫描技术获取图形信息的最快方式，这使得它们非常适合于用于处理现实世界的复杂场景的建模。

但是，点云也有一些难以解决的劣势。

1. 数据量大：点云通常包括大量的数据点，处理和存储这些数据都需要用大量的空间和时间资源。
2. 缺乏拓扑信息：通常，点云的数据点之间并没有任何连接信息，这使得在一些应用中它的使用受限。
3. 渲染挑战：直接渲染点云可能是一个对于计算资源的挑战，尤其是对于非常大的数据集合。

8.5.3 支持方法

点云提供的数据能够支持多种处理和分析方法，包括：

1. *3D* 重建：通过点云数据构建物体的三维模型。
2. 数据压缩和简化：减少数据点的数量来降低精度，简化数据集，但同时我们也保留重要的几何特征。
3. 配准：将多个点云数据集对齐，形成一个统一的三维表示。
4. 特征提取：从点云中提取有图形学意义的信息，比如边缘、角点、平面等。

总之，点云是一种表示现实世界三维空间中的物体和形状的有效方法，尤其适用于高精度的场景建模和分析。然而，由于其通常包含大量数据且缺少结构化的信息，因此在处理和渲染方面存在一定的挑战。

Chapter 9

辐射度量学

之前我们看到过光栅化以及经验模型的渲染逻辑，它们大多数都采用的是模拟的方式，以一种并不精确的语言来描述光线的行为。在现代较为前沿的渲染领域，我们更多的时候希望以更加精确的模型来描述光线的行为。我们从物理的角度出发，以描述光子的行为来定量地、精确地描述光线。这样的渲染方式被称为**基于物理的渲染**（Physically Based Rendering, PBR）[1]。在本章节中，我们将学习一些基础的 PBR 理论，从**辐射度量学**（Radiometry）出发，研究光线追踪的理论基础。

9.1 辐射度量物理量

辐射度量学是研究各种电磁辐射强弱的学科，包含可见光，在图形学中，我们就研究光子的辐射行为，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)。

下面，我们就来研究一些具体的物理量。

9.1.1 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

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

[1] 欢迎大家踊跃报名 CMU 15-468/668/868 课程学习更多的 PBR 相关理论实践。

其中，h 为普朗克常量[2]，c 为真空下的光速[3]。

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。

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，我们不研究特定波段上的光学性质，因此我们都忽略波长[4]。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。

9.1.2 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。

9.1.3 辐射照度

我们接下来定义单位面积内的通量。

定义（辐射照度）**辐射照度**（Irradiance）指的是单位面积内的辐射通量，即

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

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

[3] $c = 299792458 m/s$

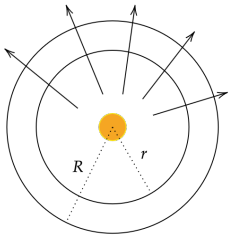
[4] 在 15-462/662 中，光谱能量（及之后的与波长相关的物理量的对应量）被提及，在 GAMES101 中则被跳过，在 Fundamentals of Computer Graphics 中则只讨论光谱能量。

简称辐照度，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。如下图所示。



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

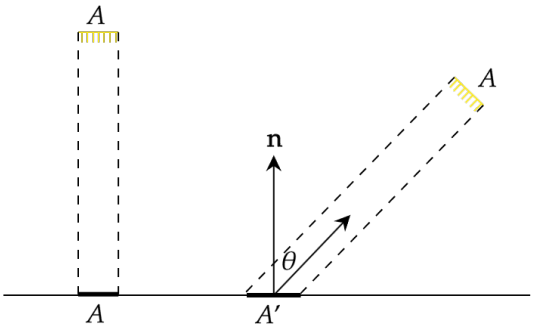
$$E(\mathbf{p}_o) = \frac{\Phi}{4\pi R^2}$$

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。

朗博定律

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。**朗博定律**描述的就是这个现象。

定义（朗博定律）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

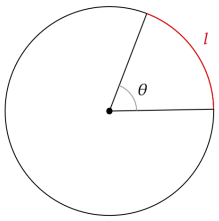
$$E = \frac{\Phi \cos\theta}{A}.$$

9.1.4 立体角与微分立体角

立体角的概念

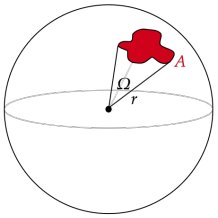
在定义下一个概念辐射亮度之前，我们需要先有一个立体的几何概念，叫做**立体角**。在研究立体角之前，我们先确认一下平面角的定义。

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。

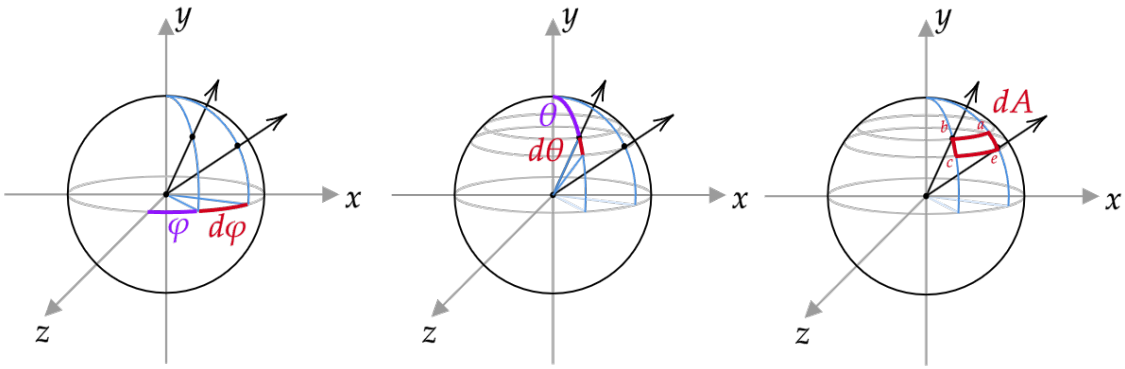
定义（立体角）**立体角**（Solid Angle）指的是该角对应不规则面的面积对球半径的平方的比值，即 $\Omega = A/r^2$ 或 $A = \Omega r^2$。立体角拥有形式单位 steradian，简写为 sr。这是一个无量纲量，量纲为 1，但是不可忽略。



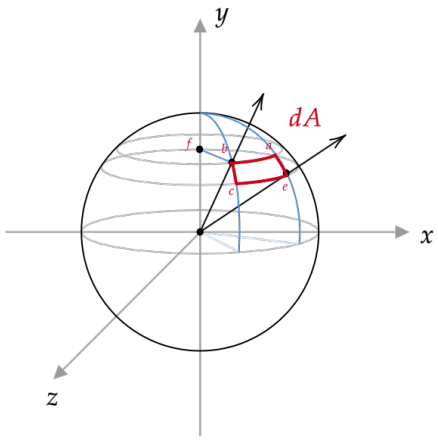
微分立体角

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。

9.1.5 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。

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

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

或者等价地，辐射照度是辐射亮度对立体角的半球积分。

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

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。对于一个给定的方向，亮度都不会发生变化。

9.2 散射模型

我们刚才研究的光线的行为都假设光线接触的物体表面是光滑的。现实中，我们实际上会通过各种各样的模型来描述不同的表面材质。

9.2.1 双向反射分布函数

因为我们比较关心物体表面最终被渲染出来的样子，所以我们需要一个模型来描述表面是如何反射光的。直觉上来说，我们关心两个方向：光线进入的方向 $\vec{\omega_i}$ 以及光线被反射后离开的方向 $\vec{\omega_o}$。我们要描述光线在方向 $(\vec{\omega_i}\to\vec{\omega_o})$ 上的**反射率**（reflectance）。

现在问题来了，用哪个物理量来描述这个反射率比较好呢？显然，如果换了一盏更亮的灯以同一个方向照射同一个表面，反射率不应该发生变化，反射率与光源有多亮显然不应该有关系。那么，可以直接检测出射方向的辐射亮度吗？我们假设我们拥有一个辐射亮度计，这个亮度计可以检测 $\Delta\sigma$ 这么大立体角的辐射亮度。为了使得这个亮度计测量精确，我们需要待检测的光线也覆盖超过 $\Delta\sigma$；然而现实中，如果出现这么一种情况，那么显然也会同时检测到来自其它方向的光。因此，使用辐射亮度并不是一个好办法。

那么，有没有一个物理量，它并需要关注被光照射的面积呢？有的，就是我们之前提到的辐照度。图形学中，我们的解决方案是使用**双向反射分布函数**。

定义（双向反射分布函数）对于一个材质，给定其接收光线的入射方向 $\vec{\omega_i}$ 以及指定的反射方向 $\vec{\omega_o}$[5]，我们定义其**双向反射分布函数**（Bidirectional Reflectance Distribution Function, BRDF）为反射辐射亮度与入射辐射照度的比值，即

$$f_r(\vec{\omega_i}\to\vec{\omega_o}) = \frac{dL_o(\vec{\omega_o})}{dE_i(\vec{\omega_i})}[sr^{-1}].$$

BRDF f_r 有如下的性质。

[5]记作 $\vec{\omega_i}\to\vec{\omega_o}$，这里的右箭头只表示方向，没有趋近的数学意义。

- $f_r(\vec{\omega_i} \to \vec{\omega_o}) \geq 0, \forall i, o.$
- $\int_{H^2} f_r(\vec{\omega_i} \to \vec{\omega_o}) \cos\theta d\vec{\omega_i} \leq 1.$
- $f_r(\vec{\omega_i} \to \vec{\omega_o}) = f_r(\vec{\omega_o} \to \vec{\omega_i}).$

9.2.2 折射

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[6]、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。

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。

斯涅尔定律

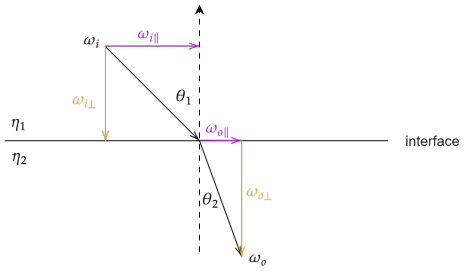
光线从一种介质进入另一种介质时会发生偏折，这种现象就叫做**折射**（refraction）。描述折射光线方向的定律就是**斯涅尔定律**。

[6]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。

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[7]，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 $\omega_{o\parallel}$

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

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.

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

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

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）。

9.2.3 散射

对于光子而言，除了被反射以及被折射，还有一部分会在另一个介质中来回弹射后再离开这个表面。这种现象就被称作**散射**（scattering）。有的时候，光甚至不会从进入的点再次离开表面，而会从表面上另外某个点上离开。这种现象就被称作**次表面散射**（subsurface scattering）。

与 BRDF 类似，给定入射方向和透射方向，我们给出透射出的亮照比的函数就被称作**双向透射分布函数**（Bidirectional Transmittance Distribution Function, BTDF）。给定入射方向、散射方向、入射点和散射点之后计算出的亮照比的函数就被称作**双向次表面散射反射分布函数**（Bidirectional Subsurface Scattering Reflectance Distribution Function, BSSRDF）。这些函数被我们统称为**双向散射分布函数**（Bidirectional Scattering Distribution Function, BSDF）.

以本笔记的范围，我们不会深入讨论 BTDF 和 BSSRDF[8]。我们仅需知道它们为渲染半透光材质（例如皮肤、玉石、参与介质）提供了重要的理论基础。

[8]欢迎大家踊跃报名 CMU 15-468/668/868 课程学习更多的 PBR 相关理论实践。

9.3 渲染方程

渲染方程是光线追踪的理论基础。它提出了辐射亮度的计算方式。

定义（渲染方程）给定所求点 $\mathbf{p}$ 以及一个从点 $\mathbf{p}$ 出发的方向向量 $\vec{\omega_o}$，我们认为其沿着这个方向的辐射亮度分为以下三个部分。

1. $L_e(\mathbf{p}, \vec{\omega_o})$：如果点 $\mathbf{p}$ 处于一个光源上，那么它拥有这一项。这一项被称为发光项，它的值为该光源从点 $\mathbf{p}$ 向方向 $\vec{\omega_o}$ 的出射辐射亮度。
2. $\int_{H^2} f_r(\mathbf{p}, \vec{\omega_i} \to \vec{\omega_o}) L_i(\mathbf{p}, \vec{\omega_i}) \cos\theta d\vec{\omega_i}$：其中 f_r 为 BRDF。这一项计算半球内（因为我们认为南半球的光线无法提供入射辐射亮度）所有方向的进入点 $\mathbf{p}$ 辐射亮度，计算的方式是通过取某一方向 BRDF 乘该方向入射辐射亮度乘法线夹角余弦的积分。

因此，**渲染方程**（Rendering Equation）表示为从某一点离开的辐射亮度等于该点自发光沿该方向离开的辐射亮度以及所有其它方向来到该点并从该点沿着该方向离开的辐射亮度的总和，即

$$L_o(\mathbf{p}, \vec{\omega_o}) = L_e(\mathbf{p}, \vec{\omega_o}) + \int_{H^2} f_r(\mathbf{p}, \vec{\omega_i} \to \vec{\omega_o}) L_i(\mathbf{p}, \vec{\omega_i}) \cos\theta d\vec{\omega_i}.$$

需要注意的是，渲染方程是递归的，因为其中 $L_i(\mathbf{p}, \vec{\omega_i})$ 一项也完全遵循着渲染方程的求解过程。如果我们要将 BRDF 换成 BTDF 或者 BSSRDF，也只需要去调整 f_r 即可。

渲染方程的递归属性使得其非常需要保证每一次迭代的性能足够优秀。否则，递归次数一旦增加，渲染方程的计算就会极其昂贵。另外，渲染方程中牵涉到积分计算，而定积分计算在计算机领域也是一个昂贵的操作。因此，我们需要简化渲染方程的实现，这一部分内容会在下一章节详细介绍。

Chapter 10

光线追踪导论

光线追踪作为目前主流的写实渲染的主要手段，在图形学和工业界都有非常重要的地位。本章主要介绍目前主流的光线追踪算法，以及背后的一些数学原理。

10.1 与光栅化的对比

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）。

10.2 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()
}
```

10.3 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。

10.3.1 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 R_i 发生的概率被记作 $P\{E_i\}$。

10.3.2 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)。

10.3.3 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

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

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

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

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 $X \sim f$。

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]对于随机变量，我们通常使用斜体大写字母表示。

1. 对于任何实数 a，$P\{X = a\} = 0$。

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。

10.3.4 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。

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，

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

10.3.5 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

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

对于 n 个独立随机变量，我们有

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

10.3.6 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

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

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。

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

10.3.7 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

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

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。

10.4 随机采样

我们的数学工具已经准备的差不多了。下面先利用以上的知识解决一些基本的问题。例如，当我们说“在一个球内随机选一个点”。我们应该如何随机才能保证球内每个点被选中的概率是一样的呢？我们在这里提供一些方法，但是不予证明，有兴趣的读者可以自行查阅概率论相关书籍文献。

在开始复杂随机采样之前，我们先假设我们有一个函数 `unif(0,1)`。这个函数可以均匀地返回一个 0 到 1 之间的随机值，也就是说 `unif(0,1)` $\sim f(x)$，其中 $f(x) = 1$ 对于 $x \in [0,1]$，$f(x) = 0$ 对于其它的 x 。

10.4.1 反密度函数

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 `unif(0,1)` 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。

Example 10.1. 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

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

置换 x 与 y，我们得到

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

利用 `unif(0,1)` 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。 ■

10.4.2 拒绝采样

拒绝采样（rejection sampling）的核心思想是我们根据某种简单的分布随机采样一些点，然后拒绝其中不服从更复杂分布的点。我们介绍三种应用场景。

单位圆内选择一点

我们先在 $[-1,1]^2$ 上随机选择 (x,y) 数对，然后剔除那些在圆外的点即可。

```
Vec2 RandPointInUnitCircle() {
    bool done = false;
    float x,y;
    while(!done) {
        x = -1 + 2 * unif(0,1);
        y = -1 + 2 * unif(0,1);
        if(x^2 + y^2 < 1){
            // in the circle
            done = true;
        }
    }
    return Vec2(x,y);
}
```

单位球面上选择一点

这个方法通常会被用于生成一个随机方向。我们在单位球内随机生成一点，然后用这个球心和这个点的连线归一化后作为生成的方向。

```
Vec3 RandPointOnUnitSphere() {
    bool done = false;
    float x,y;
    while(!done) {
        x = -1 + 2 * unif(0,1);
        y = -1 + 2 * unif(0,1);
        z = -1 + 2 * unif(0,1);
        if(x^2 + y^2 + z^2 < 1){
            // in the sphere
            done = true;
        }
    }
    return Vec3(x,y,z).normalize();
}
```

10.4.3 Metropolis 方法

Metropolis[2]方法是一种马尔科夫链蒙特卡洛方法（MCMC）的应用，这个概念我们会在接下来的后文中介绍。Metropolis 方法也提供了一种随机生成以 $f(x)$ 为密度函数的随机变量的方式。

（**初始化**）选择一个起始点 x_0 作为马尔可夫链的初始状态。

（**迭代过程**）

- 对于当前状态 x_i 进行以下步骤：从某个建议分布中生成一个候选状态 x'，这个建议分布通常是以 x_i 为中心的对称分布，比如正态分布。
- 计算通过概率 $\alpha = \min(1, f(x')/f(x_i))$，其中 f 是我们目标分布的概率密度函数。
- 生成 $u = \texttt{unif(0,1)}$。
- 如果 $u \leq \alpha$，则接受状态 x'，即令 $x_{i+1} = x'$，否则，保持当前状态不变，即 $x_{i+1} = x$。

重复上述步骤直到马尔科夫链收敛。收敛后，链中的状态可以视为来自目标分布的样本。

10.5 蒙特卡洛光线追踪

虽然我们上面介绍了这么多概率论方面的概念，但实际上用在光线追踪中的部分都是很直观、很方便理解的一些概念。要用严格的数学证明我们接下来会得出的结论，我们都要用到上面的数学工具，但很快我们就会发现实际上蒙特卡洛光线追踪是很容易理解、也很智慧的处理方式。

10.5.1 蒙特卡洛积分

在实际的计算中，我们通常采用**数值积分**（numerical integration）的方式去计算积分，即用累加的形式去近似地计算函数曲线与 x 轴围成的面积，而非精确计算积分。这样做的原因是因为积分计算实际上是很难进行的，而数值积分则可以以很高的近似程度以及快速得多的计算达到类似的效果。我们在性能和效果采取了明显性价比高的方式。

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。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。

[2]Metropolis 是个人的名字，所以我们不能译作大都市方法。

定义（蒙特卡洛估值）对于函数 $f(x)$ 在区间 $[a,b]$ 的积分估计值，它的**蒙特卡洛估值**（Monte Carlo Estimator）通过以下方式计算：
定积分：$\int_a^b f(x)dx$;
随机变量：随机采样 $X_i \sim p(x)$;
蒙特卡洛估值：$F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}$.

虽然根据这个定义以及我们刚才的分析，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计法也被叫做**重要性采样**（importance sampling），它可以显著地降低 f/p 的方差。

10.5.2 蒙特卡洛光线追踪

渲染方程的蒙特卡洛估值

终于到达了激动人心的蒙特卡洛光线追踪的部分了。蒙特卡洛光线追踪的基本思想就是利用蒙特卡洛积分去估计渲染方程中的积分数值。每次我们随机采样一些光线，然后利用我们随机采样的追踪结果的均值作为当前像素辐射亮度的结果。

回顾一下渲染方程，我们先忽略其中的发光项。

$$L_o(\mathbf{p},\omega_o) = \int_{\Omega^+} L_i(\mathbf{p},\omega_i) f_r(\mathbf{p},\omega_i \to \omega_o)(\mathbf{n}\cdot\omega_i)d\omega_i$$

这里是对半球内所有方向的积分。

我们希望求积分的函数是：

$$f(x) = L_i(\mathbf{p},\omega_i) f_r(\mathbf{p},\omega_i \to \omega_o)(\mathbf{n}\cdot\omega_i)$$

目前我们先假设半球内任何方向都是均等重要的，那么

$$p(\omega_i) = \frac{1}{2\pi}$$

因此，我们得到这个函数的蒙特卡洛估值为

$$\begin{aligned} L_o(\mathbf{p},\omega_o) &\approx \frac{1}{N}\sum_{i=1}^{N}\frac{L_i(\mathbf{p},\omega_i) f_r(\mathbf{p},\omega_i \to \omega_o)(\mathbf{n}\cdot\omega_i)}{1/2\pi} \\ &= \frac{2\pi}{N}\sum_{i=1}^{N} L_i(\mathbf{p},\omega_i) f_r(\mathbf{p},\omega_i \to \omega_o)(\mathbf{n}\cdot\omega_i) \end{aligned}$$

光线追踪算法草稿

下面，我们就根据我们刚才总结出来的知识，写一下蒙特卡洛光线追踪的伪代码算法。

```
float Radiance(Point p, Vec3 wo) {
    Randomly select N directions with wi ~ 1/(2*pi);
    float Lo = 0.0;

    for each selected direction wi{
        Ray r = Ray(p, wi);
        HitInfo hit = trace(r);
        if(hit.isHit) {
            if(hit.isLight) {
                // hit a light, direct lighting
                float Li = RadianceOfLight(hit.object, wo);
                Lo += (1/N) * Li * fr * cos * 2 * pi;
            } else {
                // hit an object, indirect lighting
                Lo += (1/N) * Radiance(hit.p, -wi) * fr * cos * 2 * pi;
            }
        }
    }

    // misses
    return Lo;
}
```

上面的这个算法中，存在两个致命的问题。

1. 每次采样 N 个方向，采样次数会指数级增加。很快就会耗尽计算机内存。
2. 迭代并不一定能到达基础情况，不一定会停止。

第一个问题很好解决。只要我们每次只采样一个方向，那么采样次数就不会随着迭代深入而增加。第二个问题的解决需要我们引入下一个概念，俄罗斯轮盘赌。

10.5.3 俄罗斯轮盘赌

俄罗斯轮盘赌在概念上很简单，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 $\frac{1}{n}$。在光线追踪的迭代过程中，我们也可以这么做。每次光线追踪有 $1-p_{rr}$ 的概率暴毙。

定义（俄罗斯轮盘赌）在迭代过程中，使用新的估值 X_{rr} 来代替原蒙特卡洛估值。以 p_{rr} 的概率，我们正常计算当前迭代的值，以 $1-p_{rr}$ 的概率直接将本次迭代的值记作 0。这样的方式就叫做**俄罗斯轮盘赌**（Russian Roulette）。

观察到，

$$E(X_{rr}) = p_{rr}E(\frac{X}{p_{rr}}) + (1-p_{rr})E(0) = E(X).$$

因此，从期望上来看，采用轮盘赌并不会影响正确性。根据这一性质我们修改上面的代码，就能确保每一个迭代都能走到最终的深度。

```
float PRR; // Russian roulette survival chance
float Radiance(Point p, Vec3 wo) {
```

```
        float a = unif(0,1);
        if(a > PRR) return 0.0;

        Randomly select 1 directions wi ~ 1/(2*pi);
        float Lo = 0.0;

        Ray r = Ray(p, wi);
        HitInfo hit = trace(r);
        if(hit.isHit) {
            if(hit.isLight) {
                // hit a light, direct lighting
                float Li = RadianceOfLight(hit.object, wo);
                Lo += ((1/N) * Li * fr * cos * 2 * pi) / PRR;
            } else {
                // hit an object, indirect lighting
                Lo += ((1/N) * Radiance(hit.p, -wi) * fr * cos * 2 * pi) / PRR;
            }
        }
        // misses
        return Lo;
    }
```

至此，我们就完成了蒙特卡洛光线的基本内容。

Chapter 11

动画

11.1 关键帧

11.2 样条

11.2.1 B 样条

11.2.2 NURBS 曲线与曲面

11.2.3 三次多项式

11.3 动力学

11.4 优化

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

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”。

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.1.1 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.1.2 隐函数求导

三元方程 $F(x,y,z)=0$ 确定一个二元隐函数 $z=f(x,y)$，其偏导数为

$$\frac{\partial z}{\partial x}=-\frac{F_x}{F_z}, \frac{\partial z}{\partial y}=-\frac{F_y}{F_z}$$

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 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.2.1 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

$$\varphi_1(x) \le y \le \varphi_2(x), a \le x \le b$$

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.2.2 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

$$\iint_D f(r\cos\theta, r\sin\theta) r dr d\theta = \int_\alpha^\beta \left[\int_{\varphi_1(\theta)}^{\varphi_2(\theta)} f(r\cos\theta, r\sin\theta) r dr \right] d\theta.$$

A.2.3 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**雅可比项**（Jacobian term）定义为

$$J(u,v) = \begin{bmatrix} \frac{\partial x}{\partial u} & \frac{\partial x}{\partial v} \\ \frac{\partial y}{\partial u} & \frac{\partial y}{\partial v} \end{bmatrix}$$

其**雅可比行列式**（Jacobian determinant）为

$$|J(u,v)| = |\det(J)| = |\frac{\partial x}{\partial u}\frac{\partial y}{\partial v} - \frac{\partial x}{\partial v}\frac{\partial y}{\partial u}|$$

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

$$\iint_D f(x,y)dxdy = \iint_{D'} f[x(u,v), y(u,v)] \cdot |J(u,v)| du dv$$

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

$$|\frac{\partial x}{\partial u}\frac{\partial y}{\partial v} - \frac{\partial x}{\partial v}\frac{\partial y}{\partial u}| = |-1/2| = 1/2.$$

利用换元公式可得，

$$\iint_D e^{\frac{y-x}{y+x}} dxdy = \iint_D e^{u/v}\frac{1}{2} du dv = \frac{1}{2}\int_0^2 dv \int_{-v}^{v} e^{u/v} du = \frac{1}{2}\int_0^2 (e - e^{-1}) v dv = e - e^{-1}.$$

■

A.3 三重积分

A.3.1 直角坐标计算三重积分

A.3.2 柱面坐标计算三重积分

A.3.3 球面坐标计算三重积分

A.3.4 重积分应用

曲面面积

质心

转动惯量

引力

A.4 曲线积分

A.4.1 对弧长曲线积分

A.4.2 对坐标曲线积分

A.4.3 格林公式

A.5 曲面积分

A.5.1 对面积曲面积分

A.5.2 对坐标曲面积分

A.5.3 高斯公式

A.6 级数

附录 B

线性代数

附录 C

C++/C++11

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。
- 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。

[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。

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
    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。

[2]事实上很多 CPU 只能读取特定位置的数据。

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。

```
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。

```
#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。

```
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

这个部分是在图形学中看似被草草带过，但实际上非常重要的一个特性。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

图形 API

图形 API 是利用了 GPU 特性而进行图形编程的接口。不同的图形 API 之间有着不同的设计哲学，但是大多数都反映对于计算机而言比较重要的三个环节——数据缓冲区（data buffer）、状态机（state machine）和着色器编程（shader programming）。我们在这里以 OpenGL 和 Apple Metal 为例，提供两种图形 API 的解读。特别地，我们还将介绍一些 Unity 的 ShaderLab 框架。必要时请查阅各自的开发文档。

对于任何图形 API，有三个根本的概念非常重要，分别是缓冲、状态以及着色器。无论是什么 API，都会需要用到它们。**缓冲**（buffer）指的是用来存储可以被 GPU 处理的数据的内存区域。**状态**（state）指的是显卡会设置好渲染状态，然后在该状态被解除之前，一直对状态指定的对象进行修改的特性。**着色器**（shader）则是 GPU 在渲染管线各个阶段进行逐顶点或是逐像素的计算的程序。

D.1 OpenGL

OpenGL 是一种 C 风格的图形 API，现代 OpenGL 几乎都使用 C++ 进行编程。在现代 OpenGL 中，立即模式（immediate mode）已经过时，取而代之的是使用顶点缓冲对象作为 CPU 至 GPU 的传达单位。OpenGL 使用的着色语言被称作 GLSL。

D.1.1 初始化

初始化步骤主要是为了打开 OpenGL 程序的窗口，开启我们的渲染循环。以 GLFW 库为例，在 OpenGL 的主程序（例如 `main.cpp`）中，我们通过以下代码设置好一个可以打开渲染窗口的程序。

```
#include <stdio.h>
#include <glew.h>
#include <glfw3.h>

// 设置窗口大小
const GLint WIDTH = 1920, HEIGHT = 1080;

int main() {
    // 通过 glfwInit() 初始化窗口，如果不成功则直接终止
    if(!glfwInit()) {
        std::cout << "GLFW␣Initialization␣failed." << endl;
        glfwTerminate();
        return 1;
```

```
}

// 设置窗口的属性
// 设置OpenGL的版本
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);

// 设置core profile，一般是不支持向后兼容的
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
// 允许向前兼容
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);

// 打开窗口
GLFWwindow *mainWindow = glfwCreateWindow(WIDTH, HEIGHT, "Hello World", NULL, NULL);
// 如果窗口创建不成功，也直接终止
if(!mainWindow) {
    std::cout << "GLFW Window creation failed." << endl;
    glfwTerminate();
    return 1;
}

// 设置帧缓存信息
int bufferWidth, bufferHeight;
glfwGetFrameBufferSize(mainWindow, &bufferWidth, &bufferHeight);

// 将上下文设置给GLEW
glfwMakeContextCurrent(mainWindow);

// 支持现代扩展
glewExperimental = GL_TRUE;

// 如果 GLEW 创建不成功，也直接终止
if (glewInit() != GLEW_OK) {
    std::cout << "GLEW initialization failed." << endl;
    // 销毁窗口
    glfwDestroyWindow(mainWindow);
    glfwTerminate();
    return 1;
}

// 创建视窗大小
glViewport(0, 0, bufferWidth, bufferHeight);

// 开启渲染循环
unsigned int buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);

// 只要窗口没有关闭，我们就不断进行这个循环
while(!glfwWindowShouldClose(mainWindow)) {
    // 获取用户输入事件的处理器
    glfwPollEvents();

    // 清除上一帧窗口的颜色和深度信息
    glClearColor(0.f, 0.f, 0.f, 1.f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // 调换前后帧
    glfwSwapBuffers(mainWindow);
}
```

```
    // 一切成功，以0值返回
    return 0;
}
```

整个初始化步骤的看似冗长，但实际上确实都是符合我们渲染管线绘制逻辑的必要步骤。在上面的步骤中，有以下几个概念需要说明。

GLEW 与 GLFW 的使用

GLEW（OpenGL Extension Wrangler Library）的主要功能是加载和管理 OpenGL 的扩展，以及管理 OpenGL 的函数指针。由于不同的平台和不同的显卡支持的 OpenGL 功能可能有所不同，GLEW 可以帮助开发者在运行时查询和加载可用的 OpenGL 扩展。

GLFW（Graphics Library Framework）则是一个专门用于 OpenGL 的轻量级工具库，提供了创建窗口、读取输入、处理事件等功能。它是一个独立于平台的 API，可以在多种操作系统上运行。OpenGL 本身不处理窗口创建和输入处理，而 GLFW 填补了这一空白，使得开发者可以在不关注特定操作系统细节的情况下，创建和管理 OpenGL 渲染窗口，处理用户输入。

帧缓存

帧缓存（frame buffer）指的是不同缓存区域（颜色缓存、深度缓存、模板缓存等）组成的缓存区域。通常，OpenGL 将场景渲染到帧缓存，然后对该帧缓存进行各种后处理效果，也可以将帧缓存的结果本身作为一张纹理在其它渲染过程中使用。

默认情况下，渲染发生在默认帧缓存（default frame buffer）上，由 OpenGL 上下文创建时自动创建，除了默认帧缓存之外，OpenGL 也允许我们创建自定义的帧缓存对象（Frame Buffer Objects, FBO）进行离屏渲染。离屏渲染通常是用来将渲染结果本身用作另一个渲染过程中的纹理，例如阴影映射、延迟渲染、屏幕空间效果、环境映射等。

双缓冲区显示状态

双缓冲区显示状态（double-buffered display state）指的是用前后两个缓冲区交替渲染的机制。在渲染循环的单次循环结束之后，前缓冲区的指针指向后缓冲区，然后后缓冲区的指针指向原来的前缓冲区，完成了两个缓冲区的交替（swap）。前缓冲区负责实际的屏幕颜色输出，而后缓冲区则负责对当前屏幕的修改。

D.1.2 顶点着色器

OpenGL 的**顶点着色器**（Vertex Shader）的作用也就是用来变换顶点、为片元着色器准备数据。通常，我们会将一个顶点着色器存入一个单独的（`.vert`）文件中。顶点着色器会以一个顶点类型作为输入，然后执行其 `main` 函数，将最终裁剪空间的顶点位置存放在 OpenGL 保留变量 `gl_Position` 中。

当我们在 OpenGL 中发出绘制命令，例如使用 `glDrawArrays` 或者 `glDrawElements` 的时候，OpenGL 就需要去处理其中的顶点，而在这个时候它就会自动调用当前激活的着色器程序去处理这个顶点。这也就意味着我们需要有一定的步骤去激活它。激活步骤我们会在下面的片元着色器中一起解释。

一个顶点着色器的代码示例如下：

```
#version 330 core // select the OpenGL version
layout(location=0) in vec3 in_Position; // declare an input vertex property
void main() {
    // save the result in  the gl_Position;
    gl_Position = vec4(in_Position, 1.0);
}
```

这里的 location=0 表示顶点位置数据从索引 0 的顶点属性中获取。

D.1.3 片元着色器

OpenGL 的**片元着色器**（Fragment Shader）的作用是用来给像素上色。通常，我们会将片元着色器存入一个单独的（.frag）文件中。片元着色器会输出到一个由用户自定名的变量中。

一个片元着色器的代码示例如下：

```
#version 330 core // select the OpenGL version
layout(location=0) out vec4 out_Color; // user defined output
void main() {
    out_Color = vec4(0.4. 0.4, 0.9, 1.0);
}
```

这里的 location 语义和顶点着色器中不同。片元着色器中的 location 指代的是输出变量缓冲区的位置。当着色器程序输出到多个渲染目标的时候，这个特性比较有用。

D.1.4 使用着色器

要完整的使用我们写好的顶点和片元着色器，我们需要经历以下的步骤：加载着色器代码、编译着色器、创建着色器程序并附加着色器、使用着色器程序。

加载着色器代码

首先，我们需要从文件中读取着色器代码。假设我们的顶点着色器为 a.vert，我们的片元着色器为 b.frag，我们可以通过下面的代码加载着色器代码。

```
std::string vertexShaderCode;
std::string fragmentShaderCode;
std::ifstream vShaderFile;
std::ifstream fShaderFile;

// 确保ifstream对象可以抛出异常：
vShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);
fShaderFile.exceptions(std::ifstream::failbit | std::ifstream::badbit);

try
{
    // 打开文件
    vShaderFile.open("a.vert");
    fShaderFile.open("b.frag");
    std::stringstream vShaderStream, fShaderStream;

    // 读取文件的缓冲内容到流中
    vShaderStream << vShaderFile.rdbuf();
    fShaderStream << fShaderFile.rdbuf();
```

```
        // 关闭文件处理器
        vShaderFile.close();
        fShaderFile.close();

        // 转换流至字符串
        vertexShaderCode   = vShaderStream.str();
        fragmentShaderCode = fShaderStream.str();
    }
    catch (std::ifstream::failure e)
    {
        std::cerr << "ERROR::SHADER::FILE_NOT_SUCCESFULLY_READ" << std::endl;
    }

    const char* vShaderCode = vertexShaderCode.c_str();
    const char* fShaderCode = fragmentShaderCode.c_str();
```

编译着色器

接下来，我们使用 OpenGL 的三个函数，可以把这个过程想象成流水线上生产一瓶饮料的过程：

- `glCreateShader`：用于创建一个处理硬件上的着色器代码的处理器，通常返回一个 `GLuint` 类型的整数 ID 作为标识。说白了就是创建了一个着色器的容器。
- `glShaderSource`：用于将着色器的代码加载进图形硬件的内存。说白了就是装填了着色器的内容。
- `glCompileShader`：用于实际上在硬件中编译着色器。说白了就是盖上了着色器的瓶盖。

编译两种着色器的代码如下：

```
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vShaderCode, NULL);
glCompileShader(vertexShader);
// 检查编译错误...

GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragmentShader, 1, &fShaderCode, NULL);
glCompileShader(fragmentShader);
// 检查编译错误...
```

创建着色器程序并附加着色器

名为着色器的饮料已经制造好了，下面就是将其提供给需要它的人去使用它了。这个过程可以理解为将货物装车的过程。这个过程我们称为链接，分为下面的几个步骤：

- `glCreateProgram`：用于创建一个装载之前已经写好的着色器的程序。说白了就是创建一辆装着色器的车。
- `glAttachShader`：用于将之前写好的着色器附加在程序上。说白了就是将造好的着色器饮料装车。
- `glLinkProgram`：用于内部着色器之间的链接。说白了就是发车。

参考代码如下：

```
GLuint shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
```

```
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
// 检查链接错误...
```

使用着色器程序

使用着色器仅需要通过 `glUseProgram()` 函数就可以实现了。

```
glUseProgram(shaderProgram);
// 现在可以渲染对象了...
```

D.1.5 顶点缓冲对象

顶点缓冲对象（Vertex Buffer Objects, VBO）是一个高性能的数据缓冲区，用来存储顶点数据，例如顶点坐标、颜色、纹理坐标、法线等。早期，我们可以通过形式上更简单的 `glBegin()` 和 `glEnd()` 调用直接传递 CPU 信息给 GPU，但是这种立即模式效率低下，因为每次都要从 CPU 向 GPU 传递顶点数据。VBO 的引入使得顶点数据可以预先存储在 GPU 内存中，避免了每次渲染的数据传输，提高了性能。

我们通常会使用一个 VBO 来存储整个三角网格的顶点数据。有的时候，在处理不同类型的顶点数据的时候，可能会使用多个 VBO，例如，将一个 VBO 用来仅存储顶点位置，另一个则仅用来存储顶点颜色，以此类推。使用不同的渲染配置的时候，我们可能也会有不同的 VBO。使用一个 VBO 的流程如下：创建 VBO、绑定 VBO、填充 VBO、配置顶点属性指针、解绑 VBO、在渲染循环中使用 VBO。

创建 VBO

首先，我们需要创建一个新的 VBO，并且获取一个唯一的 ID 来引用它。`glGenBuffers` 函数会负责生成一个合适作为新 VBO 的标识 ID 的整数。

```
GLuint VBO[1];
glGenBuffers(1, VBO);
```

如果我们一次要创建多个 VBO，我们也可以修改数组的长度，

```
GLuint VBO[4];
glGenBuffers(4, VBO);
```

绑定 VBO

由于 OpenGL 的状态机机制，我们需要让 OpenGL 知道我们接下来的操作是针对当前这个 VBO 的，因此我们需要通过 `glBindBuffer` 函数让该 VBO 成为当前的 VBO。

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

在这里，我们将 VBO 绑定到了一种缓冲类型上，在本例中是 `GL_ARRAY_BUFFER`。

填充 VBO

在前面的创建 VBO 部分，我们实际上并没有在内存中分配 VBO 的空间，而在这一步填充的过程中，我们会将顶点数据复制到 VBO 中，也会在这一步完成内存的分配。

```
GLfloat vertices[] = {-0.5f, -0.5f, 0.f, 0.5f, -0.5f, 0.f, 0.f, 0.5f, 0.f};
glBufferData(GL_ARRAY_BUFFER, sizeof(GLfloat), vertices, GL_STATIC_DRAW);
```

当顶点数据存储在 VBO 中后，我们需要告诉 OpenGL 如何解释这些数据。这是通过设置顶点属性指针完成的。

```
glVertexAttribPointer(0,3, GLfloat, 3 * sizeof(GLfloat), (void*)0);
glEnableVertexAttribArray(0);
```

在 `glVertexAttribPointer` 函数中，第一个参数代表顶点属性的索引。每个顶点属性（例如颜色、位置、纹理坐标）在着色器中都有一个对应的索引位置（也就是我们上面写过的 `location`）。我们仅仅将数据存储在 VBO 中是不足以让 OpenGL 知道如何去解释这些数据的。因此，我们需要 `glEnableVertexAttribArray(i)` 函数去启用这些顶点属性，其中 `i` 就是我们需要启用的属性的索引。

第二个参数代表顶点属性的大小，这里是 3，意味着顶点属性是一个由 3 个值组成的向量（比如一个三维向量）。第三个参数 `GLfloat` 指定数据的类型，第四个参数 `3 * sizeof(float)` 被称作**步长**（stride），指定连续顶点属性之间的间隔。这里的长度表示每个顶点的属性占用的字节大小。最后一个 `(void *) 0` 是在缓冲区起始位置的偏移量，这里是 0 表示从缓冲区的开始的位置。

解绑 VBO

完成设置后，可以解绑 VBO，以避免其他 VBO 调用意外修改这个 VBO。依然使用 `glBindBuffer` 函数。

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

在渲染循环中使用 VBO

渲染循环（Render Loop）指的是一个在应用程序运行期间不断重复的循环，在循环的每一次迭代中，您会执行渲染命令来绘制一帧。我们可以认为渲染循环的频率就是画面的帧率。在同一帧内，我们并不一定只绘制一个 VBO 里的内容，因此，在完成一个 VBO 的绘制之后，我们需要进行解绑，一来可以防止其他的绘制操作误操作在当前的 VBO 上，二来可以让我们绑定到下一个 VBO 上。

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glDrawArrays(GL_TRIANGLES, 0, 3);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

D.1.6 顶点数组对象

顶点数组对象（Vertex Array Object, VAO）用来存储一组顶点缓冲区的状态。它本质上是一个对象，存储了所有关于顶点数据布局和 VBO 状态的信息。这包括顶点属性的启用状态、顶点属性指针的位置和格式信息。通过 VAO，你可以一次设置顶点属性指针，然后在渲染时快速切换使用不同的顶点数据。这避免了每次绘制时都重复设置顶点数据状态的开销。

我们可以将 VAO 视作一本词典。当你要查阅一段 VBO 的内容的时候，你就从书架中取出一本 VAO，然后当你看完这段内容、或者想使用另一种解读方法的时候，就把这本 VAO 放回书架，然后取出另一本

VAO。对于每一本被取用的 VAO，其中的内容一旦录制完成，就不会再发生改变。

配置 VAO 的逻辑为：

- （初次）绑定 VAO。
- 绑定某个 VBO。
- 配置顶点属性（通过 `glVertexAttribPointer` 和 `glEnableVertexAttribArray`）。
- 解绑 VAO。

在渲染时使用 VAO 的逻辑：

- 绑定 VAO。
- 绑定一个 VBO（可以是与配置 VAO 时同一个 VBO，也可以是不同的）。
- 解绑 VAO。

需要注意的是，如果我们更换了 VBO，而且新的 VBO 中的顶点数据布局与初次配置 VAO 时的 VBO 不同，则可能会导致渲染错误。

```
// 创建 VAO
GLuint VAO;
glGenVertexArrays(1, &VAO);
// 初次配置 VAO
glBindVertexArray(VAO);
glEnableVertexAttribArray(0); // 激活顶点属性（索引位置0）
// 绑定 VBO
glBindBuffer(GL_ARRAY_BUFFER, triangleVBO[0]);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), 0);
// 解绑 VAO
glBindVertexArray(0);
```

D.1.7 案例 1：Blinn-Phong 着色器程序

D.1.8 案例 2：Shadow Mapping

D.2 Metal

D.3 Unity ShaderLab