第一章　二维图形

1.1 颜色

1.1.1 颜色的物理与感知

颜色的物理本质

定义 1.1 颜色 **颜色 (Color)** 是特定波长的光引起的视觉效应.

定义 1.2 可见光 可以被人眼感知的电磁波称为**可见光 (Visible Light)**. 其波长范围大约在 380nm 到 750nm 之间.

单一波长的光称为**单色光 (Monochromatic Light)**, 在自然界中比较少见. 而大多数自然界的光是由多种波长的光混合而成的, 称为**复色光 (Polychromatic Light)**[1].

复色光的形成有两种方式. 一种是发射光以加法形式组合, 例如红, 绿, 蓝三种色光混合形成白光; 另一种是吸收光以减法形式组合, 例如青, 品红, 黄三种颜色的颜料混合形成黑色.

在计算机图形学中, 我们主要研究的是显示器等发射光的设备, 因此主要使用加法颜色模型来表示颜色. 在数学上, 可以用 $I(\lambda)$ 表示复色光中各波长 λ 的光强.

颜色的感知

人眼对光的感知是通过视网膜上的视锥细胞和视杆细胞实现的.

定理 1.3 视杆细胞的作用与分布 **视杆细胞 (Rod Cells)** 对光强的敏感度高, 但不区分颜色. 视杆细胞在视网膜上分散地分布.

定理 1.4 视锥细胞的作用与分布 **视锥细胞 (Cone Cells)** 对光的波长敏感. 视锥细胞主要集中在视网膜的中央区域.

定理 1.5 视锥细胞的分类 视锥细胞按其对不同波长光的敏感性可分为三类:

1. S 视锥细胞: 对短波长 (蓝色光) 敏感, 峰值约在 420 nm.

2. M 视锥细胞: 对中波长 (绿色光) 敏感, 峰值约在 534 nm.

3. L 视锥细胞: 对长波长 (黄色光) 敏感, 峰值约在 564 nm.

尽管视锥细胞所敏感的颜色是蓝, 绿, 黄三色, 但由于黄色和绿色过于接近, 在图形学中通常使用红, 绿, 蓝三色表示各种颜色.

[1] 自然界中的白光, 例如太阳光等, 包含了可见光的所有频段的电磁波. 它的光谱几乎是连续的 (除了某些特定的, 由特定元素造成的吸收峰).

定理 1.6 色彩立体效应 光线通过角膜时会发生轻微的衍射. 眼睛通常能够将黄色波长的光 (598 nm) 调到最清晰的焦点, 从而使得波长较长的红色光波会聚在视网膜后面, 波长较短的绿色和蓝色光波会聚在视网膜前面. 这就使得人会认为较长波长的光来自更近的地方, 较短波长的光来自更远的地方. 这种现象称为**色彩立体效应 (Color Stereoscopic Effect)**.

定理 1.7 颜色恒常特性 人的视觉具有**颜色恒常特性 (Color Constancy)**, 即在不同光照条件下, 人们对物体颜色的感知基本保持不变. 例如, 在白天和黄昏时分, 我们仍然认为一件物体的颜色是相同的, 尽管其反射光谱可能有显著变化.

颜色恒常特性的一个典型的例子就是棋盘阴影错觉. 与颜色恒常特性的另一个密切相关的视觉效应是色诱导效应.

定理 1.8 色诱导 由于颜色恒常特性的存在, 人们在感知颜色时通常会扣除环境光的影响 (即倾向于将看到物体的颜色向环境光的补色靠拢). 这就导致了**色诱导 (Color Induction)** 现象.

定理 1.9 边界效应 物体边界的颜色也会影响人们对物体颜色的感知, 这称为**边界效应 (Edge Effect)**. 例如, 具有深色边界的物体会被感知为更暗的颜色, 而具有浅色边界的物体会被感知为更亮的颜色.

定理 1.10 色彩和视觉敏锐度 人对于物体清晰度的感知主要取决于物体的亮度变化, 而不是颜色变化.
另外, 人对于蓝色目标的空间敏锐度的感知要比其它颜色差得多, 这意味着蓝色的物体看起来更容易模糊.

1.1.2 颜色的离散表示

色彩空间

任意一种人眼可以分辨的颜色都可以用 (r, g, b) 三原色坐标表示, 并且它大致是线性的, 即颜色的混合可以表示为颜色坐标的加和. 这样, 任何一种颜色就对应于 $[0, 1] \times [0, 1] \times [0, 1]$ 这一立方体空间中的一个坐标. 此外, 我们还需要规定顶点处的基准颜色. 上面的所有规定构成了一个**色彩空间 (Color Space)**.

定义 1.11 色彩空间 **色彩空间**是一种用于定量描述颜色的数学模型, 它为颜色建立了一个坐标系统并规定了基准的颜色, 从而使得颜色可以被精确地表示.

常见的色彩空间有 RGB, CMY(K), HSV, HSL 等.

定义 1.12 RGB 色彩空间 **RGB 颜色空间**使用红 (Red), 绿 (Green), 蓝 (Blue) 三种颜色作为基准颜色, 通过调整这三种颜色的强度来表示各种颜色. 在计算机图形学中,RGB 颜色空间是最常用的色彩空间.

定义 1.13 HSV 与 HSL 色彩空间 **HSV 颜色空间** (Hue, Saturation, Value) 和 **HSL 颜色空间** (Hue, Saturation, Lightness) 是基于人类对颜色感知的色彩空间. 它们通过色调 (Hue), 饱和度 (Saturation), 亮度 (Lightness) 或明度 (Value) 来表示颜色, 更符合人类的直观感受.

上面两种色彩空间都采用柱坐标的方式表示颜色.

定义 1.14 CMYK 色彩空间 **CMYK 颜色空间** (Cyan, Magenta, Yellow, Key/Black) 主要用于印刷领域. 它使用青 (Cyan), 品红 (Magenta), 黄 (Yellow) 三种颜色作为基准颜色, 通过减法混合来表示各种颜色. 黑色 (Key/Black) 通道用于增强图像的对比度和深度.

色域

由于显示设备的不同, 其能够显示的颜色范围也不同. 我们可以用色域来描述显示设备能显示的颜色范围.

定义 1.15 色域 **色域 (Gamut)** 是指显示设备能够显示的颜色范围. 不同的显示设备有不同的色域, 例如 sRGB 色域,Adobe RGB 色域等.

1.2 显示

1.2.1 二维显示技术

矢量与像素

定义 2.1 矢量显示 **矢量显示**是指直接控制电子束在荧光屏上按照一定路径绘制图形的技术.

矢量显示器可以直接绘制线条和曲线, 因此在显示几何图形和文字时具有较高的精度和清晰度. 然而, 矢量显示器通常无法显示复杂的图像, 因为这时的线条数目太多, 难以处理.

定义 2.2 像素显示 **像素显示**是指将显示区域划分为小的单元格 (即**像素 (Pixel)**), 通过控制每个像素的亮度和颜色来形成图像的技术.

像素显示一般对应于阴极射线显示器的按行扫描的形式 (即**光栅显示技术**). 不管图形复杂与否, 显示的开销是不变的, 这更有利于应对复杂的图形.

定义 2.3 分辨率 屏幕显示的像素数目称为**分辨率 (Resolution)**.

常见的分辨率有 1080p(1920×1080), 2K(2048×1080), 4K(3840×2160) 等[2].

像素显示亦有缺点. 用离散的像素点近似连续的图形, 就会导致锯齿效应. 我们在后面的章节中会介绍反走样 (即抗锯齿) 技术.

显示原理

定义 2.4 帧与帧率 **帧 (Frame)** 是指屏幕上显示的单张图像, 而**帧率 (Frame Rate)** 是指每秒钟显示的帧数, 通常以 FPS(Frames Per Second) 为单位.

定义 2.5 刷新率 **刷新率 (Refresh Rate)** 是指屏幕每秒钟更新的次数, 通常以 Hz 为单位.

提高帧率可以使画面更流畅. 像素显示技术可以通过隔行扫描的方式来提高帧率.

定义 2.6 隔行扫描 **隔行扫描**是指每次只更新屏幕的一半像素, 即先更新所有奇数行, 然后更新所有偶数行的显示技术.

采用隔行扫描的方式可以提高显示设备的刷新率以及画面的帧率.

[2]有关 2K 和 4K 的命名规则比较复杂, 可以参考

颜色显示

对于现代大多数显示设备, 颜色显示的原理是一致的: 通过红, 绿, 蓝三种颜色的色光混合出 RGB 颜色空间中的各种颜色. 对于各种类型的屏幕, 其实现方式不同.

分类 2.7 现代显示器分类及显示颜色的原理

1. CRT(阴极射线显示): 通过电子束轰击荧光屏上的红, 绿, 蓝三种荧光粉来显示颜色.
2. LCD(液晶显示): 使用电压控制偏振光的偏转角度, 从而控制白光透过偏振片的强弱, 在经过红绿蓝三种颜色的滤光片之后混合出目标颜色.
3. LED(发光二极管)/OLED(有机发光二极管): 通过控制红, 绿, 蓝三种二极管的亮度来混合出各种颜色.OLED 使用有机发光材料制作发光二极管, 相较 LED 有更高的对比度和更鲜明的颜色.

定义 2.8 色域 显示设备能够显示的颜色范围称作**色域 (Gamut)**.

不同显示技术的显示设备有不同的色域, 例如 OLED 通常具有更广的色域, 能够显示更丰富的颜色.

亮度显示

人眼能感知的亮度范围很广, 而传统摄像和显示设备的亮度显示的动态范围有限, 显示效果有时不佳. 这就需要 HDR 技术.

定义 2.9 高动态范围显示 **高动态范围显示 (HDR,High Dynamic Range)** 是指能够显示更宽亮度范围的图像显示技术.

通过复杂的算法扩展和压缩图像各区域的亮度, HDR 允许在亮部和暗部表现更多的细节, 从而接近人眼感知的真实画面.

此外, 早期 CRT 显示设备的亮度和电压信号成非线性关系, 但摄像机等设备产生的信号是正比于光强的, 因此亮度显示可能有失真.CRT 显示设备的亮度 I 与电压 V 的关系通常为

$$I \propto V^{\gamma}$$

其中 γ 通常在 2.2 左右, 这会导致暗部更偏暗. 为解决这个问题, 我们需要对图像各像素的亮度值做校正, 即

$$l \to l^{1/\gamma}$$

这样会把图片先变亮, 然后再传入显示设备, 就能减小失真影响.

定义 2.10 伽马校正 上述对图像各像素的亮度值做校正的过程称为**伽马校正 (Gamma Correction)**.

尽管现在常用的显示设备, 如 LCD 显示器和 OLED 显示器等, 都不再具有上述特性, 但伽马校正已经被标准化于数字图像处理步骤中而被一直沿用. sRGB 颜色空间直接规定 $\gamma = 2.2$, 各种显示器需要做相应的伽

马校正.

在渲染时, 由于我们模拟的是现实世界的光照, 因此使用线性颜色空间是最合适的, 只需在输出图像时做伽马校正即可. 渲染中所用的各贴图也应保存在线性颜色空间中, 无需做伽马校正.

1.2.2 三维显示技术

三维显示的基本原理

人类获得立体感的途径很多.

原理 2.11 三维显示的基本原理

1. 双眼视差: 由于人眼之间有一定距离, 从而使得双眼看到的图像略有不同. 大脑通过处理这两幅图像的差异来感知深度信息.
2. 运动视差: 当我们移动头部或身体时, 近处的物体相对于远处的物体移动得更快. 大脑利用这种运动引起的视差来推断物体的距离.
3. 透视: 平行线在远处会汇聚于一点 (即**消失点**), 物体随着距离增加而变小. 大脑利用这些线索来理解空间关系.
4. 光影和遮挡: 光照和阴影提供了关于物体形状和位置的重要信息. 此外, 遮挡关系也帮助我们理解哪些物体在前面, 哪些在后面.
5. 焦点调节: 人眼通过调节晶状体的形状来聚焦不同距离的物体. 大脑利用这种调节信息来感知深度.

三维显示技术

根据前面的原理, 常见的三维显示技术主要有以下几种.

分类 2.12 常见三维显示技术

1. 立体显示: 通过为每只眼睛提供略有不同的图像来模拟双眼视差.
2. 全息显示: 利用干涉和衍射原理记录和重现光波的完整信息, 从而生成三维图像.
3. 光场显示: 通过捕捉和再现光线在空间中的传播方向和强度来创建三维图像.
4. 虚拟现实 (VR): 通过头戴式显示器和运动追踪技术创建沉浸式三维环境.
5. 增强现实 (AR): 将虚拟对象叠加到现实世界中显示.

需要注意的是, 立体显示技术与三维显示技术不同, 它只能通过双眼视差模拟固定视点的立体感, 而三维显示技术一般能提供各个视点的试图, 以求真实地还原三维空间. 立体显示技术的主要目标就是让左右眼看到不同的图像.

分类 2.13 常见立体显示技术

1. 偏振光立体显示: 使用偏振光技术, 在同一屏幕上显示具有两种偏振方向地图像, 通过偏振眼镜分别给每只眼睛传递对应图像.
2. 微柱透镜立体显示: 在屏幕前放置微柱透镜阵列, 通过透镜对光的折射使得每只眼睛只能看到特定的像素, 从而实现立体效果.
3. 光屏障立体显示: 在屏幕前放置细小光屏障, 通过屏障的遮挡使得每只眼睛只能看到特定的像素, 从而实现立体效果.

后两种技术都是裸眼 3D 技术, 不需要佩戴眼镜, 但视点位置有限制, 过大的视点移动会导致立体效果丧失, 甚至造成不适.

如果根据追踪到的视线方向将光屏障立体显示中的光屏障进行动态改变, 就能实现动态裸眼 3D 技术. 这就是三维显示技术的一种.

1.3 绘图

1.3.1 光栅化和直线绘制

光栅化

定义 3.1 光栅化 将图形转换为像素的过程称为**光栅化 (rasterization)**.

光栅化的思想在三维绘制 (即渲染) 中也会用到.

直线绘制

假定我们有一块分辨率为 $W \times H$ 的屏幕, 并且希望在屏幕上绘制直线 $y = kx + b(x_0 \leqslant x_1)$, 起止点为 $(x_0, y_0), (x_1, y_1)$. 为了简化问题, 我们假设 $0 \leqslant k \leqslant 1$.

基于浮点运算的直线绘制法 最简单的想法是遍历所有 $x \in [x_0, x_1]$, 计算对应的 y 值, 然后将 (x, y) 处的像素点填上对应的颜色.

```
def draw_line(x0:int, y0:int, k:float, b:float, x1:int):
    for x in range(x0, x1 + 1):
        y = k * x + b
        draw(x,round(y))
```

然而, 浮点数的运算开销是较大的. 为了避免浮点数的乘法, 又由于我们的直线是等间隔采样的, 因此可以使用累加法.

```
def draw_line(x0:int, y0:int, x1:int, y1:int):
    k = (y1 - y0) / (x1 - x0)
    y = y0
    for x in range(x0, x1 + 1):
        draw(x, round(y))
        y += k
```

这就是直线绘制的 **DDA 算法 (Digital Differential Analyzer)**.

但是, 上面的算法中仍然包含了浮点数的加法.

Bresenham 直线算法 Bresenham 教授在 1962 年提出了著名的**布雷森汉姆直线算法 (Bresenham's Line Algorithm)**, 能够在只计算整数加减法的情况下获得和 DDA 算法相同的结果.

令直线的起终点为 P_0, P_1. 令 $\overrightarrow{P_0P_1}$ 顺时针旋转 $90°$ 形成的向量 $\boldsymbol{n} = (y_1 - y_0, x_0 - x_1)$, 即直线的法向量, 那么直线上任意一点 P 就满足

$$F(P) = \boldsymbol{n} \cdot \overrightarrow{P_0P} = 0$$

这就是直线的隐式方程. 给出任意一点 Q, 判断 Q 在直线的上方还是下方, 只需判断 $F(Q)$ 的符号即可. 不难得出, 如果 $F(Q)$ 为负, 那么 Q 在直线上方, 反之则在直线下方.

假定已经绘制了直线上的一点 (x_i, y_i). 对于斜率 $k \in (0,1)$ 的直线, 下一个像素的坐标仅有两种可能: (x_i+1, y_i) 或 (x_i+1, y_i+1). Bresenham 认为如果上述两点的中点 $\left(x_i+1, y_i+\frac{1}{2}\right)$ 在直线上方, 就绘制 (x_i+1, y_i), 否则绘制 (x_i+1, y_i+1). 这种判断方法显然比较符合直觉.

现在我们考虑具体的实现方法. 对于已经绘制的点 $P_i(x_i, y_i)$, 需要用点 $P_i'\left(x_i+1, y_i+\frac{1}{2}\right)$ 判断下一个像素. 为了避免直接计算 $F(P_i')$(否则我们又要计算乘法), 我们考虑下面的递推方法:

$$F(P_i') = \boldsymbol{n} \cdot \overrightarrow{P_{i-1}'P_i'} = \boldsymbol{n} \cdot \left(\overrightarrow{P_0P_{i-1}'} + \overrightarrow{P_{i-1}'P_i'}\right) = F\left(P_{i-1}'\right) + \boldsymbol{n} \cdot \boldsymbol{\delta}$$

其中 $\boldsymbol{\delta} = (1,0)$ 或 $(1,1)$(判断点的移动和绘制点的移动显然是同步的). 这样, 画完之后我们可以每次更新 $F(P_i')$ 的值以进行下一个像素位置的判断.$\boldsymbol{n} \cdot \boldsymbol{\delta}$ 可以在循环前就算好, 不必重复. 递推的初值为

$$F(P_0') = F(P_0) + \boldsymbol{n} \cdot \left(1, \frac{1}{2}\right) = (y_1 - y_0) - \frac{1}{2}(x_1 - x_0)$$

为了避免 $1/2$ 带来的浮点运算, 我们可以把 F 放大 2 倍.

总结而言, 我们可以将上面的方法写成下面的程序:

```
def draw_line(x0: int, y0: int, x1: int, y1: int):
    y = y0
    dx, dy = 2 * (x1 - x0), 2 * (y1 - y0)
    dydx, F = dy - dx, dy - dx // 2
    for x in range(x0, x1 + 1):
        draw(x, y)
        if F < 0: # F(P_i') < 0
            F += dy # F(P_{i+1}') = F(P_i') + n * delta_0
        else: # F(P_i') >= 0
            y += 1
            F += dydx # F(P_{i+1}') = F(P_i') + n * delta_1
```

所有直线的绘制 我们在前面只讲了斜率 $k \in [0,1]$ 的直线的绘制. 对于其他斜率的直线, 通过适当的变换也可以做到一样的效果.

首先根据起终点计算直线的斜率 k, 如果 $|k| \leqslant 1$, 那么遍历 x 坐标并按照上述算法更新 y 坐标 (如果 $k<0$, 更新 y 时需将其递减); 如果 $|k|>1$, 那么交换前述 x 和 y 的角色即可. 代码实现如下:

```
def draw_line(x0: int, y0: int, x1: int, y1: int):
    f = (y1 - y0) < (x1 - x0)
    if not f: x0, y0, x1, y1 = y0, x0, y1, x1
    if x0 > x1: x0, y0, x1, y1 = x1, y1, x0, y0
    y, sy = y0, 1 if (y1 > y0) else -1
```

```
    dx, dy = 2 * (x1 - x0), 2 * (y1 - y0)
    dydx, F = dy - dx, dy - dx // 2
    for x in range(x0, x1 + 1):
        if f: draw(x, y)
        else: draw(y, x)
        if F < 0:
            F += dy
        else:
            y += sy
            F += dydx
```

1.3.2 多边形填充

多边形的光栅化

三角形是最简单的多边形, 也是多边形的基本组成部分. 最简单的办法是枚举所有可能的像素点, 判断其是否在三角形内.

```
def draw_triangle(x0: int, y0: int, x1: int, y1: int, x2: int, y2: int):
    xmin, xmax = min(x0, x1, x2), max(x0, x1, x2)
    ymin, ymax = min(y0, y1, y2), max(y0, y1, y2)
    for x in range(xmin, xmax + 1):
        for y in range(ymin, ymax + 1):
            if inside(x, y, x0, y0, x1, y1, x2, y2):
            draw(x, y)
```

要判断点 Q 是否在三角形内, 可以将边按逆时针顺序排列, 并将边逆时针旋转 90° 得到的法向量记作 $\boldsymbol{N}_i (i = 0, 1, 2)$. 各 $\boldsymbol{N}_i$ 都指向三角形内部, 于是只要各 $F_i(Q) = \boldsymbol{N}_i \cdot \overrightarrow{P_iQ} > 0$ 即可说明 Q 在三角形内.

然而, 上面的方法对于每个点都要计算三次点积, 效率不高. 注意到我们每次总是在按行填充像素, 因此只要维护每行的起点 x_L 和终点 x_R 即可. 我们也不必通过点积计算点的位置, 只要根据边的斜率和上一行的起终点更新即可 (即使用 DDA 算法更新起终点). 这就是**扫描线算法 (Scanline Algorithm)**.

对于多边形, 也可以用扫描线算法绘制, 但需要注意非凸的情形. 此时我们需要根据交点的数目按奇偶规则填充像素. 假定交点为 $x_1, \cdots, x_n$, 那么需要填充的部分为 x_{2k-1} 到 x_{2k} 的部分. 这可以通过简单的画图证明. 当然, 也可以将多边形分解成多个三角形进行分别绘制.

扫描线算法相比于简单的三角形算法更高效, 然而在现代计算机上, 其实使用的最多的是并行版的简单算法. 由于三角形在图形学中的重要性, 现代 GPU 搭载了用于三角形内外检测的专门模块, 从而在硬件上实现并行的简单算法.

多边形的插值

颜色插值 在绘制多边形时, 我们有时希望多边形的颜色是渐变的. 在绘制线段时, 假定两个顶点 (x_0, y_0) 和 (x_1, y_1) 的颜色为 c_0 和 c_1, 我们可以通过线性插值得到线段上 (x, y) 的颜色:

$$c = \frac{(x_1 - x)c_0 + (x - x_0)c_1}{x_1 - x_0} = c_0(1 - t) + c_1 t, t = \frac{x - x_0}{x_1 - x_0}$$

对多边形进行插值时, 我们可以应用扫描线算法, 每一行内的颜色由两个端点插值得到, 而端点的颜色又可以由对应的边的端点插值得到. 上述插值方法即**双线性插值 (Bilinear Interpolation)**.

在不使用扫描线算法的情况下, 我们需要单独确定每个点的颜色. 我们可以用**重心坐标 (Barycentric Coordinate)** 来做插值. 设三角形的顶点为 A, B, C, 那么对于三角形内的任意一点 P, 令

$$\alpha = \frac{S_{\triangle PBC}}{S_{\triangle PBC}} \qquad \beta = \frac{S_{\triangle PAC}}{S_{\triangle PBC}} \qquad \gamma = \frac{S_{\triangle PAB}}{S_{\triangle PBC}}$$

根据中学所学的平面向量知识不难得出

$$\alpha + \beta + \gamma = 1 \qquad \alpha\overrightarrow{PA} + \beta\overrightarrow{PB} + \beta\overrightarrow{PC} = \mathbf{0}$$

于是我们可以把 P 处的颜色表示为

$$c = \alpha c_A + \beta c_B + \gamma c_C$$

有关上面两种插值方式有以下结论.

定理 3.2 重心插值和双线性插值的等价性 重心插值是线性的, 即在任意一处发生相同位移时颜色的改变值一样. 特别地, 三角形的重心插值和双线性插值等价.

证明. 仍然设三角形的顶点为 $A(x_0, y_0), B(x_1, y_1), C(x_2, y_2)$, 并设 $P(x, y)$ 在三角形内. 重心坐标的表示要求

$$\begin{cases} \alpha_P(x_0 - x) + \beta_P(x_1 - x) + \gamma_P(x_2 - x) = 0 \\ \alpha_P(y_0 - y) + \beta_P(y_1 - y) + \gamma_P(y_2 - y) = 0 \\ \alpha_P + \beta_P + \gamma_P = 1 \end{cases}$$

这一线性方程组具有唯一解

$$\begin{cases} \alpha_P = \dfrac{(y - y_1)(x_2 - x_1) - (x - x_1)(y_2 - y_1)}{(y_0 - y_1)(x_2 - x_1) - (x_0 - x_1)(y_2 - y_1)} \\ \beta_P = \dfrac{(y - y_2)(x_0 - x_2) - (x - x_2)(y_0 - y_2)}{(y_1 - y_2)(x_0 - x_2) - (x_1 - x_2)(y_0 - y_2)} \\ \gamma_P = \dfrac{(y - y_0)(x_1 - x_0) - (x - x_0)(y_1 - y_0)}{(y_2 - y_0)(x_1 - x_0) - (x_2 - x_0)(y_1 - y_0)} \end{cases}$$

α_P, β_P 和 γ_P 都是关于 x, y 的线性函数, 因此 P 处的颜色 c_P 可以写为如下形式:

$$c_P = k_1 x + k_2 y + C$$

其中 k_1, k_2, C 与 x, y 无关. 因此, 重心插值是线性的.

现在来证明重心插值和双线性插值等价. 如果将颜色 c 作为三维空间中的 z 坐标, 那么这两种插值方式都在空间中描绘了一个平面. 它们都过三角形的顶点对应的 (x_i, y_i, c_i), 因而对应同一平面, 从而等价. □

多边形的拉伸 我们以图片的拉伸作为多边形插值的例子. 考虑一张四边形图片, 顶点为 A, B, C, D, 现将这张图片拉伸到屏幕上的四边形 $A'B'C'D'$. 我们可以在原图建立直角坐标系, 每个顶点对应 (u_i, v_i), 在拉伸后顶点和边上的坐标值不变 (这可以先由边的线性插值得到), 然后对四边形内的点进行双线性插值. 上面的方法称作 **UV 坐标映射 (UV Mapping)**.

尽管三角形的重心插值和双线性插值等价, 但四边形没有唯一的插值方法. 直接对其双线性插值和分解成两个三角形进行插值得到的结果不同.

1.4 曲线

1.4.1 曲线数学基础

曲线的表示

一般而言, 我们可以把曲线 (以及以后的曲面) 的表示方式分为显式表示和隐式表示.

定义 4.1 显式表示 显式表示是可以直接通过表达式得到点的表示方式. 例如, 平面上的圆的参数方程即显示表示:

$$\begin{cases} x = r\cos t \\ y = r\sin t \end{cases} \quad t \in [0, 2\pi)$$

一般的二次曲线也是显示表示:

$$y = ax^2 + bx + c$$

定义 4.2 隐式表示 隐式表示是指通过隐式方程来表示曲线 (或曲面), 而不直接给出参数到坐标的映射的表示方法. 例如, 平面上的圆可以表示为隐式方程

$$f(x, y) = x^2 + y^2 - r^2 = 0$$

隐式表示可以更容易地分辨曲线的内外侧, 但相应地不容易直接得到曲线的形状. 因此, 在曲线绘制时更常用显式表示.

曲线插值与基函数

定义 4.3 基函数 给定 $n+1$ 个点 $(x_0, y_0), \cdots, (x_n, y_n)$, 如果存在 $n+1$ 个函数 $\phi_0(x), \cdots, \phi_n(x)$, 使得

$$\phi_i(x_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

那么称 $\phi_i (i = 0, \cdots, n)$ 为这些点的**基函数 (Basis Function)**. 对这些点插值的结果可以写作

$$y = \sum_{i=0}^{n} y_i \phi_i(x)$$

这恰好是过各点的曲线, 具体形状由基函数的性质决定.

曲线的连续性

定义 4.4 参数连续性 设曲线的 n 阶导数在连接的两段曲线的交点处相等, 则称这两段曲线在该点处是 C^n 连续的.

定义 4.5 几何连续性 n 阶几何连续的定义如下:

1. G^0 连续: 两段曲线在连接点处相交.
2. G^1 连续: 两段曲线在连接点处相交, 且切线方向相同, 即曲率方向相同而大小不同.
3. G^2 连续: 两段曲线在连接点处相交, 且曲率方向和大小均相同.
4. G^3 连续: 两段曲线在连接点处相交, 且曲率方向, 大小和变化率均相同.

1.4.2 Bezier 曲线

Bezier 曲线的定义

Bezier 曲线是计算机图形学中常用的一种参数曲线, 由法国工程师 Pierre Bézier 在 20 世纪 60 年代为汽车车身设计而开发. 它们广泛应用于计算机图形学、动画、字体设计等领域.

Bezier 曲线通过一组控制点来定义. 我们先来看如何构造二阶 Bezier 曲线. 给定三个控制点 $\boldsymbol{P}_0, \boldsymbol{P}_1, \boldsymbol{P}_2$(这里的粗体表示原点到这一点的向量, 下同), 我们可以通过两轮线性插值得到曲线. 首先, 由参数 t 对 $\overrightarrow{P_0P_1}$ 和 $\overrightarrow{P_1P_2}$ 做线性插值:

$$\boldsymbol{Q}_0(t) = (1-t)\boldsymbol{P}_0 + t\boldsymbol{P}_1$$

$$\boldsymbol{Q}_1(t) = (1-t)\boldsymbol{P}_1 + t\boldsymbol{P}_2$$

然后用同一个参数 t 对 $\overrightarrow{Q_0Q_1}$ 做线性插值:

$$\boldsymbol{S}(t) = (1-t)\,\boldsymbol{Q}_0(t) + t\,\boldsymbol{Q}_1(t)$$

当 t 取遍 $[0,1]$ 时,$\boldsymbol{S}$ 对应的点的集合就是二阶 Bezier 曲线. 上述过程称作**德卡斯特里奥算法 (De Casteljau's Algorithm)**, 效率高, 编程方便, 因而被广泛使用.

类似地, n 阶 Bezier 曲线由 $n+1$ 个控制点 $\boldsymbol{P}_0, \boldsymbol{P}_1, \cdots, \boldsymbol{P}_n$ 通过 n 轮线性插值得到.

```
1 def bezier(points: list[Vec2], t: float) -> Vec2:
2     n = len(points)
3     P = points.copy()
4     for r in range(1, n):
5         for i in range(n - r):
6             P[i] = (1 - t) * P[i] + t * P[i + 1]
7     return P[0]
```

下面推导 Bezier 曲线的显式表达式. 首先考虑 $n=2$ 的情形, 将前述表达式展开可得

$$\begin{aligned}\boldsymbol{S} &= (1-t)\,\boldsymbol{Q}_0 + t\,\boldsymbol{Q}_1 \\ &= (1-t)\left[(1-t)\boldsymbol{P}_0 + t\boldsymbol{P}_1\right] + t\left[(1-t)\boldsymbol{P}_1 + t\boldsymbol{P}_2\right] \\ &= (1-t)^2\boldsymbol{P}_0 + 2t(1-t)\boldsymbol{P}_1 + t^2\boldsymbol{P}_2\end{aligned}$$

可以发现,$\boldsymbol{P}_0$,$\boldsymbol{P}_1$ 和 $\boldsymbol{P}_2$ 的系数分别是 $((1-t)+t)^2$ 进行二项式展开的系数. 因此, 我们可以猜测[3]n 阶 Bezier 曲线的显式表达式为

$$\boldsymbol{S}(t) = \sum_{k=0}^{n} \binom{n}{k} (1-t)^k t^{n-k} \boldsymbol{P}_k$$

其中

$$B_{n,k}(t) = \binom{n}{k} (1-t)^k t^{n-k}$$

被称作 n **阶伯恩斯坦多项式 (Bernstein Polynomial)**.

我们也可以用矩阵的形式来表示 Bezier 曲线. 例如, 三阶 Bezier 曲线可以通过以下的形式表示:

$$\boldsymbol{S}(t) = \boldsymbol{\tau}^{\mathrm{T}} \boldsymbol{M}_B \boldsymbol{u}$$

其中

$$\boldsymbol{\tau}^{\mathrm{T}} = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \qquad \boldsymbol{M}_B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -3 & 3 & 0 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix} \qquad \boldsymbol{u}^{\mathrm{T}} = \begin{bmatrix} \boldsymbol{P}_0 & \boldsymbol{P}_1 & \boldsymbol{P}_2 & \boldsymbol{P}_3 \end{bmatrix}$$

Bezier 曲线的性质

首先有

$$\boldsymbol{S}(0) = \boldsymbol{P}_0 \qquad \boldsymbol{S}(1) = \boldsymbol{P}_n$$

因此 Bezier 曲线经过第一个和最后一个控制点.

其次有

$$\begin{aligned}\frac{\mathrm{d}\boldsymbol{S}(t)}{\mathrm{d}t} &= \frac{\mathrm{d}}{\mathrm{d}t}\left(\sum_{k=0}^{n} \frac{n!}{k!(n-k)!}(1-t)^k t^{n-k}\boldsymbol{P}_k\right) \\ &= \sum_{k=0}^{n} \frac{n!}{k!(n-k)!}\left((n-k)(1-t)^k t^{n-k-1} - k(1-t)^{k-1}t^{n-k}\right)\boldsymbol{P}_k \\ &= n\sum_{k=0}^{n}\left(B_{n-1,k}(t) - B_{n-1,k-1}(t)\right)\boldsymbol{P}_k \\ &= n\sum_{k=0}^{n-1} B_{n-1,k}(t)(\boldsymbol{P}_{k+1} - \boldsymbol{P}_k)\end{aligned}$$

[3]事实上可以将这一过程与杨辉三角的构造相联系而证明.

于是

$$\boldsymbol{S}'(0) = n\left(\boldsymbol{P}_1 - \boldsymbol{P}_0\right) \qquad \boldsymbol{S}'(1) = n\left(\boldsymbol{P}_n - \boldsymbol{P}_{n-1}\right)$$

可见, Bezier 曲线在端点处的切线方向分别沿着 $\overrightarrow{P_0P_1}$ 和 $\overrightarrow{P_{n-1}P_n}$ 的方向.

定理 4.6 Bezier 曲线与端点的关系 设 Bezier 曲线的控制点依次为 $P_0, \cdots, P_n$, 则它经过 P_0 和 P_n, 且在端点处的切线方向分别沿着 $\overrightarrow{P_0P_1}$ 和 $\overrightarrow{P_{n-1}P_n}$ 的方向.

因此, 如果希望两条 Bezier 曲线首尾相接且切线连续, 只需让它们的端点重合且相邻的控制点共线即可.

于是, 三阶 Bezier 曲线是比较常用的, 既可以自由控制曲线的形状, 复杂度也不高.

Bezier 曲线的另一个重要性质是**凸包性质**.

定义 4.7 凸包性质 设 $\boldsymbol{P}_0, \boldsymbol{P}_1, \cdots, \boldsymbol{P}_n$ 为 Bezier 曲线的控制点, 那么 Bezier 曲线完全包含在由这些控制点构成的凸多边形内.

证明. 由于 $B_{n,k}(t) \geqslant 0$ 且 $\sum_{k=0}^{n} B_{n,k}(t) = 1$, 因此 $\boldsymbol{S}(t)$ 是控制点的凸组合, 从而在凸包内. □

Bezier 曲面

同样地, 我们可以用类似的方法绘制 Bezier 曲面. 这需要用到两个参数 u, v. 例如, 三阶 Bezier 曲面可以表示为

$$\boldsymbol{S}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_{3,i}(u)B_{3,j}(v)\boldsymbol{P}_{ij} = \boldsymbol{u}^{\mathrm{T}}\boldsymbol{M}_B\boldsymbol{P}\boldsymbol{M}_B^{\mathrm{T}}\boldsymbol{v}$$

其中

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{P}_{00} & \cdots & \boldsymbol{P}_{03} \\ \vdots & \ddots & \vdots \\ \boldsymbol{P}_{30} & \cdots & \boldsymbol{P}_{33} \end{bmatrix} \qquad \boldsymbol{v} = \begin{bmatrix} 1 \\ v \\ v^2 \\ v^3 \end{bmatrix}$$

1.4.3 样条曲线

定义 4.8 样条曲线 **样条曲线 (Spline Curve)** 是计算机图形学和数值分析中常用的一类分段定义的多项式函数, 用于平滑地插值或逼近一组离散数据点.

给定点列 $\{\boldsymbol{P}_0, \cdots, \boldsymbol{P}_m\}$, 其中 $\boldsymbol{P}_i = (x_i, y_i)$. 样条曲线的确定实际上就是在每个区间 $[x_i, x_{i+1}]$ 上构造一个多项式 $S_i(x)$, 使其经过各点并保持一定的连续性. 常见的样条曲线有以下几种.

三次样条曲线

三次样条曲线是比较简单而常用的. 假定在每一段 $\{\boldsymbol{P}_i, \boldsymbol{P}_{i+1}\}$ 上定义参数 $t \in [0,1]$, 并且假定曲线上的点可以表示为

$$\boldsymbol{S}_i(t) = \boldsymbol{a}_i t^3 + \boldsymbol{b}_i t^2 + \boldsymbol{c}_i t + \boldsymbol{d}_i$$

曲线一共有 $4m$ 个未知数, 我们需要 $4m$ 个方程来确定这些未知数.

首先, 曲线需要经过各个点, 因此有

$$\boldsymbol{S}_i(0) = \boldsymbol{P}_i,\ \boldsymbol{S}_i(1) = \boldsymbol{P}_{i+1}$$

其次, 为了保持光滑性, 我们需要保证相邻段的切线方向一致, 因此有

$$\boldsymbol{S}_i'(1) = \boldsymbol{S}_{i+1}'(0)$$

最后, 为了保持曲线的平滑性, 我们还需要保证相邻段的二阶导数一致, 因此有

$$\boldsymbol{S}_i''(1) = \boldsymbol{S}_{i+1}''(0)$$

上述三个条件一共有 $4m-2$ 个方程. 为此, 我们还需要自行指定两个边界条件, 例如令曲线在端点处的二阶导数为 0, 称作**自然边界条件**:

$$\boldsymbol{S}_0''(0) = \boldsymbol{0},\ \boldsymbol{S}_{m-1}''(1) = \boldsymbol{0}$$

求解这一线性方程组即可得到三次样条曲线的参数.

从上述过程可以看出, 三次样条曲线是插值曲线, 并且具有 C^2 连续性. 然而, 改变任意一个控制点会影响整条曲线的形状, 因此三次样条曲线不具有局部性.

定义 4.9 局部性 如果改变一个控制点只会影响曲线的局部形状而不影响整体形状, 则称该插值曲线具有**局部性**.

Hermite 样条曲线

为了避免控制点对远端曲线形状的影响, 我们可以对三次样条曲线的构造做改进. 我们不要求各段曲线的二阶导连续, 而是指定各顶点 $\boldsymbol{P}_i$ 处的切线方向为 $\boldsymbol{p}_i$, 于是对于 $(\boldsymbol{P}_i, \boldsymbol{P}_{i+1})$ 之间的三次函数有

$$\boldsymbol{S}_i(0) = \boldsymbol{P}_i \qquad \boldsymbol{S}_i(1) = \boldsymbol{P}_{i+1} \qquad \boldsymbol{S}_i'(0) = \boldsymbol{p}_i \qquad \boldsymbol{S}_i'(1) = \boldsymbol{p}_{i+1}$$

这可以直接解得

$$\boldsymbol{S}_i(t) = \left(2t^3 - 3t^2 + 1\right)\boldsymbol{P}_i + \left(t^3 - 2t^2 + t\right)\boldsymbol{p}_i + \left(-2t^3 + 3t^2\right)\boldsymbol{P}_{i+1} + \left(t^3 - t^2\right)\boldsymbol{p}_{i+1}$$

这就是三次 Hermite 样条曲线. 其它阶数的 Hermite 样条曲线也可以类似地构造.

Hermite 样条曲线各控制点的导数方向可以自行指定, 也可以由控制点得到. 著名的 Catmull-Rom 样条曲

线就是一种特殊的 Hermite 样条曲线, 其切线方向由相邻控制点决定, 即

$$\boldsymbol{p}_i = \frac{\boldsymbol{P}_{i+1} - \boldsymbol{P}_{i-1}}{2}$$

相比三次样条曲线, Hermite 样条曲线同样是插值曲线, 但同时具有局部性. 相应地, 它在光滑程度上有所牺牲, 只有 C^1 连续性. 此外, 它并不需要解线性方程组, 结果已经一定, 计算效率更高[4].

B 样条曲线

与前面两种样条曲线相比,B 样条曲线不是插值曲线, 与 Hermite 样条曲线一样具有局部性, 但光滑性更好.

选定参数区间 $[a, b]$ 内一递增的数列 $a = t_0 < t_1 < \cdots < t_m = b$, 则 n 次 B 样条曲线可以表达为

$$\boldsymbol{b}(t) = \sum_{i=0}^{m} N_{i,n}(t)\boldsymbol{P}_i$$

其中 $N_{i,n}(t)$ 为 B 样条基函数, 通过递归定义:

$$N_{i,0}(t) = \begin{cases} 1, & t_i \leqslant t < t_{i+1} \\ 0, & \text{otherwise} \end{cases}$$

$$N_{i,p}(t) = \frac{t - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t)$$

不难看出,$N_{i,p}(t)$ 是 p 阶的分段多项式, 并且在连接处保持了 C^{p-1} 连续性, 因此 n 阶 B 样条曲线具有 C^{n-1} 连续性. 此外,$N_{i,p}(t)$ 在 $[t_i, t_{i+p+1})$ 非 0, 因此改变 $\boldsymbol{P}_i$ 仅会影响 $[t_i, t_{i+p+1})$ 区间内的曲线形状. p 越大, 各个基函数和控制点影响的范围就越大, 局部性就越差.

B 样条曲线的一大优势是其灵活性. 我们可以分段选择不同阶数的基函数, 也可以选取重复节点 $t_i = t_{i+1} = \cdots = t_k$ 使得曲线其余部分连续性不变的情况下构造出尖锐的转折.

1.4.4 曲线光栅化

中点圆算法

中点圆算法 (Midpoint Circle Algorithm) 是 Bresenham 教授在 1962 年提出的一种高效的光栅化圆的算法, 其主要思路与 Bresenham 直线算法类似.

为了方便考虑, 我们假定要绘制的圆以原点为中心, 半径为 r. 由于圆的对称性, 我们只需计算第一象限的 1/8 圆, 然后将结果对称变换到其他象限即可.

考虑 $\frac{\pi}{4}$ 到 $\frac{\pi}{2}$ 的 1/8 圆, 我们从 $(0, r)$ 开始按顺时针方向绘制. 设当前绘制的点为 $P_i(x_i, y_i)$, 那么下一个点 P_{i+1} 也仅有两种可能: $P_{i+1} = (x_i + 1, y_i)$ 或 $P_{i+1} = (x_i + 1, y_i - 1)$. 我们用中点 $P_i' = \left(x_i + 1, y_i - \frac{1}{2}\right)$ 来判断下一个点的位置, 如果 P_i' 在圆内, 就向右移, 否则向右下移动. 圆的隐式方程为

$$F(x, y) = x^2 + y^2 - r^2 = 0$$

[4]Microsoft PowerPoint 提供的曲线工具就是三阶 Hermite 样条曲线, 用户可以自由编辑每个控制点的位置和对应的切线.

同样, 如果 $f\left(P_i'\right)<0$, 那么 P_i' 在圆内, 否则在圆外. 我们可以只对 $F\left(P_i'\right)$ 进行更新, 显然每次判断点的移动情况和像素的移动情况一样, 因此

$$F\left(P_{i+1}'\right)=F\left(x_{P_i'}+\Delta x, y_{P_i'}+\Delta y\right)=F\left(P_i'\right)+2x_{P_i'}\Delta x+(\Delta x)^2+2y_{P_i'}\Delta y+(\Delta y)^2$$

其中 $(\Delta x, \Delta y)=(1,0)$ 或 $(1,-1)$. 由于 P_i' 的坐标带有分数, 因此我们将上述式子转写为关于 P_i 的坐标的式子, 即

$$F\left(P_{i+1}'\right)=F\left(P_i'\right)+2x_i\Delta x+2\Delta x+(\Delta x)^2+2y_i\Delta y-\Delta y+(\Delta y)^2$$

于是向右更新像素 (即 $(\Delta x, \Delta y)=(1,0)$ 时) 需要将判断函数加上 $2x_i+3$, 向右下更新像素 (即 $(\Delta x, \Delta y)=(1,-1)$ 时) 需要将判断函数加上 $2x_i-2y_i+5$. 初始条件下有

$$F\left(1, r-\frac{1}{2}\right)=\frac{5}{4}-r$$

既然我们涉及的计算都是整数计算, 因此将初始值设为 $1-r$ 并不改变判断正负的结果. 当然, 如果半径 r 是浮点数, 就需要先计算上式后取整了.

综上, 我们可以将中点圆算法写成下面的程序:

```
1  def draw_circle(xc: int, yc: int, r: int):
2      x, y = 0, r
3      F = 1 - r
4      while x <= y:
5          draw(xc + x, yc + y); draw(xc + y, yc + x)
6          draw(xc - x, yc + y); draw(xc - y, yc + x)
7          draw(xc + x, yc - y); draw(xc + y, yc - x)
8          draw(xc - x, yc - y); draw(xc - y, yc - x)
9          if F < 0: # F(P_i') < 0
10             F += 2 * x + 3 # F(P_{i+1}) = F(P_i) + 2 * x_i + 3
11         else: # F(P_i') >= 0
12             y -= 1
13             F += 2 * (x - y) + 5 # F(P_{i+1}) = F(P_i) + 2 * (x_i - y_i) + 5
14         x += 1
```

1.5 图像处理

1.5.1 图像的表示方法

图像在本质上是连续的图形, 但计算机并不能存储无限的信息, 因此需要对图像进行离散化处理.

定义 5.1 位图与矢量图 位图 (Bitmap) 是由像素点组成的矩阵, 每个像素点有一个颜色值, 通过这些像素点的颜色值来表示图像. 矢量图 (Vector Graphics) 是由数学方程和几何图形 (如直线, 曲线, 多边形等) 来表示图像.

两者之间的差异可以由下表总结:

	位图	矢量图
优点	可以表示复杂的图像细节	可以无限放大而不失真
缺点	放大后会失真	难以表示复杂的图像细节

表 1.1: 位图与矢量图的比较

1.5.2 Fourier 变换与图像的频谱

Fourier 级数的复变函数形式

在高等数学中, 对周期为 T 的函数 $f(x)$ 的 Fourier 级数展开为

$$f(x)=\frac{a_0}{2}+\sum_{n=1}^{\infty}\left[a_n\cos\left(\frac{2n\pi x}{T}\right)+b_n\sin\left(\frac{2n\pi x}{T}\right)\right]$$

其中

$$a_0=\frac{2}{T}\int_{-T/2}^{T/2}f(x)\mathrm{d}x,\quad a_n=\frac{2}{T}\int_{-T/2}^{T/2}f(x)\cos\left(\frac{2n\pi x}{T}\right)\mathrm{d}x,\quad b_n=\frac{2}{T}\int_{-T/2}^{T/2}f(x)\sin\left(\frac{2n\pi x}{T}\right)\mathrm{d}x$$

考虑到复平面圆周与简谐波的关系 (这也方便各种变换与数学处理), 我们尝试把上面的式子改写成复变函数的形式. 由 Euler 公式

$$\mathrm{e}^{\mathrm{i}\theta}=\cos\theta+\mathrm{i}\sin\theta$$

可得

$$\cos\theta=\frac{\mathrm{e}^{\mathrm{i}\theta}+\mathrm{e}^{-\mathrm{i}\theta}}{2},\quad \sin\theta=\frac{\mathrm{e}^{\mathrm{i}\theta}-\mathrm{e}^{-\mathrm{i}\theta}}{2\mathrm{i}}$$

令 $\omega = \dfrac{2\pi}{T}$, 于是

$$\begin{aligned} f(x) &= \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos(n\omega x) + b_n \sin(n\omega x)\right] \\ &= \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \frac{\mathrm{e}^{\mathrm{i}n\omega x} + \mathrm{e}^{-\mathrm{i}n\omega x}}{2} + b_n \frac{\mathrm{e}^{\mathrm{i}n\omega x} - \mathrm{e}^{-\mathrm{i}n\omega x}}{2\mathrm{i}}\right) \\ &= \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(\frac{a_n - \mathrm{i}b_n}{2} \mathrm{e}^{\mathrm{i}n\omega x} + \frac{a_n + \mathrm{i}b_n}{2} \mathrm{e}^{-\mathrm{i}n\omega x}\right) \end{aligned}$$

为了将求和的两部分统一起来, 注意到

$$a_{-n} = \frac{2}{T} \int_{-T/2}^{T/2} f(x) \cos\left(\frac{-2n\pi x}{T}\right) \mathrm{d}x = a_n$$

$$b_{-n} = \frac{2}{T} \int_{-T/2}^{T/2} f(x) \sin\left(-\frac{2n\pi x}{T}\right) \mathrm{d}x = -b_n$$

于是上式可以改写为

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \frac{a_n - \mathrm{i}b_n}{2} \mathrm{e}^{\mathrm{i}n\omega x} + \sum_{n=1}^{\infty} \frac{a_{-n} - \mathrm{i}b_{-n}}{2} \mathrm{e}^{\mathrm{i}(-n)\omega x} = \sum_{n=-\infty}^{\infty} \frac{a_n - \mathrm{i}b_n}{2} \mathrm{e}^{\mathrm{i}n\omega x}$$

定义 $C_n = \dfrac{a_n - \mathrm{i}b_n}{2}$, 就可以得到 Fourier 级数写成复变函数的形式:

$$f(x) = \sum_{n=-\infty}^{\infty} C_n \mathrm{e}^{\mathrm{i}n\omega x}$$

现在, 我们考虑系数 C_n 的求法. 我们在上述展开式两边乘以 $\mathrm{e}^{-\mathrm{i}m\omega x}$, 然后在一个周期 (例如 $[0, T]$) 上积分, 就有

$$\int_0^T f(x) \mathrm{e}^{-\mathrm{i}m\omega x} \mathrm{d}x = \int_0^T \left(\sum_{n=-\infty}^{+\infty} C_n \mathrm{e}^{\mathrm{i}(n-m)\omega x}\right) \mathrm{d}x = \sum_{n=-\infty}^{+\infty} C_n \int_0^T \mathrm{e}^{\mathrm{i}(n-m)\omega x} \mathrm{d}x$$

当 $n = m$ 时, 上面右边的积分值为 T, 否则令 $k = n - m$ 就有

$$\int_0^T \mathrm{e}^{\mathrm{i}(n-m)\omega x} \mathrm{d}x = \int_0^T \exp\left(\frac{2\pi \mathrm{i}kx}{T}\right) \mathrm{d}x = \frac{T\left(\mathrm{e}^{2k\pi\mathrm{i}} - 1\right)}{2\pi\mathrm{i}k} = 0$$

于是

$$\int_0^T f(x) \mathrm{e}^{-\mathrm{i}m\omega x} \mathrm{d}x = TC_m$$

这样就有

$$C_n = \frac{1}{T} \int_0^T f(x) \mathrm{e}^{-\mathrm{i}n\omega x} \mathrm{d}x$$

定理 5.2 Fourier 级数展开的复变函数形式 设 $f(x)$ 为周期为 T 的满足 Dirichlet 条件的函数, 则 $f(x)$ 可以展开为 Fourier 级数

$$f(x) = \sum_{n=-\infty}^{\infty} C_n \mathrm{e}^{\mathrm{i}n\omega x}$$

其中 $\omega = \dfrac{2\pi}{T}$, 系数 C_n 由下式给出:

$$C_n = \frac{1}{T}\int_0^T f(x)\mathrm{e}^{-\mathrm{i}n\omega x}\mathrm{d}x$$

从 Fourier 级数到 Fourier 变换

那么对于非周期函数, 我们可以将其视作周期 $T \to +\infty$ 的周期函数, 此时角频率 $\dfrac{2\pi}{T} \to 0$, Fourier 级数中的 $n\omega$ 也应当取遍任意实数. 因此, 将 Fourier 展开中的求和变成积分, 并且将系数 C_n 换成有关 ω 的函数 $F(\omega)$, 即

$$f(x) = \int_{-\infty}^{+\infty} F(\omega)\mathrm{e}^{\mathrm{i}\omega x}\mathrm{d}\omega$$

同样地, 每个 ω 对应的系数 $F(\omega)$ 也可以由 $f(x)$ 求出, 即

$$F(\omega) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} f(x)\mathrm{e}^{-\mathrm{i}\omega x}\mathrm{d}x$$

在很多场合下, 通常令 $f = \dfrac{\omega}{2\pi} = \dfrac{1}{T}$ 为频率 (更经常地, f 也被记作 ω, 注意与前面推导中的角频率加以区分), 这就得到 Fourier 变换的标准形式. 同时, 我们在后面附上一个比较严谨的证明.

定理 5.3 设 $f(x)$ 满足 Fourier 变换的条件, 则 $f(x)$ 的 Fourier 变换为

$$F(\omega) = \mathcal{F}\circ f = \int_{-\infty}^{+\infty} f(x)\mathrm{e}^{-2\pi\mathrm{i}\omega x}\mathrm{d}x$$

其逆变换为

$$f(x) = \mathcal{F}^{-1}\circ F = \int_{-\infty}^{+\infty} F(\omega)\mathrm{e}^{2\pi\mathrm{i}\omega x}\mathrm{d}\omega$$

证明. 证明中的 ω 采取我们后来给出的定义. 由 Fourier 级数展开的复变函数形式, 对于周期为 T 的函数 $f(x)$, 有

$$f(x) = \sum_{n=-\infty}^{+\infty} C_n\mathrm{e}^{2\pi\mathrm{i}n\omega x} = \sum_{n=-\infty}^{+\infty} \frac{C_n}{\omega}\left(\omega\mathrm{e}^{2\pi\mathrm{i}n\omega x}\right)$$

其中 $\omega = 1/T$ 而

$$\frac{C_n}{\omega} = \frac{1}{\omega}\cdot\frac{1}{T}\int_0^T f(x)\mathrm{e}^{-2\pi\mathrm{i}n\omega x}\mathrm{d}x = \int_{-T/2}^{T/2} f(x)\mathrm{e}^{-2\pi\mathrm{i}n\omega x}\mathrm{d}x$$

现在令 $T\to\infty$, 于是 $\omega = 1/T \to 0$. 现在令 $t = n\omega$, 于是

$$\begin{aligned} f(x) &= \lim_{\omega\to 0}\sum_{n=-\infty}^{+\infty}\frac{C_n}{\omega}\left(\omega\mathrm{e}^{2\pi\mathrm{i}n\omega x}\right) \\ &= \lim_{\omega\to 0}\sum_{n=-\infty}^{+\infty}\omega\mathrm{e}^{2\pi\mathrm{i}n\omega x}\left(\int_{-T/2}^{T/2} f(x)\mathrm{e}^{-2\pi\mathrm{i}n\omega x}\mathrm{d}x\right) \\ &= \lim_{\omega\to 0}\sum_{n=-\infty}^{+\infty}\mathrm{e}^{2\pi\mathrm{i}tx}\left(\int_{-\infty}^{+\infty} f(x)\mathrm{e}^{-2\pi\mathrm{i}tx}\mathrm{d}x\right)\omega \end{aligned}$$

上面的求和与 Riemann 积分的定义一致, 因此

$$f(x)=\int_{-\infty}^{+\infty}\mathrm{e}^{2\pi\mathrm{i}tx}\left(\int_{-\infty}^{+\infty}f(x)\mathrm{e}^{-2\pi\mathrm{i}tx}\mathrm{d}x\right)\mathrm{d}t$$

把 t 换成 ω, 将中间的积分记作 $F(\omega)$, 即可证得定理. □

离散信号的 Fourier 变换

对于无限长度的离散信号 $\{x[n]\}_{n\in\mathcal{Z}}$, 我们可以把无穷积分换成无穷级数求和. 记其 Fourier 变换的结果为 $\hat{x}(\omega)$, 则

$$\hat{x}(\omega)=\sum_{n=-\infty}^{+\infty}x[n]\mathrm{e}^{-2\pi\mathrm{i}\omega n}$$

然而, 计算机存储的信号数目是有限的. 因此, 对于一段有限长度为 N 的离散信号 $\{x[n]\}_{0\leqslant n<N}$, 我们假设它可以周期延拓 (相当于 $\{x[n]\}$ 是周期为 N 的函数的 N 个采样点), 这时的 Fourier 变换即 Fourier 展开, 频率的选取是离散的; 并且由于我们只有 N 个数据点, 因此最多只有 N 个独立的频率对应的简谐波合成了这一离散信号. 于是

$$\hat{x}[k]=\sum_{n=0}^{N-1}\exp\left(-\frac{2\pi\mathrm{i}nk}{N}\right)x[n],\quad k=0,1,\cdots,N-1$$

它的逆变换为

$$x[n]=\frac{1}{N}\sum_{k=0}\exp\left(\frac{2\pi\mathrm{i}nk}{N}\right)\hat{x}[k]$$

同样地, 二维情形下的 Fourier 变换为

$$F(u,v)=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}f(x,y)\mathrm{e}^{-2\pi\mathrm{i}(ux+vy)}\mathrm{d}x\mathrm{d}y$$

其中 $\mathrm{e}^{-2\pi\mathrm{i}(ux+vy)}$ 表示方向向量为 (u,v), 频率为 $\sqrt{u^2+v^2}$ 的复平面上的波 (同样地由正弦波和余弦波组成). 也同样地, 离散情形下的二维 Fourier 变换为

$$\hat{\boldsymbol{I}}(u,v)=\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}\boldsymbol{I}(x,y)\exp\left[-2\pi\mathrm{i}\left(\frac{ux}{M}+\frac{vy}{N}\right)\right],\quad 0\leqslant u<M,0\leqslant v<N$$

于是 $\hat{\boldsymbol{I}}(u,v)$ 也是一个 $M\times N$ 的矩阵, 这就是 $\boldsymbol{I}$ 的频谱. 对频谱 $\hat{\boldsymbol{I}}$ 做逆 Fourier 变换可以还原出图像 $\boldsymbol{I}$.

1.5.3 图像滤波

卷积

卷积是图像处理中常用的操作, 用于图像的平滑, 锐化, 边缘检测等.

定义 5.4 卷积 二维离散信号 (如图像) 的卷积定义为:

$$(\boldsymbol{I}*\boldsymbol{K})(x,y)=\sum_{m=-M}^{M}\sum_{n=-N}^{N}\boldsymbol{I}(x-m,y-n)\boldsymbol{K}(m,n)$$

其中 $\boldsymbol{I}$ 是输入图像, $\boldsymbol{K}$ 是卷积核 (滤波器),(x,y) 是图像中的一个像素位置. 在图形学的实际应用中, 卷积核通常经过预翻转操作, 此时卷积定义为

$$(\boldsymbol{I}*\boldsymbol{K})(x,y)=\sum_{m=-M}^{M}\sum_{n=-N}^{N}\boldsymbol{I}(x+m,y+n)\boldsymbol{K}(m,n)$$

卷积和傅里叶变换密切相关.

定理 5.5 卷积定理 函数卷积的傅里叶变换等于各自傅里叶变换的乘积, 即

$$\mathcal{F}(f*g)=\mathcal{F}(f)\cdot\mathcal{F}(g)$$

这样, 可以观察图像和滤波器进行傅里叶变换的频谱来大致得出滤波后的结果, 从而针对性地设计滤波器.

图像模糊

最简单的模糊滤波器是均值滤波器.

定义 5.6 均值滤波器 均值滤波器的卷积核为

$$\boldsymbol{K}_{\text{mean}}=\frac{1}{(2k+1)^2}\begin{bmatrix}1&\cdots&1\\ \vdots&\ddots&\vdots\\ 1&\cdots&1\end{bmatrix}$$

根据卷积的定义, 均值滤波器的作用是取目标像素邻近的 $(2k+1)\times(2k+1)$ 个像素求平均值.

均值滤波器地频谱主要集中在低频部分, 可以去除高频噪声, 但会模糊图像细节. 然而, 它的频谱有呈十字形向外放射的部分, 因此使用均值滤波器可能导致细节错误.

高斯滤波器是一种效果更好的模糊滤波器[5]. 它的连续形式为

$$g(x,y)=\frac{1}{2\pi\sigma^2}\exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

其中 σ 为方差, 可以控制模糊程度. 实际使用时, 可以截取目标附近的 $(2k+1)\times(2k+1)$ 个像素, 根据上述函数计算权重并归一化得到卷积核, 然后按照类似均值滤波的计算方式完成卷积.

在数学上可以证明, 高斯滤波器的频谱仍然是高斯函数, 只保留低频部分, 不会出现像均值滤波器那样的走样.

[5]在 Adobe Photoshop 中对应高斯模糊选项.

边缘提取

所谓图像中的边缘, 是指图像中一个色块与另一个色块的分界线, 通常对应图像中颜色变化较大的部分. 我们可以用估计梯度的算子来提取边缘.

定义 5.7 Sobel 算子 Sobel 算子是一种常用的边缘检测算子, 它通过两个卷积核来计算图像在水平和垂直方向上的梯度:

$$\boldsymbol{K}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{K}_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

其中 $\boldsymbol{K}_x$ 用于检测水平边缘, $\boldsymbol{K}_y$ 用于检测垂直边缘.
对图像 $\boldsymbol{I}$ 进行卷积后, 可以得到水平和垂直方向的梯度:

$$G_x = \boldsymbol{I} * \boldsymbol{K}_x, \quad G_y = \boldsymbol{I} * \boldsymbol{K}_y$$

为了得出各个方向上的边缘, 可以计算指定位置的梯度向量的模长:

$$G = \sqrt{G_x^2 + G_y^2}$$

除了使用梯度提取边缘, 还可以使用二阶导数提取边缘.

定义 5.8 Laplacian 算子 Laplacian 算子是一种二阶导数算子, 用于检测图像中的边缘. 其卷积核为

$$\boldsymbol{K}_{\text{Lap}} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

对图像 $\boldsymbol{I}$ 进行卷积后, 可以得到 Laplacian 响应:

$$L = \boldsymbol{I} * \boldsymbol{K}_{\text{Lap}}$$

Laplacian 算子对噪声较敏感, 因此通常在使用前先对图像进行高斯模糊处理, 这种方法称为 LoG(Laplacian of Gaussian).

1.5.4 图像补全与融合

在图像补全时, 我们需要对图像缺失的部分进行补充, 并且使得结果看起来更为自然, 即满足下面两个条件:

1. **空间局部性**: 同一个物体上相邻的部分应当相似.
2. **奥卡姆剃刀原理**: 如无必要, 勿增实体.

在数学上, 我们可以用以下的办法描述问题: 考虑图像 I 的待补全区域 Ω 以及其边界 $\partial\Omega$. 定义 $f(x,y)$ 为 $\Omega\cup\partial\Omega$ 上的函数表示待填补的颜色, $f^*(x,y)$ 为 $\complement_I\Omega$ 上的函数表示已知的颜色. 于是优化目标为

$$\min_f \iint_\Omega ||\nabla f||^2 \,\mathrm{d}S, \quad \text{s.t.} \quad f|_{\partial\Omega} = f^*|_{\partial\Omega}$$

在图像融合时, 我们希望补全区域的颜色梯度与源图像的颜色梯度尽可能接近. 定义 $g(x,y)$ 为 Ω 上的函数表示前景图的颜色, 那么优化问题为

$$\min_f \iint_\Omega ||\nabla f - \nabla g||^2 \,\mathrm{d}S, \quad \text{s.t.} \quad f|_{\partial\Omega} = f^*|_{\partial\Omega}$$

由于图像补全是图像融合的特殊情形 ($\nabla g = 0$), 因此考虑后面的问题即可. 根据 Euler-Lagrange 方程, 上述优化问题的解满足

$$\nabla^2 f = \nabla^2 g, \quad \text{s.t.} \quad f|_{\partial\Omega} = f^*|_{\partial\Omega}$$

其中 $\nabla^2 = \dfrac{\partial^2}{\partial x^2} + \dfrac{\partial^2}{\partial y^2}$ 为 Laplacian 算子. 在离散情形下, 可以用差分法写出 $\nabla^2 f$ 的值:

$$\frac{\partial^2 f}{\partial x^2}(x,y) = [f(x+1,y) - f(x,y)] - [f(x,y) - f(x-1,y)]$$

$$\frac{\partial^2 f}{\partial y^2}(x,y) = [f(x,y+1) - f(x,y)] - [f(x,y) - f(x,y-1)]$$

于是

$$\nabla^2 f(x,y) = f(x+1,y) + f(x-1,y) + f(x,y+1) + f(x,y-1) - 4f(x,y)$$

按照一定的方式排列所有 $(x,y)\in\Omega$ 即可得到一个线性方程组. 例如, 对一个 3×3 的待求解区域 Ω 和其上待求解的像素值 $f_{11},\cdots,f_{33}$, 可以写出线性方程组:

$$\begin{bmatrix}
4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\
-1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\
0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\
0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\
0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4
\end{bmatrix}
\begin{bmatrix} f_{11} \\ f_{12} \\ f_{13} \\ f_{21} \\ f_{22} \\ f_{23} \\ f_{31} \\ f_{32} \\ f_{33} \end{bmatrix}
=
\begin{bmatrix} b_{11} \\ b_{12} \\ b_{13} \\ b_{21} \\ b_{22} \\ b_{23} \\ b_{31} \\ b_{32} \\ b_{33} \end{bmatrix}$$

记上述方程为

$$\boldsymbol{L}_\triangle \boldsymbol{f} = \boldsymbol{b}$$

其中 $\boldsymbol{L}_\triangle$ 是 Laplacian 矩阵[6].$\boldsymbol{b}$ 是根据边界和 $\nabla^2 g$ 设定的向量. 这一求解这个线性方程组即可得到每一点的像素值. 各种数值方法, 例如 Jacobi 迭代法, Gauss-Seidel 迭代法, 共轭梯度法等都可以用来求解这个线性方程

[6]Laplacian 矩阵可以被用于描述图的邻接关系, 它的 (i,i) 元表示第 i 个顶点的邻接顶点数目,(i,j) 元表示顶点 i 与顶点 j 是否相接及相接边的权重 w, 如果相接则为 w_{ij}, 否则为 0. 可以看出 Laplacian 矩阵和 Laplacian 微分算子具有密切的联系.

组.

对于形如 $\boldsymbol{Ax}=\boldsymbol{b}$ 的线性方程组, Jacobi 迭代法的迭代格式为

$$x_i^{(k+1)}=\frac{1}{a_{ii}}\left(b_i-\sum_{1\leqslant j\leqslant n,j\neq i}a_{ij}x_j^{(k)}\right)$$

可以看到每次迭代时都仅依赖上一次的结果. 如果我们及时更新迭代的结果, 就是 Gauss-Seidel 迭代法, 其迭代格式为

$$x_i^{(k+1)}=\frac{1}{a_{ii}}\left(b_i-\sum_{1\leqslant j<i}a_{ij}x_j^{(k+1)}-\sum_{i<j\leqslant n}a_{ij}x_j^{(k)}\right)$$

1.5.5 图像分割

大多数图片都有前景与背景, 但图像只有每个像素点的信息. 为了从颜色信息还原出前景与背景信息, 我们需要图像分割 (即俗称的抠图).

令 $\boldsymbol{C}$ 表示输入的图像,$\boldsymbol{F}$ 表示前景图,$\boldsymbol{B}$ 表示背景图 (假定它们都是 RGB 图像),α 表示前景的透明度, 则有

$$\boldsymbol{C}=\alpha\boldsymbol{F}+(1-\alpha)\boldsymbol{B}$$

对于某一像素点 i, 有下列方程组:

$$\begin{cases}\boldsymbol{C}_i^R=\alpha_i\boldsymbol{F}_i^R+(1-\alpha_i)\boldsymbol{B}_i^R\\ \boldsymbol{C}_i^G=\alpha_i\boldsymbol{F}_i^G+(1-\alpha_i)\boldsymbol{B}_i^G\\ \boldsymbol{C}_i^B=\alpha_i\boldsymbol{F}_i^B+(1-\alpha_i)\boldsymbol{B}_i^B\end{cases}$$

除去 $\boldsymbol{C}_i^R,\boldsymbol{C}_i^G,\boldsymbol{C}_i^B$ 为已知量外, 有 7 个未知数, 但只有 3 个方程. 即使知道背景 $\boldsymbol{B}$, 未知数数目也大于方程数目, 因此这方程组欠定, 有无穷多组解.

蓝/绿幕抠图

如果背景是纯色 (通常是蓝色或绿色), 那么可以按照下面的办法处理 (以蓝色背景为例): 假设背景图 $\boldsymbol{B}$ 只有蓝色分量 $\boldsymbol{B}_i^B$, 并且这一分量已知; 另外前景图 $\boldsymbol{F}$ 没有蓝色分量. 于是上述方程组变为

$$\begin{cases}\boldsymbol{C}_i^R=\alpha_i\boldsymbol{F}_i^R\\ \boldsymbol{C}_i^G=\alpha_i\boldsymbol{F}_i^G\\ \boldsymbol{C}_i^B=(1-\alpha_i)\boldsymbol{B}_i^B\end{cases}$$

其中 $\alpha_i,\boldsymbol{F}_i^R,\boldsymbol{F}_i^G$ 是未知量. 这样就可以解得唯一解, 从而得到前景 $\boldsymbol{F}$.

这一技术主要应用在电影特效制作中, 因为可以使用纯色幕布控制拍摄背景. 应用蓝色/绿色主要是因为人身上的颜色以红色/黄色居多, 蓝色/绿色成分相对较少.

蓝/绿幕抠图的缺点在于, 如果前景中有蓝色/绿色的部分, 那么这些部分会被误认为是背景而被抠掉. 因此这项技术不适用于透明物体的图像分割.

三角抠图

既然我们缺乏信息, 另一种简单的办法是保持 $\boldsymbol{F}$ 不变, 拍摄另一张背景 $\hat{\boldsymbol{B}}$ 对应的图像 $\hat{\boldsymbol{C}}$, 并且假定 $\boldsymbol{B}$ 和 $\hat{\boldsymbol{B}}$ 已知. 于是有

$$\begin{cases} \boldsymbol{C}_i^R = \alpha_i \boldsymbol{F}_i^R + (1-\alpha_i)\boldsymbol{B}_i^R \\ \boldsymbol{C}_i^G = \alpha_i \boldsymbol{F}_i^G + (1-\alpha_i)\boldsymbol{B}_i^G \\ \boldsymbol{C}_i^B = \alpha_i \boldsymbol{F}_i^B + (1-\alpha_i)\boldsymbol{B}_i^B \\ \hat{\boldsymbol{C}}_i^R = \alpha_i \boldsymbol{F}_i^R + (1-\alpha_i)\hat{\boldsymbol{B}}_i^R \\ \hat{\boldsymbol{C}}_i^G = \alpha_i \boldsymbol{F}_i^G + (1-\alpha_i)\hat{\boldsymbol{B}}_i^G \\ \hat{\boldsymbol{C}}_i^B = \alpha_i \boldsymbol{F}_i^B + (1-\alpha_i)\hat{\boldsymbol{B}}_i^B \end{cases}$$

这里有四个未知数和六个方程, 可以通过最小二乘法求得满意的解.

这个方法的要求相比蓝/绿幕抠图较宽松, 它可以处理各种透明的物体, 也不要求背景的颜色. 然而, 它要求拍摄图片时前景物体相对相机的位置必须一致, 在不能方便地更换背景时也比较麻烦.

Bayesian 抠图

相比前面两种方法对背景或拍摄角度的限制, Bayesian 抠图是更智能普适的算法. 除去输入图像 $\boldsymbol{C}$, 它还需要提供一张三值图, 用于标记每个像素点属于背景, 前景还是不确定点.

记输入的原图为 $\boldsymbol{C}$, 三值图为 $\boldsymbol{T}$. 我们的目标是用最大似然法求出标量场 α, 前景图 $\boldsymbol{F}$ 和背景图 $\boldsymbol{B}$, 即最大化问题

$$\max_{\alpha,\boldsymbol{F},\boldsymbol{B}} P\left(\alpha, \boldsymbol{F}, \boldsymbol{B} | \boldsymbol{C}\right)$$

假定像素点之间相互独立. 考虑像素点 i(对应的 $\boldsymbol{C}_i$ 等都是包含 RGB 通道的三维向量, 因此下面需要用到多元高斯分布), 由 Bayes 公式有

$$P\left(\alpha_i, \boldsymbol{F}_i, \boldsymbol{B}_i | \boldsymbol{C}_i\right) = \frac{P\left(\boldsymbol{C}_i | \alpha_i, \boldsymbol{F}_i, \boldsymbol{B}_i\right) P\left(\alpha_i, \boldsymbol{F}_i, \boldsymbol{B}_i\right)}{P(\boldsymbol{C}_i)}$$

其中 $P(\boldsymbol{C}_i)$ 为常数. 为了简化问题, 假定 $\alpha, \boldsymbol{F}_i$ 和 $\boldsymbol{B}_i$ 相互独立, 并且 $P\left(\alpha_i\right)$ 也是常数. 于是对上述取对数可得

$$\ln P\left(\alpha_i, \boldsymbol{F}_i, \boldsymbol{B}_i | \boldsymbol{C}_i\right) = \ln P\left(\boldsymbol{C}_i | \alpha_i, \boldsymbol{F}_i, \boldsymbol{B}_i\right) + \ln P\left(\boldsymbol{F}_i\right) + \ln P\left(\boldsymbol{B}_i\right) + \text{const}$$

现在就需要对上述目标函数中的各个概率建模. 我们假定上述三个分布都是高斯分布. 对于第一个分布 $P\left(\boldsymbol{C}_i | \alpha_i, \boldsymbol{F}_i, \boldsymbol{B}_i\right)$, 其均值为 $\alpha_i \boldsymbol{F}_i + (1-\alpha_i)\boldsymbol{B}_i$, 并且假定分布是各向同性的, 其方差为可调参数 σ_c^2. 于是

$$P\left(\boldsymbol{C}_i | \alpha_i, \boldsymbol{F}_i, \boldsymbol{B}_i\right) = \frac{1}{(2\pi)^{3/2}\sigma_c^3} \exp\left(-\frac{||\boldsymbol{C}_i - \alpha_i \boldsymbol{F}_i - (1-\alpha_i)\boldsymbol{B}_i||^2}{2\sigma_c^2}\right)$$

后面两个分布是类似的. 以第二个分布 $P\left(\boldsymbol{F}_i\right)$ 为例, 其分布可以写为如下形式:

$$P\left(\boldsymbol{F}_i\right) = \frac{1}{(2\pi)^{3/2}||\boldsymbol{\Sigma}_{\boldsymbol{F}_i}||^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{F}_i - \boldsymbol{\mu}_{\boldsymbol{F}_i})^{\mathrm{t}} \boldsymbol{\Sigma}_{\boldsymbol{F}_i}^{-1} (\boldsymbol{F}_i - \boldsymbol{\mu}_{\boldsymbol{F}_i})\right)$$

其中均值 $\boldsymbol{\mu}_{\boldsymbol{F}_i}$ 和协方差矩阵 $\boldsymbol{\Sigma}_{\boldsymbol{F}_i}$ 可以通过在三值图中标记为前景的像素点的颜色来估计. 具体而言, 考虑 i 的一个邻域 S(通常是以 i 为中心的圆或矩形) 内的所有标记为前景的像素点构成的集合 $\mathcal{N}_{\boldsymbol{F}_i}$, 则有

$$\boldsymbol{\mu}_{\boldsymbol{F}_i} = \frac{1}{W}\sum_{j\in\mathcal{N}_{\boldsymbol{F}_i}} w_j \boldsymbol{F}_j \qquad \boldsymbol{\Sigma}_{\boldsymbol{F}_i} = \frac{1}{W}\sum_{j\in\mathcal{N}_{\boldsymbol{F}_i}} w_j \left(\boldsymbol{F}_j - \boldsymbol{\mu}_{\boldsymbol{F}_i}\right)\left(\boldsymbol{F}_j - \boldsymbol{\mu}_{\boldsymbol{F}_i}\right)^{\mathrm{t}}$$

其中 w_j 为权重, $W = \sum_{j\in\mathcal{N}_{\boldsymbol{F}_i}} w_j$ 为归一化常数. 权重 w_j 由该点的透明度 α_j 和高斯距离权值 g_j(以像素 i 为中心, 方差为 8 的高斯分布上 j 所在位置的取值) 决定, 有 $w_j = \alpha_j^2 g_j$. 这样,α_j 越高, 意味着这一采样点越有可能属于前景, 权重就越大. 同样地, 距离越近, 权重 g_j 也越大.

对于背景分布 $P(\boldsymbol{B}_i)$, 计算方法也是一致的, 只需把权重换成 $w_j = (1-\alpha_j)^2 g_j$ 即可.

把上述三个分布代入目标函数, 然后通过迭代法求最大值, 即可完成目标.

1.5.6 图像抖动

将颜色深度较高的图像转换为颜色深度较低的图像时, 会出现程度不一的色带现象 (即原图中连续变化的颜色在量化后变成离散的色块). 为了减轻这种现象, 可以使用抖动技术.

我们假定目标为将一张灰度图像二值化.

有序抖动

我们从比较简单的任务开始: 假定新图像的尺寸是原图像的三倍, 那么我们可以将原图的一个像素对应到新图的 3×3 个像素. 这样, 我们可以根据原图像素的灰度值决定这 9 个像素取黑色还是白色. 按照灰度的不同, 我们可以设计抖动矩阵 $\boldsymbol{M}$, 如下所示:

$$\boldsymbol{M} = \begin{bmatrix} 6 & 8 & 4 \\ 1 & 0 & 3 \\ 5 & 2 & 7 \end{bmatrix}$$

假定像素的灰度值 g_i 取值范围为 $(0,1)$. 我们可以把这一区间划分为 9 个区域, 即将灰度值乘以 9. 如果 $9g_i > \boldsymbol{M}_{jk}$, 就把新图像中对应于原图像素 i 的 3×3 区域中的 (j,k) 像素点设为黑色, 否则设为白色. 这样, 就可以得到一张二值图像.

(a) 源图像 (b) 经过有序抖动处理后的图像

图 1.1: 有序抖动的效果演示

有序抖动实际上是给图像添加了周期性的噪声后二值化的结果. 例如, 考虑抖动矩阵的元素 $\boldsymbol{M}_{00}$, 当 $9g_i \geqslant 6$, 即 $g_i - 0.167 \geqslant 0.5$ 时, 对应的像素点被设为黑色, 也就是给该像素加上 -0.167 的噪声后进行二值化. 我们将在下一节解释为何加上噪声后进行二值化可以得到更好的效果.

基于噪声的抖动

基于前面的思想, 我们可以把图像加上随机生成的噪声后进行二值化.

噪声的作用 我们假定原图第 i 个像素的灰度值为 f_i, 二值化后图像的第 i 个像素的灰度值为 g_i. 定义量化后图像与原图的平均误差 E 如下:

$$E = \frac{1}{N}\sum_{i=0}^{N-1}(f_i - g_i)$$

其中 N 为像素总数. 我们希望 E 尽可能小. 为此, 考虑一种最简单的情形, 令 $f_i = 0.25$. 于是直接二值化的误差为

$$E = \frac{1}{N}\sum_{i=0}^{N-1}\left(\frac{1}{4} - 0\right) = \frac{1}{4}$$

现在给每个像素加上均匀分布在 $[-0.5, 0.5]$ 的噪声 ε_i, 然后再二值化. 于是

$$E_{\text{noise}} = \frac{1}{N}\sum_{i=0}^{N-1}\left(\frac{1}{4} - I_{0.25+\varepsilon_i>0.5}\right)$$

其中 I_A 为事件 A 的示性函数, 当 A 发生时取 1, 否则取 0. 由于 ε_i 是一个随机变量, 因此我们分析 E_{noise} 的数学期望:

$$\mathbb{E}(E_{\text{noise}}) = \frac{1}{4} - \frac{1}{N}\sum_{i=0}^{N}\mathbb{E}(I_{0.25+\varepsilon_i>0.5}) = \frac{1}{4} - \frac{1}{N}\sum_{i=0}^{N}P(\varepsilon_i > 0.25) = \frac{1}{4} - \frac{1}{4} = 0$$

可以发现, 在平均意义下加上噪声后二值化的误差变得更小.

白噪声与蓝噪声 上述加噪声的办法 (即给原像素加上均布于 $[-0.5, 0.5]$ 的噪声) 称作**白噪声**. 相比于直接二值化, 白噪声抖动中的色带线性大大减轻, 但仍然会出现明显的噪点.

改变噪声生成的方式, 可以得到更好的效果. 另一种常用的噪声是**蓝噪声**. 典型的蓝噪声生成方式是在图像上随机生成一系列两两距离不小于某一值 r 的点, 然后将该图像作为噪声的采样点进行二值化.

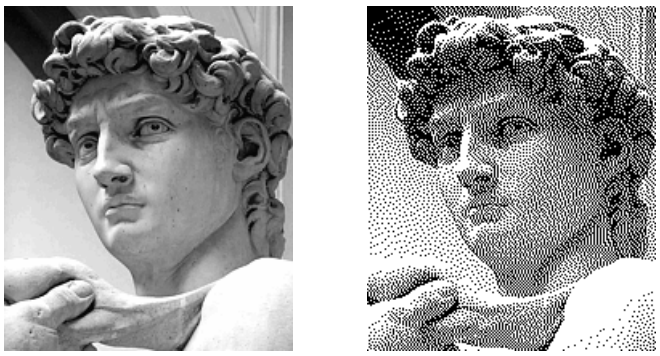
(a) 源图像 (b) 经过白噪声处理后的图像 (c) 经过蓝噪声处理后的图像

图 1.2: 噪声抖动的效果演示

从频谱上看, 白噪声的频谱是均匀分布的, 而蓝噪声的频谱在低频部分较弱, 在高频部分较强. 由于人眼更难注意到高频信号, 因此蓝噪声抖动后的图像看起来更为自然.

基于误差扩散的抖动

另一种抖动方法是 **Floyd-Steinberg 抖动算法**. 它的基本思想是, 在二值化某一像素后, 计算该像素的量化误差, 然后将该误差按一定权重分配给该像素周围尚未处理的像素. 具体而言, 我们从上到下, 从左到右地处理图像的每一个像素. 对于当前像素 i, 我们先将其灰度值 f_i 加上之前分配给它的误差 ε_i 后进行二值化, 得到二值化后的灰度值 g_i. 计算量化误差 $\delta_i = f_i + \varepsilon_i - g_i$, 然后将 δ_i 分为 $7/16, 5/16, 3/16, 1/16$ 四部分, 分别加到它右边, 左下, 下方和右下的四个像素上. 依次处理所有像素即可得到结果.



(a) 源图像 (b) 误差扩散抖动处理后的图像

图 1.3: 误差扩散抖动的效果演示

这是一个确定性的算法, 它的效果与噪声抖动相似, 但不会出现明显的噪点或规则性斑点. 然而, 由于它需要逐像素处理并不断扩散误差, 其计算速度相对更慢.

第二章　几何建模

2.1 几何表示

2.1.1 几何形状的表示方法

几何形状 (geometric shape) 是指空间中的一组特定点的集合. 对于二维空间, 常见的几何图形包括线段, 多边形, 圆等等; 对于三维空间, 常见的几何图形包括线, 面, 体等.

连续函数表示法

和平面图形一样, 空间图形也可以用连续函数表示. 例如, 空间中的直线可以表示为

$$\{(x,y,z) : Ax + By + Cz + D = 0, x, y, z \in \mathbb{R}\}$$

空间中的球面可以表示为

$$\left\{(x,y,z) : x^2 + y^2 + z^2 = R^2, x, y, z \in \mathbb{R}\right\}$$

定义 1.1 连续函数表示法 一般地, 对于空间中的几何形状 M, 可以用一个特定的连续函数 $S_M(x,y,z)$ 刻画其表面 ∂M:

$$\{(x,y,z) : S_M(x,y,z) = 0, x, y, z \in \mathbb{R}\}$$

连续函数可以精确地刻画几何形状, 也方便研究其性质. 然而, 对于复杂的图形 (尤其是复杂的曲线或曲面), 难以找到合适的函数来表示; 并且这一表达形式是隐式的, 不能直接将几何形状呈现出来. 这就需要离散化的办法.

点云

通过类似雷达的工作方式对几何形状 M 进行扫描, 可以获知 ∂M 上一系列离散的点.

定义 1.2 点云 **点云 (Point Cloud)** 是一组三维空间中有限个点构成的集合:

$$\{(x_i, y_i, z_i) : i = 1, \cdots, N\}$$

这集合描述的几何形状 M 满足: 对任意点云中的点 (x_i, y_i, z_i), 都有 $(x_i, y_i, z_i) \in \partial M$, 即

$$S_M(x_i, y_i, z_i) = 0$$

点云作为原本几何形状的采样结果, 保留了原形状的一部分几何信息，, 使得我们可以在点云上进行一些表面性质的计算, 例如计算法向和曲率.

然而采样总是伴随着信息的损失, 点云也不例外. 点云的采样密度决定了它对几何细节的表示能力. 更重要的是, 由于点云本身的非结构化和无序性, 几何形状的拓扑关系往往是最容易在点云表示中变得模糊不清的. 这为基于点云的几何形状计算和处理带来了困难.

网格模型

为了解决点云对于拓扑形状表示的不足, 我们考虑对曲面表面进行线性近似, 即用一系列小的多边形片段拼接成曲面. 为此, 把点云中的点按照 M 的形状进行连接, 可以得到一个由多边形构成的网格.

定义 1.3 网格模型 **网格模型 (Mesh Model)** 是由一组顶点, 边和面构成的三元组:

$$\mathcal{M} = (\mathcal{V}, \mathcal{E}, \mathcal{F})$$

其中 $\mathcal{V} = \{\boldsymbol{v}_i = (x_i, y_i, z_i) : i = 1, \cdots, N\}$ 是顶点集合, 每个顶点对应点云中的一个点; $\mathcal{E} = \{\boldsymbol{e}_{ij} = (v_i, v_j) : i, j = 1, \cdots, N\}$ 是边集合, 每条边连接两个顶点; $\mathcal{F} = \{\boldsymbol{f}_k\}$ 是面集合, 每个面 $\boldsymbol{f}_k$ 由数个顶点组成.
$\mathcal{M}$ 描述的几何形状满足: 对于任意面 $\boldsymbol{f}_k$, 其上所有点都在 ∂M 上, 即

$$\forall (x_k, y_k, z_k) \in \boldsymbol{f}_k, \quad S_M(x_k, y_k, z_k) = 0$$

一般而言, 我们要求网格是**流形**的, 即每条边最多被两个面共享. 有时, 我们还要求网格是**水密**的, 即一个由封闭曲面组成的, 没有孔洞的网格.

2.1.2 网格表示

网格模型是计算机图形学中最常用的几何表示方法, 我们已经在上一节介绍过其定义. 在计算机中存储时, 通常使用顶点列表, 边列表和面列表作为储存多边形网格的数据结构; 有时也会设计额外的数据结构方便邻边查找等操作.

三角网格表示法

我们从最简单的三角网格开始, 即所有面都是三角形的网格. 最简单的表示方法就是记录每个三角形的三个顶点, 即

$$\triangle_i = (\boldsymbol{v}_{i0}, \boldsymbol{v}_{i1}, \boldsymbol{v}_{i2}), \quad \text{where } \boldsymbol{v}_{ij} = (x_{ij}, y_{ij}, z_{ij})$$

定义 1.4 三角形乱序集合 **三角形乱序集合 (Triangle Soup)** 是由一组三角形构成的集合, 每个三角形包括其顶点信息.

显然, 每个顶点几乎都会被多个三角形共用, 因此上面的表示方法在空间上有很大冗余. 并且由于存储的乱序性, 我们也不易对模型进行拓扑关系的考察.

为了减少空间开销, 我们可以先单独存储顶点, 然后只存储每个三角形顶点在顶点列表中的索引.

定义 1.5 索引三角形网格 **索引三角形网格 (Indexed Triangle Set)** 由顶点列表 $\mathcal{V}$ 和三角形列表 $\mathcal{F}$ 构成. 其中 $\mathcal{V} = \{\boldsymbol{v}_i = (x_i, y_i, z_i) : i = 1, \cdots, N\}$ 是顶点列表; $\mathcal{F} = \{\triangle_k\}$ 是三角形列表, 每个三角形 $\triangle_k$ 由三个顶点索引组成.

特别地, 为了方便处理, 我们在存储顶点索引时可以按照逆时针方向存储. 这可以保证每个三角形的法向方向一致, 从而方便后续的渲染等操作.

半边数据结构

在网格中, 我们经常会面对顶点邻接关系的查询, 也需要有序地遍历顶点和面. 在一般的索引三角形网格中, 我们只能通过遍历所有三角形来找到某个顶点的邻接顶点, 这显然效率很低. **半边数据结构 (Half-Edge Data Structure)** 就是一种更高效的网格表示方法.

由于我们主要考虑流形, 因此可以把每条边拆成两个方向相反的**半边 (half-edge)**, 每个半边只属于一个面, 而每个面可以由首尾相接的数个半边表示. 对于每个半边, 我们记录其起点, 终点, 所属面, 上一个和下一个半边, 以及与其配对的另一个半边. 这样, 我们就可以通过半边快速地找到顶点的邻接顶点和邻接面.

定义 1.6 半边数据结构 **半边数据结构 (Half-Edge Data Structure)** 相比一般的网格数据结构增加了对每个边的半边表示. 每个半边 `e` 主要包含以下信息:

- `e->From()`: 半边的起点;
- `e->Twin()`: 与该半边配对的另一个半边;
- `e->Face()`: 该半边所属的面;
- `e->Next()`: 该半边在所属面中的下一个半边;
- `e->Prev`: 该半边在所属面中的上一个半边.
- `e->To()`: 半边的终点, 等于 `e->Twin()->From()`.

实现半边数据结构主要通过**双向边链表 (DCEL, Doubly Connected Edge List)** 来完成. DECL 中不仅新增了半边的信息, 其面和顶点也储存了与相关半边的信息, 从而构成了一张结构完善的图.

使用半边数据结构能完成涉及网格拓扑结构的各种操作, 例如. 下面给出两个简单的例子.

遍历面的顶点/边 给定面 `f`, 可以通过 `f->Edge()`[1]得到该面上的一个半边 `e`. 然后不断访问 `e->Next()`, 直到回到起点, 就可以遍历该面上的所有顶点和边.

```
1 DCEL::HalfEdge const * e = f->Edge();
2 DCEL::HalfEdge const * e_start = f->Edge();
3 do {
4     DCEL::VertexProxy const * v = e->From(); // 访问顶点
5     e = e->Next(); // 访问下一条边
```

[1]在 Lab 中可以直接通过 `f->Edge(i)` 访问顶点 `i` 的对边, 但这里采用更加本质的办法, 即通过类似环状链表的形式遍历.

```
} while (e != e_start);
```

围绕顶点进行遍历 前面我们给出了围绕面构造迭代器的办法. 另一个常用的迭代器是顶点迭代器, 即围绕某个顶点访问其所有邻接顶点[2]. 给定顶点 `v`, 可以通过 `v->Edge()` 得到一条以 `v` 为起点的半边 `e`. 然后不断访问 `e->Twin()->Next()`, 直到回到起点, 就可以遍历该顶点的所有邻接顶点.

```
DCEL::HalfEdge const * e = v->Edge();
DCEL::HalfEdge const * e_start = v->Edge();
do {
    DCEL::VertexProxy const * u = e->To(); // 访问邻接顶点
    e = e->Twin()->Next(); // 访问下一条邻接顶点的边
} while (e != e_start);
```

总之, 半边数据结构通过增加对边的表示, 使得网格的拓扑关系更加清晰, 从而方便了各种基于网格的计算和处理.

2.1.3 网格细分

通过增加组成几何表面的网格面片数量, 减小每个面片的面积, 可以使几何表面看起来更加光滑.

定义 1.7 网格细分 **网格细分 (Mesh Subdivision)**, 又称作网格的**上采样**, 是指通过反复细分初始的多边形网格, 不断得到更精细的网格的过程.

Catmull-Clark 细分

Catmull-Clark 细分是最常用的几何表面细分方法之一, 主要应用于四边形网格的细分上. Catmull-Clark 细分的具体步骤如下:

1. **增设面点**: 对多面体的每个面片计算一个面点, 该面点是该面片所有顶点的平均值:

$$\boldsymbol{v}_{\text{face}} = \frac{1}{N}\sum_{n=1}^{N}\boldsymbol{v}_n$$

2. **增设边点**: 对多面体的每条边计算一个边点, 该边点是该边两个端点 $\boldsymbol{v}_1$ 和 $\boldsymbol{v}_2$, 以及相邻两个面片的面点 $\boldsymbol{v}_{\text{face},1}$ 和 $\boldsymbol{v}_{\text{face},2}$ 的平均值:

$$\boldsymbol{v}_{\text{edge}} = \frac{1}{4}\left(\boldsymbol{v}_1 + \boldsymbol{v}_2 + \boldsymbol{v}_{\text{face},1} + \boldsymbol{v}_{\text{face},2}\right)$$

3. **更新顶点**: 对多面体原有的每个顶点 $\boldsymbol{v}$, 使用下面的加权平均算法更新其位置:

$$\boldsymbol{v}_{\text{vertex}} = \frac{\boldsymbol{F} + 2\boldsymbol{R} + (N-3)\boldsymbol{v}}{N}$$

[2]在 Lab 中, 可以通过 `v->Ring()` 直接访问所有相邻的面中与 `v` 不相交的, 相对的半边. 对于非边界上的 `v`, 这些半边组成了一个环.

其中 $\boldsymbol{F}$ 是所有以 $\boldsymbol{v}$ 为顶点的面片的面点的均值, $\boldsymbol{R}$ 是所有以 $\boldsymbol{v}$ 为端点的边的中点 (注意不是 **2.** 中的边点) 的均值, N 是与该顶点相邻的顶点数.

4. **重新连接**: 将每个面点 $\boldsymbol{v}_{\text{face}}$ 与该面片所有边对应的的边点 $\boldsymbol{v}_{\text{edge}}$ 相连; 将每个新顶点 $\boldsymbol{v}_{\text{vertex}}$ 与原有顶点所有相邻边的边点 $\boldsymbol{v}_{\text{edge}}$ 相连. 于是就形成新的细分过后的面片.

Loop 细分

Loop 细分是另一种常用的几何表面细分方法, 主要应用于三角形网格的细分上. Loop 细分的具体步骤如下:

1. **增设新顶点**: 对于每一条边, 如果这条边被两个三角形面包含, 则根据这条边的两个端点 $\boldsymbol{v}_0, \boldsymbol{v}_2$ 和这两个三角形除这条边外各自的顶点 $\boldsymbol{v}_1, \boldsymbol{v}_3$ 加权平均得到新顶点 $\boldsymbol{v}^*$:

$$\boldsymbol{v}^* = \frac{3}{8}(\boldsymbol{v}_0 + \boldsymbol{v}_2) + \frac{1}{8}(\boldsymbol{v}_1 + \boldsymbol{v}_3)$$

   如果这条边只被一个三角形面包含, 则取边的中点得到新顶点 $\boldsymbol{v}^*$.

2. **更新原有顶点**: 对于每个原有的顶点 $\boldsymbol{v}$, 按照下面的公式更新其位置至 $\boldsymbol{v}'$:

$$\boldsymbol{v}' = (1 - nu)\boldsymbol{v} + \sum_{i=1}^{n} u\boldsymbol{v}_i$$

   其中 n 为 $\boldsymbol{v}$ 邻接顶点的数目,$\boldsymbol{v}_i$ 是与 $\boldsymbol{v}$ 相邻的顶点. 当 $n = 3$ 时, $u = \dfrac{3}{16}$; 当 $n > 3$ 时, $u = \dfrac{3}{8n}$.

3. **重新连接**: 将每个原有三角形的三个边的构造出的新点与原有顶点更新后的位置相连, 形成四个新的三角形.

2.1.4 网格参数化

网格参数化概述

网格参数化起源于纹理映射的需要. 在为三维图形着色时 (类似地, 将三维图形展平至二维, 例如绘制世界地图等情形时), 我们需要建立三维图形表面与二维纹理图像之间的映射关系, 即建立三维图形上的点 $\boldsymbol{v}_i(x_i, y_i, z_i)$ 到二维平面上的点 $\boldsymbol{u}_i(u_i, v_i)$ 的映射 (这与 UV 坐标的思想类似).

定义 1.8 网格参数化 **网格参数化 (Mesh Parameterization)** 是指将三维空间中的网格映射到二维平面上的过程. 具体地, 对于三维空间中的网格 $\mathcal{M} =$, 我们希望找到一个映射 $f : \mathbb{R}^3 \to \mathbb{R}^2$, 使得每个顶点 $\boldsymbol{v}_i \in \mathcal{V}$ 都对应一个二维平面上的点 $\boldsymbol{u}_i = f(\boldsymbol{v}_i) \in \mathbb{R}^2$.

按照参数化中保持的几何量, 可以将网格参数化分为保长度 (Isometric) 的参数化, 保角 (Conformal) 的参数化和保面积 (Areal) 的参数化等. 保长度的参数化等价于既保角度又保面积的参数化. 理想的参数化可以保持形状不发生变化, 但只有可展曲面 (例如圆柱面) 可以进行保形参数化, 而大部分曲面在参数化时都会发生一定形变.

一般而言, 我们参数化的对象是具有边界的开网格. 对于封闭的图形, 通常先指定一条边将其裁切为开网格, 然后进行参数化.

对开网格的参数化主要分为**固定边界映射**和**自由边界映射**两类. 固定边界映射是指将边界顶点映射到二维平面的一个预设形状 (例如圆形或正方形) 上, 然后计算内部顶点映射到的坐标. 自由边界映射则不对边界顶点进行预设, 而是同时优化确定边界和内部顶点的二维坐标.

基于弹簧模型的网格参数化/重心映射

在开网格参数化中, 最基础的模型为**弹簧模型**. 我们将网格顶点作为节点, 网格的边作为连接节点的弹簧. 参数化后的系统状态, 可以由各个弹簧的弹性势能之和衡量, 最终的平衡状态 (系统总弹性势能最小的状态) 即为参数化的结果.

考虑网格 $\mathcal{M}$ 的顶点 $\boldsymbol{v}_1, \cdots, \boldsymbol{v}_n$, 各顶点参数化后的二维坐标为 $\boldsymbol{t}_1, \cdots, \boldsymbol{t}_n$. 系统的总能量 (求和时每条边的能量都被计算两遍, 因此前面还需乘以 $1/2$) 可以描述为

$$E = \frac{1}{2}\sum_{i=1}^{n}\sum_{j\in\Omega(i)}\frac{1}{2}D_{ij}\left|\left|\boldsymbol{t}_i - \boldsymbol{t}_j\right|\right|^2$$

其中 $\Omega(i)$ 表示与顶点 i 相邻的顶点集合, D_{ij} 是连接顶点 i 和 j 的弹簧的弹性系数. 系统能量最小时, 对各 $\boldsymbol{t}_i$ 的偏导均为 $\boldsymbol{0}$, 即

$$\frac{\partial E}{\partial \boldsymbol{t}_i} = \sum_{j\in\Omega(i)} D_{ij}\left(\boldsymbol{t}_i - \boldsymbol{t}_j\right) = \boldsymbol{0}$$

记组合系数

$$\lambda_{ij} = \frac{D_{ij}}{\sum\limits_{k\in\Omega(i)} D_{ik}}$$

于是上式可以整理为

$$\boldsymbol{t}_i - \sum_{j\in\Omega(i)} \lambda_{ij}\boldsymbol{t}_j = \boldsymbol{0}$$

可以看出,$\boldsymbol{t}_i$ 由其周围的点按照一定系数加权得到, 因此这一方法又称作**重心映射**. 这是一个有 $2n$ 个变量和 $2n$ 个方程的齐次线性方程组, 其平凡解为零解. 为了得到非零解, 一种简单的办法是将开网格边界上的点根据它们之间的距离映射到指定的凸图形的边界 (常见的有正方形, 圆形) 上, 然后将内部的点作为未知量按照上面的方程组求解.

另外, 可以设置合适的系数 λ_{ij} 引导参数化. 系数的设置需满足

(1) 凸组合性: 对凸多边形的加权平均仍然在凸多边形内部, 即

$$\lambda_{ij} \geqslant 0, \quad \sum_{j\in\Omega(i)} \lambda_{ij} = 1$$

(2) 线性重构性: 如果顶点 $\boldsymbol{v}_i$ 和其邻接顶点 $\boldsymbol{v}_j$ 在三维空间中共线, 则参数化后的点 $\boldsymbol{t}_i$ 和 $\boldsymbol{t}_j$ 也应当共线; 亦即平面网格映射后保持不变. 即

$$\sum_{j\in\Omega(i)} \lambda_{ij}\left(\boldsymbol{v}_i - \boldsymbol{v}_j\right) = \boldsymbol{0}$$

基于此和一些几何上的考虑, 常见的组合系数有下面几种:

(1) 平均系数: 每个邻接顶点的权重相等, 即

$$\lambda_{ij} = \frac{1}{|\Omega(i)|}$$

(2) 均值坐标系数 (Floater 权重): 基于顶点 $\boldsymbol{v}_i$ 和其邻接顶点 $\boldsymbol{v}_j$ 与 $\boldsymbol{v}_i$ 的连线与其邻接边的夹角 α_{j1} 和 α_{j2}, 定义权重为

$$\lambda_{ij} = \frac{\tan(\alpha_{j1}/2) + \tan(\alpha_{j2}/2)}{||\boldsymbol{v}_i - \boldsymbol{v}_j||}$$

(3) 调和坐标系数 (余切 Laplacian 权重): 基于顶点 $\boldsymbol{v}_i$ 和其邻接顶点 $\boldsymbol{v}_j$ 与 $\boldsymbol{v}_i$ 的连线相对的夹角 β_{j1} 和 β_{j2}, 定义权重为

$$\lambda_{ij} = \frac{\cot\beta_{j1} + \cot\beta_{j2}}{2}$$

调和坐标系数更接近保角映射, 但也可能出现权重为负值的情况, 从而导致映射出现折叠.

2.2 几何处理

2.2.1 离散微分几何

我们总是用离散的网格[3]来描述连续的曲面, 但网格定义出的几何形状只能给出 $\mathcal{C}^0$ 连续性. 如何将连续曲面的各种微分量应用在离散的情况下就属于**离散微分几何**研究的内容. 常见的各种微分量有法向量, 曲率, 拉普拉斯算子等.

局部平均区域

我们可以用下面的方法刻画三角网格顶点 $\boldsymbol{x}_i$ 处的微分量 $\boldsymbol{f}$:

$$\boldsymbol{f}(\boldsymbol{x}_i) = \left(\iint_{\Omega(\boldsymbol{x}_i)} \boldsymbol{f} \mathrm{d}S \right) \Big/ \left(\iint_{\Omega(\boldsymbol{x}_i)} \mathrm{d}S \right)$$

即在 $\boldsymbol{x}$ 的某个邻域 $\Omega(\boldsymbol{x})$ 的邻域上该微分量的平均值. 邻域的选择直接影响离散微分量的结果和准确度. 邻域越大, 结果越平滑, 但也会丢失更多的细节. 邻域越小, 结果越接近真实值, 但也对噪声更敏感. 常见的邻域选择有以下几种:

1. **重心单元 (barycentric cell)**: 把与顶点 $\boldsymbol{x}$ 相邻的所有三角形的两边的中点和重心连起来形成的多边形区域.
2. **泰森多边形单元 (Voronoi cell)**: 把与顶点 $\boldsymbol{x}$ 相邻的所有三角形的外心连起来形成的多边形区域.
3. **混合泰森多边形单元 (mixed Voronoi cell)**: Vorinoi cell 在钝角三角形中取的点可能超出三角形外, 因此在钝角三角形中用边的中点代替外心, 其余情形不变.

法向量

三角形内部点的法向量可以唯一地定义为三角形所在平面的法向量. 对于顶点 $\boldsymbol{x}$ 处的法向量 $\boldsymbol{n}(\boldsymbol{x})$, 可以通过对顶点周围三角形的法向量加权平均得到:

$$\boldsymbol{n}(\boldsymbol{x}) = \frac{\sum\limits_{T \in \Omega(\boldsymbol{x})} \alpha_T \boldsymbol{n}(T)}{\sum\limits_{T \in \Omega(\boldsymbol{x})} \alpha_T \left|\left| \boldsymbol{n}(T) \right|\right|}$$

其中 $\Omega(\boldsymbol{x})$ 是与顶点 $\boldsymbol{x}$ 邻接的三角形构成的几何,α_T 和 $\boldsymbol{n}(T)$ 是三角形 T 的权重和法向量. 常见的权重有以下几种:

1. **常数权重**: $\alpha_T = 1$. 这在计算上最简单, 但对于一些不规则网格效果不好.
2. **面积权重**: $\alpha_T = A_T$. 以三角形面积 A_T 加权, 比常数权重更合理, 但对过于不规则的网格效果也不好.
3. **角度权重**: $\alpha_T = \theta_T$, 其中 θ_T 是顶点 $\boldsymbol{x}$ 在三角形 T 中的对角. 这样加权效果最好, 但计算上也最复杂.

[3]在本节, 我们主要以三角网格作为研究对象.

梯度

梯度是非常重要的微分量, 在计算拉普拉斯算子, 网格参数化等方面有重要应用.

对于一个在三角网格上定义的标量函数 f, 我们可以通过 f 在三角形顶点上的值通过重心插值的方式得到三角形内部的值. 不妨设三角形的顶点为 $\boldsymbol{x}_a, \boldsymbol{x}_b, \boldsymbol{x}_c$, 则其内部任意一点 $\boldsymbol{x}$ 的值为:

$$f(\boldsymbol{x}) = \alpha(\boldsymbol{x})f(\boldsymbol{x}_a) + \beta(\boldsymbol{x})f(\boldsymbol{x}_b) + \gamma(\boldsymbol{x})f(\boldsymbol{x}_c)$$

其中 α, β, γ 是点 $\boldsymbol{x}$ 在三角形中的重心坐标, 在 **Lecture 3 绘图**一节中已经讲过其定义方式. 我们在那时已经证明重心坐标是线性的, 因此 f 的梯度 ∇f 也是定值. 现在来证明之.

证明. 不妨假定 $\boldsymbol{x}_a, \boldsymbol{x}_b$ 和 $\boldsymbol{x}_c$ 在三角形 T 中对应的顶点分别为 A, B 和 C, $\boldsymbol{x}$ 对应 T 内的一点 P. 以 $\alpha(\boldsymbol{x})$ 为例, 根据重心坐标与三角形面积的关系, 有

$$\alpha(\boldsymbol{x}) = \frac{A_{\triangle PBC}}{A_{\triangle ABC}} = \frac{\overline{BP} \cdot \overline{BC} \cdot \sin\angle PBC}{2A_T} = \frac{\left\|\overrightarrow{BP} \times \overrightarrow{BC}\right\|}{2A_T} = \frac{(\boldsymbol{x} - \boldsymbol{x}_b) \cdot (\boldsymbol{x}_c - \boldsymbol{x}_b)^\perp}{2A_T}$$

其中 $(\boldsymbol{x}_c - \boldsymbol{x}_b)^\perp$ 表示该向量在平面上逆时针旋转 $90°$ 后的向量. 于是

$$\nabla\alpha(\boldsymbol{x}) = \frac{(\boldsymbol{x}_c - \boldsymbol{x}_b)^\perp}{2A_T}$$

于是

$$\nabla f(\boldsymbol{x}) = \frac{(\boldsymbol{x}_c - \boldsymbol{x}_b)^\perp}{2A_T} f(\boldsymbol{x}_a) + \frac{(\boldsymbol{x}_a - \boldsymbol{x}_c)^\perp}{2A_T} f(\boldsymbol{x}_b) + \frac{(\boldsymbol{x}_b - \boldsymbol{x}_a)^\perp}{2A_T} f(\boldsymbol{x}_c)$$

这表明 $\nabla f(\boldsymbol{x})$ 与 $\boldsymbol{x}$ 无关, 即在三角形内部为常值. □

离散拉普拉斯算子

定义 2.1 拉普拉斯算子 **拉普拉斯算子 (Laplace Operator, Laplacian)** 是一个二阶微分算子, 记号为 Δ 或 ∇^2, 定义为函数梯度的散度:

$$\Delta f = \operatorname{div}(\operatorname{grad} f)$$

在 n 维欧氏空间中, 对于一个标量函数 $f(x_1, \cdots, x_n)$, 拉普拉斯算子可以表示为:

$$\Delta = \sum_{i=1}^{n} \frac{\partial^2}{\partial x_i^2}$$

我们知道, 对于一维函数, 其二阶导数表示该函数的凹凸程度; 而 Laplacian 是多维函数在各个正交的方向上的二阶导数之和, 也就刻画了该函数在某一点整体的凹凸程度.

我们也可以用梯度和散度的几何意义理解 Laplacian. 梯度作为向量场衡量了函数在各点的变化方向, 再对其求散度就可以得到函数在该点附近梯度的情况. 如果梯度场流入某一点更多, 此时该点的 Laplacian 为正, 表明此处函数内凹 (类似于山谷); 反之, 如果梯度场流出某一点更多, 此时该点的 Laplacian 为负, 表明此处函

数外凸 (类似于山峰). 当梯度场流入和流出的量近似时, 该点的 Laplacian 接近零, 表明此处函数比较平坦[4](类似于山坡).

Laplacian 对于刻画曲面性质有重要作用. 例如, 我们可以用 Laplacian 计算由参数方程 $\boldsymbol{S}(u,v)$ 在三维空间中定义的曲面 S 的平均曲率[5]:

$$H = -\frac{\Delta \boldsymbol{S}}{2\boldsymbol{n}}$$

在离散情况下, 我们可以得到如下结论.

定义 2.2 离散拉普拉斯算子 在三角形网格中, 函数 f 在顶点 $\boldsymbol{x}_i$ 处的**离散拉普拉斯算子 (Discrete Laplace Operator)** 定义为:

$$\Delta f(\boldsymbol{x}_i) = \sum_{\boldsymbol{x}_j \in \Omega(\boldsymbol{x}_i)} w_{ij} \left(f(\boldsymbol{x}_j) - f(\boldsymbol{x}_i) \right)$$

其中 $\Omega(\boldsymbol{x}_i)$ 是与顶点 $\boldsymbol{x}_i$ 邻接的顶点集合, w_{ij} 是权重.

定义 2.3 均匀拉普拉斯算子 **均匀拉普拉斯算子 (Uniform Laplace Operator)** 的权重定义为:

$$w_{ij} = \frac{1}{|\Omega(\boldsymbol{x}_i)|}$$

其中 $|\Omega(\boldsymbol{x}_i)|$ 是邻接顶点的个数.

定义 2.4 余切拉普拉斯算子 **余切拉普拉斯算子 (Cotangent Laplace Operator)** 的权重定义为:

$$w_{ij} = \frac{1}{2} \left(\cot \alpha_{ij} + \cot \beta_{ij} \right)$$

其中 α_{ij} 和 β_{ij} 是与边 $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ 相邻的两个三角形中该边所对的角.

我们现在来证明上述定义 (尤其是余切 Laplacian) 的合理性[6].

证明. 我们首先推导散度定理在二维情形下 (即格林公式) 的向量表述[7]. 对于一个在区域 Ω 上定义的向量场 $\boldsymbol{v}(x,y) = (P(x,y), Q(x,y))$, 设 $\boldsymbol{n}(x,y) = (a,b)$ 为 $\partial\Omega$ 上 (x,y) 处的单位外法向量, 此处的单位切向量 $\boldsymbol{t}(x,y) = (\cos\alpha, \cos\beta)$. 于是有

$$\boldsymbol{k} = \boldsymbol{n} \times \boldsymbol{t} = \begin{vmatrix} \boldsymbol{i} & \boldsymbol{j} & \boldsymbol{k} \\ a & b & 0 \\ \cos\alpha & \cos\beta & 0 \end{vmatrix} = (a\cos\beta - b\cos\alpha)\,\boldsymbol{k}$$

[4]这里的平坦不是指函数值变化不大, 而是指函数值的变化速率变化不大, 类似于一次函数.

[5]对于一个曲面 S 上某一点 P, 过 P 的法向量做平面可以与 S 交出一条曲线 C. 改变平面的方向所得的 C 中在 P 处的曲率有极大值 k_1 和极小值 k_2, 称作曲面 S 在点 P 处的**主曲率** (Principal Curvature). 曲面 S 在 P 处的**平均曲率** (Mean Curvature) 定义为 $H = (k_1 + k_2)/2$. 以后也许还会用到**高斯曲率** (Gaussian Curvature), 定义为 $K = k_1 k_2$.

[6]依照下面的推导, 余切 Laplacian 的系数的归一化与邻域面积有关, 但实际实现时往往会直接进行归一化操作; 并且在计算得到的余切值显著较大时还需要进行修正.

[7]这在高等数学习题中亦有相应的习题, 此处再推导一遍.

其中 $\boldsymbol{i},\boldsymbol{j},\boldsymbol{k}$ 分别为 x,y,z 轴的单位向量. 解上述方程从而表明 $\boldsymbol{n}=(\cos\beta,-\cos\alpha)$. 于是就有

$$\oint_{(\partial\Omega)^+}\boldsymbol{v}\cdot\boldsymbol{n}\mathrm{d}s=\oint_{(\partial\Omega)^+}(P\cos\beta-Q\cos\alpha)\mathrm{d}s=\oint_{(\partial\Omega)^+}(P\mathrm{d}y-Q\mathrm{d}x)$$
$$\xlongequal{\text{格林公式}}\iint_\Omega\left(\frac{\partial P}{\partial x}+\frac{\partial Q}{\partial y}\right)\mathrm{d}S=\iint_\Omega\operatorname{div}\ \boldsymbol{v}\mathrm{d}S=\iint_\Omega(\nabla\cdot\boldsymbol{v})\mathrm{d}S$$

现在回到对于网格的讨论来. 对于顶点 $\boldsymbol{x}_i$ 的一个邻域 $\Omega(\boldsymbol{x}_i)$, 根据散度定理有

$$\iint_{\Omega(\boldsymbol{x}_i)}\Delta f\mathrm{d}S=\iint_{\Omega(\boldsymbol{x}_i)}\nabla(\nabla f)\mathrm{d}S=\int_{\partial\Omega(\boldsymbol{x}_i)}\nabla f\cdot\boldsymbol{n}\mathrm{d}s$$

其中 $\boldsymbol{n}$ 表示给定点处的法向量. 我们分别考虑边界[8]$\partial\Omega(\boldsymbol{x}_i)$ 在 $\boldsymbol{x}_i$ 邻接的各个三角形 T_m 中的一段 $\partial\Omega(\boldsymbol{x}_i)\cap T_m$, 设 T_m 的顶点为 $\boldsymbol{x}_i,\boldsymbol{x}_j,\boldsymbol{x}_k$.

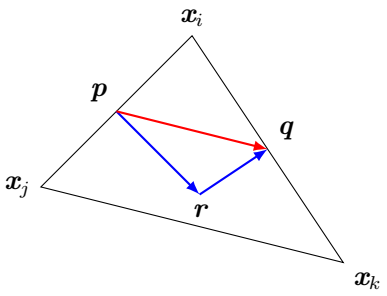


图 2.1: $\partial\Omega(\boldsymbol{x}_i)$ 在 T_m 中的一段 (图中的蓝色有向线段)

在梯度一节中我们已经证明 ∇f 在给定三角形上是定值, 于是依照上图有

$$\int_{\partial\Omega(\boldsymbol{x}_i)\cap T_m}\nabla f\cdot\boldsymbol{n}\mathrm{d}s=\nabla f\cdot\int_{\partial\Omega(\boldsymbol{x}_i)}\boldsymbol{n}\mathrm{d}s=\nabla f\cdot\left(\int_{PR}\boldsymbol{n}\mathrm{d}s+\int_{RQ}\boldsymbol{n}\mathrm{d}s\right)$$

而对于有向线段 PR 而言, 其方向向量为 $\boldsymbol{t}=\dfrac{\boldsymbol{r}-\boldsymbol{p}}{||\boldsymbol{r}-\boldsymbol{p}||}$, 于是

$$\int_{PR}\boldsymbol{n}\mathrm{d}s=\int_{PR}\left(-\boldsymbol{t}^\perp\right)\mathrm{d}s=\frac{(\boldsymbol{p}-\boldsymbol{r})^\perp}{||\boldsymbol{r}-\boldsymbol{p}||}\cdot\int_{PR}\mathrm{d}s=(\boldsymbol{p}-\boldsymbol{r})^\perp$$

同理有 $\displaystyle\int_{RQ}\boldsymbol{n}\mathrm{d}s=(\boldsymbol{r}-\boldsymbol{q})^\perp$. 于是

$$\int_{\partial\Omega(\boldsymbol{x}_i)\cap T_m}\nabla f\cdot\boldsymbol{n}\mathrm{d}s=\nabla f\cdot\left((\boldsymbol{p}-\boldsymbol{r})^\perp+(\boldsymbol{r}-\boldsymbol{q})^\perp\right)=\nabla f\cdot(\boldsymbol{p}-\boldsymbol{q})^\perp$$

按照**局部平均区域**一节中所述的 $\Omega(\boldsymbol{x}_i)$ 的取法,$\boldsymbol{p}$ 和 $\boldsymbol{q}$ 分别为各自边的中点. 于是

$$\int_{\partial\Omega(\boldsymbol{x}_i)\cap T_m}\nabla f\cdot\boldsymbol{n}\mathrm{d}s=\frac{1}{2}\nabla f\cdot(\boldsymbol{x}_j-\boldsymbol{x}_k)^\perp$$

现在再将[9]上一节中推导的梯度公式变形可得

$$\nabla f=(f_j-f_i)\frac{(\boldsymbol{x}_i-\boldsymbol{x}_k)^\perp}{2A_m}+(f_k-f_i)\frac{(\boldsymbol{x}_j-\boldsymbol{x}_i)^\perp}{2A_m}$$

[8]这里默认取正向边界方向. 后文的顶点选取的顺序也依照正向边界的方向.

[9]为了简洁考虑, 记 f 在 $\boldsymbol{x}_i$ 处的值为 f_i, 其余类似

代入上式可得

$$\begin{aligned}\int_{\partial\Omega(\boldsymbol{x}_i)\cap T_m}\nabla f\cdot\boldsymbol{n}\mathrm{d}s &= \frac{1}{4A_m}\left[(f_j-f_i)(\boldsymbol{x}_i-\boldsymbol{x}_k)^{\perp}(\boldsymbol{x}_j-\boldsymbol{x}_k)^{\perp}+(f_k-f_i)(\boldsymbol{x}_j-\boldsymbol{x}_i)^{\perp}(\boldsymbol{x}_j-\boldsymbol{x}_k)^{\perp}\right]\\ &= \frac{1}{4A_m}\left[(f_j-f_i)(\boldsymbol{x}_i-\boldsymbol{x}_k)\cdot(\boldsymbol{x}_j-\boldsymbol{x}_k)+(f_k-f_i)(\boldsymbol{x}_i-\boldsymbol{x}_j)\cdot(\boldsymbol{x}_k-\boldsymbol{x}_j)\right]\end{aligned}$$

上面的式子中恰好存在边的点乘与面积之比, 因此考虑用三角函数表示. 事实上, 对于任意的 $\triangle ABC$ 总有

$$\frac{\overrightarrow{AB}\cdot\overrightarrow{AC}}{A_{\triangle ABC}} = \frac{\overline{AB}\cdot\overline{AC}\cdot\cos A}{\frac{1}{2}\overline{AB}\cdot\overline{AC}\cdot\sin A} = 2\cot A$$

于是把这一结论运用到上式即可得

$$\int_{\partial\Omega(\boldsymbol{x}_i)\cap T_m}\nabla f\cdot\boldsymbol{n}\mathrm{d}s = \frac{1}{2}\left[(f_j-f_i)\cot\gamma_k+(f_k-f_i)\cot\gamma_j\right]$$

其中 γ_j 和 γ_k 分别是顶点 $\boldsymbol{x}_j$ 和 $\boldsymbol{x}_k$ 处的角. 现在, 我们将 $\partial\Omega(\boldsymbol{x}_i)$ 在所有邻接三角形中的部分累加起来, 即可得

$$\int_{\partial\Omega(\boldsymbol{x}_i)}\nabla f\cdot\boldsymbol{n}\mathrm{d}s = \frac{1}{2}\sum_{j\in\Omega(i)}(\cot\alpha_{ij}+\cot\beta_{ij})(f_j-f_i)$$

其中 α_{ij} 和 β_{ij} 是与边 $(\boldsymbol{x}_i,\boldsymbol{x}_j)$ 相邻的两个三角形中该边所对的角. 最后, 我们只需要将上式除以邻域 $\Omega(\boldsymbol{x}_i)$ 的面积 $A(\boldsymbol{x}_i)$ 即可得到余切 Laplacian 的定义, 即

$$\Delta f(x_i) = \frac{1}{2A(\boldsymbol{x}_i)}\sum_{j\in\Omega(i)}(\cot\alpha_{ij}+\cot\beta_{ij})(f_j-f_i)$$

在实际应用中, 由于 $A(\boldsymbol{x}_i)$ 是定值, 因此我们通常令权重 w_{ij} 为

$$w_{ij} = \frac{1}{2}(\cot\alpha_{ij}+\cot\beta_{ij})$$

后进行归一化处理.

对权值进行简化, 我们可以得到均匀 Laplacian 的定义. 均匀 Laplacian 的计算量较小, 但没有恰当的几何意义, 在很多时候也经常导致问题. □

2.2.2 网格平滑

随着三维扫描和曲面重建技术的发展, 得到实体表面的多边形网格表示已经不是难事, 但所得到的表面往往包含噪声. 为了将其转化为更光滑的网格以便后续处理, 我们需要用到**网格平滑**技术.

定义 2.5 网格平滑 **网格平滑 (Mesh Smoothing)**(又称为**网格降噪 (Mesh Deinoising)** 通过) 对多边形网格进行处理, 以减少噪声, 从而得到更光滑, 性质更优良的网格表示.

一般而言, 噪声所在的位置都是曲率较大, 形状比较尖锐的地方. 因此, 网格平滑的目标是减少这些高频噪声, 同时尽可能地保留网格的整体形状和细节.

我们在本节主要介绍**拉普拉斯平滑 (Laplacian Smoothing)**.

我们用扩散流的数学模型来描述网格平滑的过程. 考虑热量传导的物理模型, 设 $f(\boldsymbol{x},t)$ 表示时间 t 时刻位置 $\boldsymbol{x}$ 处的温度, 则扩散方程为:

$$\frac{\partial f(\boldsymbol{x},t)}{\partial t}=\lambda\Delta f(\boldsymbol{x},t)$$

在网格中, 高频的噪声就对应于杂乱分布的高温点. 随着扩散的进行, 热量分布趋于均匀, 对应顶点构成的表面也趋于平滑.

现在, 我们需要将扩散流公式在时间和空间上进行离散化, 以便在计算机中处理. 在空间上的 Laplacian 离散化已经在前面一节中介绍, 即

$$\frac{\partial f\left(\boldsymbol{x}_i,t\right)}{\partial t}=\lambda\Delta f\left(\boldsymbol{x}_i,t\right)=\lambda\sum_{\boldsymbol{x}_j\in\Omega(\boldsymbol{x}_i)}w_{ij}\left(f\left(\boldsymbol{x}_j,t\right)-f\left(\boldsymbol{x}_i,t\right)\right)$$

而在时间上的离散化, 可以将 t 划分为步长为 h 的若干个间隔, 用有限差分的思想对微分方程进行近似:

$$\frac{\partial f\left(\boldsymbol{x}_i,t\right)}{\partial t}=\frac{f\left(\boldsymbol{x}_i,t+h\right)-f\left(\boldsymbol{x}_i,t\right)}{h}$$

于是

$$f\left(\boldsymbol{x}_i,t+h\right)=f\left(\boldsymbol{x}_i,t\right)+h\frac{\partial f\left(\boldsymbol{x}_i,t\right)}{\partial t}=f\left(\boldsymbol{x}_i,t\right)+h\lambda\sum_{\boldsymbol{x}_j\in\Omega(\boldsymbol{x}_i)}w_{ij}\left(f\left(\boldsymbol{x}_j,t\right)-f\left(\boldsymbol{x}_i,t\right)\right)$$

在网格简化中, 我们考虑的 “温度” 就是顶点的空间坐标. 于是, 在每次时间间隔为 h 的第 k 次迭代中, 每个顶点 $\boldsymbol{x}_i$ 的位置按照如下的方式更新:

$$\boldsymbol{x}_i^{(k+1)}\leftarrow\boldsymbol{x}_i^{(k)}+\lambda\Delta\boldsymbol{x}_i^{(k)}$$

其中 λ 为**平滑参数**即为前面推导中的 $h\lambda$.

在两种 Laplacian 下, 上述迭代方法可以得到不同的平滑效果. 均匀 Laplacian 下, 每个顶点都向其邻居的均值移动, 这种方法简单且计算效率高, 但容易导致网格收缩和细节丢失. 余切 Laplacian 下, 每个顶点向平均曲率的方向移动, 也被称作**平均曲率流 (Mean Curvature Flow)**. 这种方法能够更好地保留网格的形状和细节, 但是计算复杂度较高.

2.2.3 网格编辑

定义 2.6 网格编辑 **网格编辑 (Mesh Editing)** 是操纵和修改网格表面的几何形状, 同时能够保留原始网格几何细节的操作.

回顾我们对二维图像进行 Poisson 编辑的过程, 这可以描述为下面的最小化问题:

$$\arg\min_f\iint_\Omega\left|\left|\nabla f-\nabla g\right|\right|^2\mathrm{d}S,\quad\text{s.t.}\quad f|_{\partial\Omega}=f^*|_{\partial\Omega}$$

类似的思想在三维情形下的应用就是 Laplacian 网格编辑. 它通过尽可能使得编辑后网格上的 Laplace 微分坐标与原始网格上的 Laplace 微分坐标保持一致保留网格的细节. Laplace 微分坐标在本质上就是对网格顶点施加均匀 Laplacian 的结果, 即

$$\mathcal{L}(\boldsymbol{v}_i) = \boldsymbol{v}_i - \frac{1}{N_i}\sum_{j\in\Omega(i)}\boldsymbol{v}_j$$

2.2.4 网格简化

在图形学中, 渲染效率与渲染质量需要兼顾. 同样的物体在近处时需要使用精度较高, 面数较多的模型以保证真实; 而在远处时由于物体在屏幕上占据的像素较少, 使用精度较低, 面数较少的模型也能保证视觉效果. 因此, 我们需要对高精度的网格进行简化以提高渲染效率. 这时就需要用到**网格简化**算法来生成不同**细节层次 (Level of Detail, LoD)** 的模型用于不同情况下的渲染.

定义 2.7 网格简化 (Mesh Simplification) 通过对多边形网格进行处理, 以减少其面数, 从而得到更简洁, 渲染效率更高的网格表示.

网格简化一般可以通过移除顶点或坍缩边实现, 实际应用中后者更容易实现, 也更常用. 边坍缩算法的核心是设计方法找出哪些边的坍缩对网格形状的影响最小, 以及确定坍缩后新顶点的位置, 从而在简化时尽量保留网格的整体形状.

我们在本节主要介绍**二次误差度量 (Quadric Error Metrics, QEM)** 方法[10]. 对于一次边坍缩, 考虑将 $\boldsymbol{v}_1, \boldsymbol{v}_2$ 收缩为 $\boldsymbol{v}$. 为了度量收缩操作与原网格的差异, 我们可以用简化后的点与原网格中 $\boldsymbol{v}_1, \boldsymbol{v}_2$ 邻接的所有面的距离之和来表示.

定义 $\text{plain}(\boldsymbol{v}_i)$ 表示所有与顶点 $\boldsymbol{v}_i$ 邻接的面的集合, 其中的元素 $\boldsymbol{p} = \begin{bmatrix} a & b & c & d \end{bmatrix}^{\mathrm{t}}$ 表示对应平面的方程为 $ax + by + cz + d = 0$, 并且 $a^2 + b^2 + c^2 = 1$. 于是优化目标为:

$$\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \sum_{\boldsymbol{p}\in\text{plain}(\boldsymbol{v}_1)\cup\text{plain}(\boldsymbol{v}_2)} d_{\boldsymbol{v},\boldsymbol{p}}^2$$

在这里, 为了计算方便, 我们将 $\boldsymbol{v}$ 扩展为具有四个分量的向量 $\boldsymbol{v} = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^{\mathrm{t}}$, 其中前三个分量表示 $\boldsymbol{v}$ 的位置. 于是点 $\boldsymbol{v}$ 到平面 $\boldsymbol{p}$ 的距离为:

$$d_{\boldsymbol{v},\boldsymbol{p}}^2 = \left(\boldsymbol{v}^{\mathrm{t}}\boldsymbol{p}\right)^2 = \boldsymbol{v}^{\mathrm{t}}\boldsymbol{p}\boldsymbol{p}^{\mathrm{t}}\boldsymbol{v}$$

令 $\boldsymbol{K}_{\boldsymbol{p}} = \boldsymbol{p}\boldsymbol{p}^{\mathrm{t}}$, 则有

$$\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \boldsymbol{v}^{\mathrm{t}} \left(\sum_{\boldsymbol{p}\in\text{plain}(\boldsymbol{v}_1)\cup\text{plain}(\boldsymbol{v}_2)} \boldsymbol{K}_{\boldsymbol{p}} \right) \boldsymbol{v}$$

为了简化计算, 我们将上式近似为

$$\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \boldsymbol{v}^{\mathrm{t}} \left(\sum_{\boldsymbol{p}\in\text{plain}(\boldsymbol{v}_1)} \boldsymbol{K}_{\boldsymbol{p}} + \sum_{\boldsymbol{p}\in\text{plain}(\boldsymbol{v}_2)} \boldsymbol{K}_{\boldsymbol{p}} \right) \boldsymbol{v}$$

[10] Garland M, Heckbert PS. Surface simplification using quadric error metrics. In: Proceedings of the 24th annual conference on computer graphics and interactive techniques. 1997, p. 209-16.

实际上至多只会重复以 $\boldsymbol{v}_1, \boldsymbol{v}_2$ 为顶点的两个三角形面, 但将误差转化为仅与顶点相关的信息, 因而简化了计算. 令 $\boldsymbol{Q}_i = \sum_{\boldsymbol{p}\in\mathrm{plain}(\boldsymbol{v}_i)} \boldsymbol{K}_{\boldsymbol{p}}$, 于是就有

$$\boldsymbol{v}^* = \arg\min_{\boldsymbol{v}} \boldsymbol{v}^{\mathrm{t}} \left(\boldsymbol{Q}_1 + \boldsymbol{Q}_2\right) \boldsymbol{v}$$

现在来考虑这一二次型的最小值何时取到. 令 $\boldsymbol{Q} = \boldsymbol{Q}_1 + \boldsymbol{Q}_2$, 将上述式子展开可得

$$\boldsymbol{v}^{\mathrm{t}} \boldsymbol{Q} \boldsymbol{v} = q_{11}x^2 + 2q_{12}xy + 2q_{13}xz + 2q_{14}x + q_{22}y^2 + 2q_{23}yz + 2q_{24}y + q_{33}z^2 + 2q_{34}z + q_{44}$$

并且要求

$$\frac{\partial \boldsymbol{v}^{\mathrm{t}} \boldsymbol{Q} \boldsymbol{v}}{\partial x} = \frac{\partial \boldsymbol{v}^{\mathrm{t}} \boldsymbol{Q} \boldsymbol{v}}{\partial y} = \frac{\partial \boldsymbol{v}^{\mathrm{t}} \boldsymbol{Q} \boldsymbol{v}}{\partial z} = 0$$

于是可以得到如下线性方程组:

$$\begin{bmatrix} q_{11} & q_{12} & q_{13} & q_{14} \\ q_{21} & q_{22} & q_{23} & q_{24} \\ q_{31} & q_{32} & q_{33} & q_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

令 $\boldsymbol{Q}'$ 为前面的矩阵, 则有 $\boldsymbol{Q}'\boldsymbol{v}^* = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^{\mathrm{t}}$. 如果 $\boldsymbol{Q}'$ 可逆, 就有

$$\boldsymbol{v}^* = \left(\boldsymbol{Q}'\right)^{-1} \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^{\mathrm{t}}$$

如果 $\boldsymbol{Q}'$ 不可逆, 这就说明我们无法找出使误差最小的 $\boldsymbol{v}$. 此时, 我们可以简单地选择 $\boldsymbol{v}_1, \boldsymbol{v}_2$ 以及它们的中点 $\dfrac{\boldsymbol{v}_1 + \boldsymbol{v}_2}{2}$ 中使得误差 $\boldsymbol{v}^{\mathrm{t}} \boldsymbol{Q} \boldsymbol{v}$ 最小的点作为优化的结果 $\boldsymbol{v}^*$. 需要注意的是, 这里的 $\boldsymbol{Q}'$ 仅用于求解 $\boldsymbol{v}^*$ 使用. 计算误差仍然使用 $\boldsymbol{Q}$.

于是, QEM 算法的基本流程为:

1. 初始化每个顶点的误差矩阵 $\boldsymbol{Q}_i$.
2. 对每组合法的 (可收缩而不改变网格拓扑结构的) 点对 $(\boldsymbol{v}_1, \boldsymbol{v}_2)$, 计算其误差矩阵 $\boldsymbol{Q} = \boldsymbol{Q}_1 + \boldsymbol{Q}_2$ 以及最优收缩点 $\boldsymbol{v}^*$ 和误差 $\mathrm{err}(\boldsymbol{v}^*) = \left(\boldsymbol{v}^*\right)^{\mathrm{t}} \boldsymbol{Q} \boldsymbol{v}^*$.
3. 将所有点对按误差从小到大存入优先队列中.
4. 重复以下操作直到达到预定的顶点数:
   a. 从优先队列中取出误差最小的点对 $(\boldsymbol{v}_1, \boldsymbol{v}_2)$, 并将其坍缩为 $\boldsymbol{v}^*$, 在网格结构中实现这一操作.
   b. 考虑所有坍缩前与 $\boldsymbol{v}_1, \boldsymbol{v}_2$ 相邻的顶点 $\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n$. 首先, 由于 $\boldsymbol{u}_i$ 相邻的点已经更新为 $\boldsymbol{v}^*$, 因此其相邻的面已经发生改变, 因此需要重新计算 $\boldsymbol{u}_i$ 的误差矩阵. 然后考虑所有与 $\boldsymbol{u}_i$ 相邻的顶点 $\boldsymbol{w}_{i1}, \cdots, \boldsymbol{w}_{im}$, 对于顶点对 $(\boldsymbol{u}_i, \boldsymbol{w}_{ij})$, 由于 $\boldsymbol{u}_i$ 的误差矩阵发生变化, 因此需要重新判断点对是否合法, 并重新计算收缩点和误差, 最后将其更新到优先队列中.

2.3 几何重建

2.3.1 几何重建概述

现实世界中的三维物体和场景无法直接以数学模型或几何形式被计算机存储或处理. 大多数时候, 我们只能通过图像, 视频或点云数据间接地获取三维信息, 这些数据本身离散而稀疏, 带有噪声, 也没有拓扑结构. 为了用计算机处理形状, 空间结构和几何关系, 必须将这些数据重建为连续的几何模型.

定义 3.1 几何重建 几何重建 (Geometry Reconstruction) 从离散的三维数据 (如点云, 深度图像或多视图图像) 中, 重建出物体或场景的连续几何形状与结构的过程.

2.3.2 点云的配准/注册

在几何重建中, 我们往往会从不同角度和位置对同一场景或物体分别采集点云数据. 将不同视角下的点云对齐到同一坐标系下, 才能得到完整的重建数据.

定义 3.2 点云配准/注册 点云配准/注册 (Point Cloud Registration) 是将来自不同视角或传感器的多个点云数据集对齐到同一坐标系下的过程.

现在我们来介绍一种经典的点云配准算法, 即 ICP(Iterative Closest Point) 算法.

点云配准的 ICP 算法

我们在不同角度测得的点云总是保长度的, 这意味着描述同一物体的点云之间总是可以通过平移和旋转变换对齐.

给定待变换点云 S 和目标点云 T, ICP 算法的目标是找到一组最佳的旋转和平移变换 $\boldsymbol{R}, \boldsymbol{t}$ 使得 S 经变换后最大程度上地与 T 对其. ICP 算法的具体流程如下:

1. 通过主成分分析 (PCA, Principal Component Analysis) 初始化一组变换 $\boldsymbol{R}, \boldsymbol{t}$ 作为估计变换的初值.

2. 将当前求得的变换 $\boldsymbol{R}, \boldsymbol{t}$ 应用于 S, 对于 S 中的各点 $\boldsymbol{p}_i$ 变换后得到 $\boldsymbol{p}_i' = \boldsymbol{R}\boldsymbol{p}_i + \boldsymbol{t}$. 然后找到 T 中与 $\boldsymbol{p}_i'$ 中最接近的点 $\boldsymbol{q}_i$, 如此构成 S' 与 T 的一一对应. 同时, 舍去距离过远的点对.

3. 在余下的 N 组点对 $\{(\boldsymbol{p}_i', \boldsymbol{q}_i) : i = 1, \cdots, N\}$ 中, 构造累计误差函数

$$E = \sum_{i=1}^{N} ||\boldsymbol{p}_i' - \boldsymbol{q}_i||^2 = \sum_{i=1}^{N} ||\boldsymbol{R}\boldsymbol{p}_i + \boldsymbol{t} - \boldsymbol{q}_i||^2$$

注意这里的 $\boldsymbol{R}, \boldsymbol{t}$ 是需要求解的变量, 与 **2.** 中前一步迭代求得的 $\boldsymbol{R}, \boldsymbol{t}$ 是不同的. 前一步求得的 $\boldsymbol{R}, \boldsymbol{t}$ 只用于对应关系的确定, 不直接参与后续的最小化计算.

然后, 通过奇异值分解 (SVD, Singular value decomposition) 求解最小化问题:

$$(\boldsymbol{R}, \boldsymbol{t}) = \arg\min_{\boldsymbol{R}, \boldsymbol{t}} E$$

4. 重复步骤 2 和 3, 直到累计误差 E 小于预设阈值或达到最大迭代次数.

现在我们具体地介绍上述算法中重要的两步, 即 PCA 和 SVD.

主成分分析

定义 3.3 主成分分析 **主成分分析 (Principal Component Analysis, PCA)** 是一种统计方法, 用于通过线性变换将数据从高维空间映射到低维空间, 以保留数据的主要特征和结构.

一般而言, 点云的分布在空间中是各向异性的, 即在某些方向上点云的变化更显著. 通过 PCA 可以找到点云数据的主要方向 (即主轴), 从而为 ICP 算法提供一个合理的初始变换估计. 现在给出 PCA 的具体步骤.

首先, 考虑给定的一组点[11] $\boldsymbol{p}_1, \cdots, \boldsymbol{p}_N$ 以及中心坐标

$$\bar{\boldsymbol{p}} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{p}_i$$

首先, 为了防止平移对 PCA 结果的影响, 我们需要将点云数据进行中心化处理, 即将每个点减去中心坐标:

$$\boldsymbol{p}_i' = \boldsymbol{p}_i - \bar{\boldsymbol{p}}, \quad i = 1, \cdots, N$$

构造矩阵

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{p}_1' \\ \boldsymbol{p}_2' \\ \vdots \\ \boldsymbol{p}_N' \end{bmatrix}$$

然后计算 $\boldsymbol{P}$ 的协方差矩阵

$$\boldsymbol{\Sigma}_{\boldsymbol{P}} = \boldsymbol{P}\boldsymbol{P}^{\mathrm{t}}$$

二维情况下的协方差矩阵的两个特征向量 $\boldsymbol{v}_1, \boldsymbol{v}_2$ 表示数据的两条轴线. 对应于更大的特征值的特征向量 $\boldsymbol{v}_1$ 表示数据变化更显著的方向, 即主成分方向, 而 $\boldsymbol{v}_2$ 则表示数据变化最小的方向[12]. 对于三维情况也是同理.

对协方差矩阵 $\boldsymbol{\Sigma}_{\boldsymbol{P}}$ 进行特征值分解, 可得:

$$\boldsymbol{\Sigma}_{\boldsymbol{P}} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^{\mathrm{t}}$$

其中矩阵 $\boldsymbol{V}$ 的列向量即为 $\boldsymbol{\Sigma}_{\boldsymbol{P}}$ 的特征向量, 它们也就确定了点云的主轴方向.

现在, 对给定的两个点云 S 和 T, 我们分别对它们进行 PCA, 得到各自的中心 $\bar{\boldsymbol{p}}, \bar{\boldsymbol{q}}$ 以及主轴 $\boldsymbol{v}_{s1}, \boldsymbol{v}_{s2}, \boldsymbol{v}_{s3}$ 和 $\boldsymbol{v}_{t1}, \boldsymbol{v}_{t2}, \boldsymbol{v}_{t3}$. 既然变换仅由平移和旋转构成, 我们可以认为 S 和 T 的主轴应当近似地重合. 于是, 先通过平移使得两个点云的中心重合, 即

$$\boldsymbol{t} = \bar{\boldsymbol{q}} - \bar{\boldsymbol{p}}$$

[11] 每个点都是一个三维向量, 表示该点在三维空间中的位置.

[12] 例如, 如果数据点沿一条直线附近随机分布, 那么 $\boldsymbol{v}_1$ 就是该直线的法向量,$\boldsymbol{v}_2$ 是该直线的方向向量

然后, 通过旋转使得两个点云的主轴方向对齐. 设旋转矩阵为 $\boldsymbol{R}$, 则有

$$\boldsymbol{R}\boldsymbol{v}_{si} = \boldsymbol{v}_{ti}, \quad i = 1, 2, 3$$

这样求得的 $\boldsymbol{R}, \boldsymbol{t}$ 即可作为 ICP 算法的初始变换估计.

奇异值分解

对于前述的最小化问题, 事实上是具有唯一解的. 我们可以用奇异值分解来求解.

注意到 E 中含有两个变量 $\boldsymbol{R}, \boldsymbol{t}$. 我们先求解使得 E 最小的 $\boldsymbol{t}$, 然后再求解 $\boldsymbol{R}$. 这是为了对齐质心后才能进行旋转矩阵的求解.

设 S 和 T 的中心分别为 $\bar{\boldsymbol{p}}, \bar{\boldsymbol{q}}$. 令

$$\tilde{\boldsymbol{p}}_i = \boldsymbol{p}_i - \bar{\boldsymbol{p}}, \quad \tilde{\boldsymbol{q}}_i = \boldsymbol{q}_i - \bar{\boldsymbol{q}}$$

于是

$$E = \sum_{i=1}^{N} ||\boldsymbol{R}\boldsymbol{p}_i + \boldsymbol{t} - \boldsymbol{q}_i||^2 = \sum_{i=1}^{N} ||\boldsymbol{R}\tilde{\boldsymbol{p}}_i - \tilde{\boldsymbol{q}}_i + (\boldsymbol{R}\bar{\boldsymbol{p}} + \boldsymbol{t} - \bar{\boldsymbol{q}})||^2$$

关于 $\boldsymbol{R}\bar{\boldsymbol{p}} + \boldsymbol{t} - \bar{\boldsymbol{q}}$ 展开后对 $\boldsymbol{t}$ 求导并令其为 0, 可得 (实际上直接观察也容易得到)

$$\boldsymbol{R}\bar{\boldsymbol{p}} + \boldsymbol{t} - \bar{\boldsymbol{q}} = \boldsymbol{0}$$

于是

$$\boldsymbol{t} = \bar{\boldsymbol{q}} - \boldsymbol{R}\bar{\boldsymbol{p}}$$

注意这里的 $\boldsymbol{t}$ 是关于 $\boldsymbol{R}$ 的函数, 因此求解 $\boldsymbol{R}$ 完毕后需要重新进行带入. 现在把 $\boldsymbol{t}$ 代入 E, 可得

$$E = \sum_{i=1}^{N} ||\boldsymbol{R}\tilde{\boldsymbol{p}}_i - \tilde{\boldsymbol{q}}_i||^2$$

上面的式子中已经没有 $\boldsymbol{t}$, 即已经完成对齐质心的工作. 我们只需要最小化 E 关于 $\boldsymbol{R}$ 的部分. 将平方展开, 并且注意到 $||\boldsymbol{R}\tilde{\boldsymbol{p}}_i|| = ||\tilde{\boldsymbol{p}}_i||$(旋转显然不改变向量的模长), 于是可得

$$E = \sum_{i=1}^{N} \left(||\tilde{\boldsymbol{p}}_i||^2 + ||\tilde{\boldsymbol{q}}_i||^2 \right) - 2\sum_{i=1}^{N} \tilde{\boldsymbol{q}}_i^{\mathrm{t}} \boldsymbol{R}\tilde{\boldsymbol{p}}_i$$

注意到上式中只有最后一项与 $\boldsymbol{R}$ 有关, 因此等价于最大化

$$\boldsymbol{R}^* = \arg\max_{\boldsymbol{R}} \sum_{i=1}^{N} \tilde{\boldsymbol{q}}_i^{\mathrm{t}} \boldsymbol{R}\tilde{\boldsymbol{p}}_i$$

自然地, 我们会想到标量与矩阵的迹之间的关联. 由于每个求和项都是向量的内积, 可以视作 1×1 的矩阵, 于是自然可以将它写做迹的形式:

$$\tilde{\boldsymbol{q}}_i^{\mathrm{t}} \boldsymbol{R}\tilde{\boldsymbol{p}}_i = \mathrm{trace}\left(\tilde{\boldsymbol{q}}_i^{\mathrm{t}} \boldsymbol{R}\tilde{\boldsymbol{p}}_i \right) = \mathrm{trace}\left(\tilde{\boldsymbol{p}}_i \tilde{\boldsymbol{q}}_i^{\mathrm{t}} \boldsymbol{R} \right)$$

于是求和后就有

$$\sum_{i=1}^{N} \tilde{\boldsymbol{q}}_i^{\mathrm{t}} \boldsymbol{R} \tilde{\boldsymbol{p}}_i = \operatorname{trace}\left(\sum_{i=1}^{N} \tilde{\boldsymbol{p}}_i \tilde{\boldsymbol{q}}_i^{\mathrm{t}} \boldsymbol{R}\right) = \operatorname{trace}\left(\boldsymbol{R} \sum_{i=1}^{N} \tilde{\boldsymbol{p}}_i \tilde{\boldsymbol{q}}_i^{\mathrm{t}}\right)$$

为了写成矩阵形式, 定义

$$\boldsymbol{M} = \sum_{i=1}^{N} \tilde{\boldsymbol{p}}_i \tilde{\boldsymbol{q}}_i^{\mathrm{t}}$$

这是一个 3×3 的矩阵, 对其进行奇异值分解, 可得

$$\boldsymbol{M} = \boldsymbol{U} \boldsymbol{\Sigma} \boldsymbol{V}^{\mathrm{t}}$$

其中 $\boldsymbol{U}, \boldsymbol{V}$ 是正交矩阵, $\boldsymbol{\Sigma}$ 是对角矩阵, 其对角线元素为非负的奇异值. 于是

$$\operatorname{trace}\left(\boldsymbol{R} \sum_{i=1}^{N} \tilde{\boldsymbol{p}}_i \tilde{\boldsymbol{q}}_i^{\mathrm{t}}\right) = \operatorname{trace}(\boldsymbol{R}\boldsymbol{M}) = \operatorname{trace}\left(\boldsymbol{R}\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{t}}\right) = \operatorname{trace}\left(\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{t}}\boldsymbol{R}\boldsymbol{U}\right)$$

由于 $\boldsymbol{R}$ 是旋转矩阵, 因此它自然也是正交矩阵. 于是设 $\boldsymbol{X} = \boldsymbol{V}^{\mathrm{t}}\boldsymbol{R}\boldsymbol{U}$, 则 $\boldsymbol{X}$ 也是正交矩阵. 于是

$$\operatorname{trace}\left(\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{t}}\boldsymbol{R}\boldsymbol{U}\right) = \operatorname{trace}(\boldsymbol{\Sigma}\boldsymbol{X}) = \sigma_1 x_{11} + \sigma_2 x_{22} + \sigma_3 x_{33}$$

其中 σ_i 为 $\boldsymbol{M}$ 的奇异值 (即 $\boldsymbol{\Sigma}$ 的对角线元素), x_{ii} 为 $\boldsymbol{X}$ 的对角线元素. 由于 $\boldsymbol{X}$ 是正交矩阵, 因此其元素的绝对值均不大于 1, 即 $|x_{ii}| \leqslant 1$. 为了最大化上式, 显然需要 $x_{ii} = 1$. 这就要求 $\boldsymbol{X} = \boldsymbol{I}$, 即

$$\boldsymbol{V}^{\mathrm{t}}\boldsymbol{R}\boldsymbol{U} = \boldsymbol{I} \Rightarrow \boldsymbol{R} = \boldsymbol{V}\boldsymbol{U}^{\mathrm{t}}$$

最后, 还需确保 $\boldsymbol{R}$ 是合法的旋转矩阵, 即要求 $\det \boldsymbol{R} = 1$. 如果 $\det \boldsymbol{R} = -1$, 这样的 $\boldsymbol{R}$ 表示了一个反射变换. 这里直接给出修正后的结果 (这涉及到一些线性代数知识):

$$\boldsymbol{R}^* = \boldsymbol{V} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \det \boldsymbol{V}\boldsymbol{U}^{\mathrm{t}} \end{bmatrix} = \boldsymbol{U}^{\mathrm{t}}$$

当然, 在课程中并没有讲到这种情况, 因此有

$$\boldsymbol{R}^* = \boldsymbol{V}\boldsymbol{U}^{\mathrm{t}}$$

回代求出 $\boldsymbol{t}^*$, 有

$$\boldsymbol{t}^* = \bar{\boldsymbol{q}} - \boldsymbol{R}^* \bar{\boldsymbol{p}}$$

这样得到的 $\boldsymbol{R}^*, \boldsymbol{t}^*$ 即可作为下一轮迭代用于确定对应关系的变换.

2.3.3 点云的表面重建

Voronoi 图

对于给定的点云 S, 一种自然地划分空间的方式是将空间中每个点分配给距离其最近的点云中的点.

定义 3.4 Voronoi 图 Voronoi 图 (Voronoi Diagram) 是根据点集 P 将空间划分为若干个区域的几何结构, 其中每个区域 Ω_i 对应于 P 中一个给定的顶点 $\boldsymbol{v}_i$, 使得 Ω_i 内的所有点都比 P 中其他顶点 $\boldsymbol{v}_j$ 更接近该顶点 $\boldsymbol{v}_i$.

事实上, 在二维情形中的 Voronoi 图正是每个点与它临近的点的垂直平分线所构成的图形. 尽管如此的定义很简单, 但在实现上却仍比较复杂. 我们可以用另外一种方式间接地实现 Voronoi 图的构造.

Delaunay 三角剖分

定义 3.5 Delaunay 三角剖分 Delaunay 三角剖分 (Delaunay Triangulation) 是点集 $\mathcal{P}$ 的剖分, 它将 $\mathcal{P}$ 连接成三角形网格, 使得每个三角形的外接圆内不包含其他点.

Delaunay 三角剖分是 Voronoi 图的对偶图.Voronoi 图的每个顶点对应 Delaunay 三角剖分中的一个三角形的外心, Delaunay 三角剖分中的每个顶点即为 Voronoi 图中的一个区域对应的点.

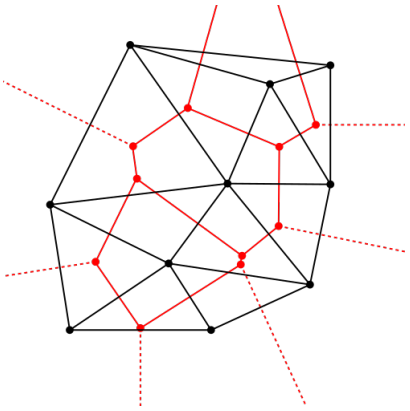


图 2.2: Delaunay 三角剖分与 Voronoi 图的关系

可以证明 Delaunay 三角剖分是使得所有三角形中最小内角最大化的剖分方式, 因而尽可能避免了细长三角形的出现. 结合三角形外接圆的形式, 我们可以得出: 在 Delaunay 三角剖分中, 任一被两个三角形共用的边所对的两个角之和小于 180°(否则这四点将落在同一圆内, 这是与定义矛盾的.). 然而, 如果将这一条边所在的四边形替换为由另一条对角线所构成的两个三角形, 则这两个新三角形就符合剖分性质了. 据此, 我们可以定义 Delaunay 三角剖分中最基本的操作——边翻转 (Edge Flip), 如下图所示:

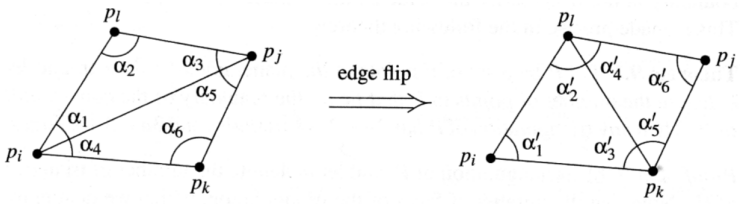


图 2.3: 边翻转操作

于是, 这样的思路产生了一个非常简单的算法: 任一构造一个三角剖分, 检查所有被共用的边, 如果不满足上述条件就进行边翻转操作, 直到所有边都满足条件为止.

遗憾的是这种算法效率并不高. 我们可以采取增量式算法, 每次向已有的三角剖分中加入一个点, 并检查受影响的三角形和进行必要的边翻转操作. 效率最高的算法是分治算法, 通过每次将点集划分为两个子集, 递归地对两个子集进行 Delaunay 三角剖分, 然后将两个子集的剖分结果合并起来. 运用一些巧妙的技巧可以使得合并操作的复杂度为 $O(n)$, 于是算法的复杂度为 $O(n\log n)$, 并可以进一步优化为 $O(n\log\log n)$.

Poisson 表面重建

与 Delaunay 三角剖分这种由点云直接构造三角网格的方法相比, Poisson 表面重建则采取了相对间接的方法, 先构建符号距离函数 (Signed Distance Function, SDF), 然后通过提取等值面来获得表面模型. 这可以避免对点云直接建模时可能出现的孔洞和噪声问题.

定义 3.6 Poisson 表面重建 **Poisson 表面重建 (Poisson Surface Reconstruction)** 是一种基于点云法线信息, 通过求解 Poisson 方程来重建连续表面模型的方法.

具体而言[13][14], 对于空间中的几何体 M, 定义函数 χ_M 使得

$$\chi_M(\boldsymbol{v})=\begin{cases}1, & \boldsymbol{v}\in M\\ 0, & \boldsymbol{v}\notin M\end{cases}$$

该函数称为 M 的指示函数 (Indicator Function). 注意到 χ_M 在 M 的表面 ∂M 处是不连续的, 为了将其与表面的梯度场联系起来, 我们可以对其进行平滑处理, 即对 χ_M 与高斯函数进行卷积操作:

$$\left(\chi_M\circ\tilde{F}\right)(\boldsymbol{v})=\int\tilde{F}\left(\boldsymbol{v}-\boldsymbol{u}\right)\chi_M(\boldsymbol{u})\mathrm{d}\boldsymbol{u}=\int_M\tilde{F}_{\boldsymbol{v}}(\boldsymbol{u})\mathrm{d}\boldsymbol{u},\quad \tilde{F}_{\boldsymbol{v}}(\boldsymbol{u})=\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{||\boldsymbol{u}-\boldsymbol{v}||^2}{2\sigma^2}\right)$$

其中已经带入了 χ_M 的定义. 以下为了方便考虑, 重新定义 $\chi_M=\chi_M\circ\tilde{F}$. 通过散度定理可以证明:

$$\nabla\chi_M(\boldsymbol{v})=\int_{\partial M}\tilde{F}_{\boldsymbol{v}}(\boldsymbol{u})\boldsymbol{n}_{\boldsymbol{u}}\mathrm{d}\boldsymbol{u}$$

其中 $\boldsymbol{n}_{\boldsymbol{u}}$ 是 ∂M 在点 $\boldsymbol{u}$ 处的内法向量[15].

现在, 我们需要将连续的梯度场 $\nabla\chi_M$ 与离散的, 带法向量的点云 S 联系起来. 首先, 需要根据带法向量的点云 S 重建一个向量场 $\boldsymbol{V}$ 用于之后的最小化运算. 根据 S 中的点 $\boldsymbol{v}_i$ 将曲面划分为不相交的局部区域 $\mathcal{P}_i$, 于是对于空间中任意一点 $\boldsymbol{v}$ 有

$$\nabla\chi_M(\boldsymbol{v})=\sum_{\boldsymbol{v}_i\in S}\int_{\mathcal{P}_i}\tilde{F}_{\boldsymbol{v}}(\boldsymbol{u})\boldsymbol{n}_{\boldsymbol{u}}\mathrm{d}\boldsymbol{u}$$

[13]Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In Proceedings of the Fourth Eurographics Symposium on Geometry Processing, SGP '06, 61-70. Goslar, DEU, 2006. Eurographics Association.

[14]文献作者总是采用一些花哨的符号以展现他的数学功底, 可惜这对读者们造成了很大困扰.

[15]这里取法向量的方法与一般的习惯不同, 需要加以注意.

在每个 $\mathcal{P}_i$ 内, 我们近似地用 $\boldsymbol{v}_i$ 代替式中的 $\boldsymbol{u}$, 从而得到

$$\nabla\chi_M(\boldsymbol{v}) \approx \sum_{\boldsymbol{v}_i\in S}\int_{\mathcal{P}_i}\tilde{F}_{\boldsymbol{v}}(\boldsymbol{v}_i)\,\boldsymbol{n}_i\mathrm{d}\boldsymbol{v}_i = \sum_{\boldsymbol{v}_i\in S}\tilde{F}_{\boldsymbol{v}}(\boldsymbol{v}_i)\,\boldsymbol{n}_i\int_{\mathcal{P}_i}\mathrm{d}\boldsymbol{v}_i = \sum_{\boldsymbol{v}_i\in S}\tilde{F}_{\boldsymbol{v}}(\boldsymbol{v}_i)\,\boldsymbol{n}_i\,|\mathcal{P}_i| \xlongequal{\text{定义为}} \boldsymbol{V}(\boldsymbol{v})$$

其中 $|\mathcal{P}_i|$ 为该邻域的面积. 于是重建的向量场 $\boldsymbol{V}(\boldsymbol{v})$ 可以通过点云加权平均得到.

现在的目标是根据重建的向量场 $\boldsymbol{V}$ 来求解指示函数 χ_M. 根据上式, 我们需要解如下的梯度场方程:

$$\nabla\chi_M = \boldsymbol{V}$$

由于右侧的 $\boldsymbol{V}$ 不一定可积, 因此可能无法找到方程的精确解. 于是我们转而求解下面的最小化问题:

$$\arg\min_{\chi_M} = \int_M ||\nabla\chi_M - \boldsymbol{V}||^2\,\mathrm{d}\boldsymbol{v}$$

这一最小化问题的解为:

$$\Delta\chi_M = \nabla\cdot\boldsymbol{V}$$

通过最小二乘法求解该方程, 即可得到 χ_M 的近似解. 最后, 通过 Marching Cubes 等等值面提取算法, 从 χ_M 中提取出表面模型.

Poisson 表面重建依赖于带法向量的点云 $\boldsymbol{V}$ 的构建. 由于点云在空间上不是均布的, 我们需要高效的对其离散化和处理. 常用的办法是八叉树 (Octree) 结构, 它能够自适应地对点云进行空间划分, 在点云密集的区域使用更细的划分, 而在点云稀疏的区域使用较粗的划分. 这样不仅节省了存储空间, 也提高了计算效率. 八叉树的构造方法如下:

1. 初始化一个包含所有点的立方体区域作为八叉树的根节点.

2. 递归地将每个节点划分为八个子节点, 直到满足以下条件之一:

- 节点内的点数小于预设的阈值.
- 达到预设的最大深度.

3. 对于每个叶节点, 计算其包含的点的法向量信息, 并将其存储在节点中.

在八叉树的每个节点上定义局部基函数 (通常是线性函数)$\phi_i(\boldsymbol{v})$, 那么上述问题即可转化为线性方程组的求解问题, 从而得到最终的 χ_M.

2.3.4 隐式表面重建——Marching Cubes 算法

符号距离场函数 f 的定义如下:

$$f_M(\boldsymbol{v}) = \begin{cases} \min\limits_{\boldsymbol{v}'\in\partial M}||\boldsymbol{v}-\boldsymbol{v}'||, & \boldsymbol{v}\in M \\ -\min\limits_{\boldsymbol{v}'\in\partial M}||\boldsymbol{v}-\boldsymbol{v}'||, & \boldsymbol{v}\notin M \end{cases}$$

它本身隐式地表示了几何形状, 其 0-等值面对应着几何形状的表面. 许多方法能够通过对符号距离场求值来计算几何表面的位置. 行进立方体 (Marching Cubes) 算法就是这样一种方法, 它能够从符号距离函数中重建网格模型, 被广泛使用于整个几何建模领域.

Marching Cubes 算法的基本思想是将三维空间划分为一系列立方体单元, 然后在每个立方体内根据符号距离函数的值来确定表面与立方体的交点. 具体步骤如下:

1. 将三维空间划分为均匀的立方体网格.

2. 对于每个立方体的所有顶点 $\boldsymbol{v}_i(i=0,\cdots,7)$, 其 SDF 值 $sdf\left(\boldsymbol{v}_i\right)$ 的正负值决定了其在几何体的外部或内部. 如果一条边的两个顶点的 SDF 值符号不同, 则表明该边与几何体的表面相交. 如此, 每个立方体的顶点的符号组合共有 $2^8=256$ 种可能, 可以将其编码为一个 8 位的二进制数 c. Marching Cubes 算法预先定义了两张查找表: 一张是边状态表 $\mathcal{E}$, 用于指示哪些边与表面相交; 另一张是三角形表 $\mathcal{T}$, 用于指示如何连接这些交点以形成三角形网格.

3. 首先, 根据立方体顶点的符号组合计算出索引 c. 然后, 查找边状态表 $\mathcal{E}[c]$ 以确定哪些边与表面相交. 对于每条相交的边, 通过线性插值计算出交点的位置, 即

$$\boldsymbol{v}^*=\frac{v_2-v^*}{v_2-v_1}\boldsymbol{v}_1+\frac{v^*-v_1}{v_2-v_1}\boldsymbol{v}_2$$

其中 $\boldsymbol{v}^*$ 是交点位置, $\boldsymbol{v}_1,\boldsymbol{v}_2$ 是边的两个端点, v_1,v_2 是顶点的 SDF 值, $v^*=0$ 表示表面的 SDF 值.

然后, 根据三角形表 $\mathcal{T}[c]$ 确定需要将哪些交点连接成三角形, 将交点位置和生成的三角形添加到最终的网格模型中.

4. 重复步骤 2 和 3, 直到处理完所有立方体单元.

2.3.5 点云的模型拟合

模型拟合指的是从给定的点云中提取出平面或球体, 圆柱体等几何基元的过程. 它在低层次的三维数据 (无序的点集) 和高层次的三维形状的结构信息之间建立了联系, 为许多下游的应用提供基础.

我们可以如下定义模型拟合问题[16]: 对于给定的点云 $P=\{\boldsymbol{v}_i:i=1,\cdots,N\}$ 和给定的一类几何形状 S, 求 P 的子集 P' 和 S 的方程 $F(\boldsymbol{v})$ 的参数, 使得 $\forall\boldsymbol{v}_j\in P',F\left(\boldsymbol{v}_j\right)=0$. 由于实际的点云有一定的噪声, 因此通常不能做到完美拟合. 我们一般定义一个误差函数, 然后通过最小二乘法等数值方法近似求解.

拟合方法示例——平面拟合

我们考虑一个比较简单的例子: 输入的点云 P 近似在一个平面 $ax+by+cz+d=0$ 上. 定义误差函数 E 为点到平面的距离的平方和:

$$E=\sum_{i=1}^{N}d_{\boldsymbol{v}_i}=\sum_{i=1}^{N}\left(ax_i+by_i+cz_i+d\right)^2$$

写成矩阵乘法的形式即

$$E=||\boldsymbol{PM}||^2,\quad \boldsymbol{P}=\begin{bmatrix}\boldsymbol{v}_1^{\mathrm{t}} & 1\\ \vdots & \vdots\\ \boldsymbol{v}_N^{\mathrm{t}} & 1\end{bmatrix},\quad \boldsymbol{M}=\begin{bmatrix}a\\ b\\ c\\ d\end{bmatrix}$$

[16]对二维空间中的数据进行线性拟合实际上就可以视作对二维点云的直线拟合.

对 $\boldsymbol{P}$ 奇异值分解, 有

$$\boldsymbol{P} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{t}}$$

由于 $E = \left|\left|\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{t}}\boldsymbol{M}\right|\right|^2 = \left|\left|\boldsymbol{\Sigma}\boldsymbol{V}^{\mathrm{t}}\boldsymbol{M}\right|\right|^2$(正交矩阵不改变向量的模长), 因此为了最小化 E, 需要 $\boldsymbol{M}$ 应当是 $\boldsymbol{V}^{\mathrm{t}}$ 的最后一列 (这样才能对应于最小的奇异值). 这样就得到了平面的参数 a, b, c, d.

随机抽样一致性 (RANSAC) 算法

实际问题是复杂的, 因为 P 中的某些点并不属于待拟合的几何形状[17]. 为了解决这个问题, 我们可以使用随机抽样一致性 (RANSAC, Random Sample Consensus) 算法.

对于点云 P 中的点和待求的形状 S, 显然有相当一部分点落在 S 上 (在实际情况中也可能是 S 附近的一个区域, 下文的落在 S 上也即这种含义). 仍然以平面的拟合为例, 如果落在 S 上的概率为 w, 那么从点云中随机选择 n 个点, 恰好确定目标平面的概率为 w^n. 重复这一过程 k 次, 成功使得取样点均在 S 上的概率为

$$p = 1 - (1 - w^n)^k$$

RANSAC 算法的基本思想正是通过多次随机采样来增加找到正确模型的概率. 具体步骤如下:

1. 随机从点云 P 中选择一定数量的点 (通常选取的较少, 以免需要的采样次数过多. 对于平面这种简单的图形, 一般选取三个点即可) 来确定一个候选模型 S'.
2. 计算点云 P 中每个点到模型 S' 的距离, 并将距离小于预设阈值的点视为内点 (即属于模型的点). 统计内点的数量.
3. 重复步骤 1 和 2 多次, 记录内点数量最多的模型 S^* 及其对应的内点集 P^*.
4. 使用内点集 P^* 重新拟合模型, 得到最终的几何形状.

[17] 例如一个建筑物的点云中不可能所有点都属于某一特定的墙面, 除非这个建筑物是一堵墙.

2.4 几何变换

2.4.1 几何变换基础

定理 4.1 平移 平移对应的数学操作为向量加法. 设物体上任意一点 P 的坐标为 $\boldsymbol{p}$, 平移向量为 $\boldsymbol{t}$, 则平移后的点 P' 的坐标 $\boldsymbol{p}' = \boldsymbol{p} + \boldsymbol{t}$. 例如, 对于三维空间而言有

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \end{bmatrix}$$

定理 4.2 旋转 旋转对应的数学操作为矩阵乘法. 设物体上任意一点 P 的坐标为 $\boldsymbol{p}$, 旋转矩阵为 $\boldsymbol{R}$, 则旋转后的点 P' 的坐标 $\boldsymbol{p}' = \boldsymbol{R}\boldsymbol{p}$. 在二维平面中, 绕原点逆时针旋转 θ 角度的旋转矩阵为

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

在三维空间中, 绕 x, y, z 轴分别旋转 α, β, γ 角度的旋转矩阵分别为

$$\boldsymbol{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix}, \quad \boldsymbol{R}_y = \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}, \quad \boldsymbol{R}_z = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

定理 4.3 绕任意轴的旋转 在三维空间中, 绕单位向量 $\boldsymbol{a} = (a_x, a_y, a_z)$ 定义的轴逆时针旋转 θ 角度的旋转矩阵 $\boldsymbol{R}$ 为

$$\boldsymbol{R} = \cos\theta \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} + (1 - \cos\theta) \begin{bmatrix} a_x^2 & a_x a_y & a_x a_z \\ a_x a_y & a_y^2 & a_y a_z \\ a_x a_z & a_y a_z & a_z^2 \end{bmatrix} + \sin\theta \begin{bmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{bmatrix}$$

证明. 考虑三维空间中的点 $\boldsymbol{v}$, 旋转轴 $\boldsymbol{a}$ 和旋转角度 θ. 从 $\boldsymbol{v}$ 旋转至 $\boldsymbol{v}'$ 的操作可以分解为平行于 $\boldsymbol{a}$ 的分量 $\boldsymbol{r}$ 和垂直于 $\boldsymbol{a}$ 的分量 $\boldsymbol{u}$.

根据投影向量的定义, $\boldsymbol{r} = (\boldsymbol{v} \cdot \boldsymbol{a})\boldsymbol{a}$, $\boldsymbol{u} = \boldsymbol{v} - (\boldsymbol{v} \cdot \boldsymbol{a})\boldsymbol{a}$. 旋转过程中, $\boldsymbol{r}$ 保持不变, 而 $\boldsymbol{u}$ 则绕 $\boldsymbol{a}$ 旋转 θ 角度, 这可以由下面的方式得到

$$\boldsymbol{u}' = \boldsymbol{u}\cos\theta + \boldsymbol{u}^{\perp}\sin\theta$$

注意到 $\boldsymbol{u}^{\perp}=\boldsymbol{a}\times\boldsymbol{v}$, 于是可得

$$\begin{aligned}\boldsymbol{v}'&=[(\boldsymbol{v}-(\boldsymbol{v}\cdot\boldsymbol{a})\boldsymbol{a})\cos\theta]+(\boldsymbol{a}\times\boldsymbol{v})\sin\theta+(\boldsymbol{v}\cdot\boldsymbol{a})\boldsymbol{a}\\&=\boldsymbol{v}\cos\theta+(1-\cos\theta)(\boldsymbol{v}\cdot\boldsymbol{a})\boldsymbol{a}+(\boldsymbol{a}\times\boldsymbol{v})\sin\theta\end{aligned}$$

定义 $\boldsymbol{a}$ 的反对称矩阵为

$$\boldsymbol{a}^{*}=\begin{bmatrix}0&-a_z&a_y\\a_z&0&-a_x\\-a_y&a_x&0\end{bmatrix}$$

不难根据矩阵乘法和向量叉乘的定义得出 $\boldsymbol{a}\times\boldsymbol{v}=\boldsymbol{a}^{*}\boldsymbol{v}$.
根据点积的定义有

$$(\boldsymbol{a}\cdot\boldsymbol{x})\boldsymbol{a}=\left(\boldsymbol{a}^{\mathrm{t}}\boldsymbol{x}\right)\boldsymbol{a}=\boldsymbol{a}\left(\boldsymbol{a}^{\mathrm{t}}\boldsymbol{x}\right)=\left(\boldsymbol{a}\boldsymbol{a}^{\mathrm{t}}\right)\boldsymbol{x}$$

于是将上述 $\boldsymbol{v}'$ 的式子写作矩阵乘法即有

$$\boldsymbol{v}'=\left[\boldsymbol{I}\cos\theta+\boldsymbol{a}\boldsymbol{a}^{\mathrm{t}}(1-\cos\theta)+\boldsymbol{a}^{*}\sin\theta\right]\boldsymbol{v}$$

于是旋转矩阵即为

$$\boldsymbol{R}=\boldsymbol{I}\cos\theta+\boldsymbol{a}\boldsymbol{a}^{\mathrm{t}}(1-\cos\theta)+\boldsymbol{a}^{*}\sin\theta$$

□

2.4.2 齐次坐标系及其中的几何变换

我们在上一节中简单介绍了几何变换对应的数学操作. 例如, 平移可以视作向量的加法, 旋转可以视作旋转矩阵与向量的乘法. 但是, 这些操作并不能统一表示, 因而带来了额外的麻烦. 例如, 平移无法表示为矩阵与向量的乘法. 为了解决这个问题, 我们可以引入齐次坐标系的概念.

齐次坐标系的定义

齐次坐标系的基本思想是, 通过引入额外的维度, 将所有的几何变换都表示为矩阵与向量的乘法.

定义 4.4 齐次坐标系 在 n 维空间 R^n 中, 点 $\boldsymbol{v}=(x_1,\cdots,x_n)$ 的齐次坐标表示为 $(x_1,\cdots,x_n,1)$, 即在原有坐标的基础上增加一个分量恒为 1 的维度.
相应地, 齐次坐标系中的向量 $\boldsymbol{v}=(x_1,\cdots,x_n)$ 表示为 $(x_1,\cdots,x_n,0)$, 即在原有坐标的基础上增加一个分量恒为 0 的维度.

如此, 我们可以将所有的几何变换都表示为 $(n+1)\times(n+1)$ 的矩阵与齐次坐标向量的乘法.

齐次坐标系中几何变换的表示方式

这里以三维空间为例介绍齐次坐标系中几何变换的表示方式.

定理 4.5 齐次坐标系中的平移矩阵 平移向量 $\boldsymbol{t}=(t_x,t_y,t_z)$ 的平移矩阵为

$$\boldsymbol{T}=\boldsymbol{I}+\begin{bmatrix}\boldsymbol{t}\\0\end{bmatrix}$$

对于空间中任意一点 $\boldsymbol{p}$, 在齐次坐标系下总有

$$\begin{bmatrix}\boldsymbol{p}+\boldsymbol{t}\\1\end{bmatrix}=\boldsymbol{T}\begin{bmatrix}\boldsymbol{p}\\1\end{bmatrix}$$

证明. 简单地运用矩阵乘法的定义就能得到. □

定理 4.6 齐次坐标系中的旋转矩阵 齐次坐标系中的旋转矩阵为

$$\boldsymbol{R}=\begin{bmatrix}\boldsymbol{R}'&\boldsymbol{0}\\\boldsymbol{0}&1\end{bmatrix}$$

其中 $\boldsymbol{R}'$ 为三维空间中的旋转矩阵, 在前面已经推出其具体形式.

于是在齐次坐标系下, 任何平移与旋转变换都可以写作矩阵

$$\begin{bmatrix}\boldsymbol{R}&\boldsymbol{t}\\\boldsymbol{0}&1\end{bmatrix}$$

与齐次坐标向量的乘法形式.

2.4.3 几何变换在模型渲染中的应用

对于渲染这一过程来说, 将三维模型投影到二维平面上是一个重要的步骤. 观察者的位置和视角决定了投影的结果. 通常, 我们需要观察者的位置 $\boldsymbol{p}$, 观察平面的法向量 $\boldsymbol{n}$ 和视角正上方的方向 $\boldsymbol{v}$ 决定.

坐标变换

在观察者的坐标中,$\boldsymbol{n}$ 通常指向 z 轴正方向,$\boldsymbol{v}$ 指向 y 轴正方向. 而 x 轴方向根据左右手坐标系而确定. 这里假定 x 轴的方向为 $\boldsymbol{u}=\boldsymbol{v}\times\boldsymbol{n}$. 因此, 我们先需要把模型从世界坐标系 $\mathcal{C}_{\text{world}}$ 变换到观察坐标系 $\mathcal{C}_{\text{obs}}$ 上.

二维坐标变换 为了研究坐标系变换问题, 我们从二维坐标系变换开始.

考虑 $\boldsymbol{p}$ 在 uv 坐标和 xy 坐标下的表示分别为

$$(p_u, p_v) = \boldsymbol{o}_{uv} + p_u\boldsymbol{u} + p_v\boldsymbol{v}$$

$$(p_x, p_y) = \boldsymbol{o}_{xy} + p_x\boldsymbol{x} + p_y\boldsymbol{y}$$

其中 $\boldsymbol{o}_{uv}$ 和 $\boldsymbol{o}_{xy}$ 分别为两个坐标系的原点位置. 并且有

$$\boldsymbol{u} = u_x\boldsymbol{x} + u_y\boldsymbol{y}$$

$$\boldsymbol{v} = v_x\boldsymbol{x} + v_y\boldsymbol{y}$$

于是可以列出方程

$$\boldsymbol{o}_{uv} + p_u(u_x\boldsymbol{x} + u_y\boldsymbol{y}) + p_v(v_x\boldsymbol{x} + v_y\boldsymbol{y}) = \boldsymbol{o}_{xy} + p_x\boldsymbol{x} + p_y\boldsymbol{y}$$

即

$$\boldsymbol{o}_{uv} - \boldsymbol{o}_{xy} = (p_x - p_u u_x - p_v v_x)\boldsymbol{x} + (p_y - p_u u_y - p_v v_y)\boldsymbol{y}$$

设 uv 坐标的原点在 xy 坐标系下的表示为 (e_x, e_y), 则有

$$\begin{cases} e_x = p_x - p_u u_x - p_v v_x \\ e_y = p_y - p_u u_y - p_v v_y \end{cases}$$

于是可得

$$\begin{cases} p_x = e_x + p_u u_x + p_v v_x \\ p_y = e_y + p_u u_y + p_v v_y \end{cases}$$

我们发现上述结果恰好对应于下面的几何变换:

$$\begin{bmatrix} p_x \\ p_y \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & e_x \\ 0 & 1 & e_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & 0 \\ u_y & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_u \\ p_v \\ 1 \end{bmatrix}$$

需要注意的是, 以矩阵乘法表示的几何变换只能在同一坐标系下才具有几何意义. 因此, 上述等式左右两边的点都应在 xy 坐标系下描述其坐标. 也即, 我们把 xy 坐标系中的点 $(p_u, p_v)_{xy}$ 先旋转, 使得旋转后的 $\boldsymbol{x}$, $\boldsymbol{y}$ 轴分别与 $\boldsymbol{u}$, $\boldsymbol{v}$ 轴对齐, 再平移使得 xy 坐标系的原点与 uv 坐标系的原点重合. 此时,$(p_u, p_v)_{xy}$ 将被变换到 $(p_x, p_y)_{xy}$, 这一点的即 uv 坐标系中的点 $(p_u, p_v)_{uv}$ 的位置. 于是我们就完成了坐标系变换. 上述过程如下图所示.

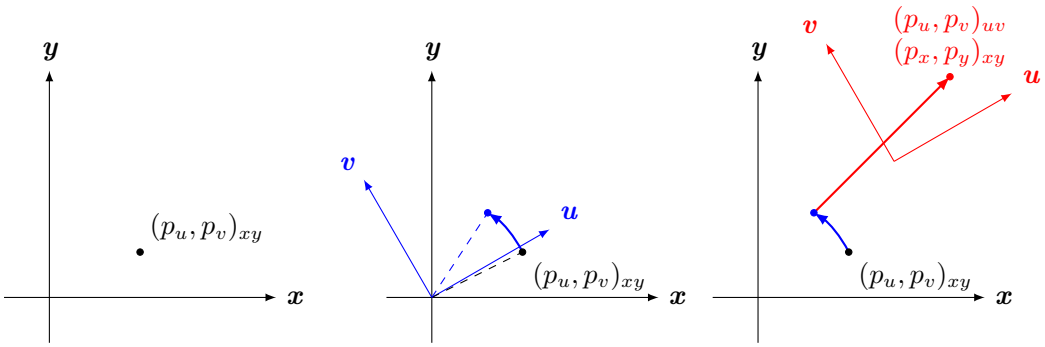


图 2.4: 二维坐标系变换的示意图

三维坐标变换 同样地, 可以推出三维坐标系的矩阵表示为

$$\begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & e_x \\ 0 & 1 & 0 & e_y \\ 0 & 0 & 1 & e_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_u \\ p_v \\ p_w \\ 1 \end{bmatrix}$$

投影变换

在解决坐标变换的问题后, 我们需要将物体投影到观察平面上. 这里我们介绍两种常见的投影方式: 正交和透视投影. 在此之前, 需要明确一些投影中的概念.

定义 4.7 投影线 连接对象点和投影点的直线称作**投影线**.

定义 4.8 平行投影 投影线相互平行的投影方式称作**平行投影**

定义 4.9 (投影) 观察体 观察得到的图像对应在三维空间中的区域称作 **(投影) 观察体**.

定义 4.10 裁剪平面 观察体在 z 方向 (也就是投影平面法向) 的边缘通过选取平行于投影平面的两个平面决定, 这两个平面分别称作**近裁剪平面**和**远裁剪平面**, 分别记作 z_{near} 和 z_{far}.

有时, z_{near} 和 z_{far} 也指两个裁剪平面的 z 轴坐标. 这需要依据上下文确定其含义.

正交投影 我们先来介绍比较简单的投影方式.

定义 4.11 正交投影 **正交投影**属于平行投影的一种, 它的投影线全部与投影平面垂直.

不难看出正交投影是保长度的, 因此工程和建筑测绘常用正交投影.

正交投影从观察坐标到观察平面的变换很简单, 任意一点 $\boldsymbol{v}=(x,y,z)$ 的投影点就是 $\boldsymbol{i}=(x,y)$. 通常, 需要将对象描述转化到规范化坐标系, 即建立 $\boldsymbol{v}\to[-1,1]^3$ 的映射 (即将观察体映射到边长为 2, 范围从 $(-1,-1,-1)$ 到 $(1,1,1)$ 的立方体中). 这对应于一个放缩操作, 因此在齐次坐标系下正交投影的变换矩阵为

$$\boldsymbol{M}_{\text{ortho}}=\begin{bmatrix}\frac{2}{x_{\max}-x_{\min}} & 0 & 0 & -\frac{x_{\max}+x_{\min}}{x_{\max}-x_{\min}}\\ 0 & \frac{2}{y_{\max}-y_{\min}} & 0 & -\frac{y_{\max}+y_{\min}}{y_{\max}-y_{\min}}\\ 0 & 0 & \frac{2}{z_{\max}-z_{\min}} & -\frac{z_{\max}+z_{\min}}{z_{\max}-z_{\min}}\\ 0 & 0 & 0 & 1\end{bmatrix}$$

其中 $x_{\max},x_{\min},y_{\max},y_{\min}$ 分别为观察体在 x 和 y 方向上的上下界, $z_{\min}$ 和 $z_{\max}$ 即为 z_{near} 和 z_{far}. 这样, 在规范化坐标系中的坐标 (x',y',z') 和规范化前的坐标 (x,y,z) 就满足

$$\begin{bmatrix}x' & y' & z' & 1\end{bmatrix}^{\mathrm{t}}=\boldsymbol{M}_{\text{ortho}}\begin{bmatrix}x & y & z & 1\end{bmatrix}^{\mathrm{t}}$$

透视投影 尽管正交投影容易生成, 且可以保持对象的比例不变, 但它的成像缺发真实感. 人眼观察和相机拍摄到的图像通常符合透视投影规律.

定义 4.12 透视投影 透视投影的投影线投影线汇聚于投影中心 C, 投影中心到投影平面的距离称为焦距, 记作 f.

透视投影观察体是棱台形状, 近剪切面 z_{near} 小, 远剪切面 z_{far} 大. 我们需要将观察体映射到一个适合正交投影的区域内 (即将棱台映射到一个长方体内). 具体而言, 是把观察体挤压到一个以 z_{near} 为底面的长方体内. 设观察体内一点 $\boldsymbol{v}=(x,y,z)$ 变换到正交投影区域内的点 $\boldsymbol{u}=(x',y',z')$. 根据相似三角形的性质, 有

$$\frac{x'}{z_{\text{near}}}=\frac{x}{z},\quad \frac{y'}{z_{\text{near}}}=\frac{y}{z}$$

即

$$x'=\frac{xz_{\text{near}}}{z},\quad y'=\frac{yz_{\text{near}}}{z}$$

我们现在求矩阵 $\boldsymbol{M}_{\text{persp}\to\text{ortho}}$ 使得在齐次坐标系下有

$$\begin{bmatrix}\boldsymbol{u}\\ 1\end{bmatrix}=\boldsymbol{M}_{\text{persp}\to\text{ortho}}\begin{bmatrix}\boldsymbol{v}\\ 1\end{bmatrix}$$

注意到 x',y' 的表达式中 z 在分母, 这意味着映射可能是非线性的. 为了避免这一情况, 注意到对齐次坐标系

的各个分量同乘一数不改变其含义, 于是将两边同乘 z 可得 (这里把 z 合并进矩阵中)

$$\begin{bmatrix} xz_{\text{near}} \\ yz_{\text{near}} \\ z'z \\ z \end{bmatrix} = \boldsymbol{M}_{\text{persp}\to\text{ortho}} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

我们可以逆推出矩阵除第三行以外的形式. 于是有

$$\begin{bmatrix} xz_{\text{near}} \\ yz_{\text{near}} \\ z'z \\ z \end{bmatrix} = \begin{bmatrix} z_{\text{near}} & 0 & 0 & 0 \\ 0 & z_{\text{near}} & 0 & 0 \\ m_{31} & m_{32} & m_{33} & m_{34} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

由于 z 坐标的变换与 x, y 坐标无关, 因此 $m_{31} = m_{32} = 0$. 自然, 我们希望映射是线性的, 不应该改变两个裁剪平面的 z 坐标值. 于是分别将 $z = z' = z_{\text{near}}$ 和 $z = z' = z_{\text{far}}$ 代入矩阵的第三行可得

$$\begin{cases} z_{\text{near}} m_{33} + m_{34} = z_{\text{near}}^2 \\ z_{\text{far}} m_{33} + m_{34} = z_{\text{far}}^2 \end{cases}$$

解得 $m_{33} = z_{\text{near}} + z_{\text{far}}, m_{34} = -z_{\text{near}} z_{\text{far}}$. 这样, 观察体就被映射到一个以近裁剪平面为底面,z 坐标取值为 $[z_{\text{near}}, z_{\text{far}}]$ 的长方体内. 再对此长方体应用前面求出的正交投影的变换矩阵, 可知投射投影的变换矩阵为

$$\begin{aligned} \boldsymbol{M}_{\text{persp}} &= \boldsymbol{M}_{\text{ortho}} \boldsymbol{M}_{\text{persp}\to\text{ortho}} \\ &= \begin{bmatrix} \frac{2}{x_{\max}-x_{\min}} & 0 & 0 & -\frac{x_{\max}+x_{\min}}{x_{\max}-x_{\min}} \\ 0 & \frac{2}{y_{\max}-y_{\min}} & 0 & -\frac{y_{\max}+y_{\min}}{y_{\max}-y_{\min}} \\ 0 & 0 & \frac{2}{z_{\text{far}}-z_{\text{near}}} & -\frac{z_{\text{far}}+z_{\text{near}}}{z_{\text{far}}-z_{\text{near}}} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} z_{\text{near}} & 0 & 0 & 0 \\ 0 & z_{\text{near}} & 0 & 0 \\ 0 & 0 & z_{\text{near}}+z_{\text{far}} & -z_{\text{near}} z_{\text{far}} \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{aligned}$$

这就完成了把观察体映射到规范化坐标系的立方体内的过程.

需要注意的是, 在透视投影后的第四个分量即为深度值 z, 这在后续的深度测试中会用到. 此外, 根据齐次坐标的定义, 在使用透视投影矩阵后需要再做一次齐次除法, 即将前三个分量分别除以第四个分量, 才能得到最终的规范化坐标系下的坐标值.

视口映射 在完成将观察体映射到规范化坐标系的工作后, 由于不同设备的屏幕分辨率大小不同, 我们需要将规范化坐标系内的物体再映射到屏幕坐标上 (这里不对 z 坐标做变换, 在以后的步骤中另有他用). 假定屏幕的宽度和高度分别为 w, h, 那么视口变换就是把 xy 坐标 $[-1, 1] \times [-1, 1]$ 映射到 $[0, w] \times [0, h]$ 上. 这也是一个放

缩操作, 因此不难写出视口变换的矩阵为

$$\boldsymbol{M}_{\text{view}} = \begin{bmatrix} w/2 & 0 & 0 & w/2 \\ 0 & h/2 & 0 & h/2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

第三章　渲染

3.1 光照与着色

3.1.1 材质与着色

我们已经学习了如何在计算机中储存和表示三维物体. 然而, 现实中的物体除几何属性外, 还存在**材质 (Material)** 属性. 材质决定了物体表面对光的反射和吸收特性, 从而影响我们最终看到的物体的颜色, 纹理等视觉信息. 在计算机图形学中, 为几何体的表面加上材质的过程称为**着色 (Shading)**.

定义 1.1 材质与着色
材质 (Material) 是物体表面的光学属性, 决定物体如何与光相互作用, 也即决定物体呈现的视觉效果.
着色 (Shading) 是绘制几何体表面的颜色使得它的视觉效果符合设定材质的过程.

3.1.2 光照

光源

按照光源发出光线的不同, 光源可以分为以下几类.

(1) 点光源 (Point Light Source): 点光源在空间中可以看作一个无大小的点, 从该点向各个方向均匀发射光线. 点光源的光强度 I 只与距离有关, 并且正比于 $1/r^2$, 而与方向无关.

(2) 平行光源 (Directional Light Source): 平行光源发出的光线在空间中是平行的. 平行光源的光强度 I 与距离无关, 但与接受平面与光线方向的夹角 θ 有关, 即 $I_{\text{eff}} = I\cos\theta$.

(3) 环境光 (Ambient Light Source): 环境光源表示来自环境中各个方向的漫反射光线的总和, 可以认为在空间中均匀分布, 因此其光强度 I 与距离和方向均无关.

反射

与研究光照类似, 我们先考虑简单情况下的反射规律.

漫反射 当物体表面粗糙不平时, 入射到表面的光线会被反射到各个方向, 称为**漫反射 (Diffuse Reflection)**.

定义 1.2 漫反射与朗博体
漫反射 (Diffuse Reflection) 是指入射光线被物体表面反射到各个方向的现象.
朗博体 (Lambertian Surface) 是表面只有完全随机的漫反射的物体.

对于朗博体的漫反射而言, 我们有如下简单的公式以计算其反射的光强:

$$L_d = k_a I_a + k_d \left(I_d + \frac{I_p}{r^2} \right) \max(\cos\theta, 0)$$

其中 L_d 表示漫反射后的光强 (下标 d 意为 diffuse, 表示漫反射), k_a 和 k_d 分别是物体对环境光的反射率和物体表面的漫反射率. 简单而言[1], k_a 和 k_d 越小, 物体表面吸收的光越多, 反射光强 L_d 也越小, 物体就越暗; 反之物体就越亮. I_a, I_d 和 I_p 分别表示环境光, 平行光和点光源强度, r 表示点光源到物体的距离.

上述式子中的角度 θ 表示物体指向光源的单位向量 $\boldsymbol{l}$ 和物体表面的单位法向量 $\boldsymbol{n}$ 的夹角, 因此式子中的 $\cos\theta$ 可以替换为 $\boldsymbol{l}\cdot\boldsymbol{n}$. 注意, 当 $\theta > \pi/2$ 时, 光线入射到物体的背面, 此时物体并不接受光照, 因此我们取 $\max(\cos\theta, 0)$ 以舍弃此种情形.

镜面反射 当物体表面光滑时, 入射到表面的光线会按照一定的规律被反射, 称为**镜面反射 (Specular Reflection)**.

定义 1.3 镜面反射 **镜面反射 (Specular Reflection)** 是指入射光线按照反射定律被反射的现象.

一般而言, 由于物体表面并不能完全光滑, 镜面反射并不总是完全符合反射定律, 而是分布于镜面反射光线 $\boldsymbol{r}$ 附近. 如果物体表面指向观察者的方向为 $\boldsymbol{v}$, 那么应当有 $\boldsymbol{v}$ 与 $\boldsymbol{r}$ 的夹角越小, 反射光强越大. 这样, 我们可以写出镜面反射的光强计算公式:

$$L_s = k_s\left(I_d + \frac{I_p}{r^2}\right)\max(\boldsymbol{v}\cdot\boldsymbol{r}, 0)^\alpha$$

其中 L_s 表示镜面反射后的光强 (下标 s 意为 specular, 表示镜面反射), k_s 是物体表面的镜面反射率. α 是镜面反射指数, α 越大, 视线远离镜面反射方向时光强衰减地越快, 高光区域越集中, 反之则越分散. 理想镜面反射光线 $\boldsymbol{r}$ 可由

$$\boldsymbol{r} = 2(\boldsymbol{l}\cdot\boldsymbol{n})\,\boldsymbol{n} - \boldsymbol{l}$$

给出.

除去这种办法之外, 人们提出了一种巧妙的计算方法: 定义半程向量

$$\boldsymbol{h} = \frac{\boldsymbol{l}+\boldsymbol{v}}{||\boldsymbol{l}+\boldsymbol{v}||}$$

表示光源方向 $\boldsymbol{l}$ 与观察方向 $\boldsymbol{v}$ 的角平分线方向. 观察方向 $\boldsymbol{v}$ 越接近 $\boldsymbol{r}$, 半程向量 $\boldsymbol{h}$ 就越接近法线 $\boldsymbol{n}$, 因此可以将上式改写为

$$L_s = k_s\left(I_d + \frac{I_p}{r^2}\right)\max(\boldsymbol{n}\cdot\boldsymbol{h}, 0)^\beta$$

其中 β 也是镜面反射指数.

[1] 物体表面的颜色也会影响对各种波长的光的反射率, 不过也可以逐颜色分量地应用上述公式, 因此通常使用的 k 包含 RGB 三个分量. 此时, 如果光源的 RGB 分量表示为向量 $\boldsymbol{I}$, 反射率表示为向量 $\boldsymbol{k}$, 那么最终的反射光颜色可以表示为

$$\boldsymbol{I}\circ\boldsymbol{k}$$

其中 $\circ$ 表示向量逐分量相乘. 因此, 上述公式在各个颜色分量不同时可以写成

$$\boldsymbol{L}_d = \boldsymbol{k}_a\circ\boldsymbol{I}_a + \boldsymbol{k}_d\circ\left(\boldsymbol{I}_d + \frac{\boldsymbol{I}_p}{r^2}\right)\max(\cos\theta, 0)$$

在下面的镜面反射的公式中也类似.

Phong 光照模型和 Blinn-Phong 光照模型 把前述的漫反射和镜面反射结合起来, 我们就得到了常用的光照模型.

定义 1.4 Phong 光照模型 **Phong 光照模型 (Phong Reflection Model)** 结合了漫反射和镜面反射, 其光强计算公式为:

$$L = k_a I_a + k_d \left(I_d + \frac{I_p}{r^2} \right) \max(\boldsymbol{l} \cdot \boldsymbol{n}, 0) + k_s \left(I_d + \frac{I_p}{r^2} \right) \max(\boldsymbol{v} \cdot \boldsymbol{r}, 0)^\alpha$$

定义 1.5 Blinn-Phong 光照模型 **Blinn-Phong 光照模型 (Blinn-Phong Reflection Model)** 是 Phong 光照模型的变种, 其光强计算公式为:

$$L = k_a I_a + k_d \left(I_d + \frac{I_p}{r^2} \right) \max(\boldsymbol{l} \cdot \boldsymbol{n}, 0) + k_s \left(I_d + \frac{I_p}{r^2} \right) \max(\boldsymbol{n} \cdot \boldsymbol{h}, 0)^\beta$$

可以看出, 两者的区别仅在于镜面反射部分. Blinn-Phong 模型使用了半程向量 $\boldsymbol{h}$, 这在高光表现的效果上更平滑自然, 因而更多地被使用.

3.1.3 渲染的光栅化

与二维图形一样, 我们可以通过**光栅化**将虚拟的几何与材质转化为像素. 这主要包含两个问题: 确定像素对应的三维位置 (也即正确处理遮挡关系), 以及确定该像素的颜色.

深度缓存

画家算法 为了确定物体的前后关系, 最初想到的办法是给每个形状/物体一个额外的深度属性 z, 然后对所有形状的深度排序, 按照 z 从大到小 (即从远到近) 的顺序进行绘制. 这和画家的绘画方法类似.

定义 1.6 画家算法 上述过程称作**画家算法 (Painter's Algorithm)**.

然而, 真实世界中物体各处的深度值可能并不相同, 因此可能出现复杂的遮挡关系, 使得画家算法难以实现.

深度缓存 为了解决这个问题, 我们不妨转换思路, 由记录每个物体的深度转为记录每个像素的深度值, 所有像素的深度值构成**深度缓存 (Depth Buffer)**. 每个图形各处的深度值也不是固定的. 在绘制时, 我们对每个像素独立检测, 如果发现等待绘制的图形上的深度小于当前像素的深度, 则覆盖当前像素的颜色和深度值, 否则表示图形在此处被遮挡, 不更新像素.

定义 1.7 上述过程称作**深度缓存 (Depth Buffer)** 算法.

深度缓存技术可以处理更复杂的情况, 并且只需要维护所有像素的缓存, 是一种更高效的办法.

然而, 深度缓存技术对于半透明物体的处理并不理想, 因为半透明物体叠加后的颜色仍密切取决于此处各物体的前后关系. 因此, 通常在这些地方仍然需要使用画家算法.

着色模型

我们现在考虑第二个问题: 确定像素的颜色. 这可以用各种着色模型来实现. 下面给出几种常见的着色模型.

平面着色 最简单而直接的办法就是使用网格的每个面的法向量进行光照计算, 并为整个面赋予同样的颜色.

定义 1.8 平面着色 上述过程称作**平面着色 (Flat Shading)**.

在平面着色中, 每个面只有一个法向, 因而只有一个颜色, 于是我们能在渲染结果里看到很多独立而不连续的面片, 但是我们还是可以直观看到整个形状上明暗的过渡以及高光. 如果要得到比较平滑的结果, 就需要更精细的网格. 为了解决这一问题, 人们提出了许多其它的着色模型.

Gouraud 着色 与平面着色不同, 我们可以根据每个顶点的法向量进行光照计算, 并为顶点赋予颜色. 然后在渲染时, 通过重心插值等插值方法得出三角形面片内部各点的颜色.

定义 1.9 Gouraud 着色 上述过程称作 **Gouraud 着色 (Gouraud Shading)**.

Gouraud 着色在曲面的大部分区域里都可以得到光滑的颜色过渡, 但是在高光区域我们能明显看到三角形插值的痕迹. 这是因为在高光区域附近, 颜色变化随着法向量变化比较明显, 并且这不是线性关系 (例如, 如果高光中心在三角形面片的中心, 而顶点处颜色较暗, 显然不能通过插值表现出中心更亮的颜色).

Phong 着色 为了克服 Gouraud 着色在高光区域的不足, 我们可以直接在三角形面片的**每个像素**处计算法向量, 并根据该法向量计算颜色[2].

定义 1.10 Phong 着色 上述过程称作 **Phong 着色 (Phong Shading)**.

3.1.4 风格化渲染

应用于不同场景的渲染并不一定追求真实感, 有时希望突出/隐去一些细节或展现特定的艺术风格. 这类渲染称为**风格化渲染 (Stylized Rendering)** 或**非真实感渲染 (Non-photorealistic Rendering, NPR)**.

[2]注意与前面的 Blinn-Phong 反射模型加以区分. 两者分别是反射模型和着色模型, 并没有直接联系.

简单而言, 风格化渲染中比较重要的两个特征分别是**线**和**风格化的着色模型**. 我们现在分别介绍这两种特征相关的处理方法.

轮廓线提取

物体的轮廓线对描述物体的信息非常重要. 在风格化渲染中通常会对轮廓线进行特殊处理.

定理 1.11 轮廓线的提取方法 I 物体的边缘位置的法线 $\boldsymbol{n}$ 应当与视线方向 $\boldsymbol{v}$ 近似地垂直, 因此我们可以判断物体上所有满足 $|\boldsymbol{n}\cdot\boldsymbol{v}|<\varepsilon$ 的点, 这些点就可以视作物体的轮廓线.

上述办法依赖于 ε 的设定, 并且得到的边缘并不一定粗细均匀. 由此, 我们还有另一种办法.

定理 1.12 轮廓线的提取办法 II 我们绘制两遍几何体. 第一遍只绘制背面, 并且绘制的颜色为轮廓线的颜色, 同时将几何体向外扩展一些; 第二遍再在背面上绘制正面的几何体, 如此, 没有被遮挡的部分就是轮廓线. 这种办法又被称作**程序化几何法**.

使用上述办法时, 需要注意向外扩展的实现办法, 也就是将顶点沿它的法向移动一定距离, 并且这段距离在屏幕上最终呈现的长度是固定的, 因此这需要考虑投影变换带来的影响; 此外, 绘制正面的几何体时, 需要使用深度缓存, 否则可能丢失部分轮廓线.

风格化着色模型示例——Gooch 着色

除去用光影表现物体的形状, 我们还可以使用颜色的冷暖变化来表现物体的形状. 一种常见的风格化着色模型是 **Gooch 着色 (Gooch Shading)**.

定义 1.13 Gooch 着色 取定冷色 k_c 和暖色 k_w, 物体上各点的颜色通过插值得到:

$$k=\left(\frac{1+\boldsymbol{l}\cdot\boldsymbol{n}}{2}\right)k_c+\left(\frac{1-\boldsymbol{l}\cdot\boldsymbol{n}}{2}\right)k_w$$

其中 $\boldsymbol{l}$ 是物体上的点指向光源的方向, $\boldsymbol{n}$ 是该点表面法线的方向.

3.2 图形管线

我们已经大致学习了从二维图形的绘制, 再到三维图形的建模, 再到本章的三维场景的渲染. 现代的大型实时游戏场景中包含大量的三角形面片, 并且为了看起来流畅, 每秒渲染的帧数常常需要几十上百帧. 为了高效完成图形渲染的工作, 人们对渲染的流程进行了长时间的优化和演进, 最终形成了**图形管线 (Graphic Pipeline)** 的概念.

定义 2.1 图形管线 **图形管线**是在 GPU/图形硬件层面上, 将顶点与图元的输入经过一系列确定的硬件/着色器阶段 (顶点处理, 图元装配, 裁剪/投影, 光栅化, 片段处理, 像素测试与输出合并等), 以产生最终帧缓冲中像素值的有序数据处理流水线.

通常, 我们所说的**渲染管线 (Rendering Pipeline)** 大致与图形管线相同.

3.2.1 渲染管线的一般过程

通常, 渲染管线可以分为以下几个步骤, 每一个步骤接受的输入都是上一步骤的输出.

(1) 应用 (Application)

- 目的: 构建和提交需要渲染的几何数据和渲染状态.
- 输出: 由需要渲染的 3D 应用 (例如游戏) 给出, 包括模型数据, 纹理数据, 相机参数, 光源信息等等[3].

(2) 顶点处理 (Vertex Processing)

- 目的: 将模型变换到裁剪空间[4]; 然后计算顶点处的各种属性 (包括法线, UV 坐标, 颜色等).
- 输出: 顶点在裁剪空间中的位置及其它属性.

(3) 三角形处理 (Triangle Processing)

- 目的: 将顶点组装成三角形图元, 并进行裁剪和投影变换.
- 过程:
  - **a. 图元装配**: 将顶点按预定的模式组装成三角形图元.
  - **b. 背面剔除**: 基于顶点顺序与法线方向的关系删除不可见的三角形.
  - **c. 裁剪**: 将超出视锥的三角形裁剪裁剪成视锥内的部分[5].
  - **d. 透视除法与视口变换**: 将裁剪空间中的顶点通过透视除法转换到 NDC 空间, 然后通过视口变换映射到屏幕空间.

[3]通常, 在渲染过程中采取的加速算法也由 CPU 执行, 因此包含在此部分内.

[4]通过我们在几何变换中所学的投影变换的方法转换到裁剪空间中. 裁剪空间中顶点以齐次坐标的形式表示, 并且第四个分量 w 尚未进行归一化. 对裁剪空间中顶点坐标做下述变换

$$\begin{bmatrix} x & y & z & w \end{bmatrix}^{\mathrm{t}} \longrightarrow \begin{bmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} \end{bmatrix}^{\mathrm{t}}$$

即归一化并除去第四个分量后才能转换成规范化坐标系, 即 **NDC(Normalized Device Coordinates) 空间**. 这一步被称作**透视除法**, 是下一步三角形处理中的操作.

[5]这一步就是裁剪空间这一名称的由来.

- 输出: 屏幕空间中的三角形片元 (包含各种属性).

(4) 光栅化 (Rasterization)

- 目的: 将三角形片元转换为屏幕的片元 (Fragments, 可以理解为采样点), 并生成用于插值的片元数据[6].
- 输出: 片元流 (Fragment Stream), 其中每个片元包含插值后的属性 (颜色, 法线, UV 坐标, 深度等).

(5) 片元处理 (Fragment Processing)

- 目的: 通过给定的着色器, 以及片元的各种数据 (包括纹理坐标, 法线, 顶点颜色, 深度等) 计算片元的最终颜色值.
- 输出: 每个片元的最终颜色值和深度值.

(6) 帧缓冲操作 (Framebuffer Operations)

- 目的: 将片元的颜色和深度值与帧缓冲中的现有值进行测试和合并, 以生成最终的像素值.
- 过程:
  a. **深度测试**: 比较片元的深度值与帧缓冲中对应像素的深度值, 以确定片元是否可见.
  b. **模板测试**: 基于模板缓冲的值决定片元是否应被绘制. 通常用于复杂遮罩, 轮廓等高级效果的控制.
  c. **混合**: 将片元的颜色值与帧缓冲中现有的颜色值进行混合 (如果需要), 以实现透明度等效果.

(7) 显示 (Display)

- 目的: 将帧缓冲中的最终像素值传输到显示设备 (例如监视器) 以进行可视化.
- 输出: 显示设备上呈现的最终图像.

3.2.2 图形 API

为了实现对图形管线的控制和使用, 现代计算机图形学中引入了**图形 API(Graphic API)** 的概念.

定义 2.2 图形 API **图形 API** 是指应用程序与图形硬件之间的抽象接口, 它通过一组函数或指令集, 使开发者能够控制 GPU 执行图形渲染, 计算和资源管理等操作, 而无需直接编写底层驱动代码.
常见图形 API 包括 OpenGL, Vulkan, DirectX, Metal 等.

上面介绍的渲染管线仅为通用的大致结构, 各个图形 API 对管线的具体实现过程是不同的. 以 OpenGL 为例, 其渲染管线如下所示.

[6] 一些超采样算法, 例如 MSAA, 需要在此时进行覆盖测试; 此外, 片元的插值需要经过透视矫正.

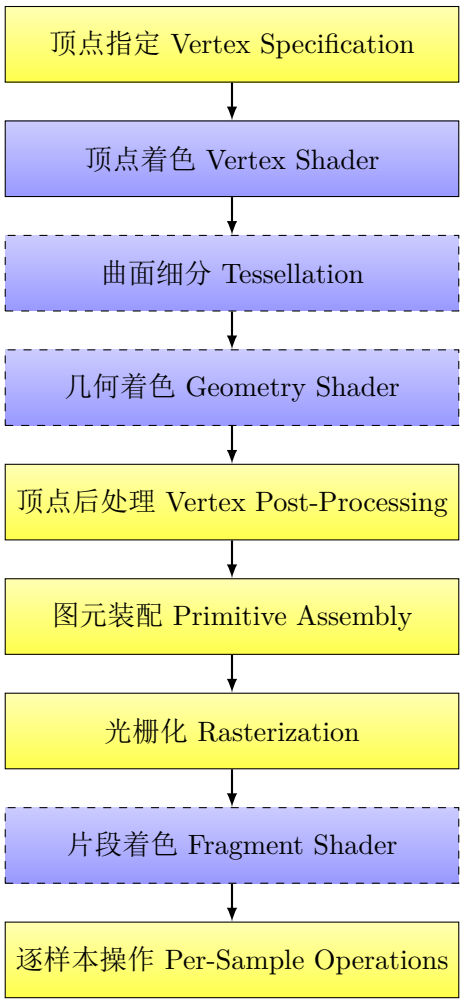


图 3.1: OpenGL 渲染管线示意图

上述流程图中仅有标蓝色框的部分是可编程的, 其余步骤均在硬件驱动中实现, 仅能通过一些选项调整参数. 大部分情况下需要手动编写的仅有**顶点着色器**.

3.3 纹理映射

通过前面着色部分的介绍, 我们可以绘制出光滑均匀的表面. 然而, 现实中诸如木头, 锈铁这样的材质包含了丰富的细节, 表面的颜色和粗糙度等属性在表面上都不均匀. 可以想见, 想用高精度的几何表示表面的这些信息是不现实的, 更为实际的做法是将这些信息记录在二维的图像中, 然后将其附加到几何表面上. 记录表面颜色, 粗糙度等信息的图像就称为**纹理 (Texture)**, 或者称为**贴图**; 而将其映射到三维几何体上的过程就称为**纹理映射 (Texture Mapping)**.

3.3.1 纹理映射

定义 3.1 纹理坐标 纹理是二维图像, 我们可以通过直角坐标系中的点 (u, v) 访问其上的颜色, 一般称作**纹理坐标 (Texture Coordinate)**.

纹理映射的过程实际上就是将三维几何体上的点 (x, y, z) 映射到二维纹理图像上的点 (u, v) 的过程. 对于三角形网格模型, 我们只需要为每个顶点记录纹理坐标 (u, v), 然后通过重心插值的办法即可得到表面上所有点的 uv 坐标.

上面的过程在三维空间中是显然的. 然而, 在屏幕空间上, 我们应该如何确定每个像素对应的纹理坐标? 我们已经知道每个像素在屏幕空间中的位置, 也知道每个顶点在屏幕空间中的位置和纹理坐标. 一种简单的想法是直接在屏幕空间中进行重心插值, 然而这样得到的深度是不准确的.

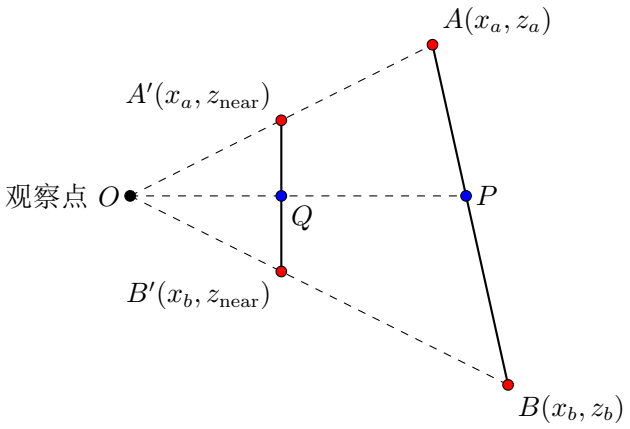


图 3.2: 投影变换与插值

如上图所示, 空间中的顶点 A 和 B 经过投影变换后映射到屏幕上的 A' 和 B'. 然而, 容易看出 A' 和 B' 的中点 Q 所对应的空间中的点 P 却不是 A 和 B 的中点.

看起来只有在投影变换前先进行插值才能解决这个问题, 不过这样做的开销会比较大. 实际上, 我们可以采用一种更为高效的做法, 即对屏幕空间的插值结果进行修正, 称作**透视校正插值 (Perspective-Correct**

Interpolation). 具体而言, 假定用屏幕上的坐标求得以 $\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3$ 为顶点的三角形内某一点 $\boldsymbol{q}$ 的重心坐标为

$$\boldsymbol{q} = \alpha \boldsymbol{p}_1 + \beta \boldsymbol{p}_2 + \gamma \boldsymbol{p}_3$$

那么该点经过校正的插值结果为

$$f(\boldsymbol{q}) = \frac{\dfrac{\alpha}{w_1} f(\boldsymbol{p}_1) + \dfrac{\beta}{w_2} f(\boldsymbol{p}_2) + \dfrac{\gamma}{w_3} f(\boldsymbol{p}_3)}{\dfrac{\alpha}{w_1} + \dfrac{\beta}{w_2} + \dfrac{\gamma}{w_3}}$$

其中 w_i 为顶点经过投影变换后齐次坐标未经归一化的第四个分量. 通过这种方式, 我们可以在屏幕空间高效地进行纹理坐标的插值.

3.3.2 纹理坐标插值

使用前面的经过校正线性插值获得的纹理坐标 (u, v) 是浮点数坐标, 而大部分纹理和正常的图像一样都是像素化存储的. 我们需要继续通过插值的方法获取该位置的颜色值.

回想 Lab 1 中神人一般的反走样, 我们可以通过双线性插值的办法获取纹理坐标的信息. 假定 (u, v) 位于纹理图像中四个像素 $(i, j), (i+1, j), (i, j+1), (i+1, j+1)$ 所围成的正方形内,(u, v) 在这一单位正方形中的坐标为 (s, t), 那么可以通过下面的公式计算出该位置的颜色:

$$\boldsymbol{c}_{uv} = (1-t)(1-s)\boldsymbol{c}_{ij} + (1-t)s\boldsymbol{c}_{i(j+1)} + (1-s)t\boldsymbol{c}_{(i+1)j} + st\boldsymbol{c}_{(i+1)(j+1)}$$

相比于寻找最近邻的像素, 这一方法能更好地避免锯齿, 得到较为平滑地结果.

3.3.3 纹理反走样

我们在 **Lecture 6 反走样**中已经介绍过使用 MIPMAP 进行反走样的办法, 这里就不再重复了.

3.3.4 纹理应用

除去用纹理记录颜色信息外, 我们还可以用纹理储存更多信息, 以实现更真实或更高效的渲染效果.

凹凸贴图

如果物体的表面凹凸不平, 我们不得不采用更高精度的模型来表示这些细节, 从而增加开销. 然而, 如果这些凹凸结构只影响最后的渲染, 而不影响物体的其它效果 (比如物体的运动和碰撞等), 我们就可以使用精度较低的模型, 然后将这些表面的凹凸细节记录在纹理中. 这种技术称作**凹凸贴图 (Bump Mapping)**.

实现凹凸贴图的办法有很多. 一种办法是直接记录每个点的法向量 (这是三维空间中的向量, 因此恰好可以表示为 RGB 通道上的分量, 从而以图像的形式存储); 另一种办法是通过灰度图像记录高度信息, 然后通过差分计算法向量. 无论采用哪种办法, 最终我们都可以在片元着色器中使用凹凸贴图后的法向量进行光照计算, 从而得到更加真实的表面效果.

凹凸贴图的数学原理和传统方法 考虑模型的法线 $\boldsymbol{n}$. 我们希望通过凹凸贴图对法向量施加一个扰动 $\Delta\boldsymbol{n}$, 即

$$\boldsymbol{n}' = \boldsymbol{n} + \Delta\boldsymbol{n}$$

现在的目标是求出法线 $\boldsymbol{n}$. 对于模型上任意一点, 将其位置 $\boldsymbol{p}$ 写做 UV 坐标系下的参数方程 $\boldsymbol{p} = \boldsymbol{p}(u, v)$. 为了求出这一点的法向量 $\boldsymbol{n}$, 我们考虑固定 u 得到的曲线 $\boldsymbol{p}_v = \boldsymbol{p}(u_0, v)$ 和固定 v 得到的曲线 $\boldsymbol{p}_u = \boldsymbol{p}(u, v_0)$. 对这两条曲线分别求导, 可得它们在此处的切向方向为

$$\boldsymbol{t} = \frac{\partial \boldsymbol{p}}{\partial u}, \quad \boldsymbol{b} = \frac{\partial \boldsymbol{p}}{\partial v}$$

这两条切线分别被称作切线和副切线. 根据法线 $\boldsymbol{n}$ 的定义, 它应当与经过该点的所有曲线在此处的切线均垂直, 特别地, 它应当与切线 $\boldsymbol{t}$ 和副切线 $\boldsymbol{b}$ 均垂直. 于是我们可以用向量的叉积求出法线, 其方向即为

$$\boldsymbol{n} = \frac{\boldsymbol{t} \times \boldsymbol{b}}{||\boldsymbol{t} \times \boldsymbol{b}||}$$

现在假定每个点的高度由纹理函数 $h(u, v)$[7]给出, 那么该点处真实的高度即为模型上的点沿模型法向量 $\boldsymbol{n}$ 方向移动 $h(u, v)$ 长度, 于是新的位置为

$$\boldsymbol{p}'(u, v) = \boldsymbol{p}(u, v) + h(u, v)\boldsymbol{n}$$

现在我们来计算新的法向量 $\boldsymbol{n}'$. 对 $\boldsymbol{p}'$ 求偏导数, 可得新的切线 $\boldsymbol{t}'$ 为

$$\boldsymbol{t}' = \frac{\partial \boldsymbol{p}'}{\partial u} = \frac{\partial \boldsymbol{p}}{\partial u} + \frac{\partial h}{\partial u}\boldsymbol{n} + \frac{\partial \boldsymbol{n}}{\partial u}h = \boldsymbol{t} + \frac{\partial h}{\partial u}\boldsymbol{n} + \frac{\partial \boldsymbol{n}}{\partial u}h$$

类似地可得新的副切线 $\boldsymbol{b}'$ 为

$$\boldsymbol{b}' = \boldsymbol{b} + \frac{\partial h}{\partial v}\boldsymbol{n} + \frac{\partial \boldsymbol{n}}{\partial v}h$$

一般而言, 法向量 $\boldsymbol{n}$ 的变化 (尤其是在精度不高的模型上) 是比较小的, 因此可以忽略上式中带有 $\dfrac{\partial \boldsymbol{n}}{\partial u, v}$ 的项. 于是新的法向量可以由下式给出:

$$\begin{aligned}\boldsymbol{n}' = \boldsymbol{t}' \times \boldsymbol{b}' &= \left(\boldsymbol{t} + \frac{\partial h}{\partial u}\boldsymbol{n}\right) \times \left(\boldsymbol{b} + \frac{\partial h}{\partial v}\boldsymbol{n}\right) \\ &= \boldsymbol{t} \times \boldsymbol{b} + \frac{\partial h}{\partial u}\boldsymbol{n} \times \boldsymbol{b} + \frac{\partial h}{\partial v}\boldsymbol{t} \times \boldsymbol{n}\end{aligned}$$

最后归一化即可得到新的法向量:

$$\boldsymbol{n}' = \text{normalize}\left(\boldsymbol{t} \times \boldsymbol{b} - \frac{\partial h}{\partial u}\boldsymbol{b} \times \boldsymbol{n} - \frac{\partial h}{\partial v}\boldsymbol{n} \times \boldsymbol{t}\right) = \boldsymbol{n} + \frac{\dfrac{\partial h}{\partial u}\boldsymbol{n} \times \boldsymbol{b} + \dfrac{\partial h}{\partial v}\boldsymbol{t} \times \boldsymbol{n}}{\boldsymbol{n} \cdot (\boldsymbol{t} \times \boldsymbol{b})}$$

基于表面梯度的凹凸贴图办法 传统的凹凸贴图依赖于已知的表面参数化, 在网格中需要预计算和存储切向量, 增加了内存和计算开销. 我们现在介绍一种基于表面梯度的凹凸贴图办法.

仍然考虑物体表面的参数化. 但是现在, 我们不一定要采用 UV 坐标, 而是考虑一个任意的参数化方式 $\boldsymbol{p}(s, t)$. 通过类似的方式仍然能求出其切线 $\boldsymbol{t}$, 副切线 $\boldsymbol{b}$ 以及法线 $\boldsymbol{n}$. 对于定义在物体表面 S 上的高度函数 $\tilde{h}$,

[7]在下一部分的改进办法中, $\tilde{h}$ 代指对物体表面上一点 (x, y, z) 的纹理高度, 即 $h(s, t) = \tilde{h}(\boldsymbol{p}(s, t)) = \tilde{h}(x, y, z)$.

其表面梯度 $\nabla_S\tilde{h}$ 是一个切于表面的向量, 沿该方向上函数变化的速率最快. 事实上,$\nabla_S\tilde{h}$ 正是 $\nabla\tilde{h}$ 在 S 的切平面上的投影向量, 即

$$\nabla_S\tilde{h} = \nabla\tilde{h} - \left(\boldsymbol{n}\cdot\nabla\tilde{h}\right)\boldsymbol{n}$$

我们现在先来求 $\nabla_S\tilde{h}$. 将 $\tilde{h}$ 视作 s,t 的函数, 即 $h(s,t)=\tilde{h}(\boldsymbol{p}(s,t))$, 于是对 h 求偏导数可得

$$\begin{aligned}\frac{\partial h}{\partial s} &= \frac{\partial}{\partial s}\tilde{h}(\boldsymbol{p}(s,t)) = \frac{\partial}{\partial s}\tilde{h}(x(s,t),y(s,t),z(s,t))\\ &= \frac{\partial\tilde{h}}{\partial x}\frac{\partial x}{\partial s}+\frac{\partial\tilde{h}}{\partial y}\frac{\partial y}{\partial s}+\frac{\partial\tilde{h}}{\partial z}\frac{\partial z}{\partial s}\\ &= \nabla\tilde{h}\cdot\frac{\partial\boldsymbol{p}}{\partial s} = \nabla\tilde{h}\cdot\boldsymbol{t}\end{aligned}$$

同样地有

$$\frac{\partial h}{\partial t} = \nabla\tilde{h}\cdot\frac{\partial\boldsymbol{p}}{\partial t} = \nabla\tilde{h}\cdot\boldsymbol{b}$$

根据 $\nabla_S\tilde{h}$ 与 $\nabla\tilde{h}$ 的关系可得

$$\nabla_S\tilde{h}\cdot\boldsymbol{t} = \left(\nabla\tilde{h}-\left(\boldsymbol{n}\cdot\nabla\tilde{h}\right)\boldsymbol{n}\right)\cdot\boldsymbol{t} = \nabla_S\tilde{h}\cdot\boldsymbol{t}-\left(\boldsymbol{n}\cdot\nabla\tilde{h}\right)\boldsymbol{n}\times\boldsymbol{t} = \nabla_S\tilde{h}\cdot\boldsymbol{t} = \frac{\partial h}{\partial s}$$

同样地有

$$\nabla_S\tilde{h}\cdot\boldsymbol{b} = \frac{\partial h}{\partial t}$$

此外, $\nabla_S\tilde{h}$ 本身也位于切平面上, 因此可以将它表示为 $\boldsymbol{t}$ 和 $\boldsymbol{b}$ 的线性组合. 为了接下来的计算方便, 我们引入 $\boldsymbol{t},\boldsymbol{b}$ 的对偶基 $\boldsymbol{t}^*,\boldsymbol{b}^*$, 它们满足

$$\boldsymbol{t}^* = \boldsymbol{b}\times\boldsymbol{n},\quad \boldsymbol{b}^* = \boldsymbol{n}\times\boldsymbol{t}$$

令 $D=\boldsymbol{n}\cdot(\boldsymbol{t}\times\boldsymbol{b})$ 为 $\boldsymbol{t},\boldsymbol{b}$ 与 $\boldsymbol{n}$ 的标量三重积. 对偶基 $\boldsymbol{t}^*,\boldsymbol{b}^*$ 有如下性质:

$$\boldsymbol{t}^*\cdot\boldsymbol{t} = \boldsymbol{b}^*\cdot\boldsymbol{b} = D,\quad \boldsymbol{t}^*\cdot\boldsymbol{b} = \boldsymbol{b}^*\cdot\boldsymbol{t} = 0$$

并且容易看出 $\boldsymbol{t}^*$ 和 $\boldsymbol{b}^*$ 也是切平面的一组基. 由于 $\nabla_S\tilde{h}$ 也位于切平面上, 因此存在唯一的 α,β 使得

$$\nabla_S\tilde{h} = \alpha\boldsymbol{t}^*+\beta\boldsymbol{b}^*$$

利用我们之前的结论, 将其分别于 $\boldsymbol{t}$ 和 $\boldsymbol{b}$ 做点积可得

$$\begin{cases}\nabla_S\tilde{h}\cdot\boldsymbol{t} = \alpha\boldsymbol{t}^*\cdot\boldsymbol{t}+\beta\boldsymbol{b}^*\cdot\boldsymbol{t} = \alpha D = \dfrac{\partial h}{\partial s}\\ \nabla_S\tilde{h}\cdot\boldsymbol{b} = \alpha\boldsymbol{t}^*\cdot\boldsymbol{b}+\beta\boldsymbol{b}^*\cdot\boldsymbol{b} = \beta D = \dfrac{\partial h}{\partial t}\end{cases}$$

于是就有

$$\nabla_S\tilde{h} = \frac{\dfrac{\partial h}{\partial s}\boldsymbol{t}^*+\dfrac{\partial h}{\partial t}\boldsymbol{b}^*}{D} = \frac{\dfrac{\partial h}{\partial s}\boldsymbol{b}\times\boldsymbol{n}+\dfrac{\partial h}{\partial t}\boldsymbol{n}\times\boldsymbol{t}}{\boldsymbol{n}\cdot(\boldsymbol{t}\times\boldsymbol{b})}$$

于是回顾我们前面推出的凹凸贴图法线扰动公式, 就可得

$$\boldsymbol{n}' = \boldsymbol{n}-\nabla_S h$$

既然参数化方法 $\boldsymbol{p}(s,t)$ 是任意选取的, 那么最直接的想法就是令 s,t 分别为屏幕空间的 x,y 坐标. 这样, 我们可以在屏幕空间中通过微分或差分的方式直接计算上述式子中需要的参量 $\boldsymbol{t},\boldsymbol{b},\dfrac{\partial h}{\partial s},\dfrac{\partial h}{\partial t}$. 这就是基于表面梯度的凹凸贴图的实现原理.

立方体贴图与天空盒

简单的纹理通常是一张二维图形. 在本节我们将讨论将多个纹理组合起来映射到一张纹理上的贴图: **立方体贴图 (Cube Map)**.

立方体贴图包含六张等大的正方形纹理图片, 对应立方体的六个面. 这里略去立方体贴图的实现方式. 我们现在讨论其一个重要应用: **天空盒 (Skybox)**.

天空盒是一个包含了整个场景的立方体, 它包含周围环境的 6 个图像, 让玩家以为他处在一个比实际大得多的环境当中.

3.4 全局光照

3.4.1 光线投射与光线追踪

光线投射

大部分环境中的光都不能被镜头所捕捉, 因此相比于考虑每个光源发出的每条光线, 我们不妨采取逆向思维, 考虑那些最终到达镜头的光线. 由于光路是可逆的, 因此我们从镜头向屏幕上的点连一条线, 该射线在场景中击中的第一个物体将决定该点的颜色.

定义 4.1 光线投射 光线投射 (Ray Casting) 是一种通过从观察点向场景中投射射线来确定可见表面和颜色的技术.

因此, 光线投射算法的基本流程如下:

1. 对于屏幕的每个像素点 (x, y), 从相机位置向场景中投射一条穿过 (x, y) 的射线 $\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$.
2. 计算射线 $\boldsymbol{r}(t)$ 与场景中的物体第一次发生相交的位置 $\boldsymbol{p}$.
3. 将交点 $\boldsymbol{p}$ 与所有光源相连, 分别得到一根 shadow ray. 判断 shadow ray 在到达光源之前是否与其他物体相交, 如果相交则该光源对点 $\boldsymbol{p}$ 不可见, 否则可见.
4. 在可见的情况下, 我们已经知道了指向光源的方向 $\boldsymbol{l}$, 指向观察者的方向 $-\boldsymbol{d}$, 以及表面法线 $\boldsymbol{n}$. 使用在光照与着色中介绍的各种光照模型即可计算光源对点 $\boldsymbol{p}$ 贡献的颜色.
5. 将所有光源的贡献累加, 得到最终的像素颜色.

上述过程的伪代码可以表示如下:

```
for each pixel (x, y) do
    # Generate primary ray from camera through pixel
    ray = generateRay(camera, x, y)
    # Find intersection with scene.
    hitInfo = intersectScene(ray)
    if hitInfo.hit then
        color = vec3(0, 0, 0)
        # For each light source, check visibility and compute lighting.
        for each light in scene.lights do
            shadowRay = generateShadowRay(hitInfo.position, light.position)
            if not intersectScene(shadowRay).hit then
                color += computeLighting(hitInfo, light)
        setPixelColor(x, y, color)
    else
        setPixelColor(x, y, backgroundColor)
```

光线追踪

在实际情况下, 物体接受的光照可能并不直接来自于光源, 而可能经过其它物体的反射/折射再到达该物体上. 为了考虑这些间接光照, 我们需要引入光线追踪技术.

定义 4.2 光线追踪 光线追踪 (Ray Tracing) 通过模拟光线在场景中的传播路径, 包括反射和折射, 来生成更加逼真的图像.

光线追踪算法的基本流程如下:

1. 对于屏幕的每个像素点 (x, y), 从相机位置向场景中投射一条穿过 (x, y) 的射线 $\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}$.
2. 计算射线 $\boldsymbol{r}(t)$ 与场景中的物体第一次发生相交的位置 $\boldsymbol{p}$.
3. 按照光线投射中介绍的办法计算交点 $\boldsymbol{p}$ 处直接来自于光源形成的颜色.
4. 根据材质属性, 生成反射射线和折射射线, 递归地追踪这些射线以计算间接光照形成的颜色.
5. 将局部光照和间接光照形成的颜色累加, 得到最终的像素颜色.

上述过程的递归形式的伪代码可以表示如下:

```
# Define recursively ray tracing function.
function traceRay(ray, depth):
    if depth > maxDepth then
        return backgroundColor
    hitInfo = intersectScene(ray)
    if hitInfo.hit then
        color = computeLocalLighting(hitInfo)
        # Compute reflection
        if hitInfo.material.reflective then
            reflectRay = generateReflectRay(hitInfo)
            color += hitInfo.material.reflectivity * traceRay(reflectRay, depth + 1)
        # Compute refraction
        if hitInfo.material.refractive then
            refractRay = generateRefractRay(hitInfo)
            color += hitInfo.material.transparency * traceRay(refractRay, depth + 1)
        return color
    else
        return backgroundColor
for each pixel (x, y) do
    ray = generateRay(camera, x, y)
    color = traceRay(ray, 0)
    setPixelColor(x, y, color)
```

光线追踪算法能够生成高度逼真的图像, 但计算开销较大, 因此在实际应用中通常会结合其他算法进行优化. 我们将在之后讲到优化的方式. 在此之前, 我们先来了解光线投射/追踪的基本步骤, 即光线求交.

3.4.2 光线求交

可以看出, 在光线投射与光线追踪算法中, 计算射线与场景中物体的交点是一个核心步骤. 高效地实现光线求交对于提升渲染性能至关重要. 下面我们将介绍几种常见的几何体与光线求交点的办法. 在本节的推导中, 我们假定光线的方程为

$$\boldsymbol{r}(t) = \boldsymbol{o} + t\boldsymbol{d}, \quad 0 \leqslant t < +\infty$$

其中 $\boldsymbol{o}$ 意为 Origin, 即光线的起点, 而 $\boldsymbol{d}$ 意为 Direction, 即光线的方向向量. 求得的 t 即与物体的交点到光线起点的距离.

光线与球面求交

考虑球心为 $\boldsymbol{c}$, 半径为 r 的球面, 其隐式方程为

$$||\boldsymbol{p} - \boldsymbol{c}||^2 = r^2$$

代入光线方程, 我们有

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

展开后得到关于 t 的二次方程

$$t^2\boldsymbol{d}\cdot\boldsymbol{d} + 2t\boldsymbol{d}\cdot(\boldsymbol{o}-\boldsymbol{c}) + (\boldsymbol{o}-\boldsymbol{c})\cdot(\boldsymbol{o}-\boldsymbol{c}) - r^2 = 0$$

设 $a = \boldsymbol{d}\cdot\boldsymbol{d}$, $b = 2\boldsymbol{d}\cdot(\boldsymbol{o}-\boldsymbol{c})$, $c = (\boldsymbol{o}-\boldsymbol{c})\cdot(\boldsymbol{o}-\boldsymbol{c}) - r^2$, 则方程的解为

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

根据判别式 $b^2 - 4ac$ 的值, 我们可以判断光线与球面的交点情况是相离, 相切还是相交, 然后根据求根公式即可求出交点.

光线与长方体求交

为了简化光线与物体的求交计算, 我们通常对物体创建一个规则的几何外形将其包围. 最常见的包围体是**轴对齐包围盒 (Axis-Aligned Bounding Box, AABB)**.

定义 4.3 轴对齐包围盒 轴对齐包围盒是指其边界与坐标轴平行的长方体, 通常由最小点 $\boldsymbol{p}_{\min} = (x_{\min}, y_{\min}, z_{\min})$ 和最大点 $\boldsymbol{p}_{\max} = (x_{\max}, y_{\max}, z_{\max})$ 定义.

这里介绍一种高效的光线与 AABB 求交的方法, 即 **Slabs Method**. 不失一般性地, 我们考虑长方体中平行于 xy 平面的表面. 设表面所在的方程分别为 $z = z_{\min}$ 和 $z = z_{\max}$, 则光线与这两个平面的交点 t 值分别为

$$t_{z_{\min}} = \frac{z_{\min} - \boldsymbol{o}_z}{\boldsymbol{d}_z}, \quad t_{z_{\max}} = \frac{z_{\max} - \boldsymbol{o}_z}{\boldsymbol{d}_z}$$

于是光线处于这两个平面之间时总有

$$t_{z_{\min}} \leqslant t \leqslant t_{z_{\max}}$$

类似地, 我们可以计算出光线与平行于 xy 平面和 yz 平面的交点 t 值, 分别记为 $t_{x_{\min}}, t_{x_{\max}}$ 和 $t_{y_{\min}}, t_{y_{\max}}$. 如果光线与长方体相交, 那么它同时处于这三组平面之间, 于是

$$[t_{x_{\min}}, t_{x_{\max}}] \cap [t_{y_{\min}}, t_{y_{\max}}] \cap [t_{z_{\min}}, t_{z_{\max}}] \neq \varnothing$$

于是令

$$t_{\min} = \max\left\{t_{x_{\min}}, t_{y_{\min}}, t_{z_{\min}}\right\}, \quad t_{\max} = \min\left\{t_{x_{\max}}, t_{y_{\max}}, t_{z_{\max}}\right\}$$

只需要 $t_{\min} \leqslant t_{\max}$, 则光线与长方体相交, 交出的线段即为

$$\boldsymbol{r}'(t) = \boldsymbol{o} + t\boldsymbol{d}, \quad t_{\min} \leqslant t \leqslant t_{\max}$$

光线与三角面片求交

三角面片是计算机图形学中最常用的基本几何体之一, 因此高效地实现光线与三角面片的求交对于光线追踪算法至关重要. 这里介绍一种常用的光线与三角面片求交算法, 即 **Möller-Trumbore 算法**. 对于三角形上任意一点 $\boldsymbol{t}(u, v)$, 其满足

$$\boldsymbol{t}(u, v) = (1 - u - v)\boldsymbol{v}_0 + u\boldsymbol{v}_1 + v\boldsymbol{v}_2$$

这里的 u, v 即为点在三角形中的重心坐标, 满足 $u \geqslant 0, v \geqslant 0$ 且 $u + v \leqslant 1$. 将上述方程与光线方程联立, 可得

$$\boldsymbol{o} + t\boldsymbol{d} = (1 - u - v)\boldsymbol{v}_0 + u\boldsymbol{v}_1 + v\boldsymbol{v}_2$$

整理后得到

$$-t\boldsymbol{d} + u(\boldsymbol{v}_1 - \boldsymbol{v}_0) + v(\boldsymbol{v}_2 - \boldsymbol{v}_0) = \boldsymbol{o} - \boldsymbol{v}_0$$

这一线性方程组的矩阵形式为

$$\begin{bmatrix} -\boldsymbol{d} & \boldsymbol{v}_1 - \boldsymbol{v}_0 & \boldsymbol{v}_2 - \boldsymbol{v}_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \boldsymbol{o} - \boldsymbol{v}_0$$

令

$$\boldsymbol{D} = \boldsymbol{d}, \quad \boldsymbol{E}_1 = \boldsymbol{v}_1 - \boldsymbol{v}_0, \quad \boldsymbol{E}_2 = \boldsymbol{v}_2 - \boldsymbol{v}_0, \quad \boldsymbol{T} = \boldsymbol{o} - \boldsymbol{v}_0$$

根据 Cramer 法则可知线性方程组的解为

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\det\begin{bmatrix} -\boldsymbol{D} & \boldsymbol{E}_1 & \boldsymbol{E}_2 \end{bmatrix}} \begin{bmatrix} \det\begin{bmatrix} \boldsymbol{T} & \boldsymbol{E}_1 & \boldsymbol{E}_2 \end{bmatrix} \\ \det\begin{bmatrix} -\boldsymbol{D} & \boldsymbol{T} & \boldsymbol{E}_2 \end{bmatrix} \\ \det\begin{bmatrix} -\boldsymbol{D} & \boldsymbol{E}_1 & \boldsymbol{T} \end{bmatrix} \end{bmatrix}$$

根据我们在线性代数中所学的知识有

$$\det\begin{bmatrix}\boldsymbol{A} & \boldsymbol{B} & \boldsymbol{C}\end{bmatrix} = -(\boldsymbol{A}\times\boldsymbol{C})\cdot\boldsymbol{B} = -(\boldsymbol{C}\times\boldsymbol{B})\cdot\boldsymbol{A}$$

于是上式可以改写为

$$\begin{bmatrix}t\\u\\v\end{bmatrix} = \frac{1}{(\boldsymbol{D}\times\boldsymbol{E}_2)\cdot\boldsymbol{E}_1}\begin{bmatrix}(\boldsymbol{T}\times\boldsymbol{E}_1)\cdot\boldsymbol{E}_2\\(\boldsymbol{D}\times\boldsymbol{E}_2)\cdot\boldsymbol{T}\\(\boldsymbol{T}\times\boldsymbol{E}_1)\cdot\boldsymbol{D}\end{bmatrix}$$

令 $\boldsymbol{P}=\boldsymbol{D}\times\boldsymbol{E}_2, \boldsymbol{Q}=\boldsymbol{T}\times\boldsymbol{E}_1$, 则有

$$t=\frac{\boldsymbol{Q}\cdot\boldsymbol{E}_2}{\boldsymbol{P}\cdot\boldsymbol{E}_1},\quad u=\frac{\boldsymbol{P}\cdot\boldsymbol{T}}{\boldsymbol{P}\cdot\boldsymbol{E}_1},\quad v=\frac{\boldsymbol{Q}\cdot\boldsymbol{D}}{\boldsymbol{P}\cdot\boldsymbol{E}_1}$$

最后判断 $u\geqslant 0, v\geqslant 0, u+v\leqslant 1$ 且 $t\geqslant 0$ 即可确定光线与三角面片是否相交, 如果相交则交点为 $\boldsymbol{r}(t)=\boldsymbol{o}+t\boldsymbol{d}$, 在三角面片上的重心坐标为 (u,v).

上述过程的算法流程如下:

1. 按照定义计算辅助向量 $\boldsymbol{D}, \boldsymbol{E}_1, \boldsymbol{E}_2, \boldsymbol{T}, \boldsymbol{P}, \boldsymbol{Q}$.

2. 判断行列式 $\boldsymbol{P}\cdot\boldsymbol{E}_1$ 是否接近于 0, 如果是则光线与三角面片平行, 不相交.

3. 计算参数 t, u, v 的值.

4. 判断 $u\geqslant 0, v\geqslant 0, u+v\leqslant 1$ 且 $t\geqslant 0$ 是否成立, 如果成立则光线与三角面片相交, 否则不相交.

3.4.3 空间加速结构

如果每一次光线求交都要遍历场景中的所有三角面片, 计算效率将非常低下. 为了提升光线求交的效率, 我们通常会使用空间加速结构对场景进行划分, 从而减少每次求交时需要检查的三角面片数量. 下面介绍几种常见的空间加速结构. 从原理上而言, 它们都是将空间划分为若干个区域, 先判断光线是否与区域相交, 如果相交才会进一步检查该区域内的物体, 从而避免了很多不必要的求交计算. 它们之间的区别主要在于区域划分的方式.

层次包围盒

层次包围盒 (Hierarchical Bounding Boxes) 是一种通过构建包围盒[8]树来组织场景中物体的空间加速结构. 包围盒树的每个节点表示一个包围盒, 其子节点表示该包围盒内包含的更小的包围盒或物体. 在光线求交时, 首先检查光线与根节点的包围盒是否相交, 如果相交则递归地检查子节点, 否则跳过该子树.

容易看出建立查找树可以使得求交的时间复杂度从 $O(n)$ 降低到 $O(\log n)$, 因此对于渲染效率有显著提升. 我们现在来介绍层次包围盒构建的具体办法:

1. 计算当前节点包含的所有物体的包围盒并存储.

2. 按照一定的划分策略将物体划分为两个子集, 作为该节点的子节点.

[8]一般而言, 包围盒是一个包含其中所有物体的最小的长方体. 光线与包围盒的求交可以用前面介绍的 **Slabs** 算法实现.

3. 递归地对每个子节点重复上述过程, 直到满足终止条件 (如节点包含的物体数量小于某个阈值).

在光线求交时, 我们可以按照如下流程进行:

1. 从根节点开始, 检查光线与当前节点的包围盒是否相交.

2. 如果相交, 则递归地检查子节点; 如果不相交, 则跳过该子树.

3. 当到达叶子节点时, 对该节点包含的所有物体进行求交测试, 记录光源到交点的距离.

4. 递归结束后对最近的交点进行处理, 计算颜色等信息, 然后返回.

需要注意的是, 在递归的过程中并不需要对每个交点进行精确的计算 (例如插值计算颜色等信息), 这会严重影响效率, 只需要更新最近的交点即可.

八叉树

我们在前面已经介绍过八叉树这一数据结构, 它也可以用作空间加速. 八叉树的每个节点表示一个立方体空间, 其子节点表示该立方体被划分后的八个子立方体. 在光线求交时, 首先检查光线与根节点的立方体是否相交, 如果相交则递归地检查子节点, 否则跳过该子树.

BSP 树和 KD 树

BSP 树 (Binary Space Partitioning Tree) 和 KD 树 (K-Dimensional Tree) 是两种基于空间划分的加速结构. 它们通过选择一个平面[9]将空间划分为两个子空间, 并递归地对每个子空间进行划分, 直到满足终止条件. 在光线求交时, 首先计算光线与当前节点的划分平面的交点, 据此可以判断它与该平面划分出的两个子节点的相交关系, 然后递归地检查相交的子节点.

两者的区别主要在于 KD 树的划分平面要求平行于 xy, yz 或 xz 平面 (也即只能是 $x = c$, $y = c$ 或 $z = c$ 中的一种), 而 BSP 树的划分平面可以是任意方向的平面. 一般而言, KD 树的构建和求交效率更高, 而 BSP 树仅在一些特殊的场景下使用 (例如场景中有很多倾斜的物体).

[9]与 BVH 树不同的是 BSP 树和 KD 树并不需要重新计算每个子节点的包围盒, 子节点的几何形状已经由划分平面天然地确定了.

3.5 高级渲染

我们需要用更精确的模型描述和模拟光在环境中的传输过程.

3.5.1 辐射度量学

为了描述光的传输过程, 我们需要引入一些**辐射度量学 (Radiometry)** 的概念. 辐射度量学提供了一系列思想和数学工具, 来描述光的传播和反射. 它构成了推导本章余下部分将使用的渲染算法的基础.

基本假设

首先, 我们在渲染时总是假设几何光学成立, 因此有如下基本假设:

定理 5.1 光的基本假设

1. **线性**: 光的传输是线性的, 即多束光的叠加等于各束光的单独传输之和.
2. **能量守恒**: 光经过散射后, 散射光的能量不大于入射光的能量.
3. **无偏振**: 忽略光的偏振效应.
4. **稳态**: 环境中光的分布不随时间变化.

基本物理量

在辐射度量学中, 我们主要关心以下物理量: 通量, 辐照度, 强度和辐射率. 需要注意的是这些度量都依赖于波长, 但在本章的讨论中我们通常不表明这一依赖关系.

定义 5.2 通量 通量 (Flux)Φ 表示单位时间内通过某一给定面的光的能量, 即

$$\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t}$$

光源的总发射常用通量表示, 因为从上述定义可以看出通量与我们选取的截面无关.

定义 5.3 辐照度与辐射出射度 辐照度 (Irradiance)E 表示单位面积上接收到的通量, **辐射出射度 (Radiant Exitance)**M 表示单位面积上发出的通量. 对于空间中给定表面上的一点 $\boldsymbol{p}$, 其辐照度可以通过微分定义如下:

$$E(\boldsymbol{p}) = \frac{\mathrm{d}\Phi(\boldsymbol{p})}{\mathrm{d}A}$$

需要注意的是, 辐照度 (包括辐射出射度) 都是基于表面定义的, 因此一般我们都只在物体的表面考虑辐照度. 如果是空间中的某一点, 那么一般需要额外确定所取的面积元的方向 (即确定一个虚平面).

定义 5.4 辐射率 辐射率 (Radiance) L 相比辐照度考虑了光在不同方向上的分布. 给定平面上某一点 $\boldsymbol{p}$ 在方向 $\boldsymbol{\omega}$ 上的辐射率可以定义如下:

$$L(\boldsymbol{p},\boldsymbol{\omega})=\frac{\mathrm{d}^2\Phi(\boldsymbol{p},\boldsymbol{\omega})}{\mathrm{d}A^{\perp}\mathrm{d}\boldsymbol{\omega}}$$

其中 $\mathrm{d}A^{\perp}=\mathrm{d}A\cos\theta$ 表示垂直于 $\boldsymbol{\omega}$ 的面积元, 这里 θ 为 $\boldsymbol{\omega}$ 与平面在 $\boldsymbol{p}$ 处的法向量 $\boldsymbol{n}$ 的夹角.

也即, 给定点 $\boldsymbol{p}$ 处的指定方向 $\boldsymbol{\omega}$ 上的辐射率 $L(\boldsymbol{p},\boldsymbol{\omega})$ 表示 $\boldsymbol{\omega}$ 上单位立体角和垂直 $\boldsymbol{\omega}$ 上单位面积上的通量.

在所有这些辐射度量中, 辐射率将在本章中最频繁地使用. 某种意义上它是所有辐射度量中最基本的, 如果给定了辐射率, 那么所有其他值都可以通过对辐射率在区域和方向上的积分来计算.

辐射率的另一个良好属性是在通过空间中的射线上保持不变. 这意味着我们只需考虑 $L(\boldsymbol{p},\boldsymbol{\omega})$ 在物体表面上的取值, 而不必考虑光在空间中传播的过程.

在上述所有物理量中, 比较重要的是辐射率 L 与辐照度 E 的关系. 对于给定面上的一点 $\boldsymbol{p}$ 以及此处的法向量 $\boldsymbol{n}$, 入射到该点的辐照度 $E(\boldsymbol{p})$ 可以通过该点在各个方向上的辐射率积分得到 (这里的下标 i 表示入射光, 以后的下标 o 表示出射光):

$$E(\boldsymbol{p})=\int_{\Omega}\mathrm{d}E_{\boldsymbol{\omega}}(\boldsymbol{p})=\int_{\Omega}L_i(\boldsymbol{p},\boldsymbol{\omega}_i)\cos\theta_i\mathrm{d}\boldsymbol{\omega}_i$$

其中 Ω 表示以 $\boldsymbol{p}$ 为顶点的半空间[10], 通常选择法向量 $\boldsymbol{n}$ 对应的半球, 记作 $H^2(\boldsymbol{n})$. θ_i 表示入射方向 $\boldsymbol{\omega}_i$ 与法向量 $\boldsymbol{n}$ 之间的夹角.

3.5.2 反射与折射模型

当光线入射到表面时, 表面会散射光线, 将部分光线反射回环境中, 部分光线则射入物体中 (如果物体是半透明的). 建立这种模型需要描述两种主要效应: 反射/折射光的光谱分布和方向分布.

双向反射分布函数 BRDF

为了描述表面对光的反射特性, 我们引入**双向反射分布函数 (Bidirectional Reflectance Distribution Function, BRDF)**. BRDF 描述了入射光线和出射光线之间的关系. 具体而言, 对于物体表面上的一点 $\boldsymbol{p}$, 我们希望知道沿 $\boldsymbol{\omega}_i$ 方向入射光的辐照度 $E_{\boldsymbol{\omega}_i}(\boldsymbol{p})$ 在沿 $\boldsymbol{\omega}_o$ 方向上造成的出射光的辐射率 $L_o(\boldsymbol{p},\boldsymbol{\omega}_o)$.

根据几何光学的线性假设, 在 $\boldsymbol{\omega}_o$ 方向上出射光的辐照率, 应当等于各入射光造成 $\boldsymbol{\omega}_o$ 方向上反射光的总和; 此外, 增强入射光的辐照度 $E_{\boldsymbol{\omega}_i}(\boldsymbol{p})$, 也应当线性地增强出射光的辐射率. 我们把上述关系表示为积分形式:

$$L_o(\boldsymbol{p},\boldsymbol{\omega}_o)=\int_{\Omega}k\mathrm{d}E_{\boldsymbol{\omega}_i}(\boldsymbol{p})=\int_{\Omega}kL_i(\boldsymbol{p},\boldsymbol{\omega}_i)\cos\theta_i\mathrm{d}\boldsymbol{\omega}_i$$

于是

$$\mathrm{d}L_o(\boldsymbol{p},\boldsymbol{\omega}_o)=kL_i(\boldsymbol{p},\boldsymbol{\omega}_i)\cos\theta_i\mathrm{d}\boldsymbol{\omega}_i$$

根据这一关系, 我们可以定义 BRDF 如下:

[10] 如果 $\boldsymbol{\omega}$ 与法向量 $\boldsymbol{n}$ 的夹角大于 $\pi/2$, 这意味着这部分光线会被遮挡, 因此不用考虑.

定义 5.5 双向反射分布函数 给定物体表面上某一点 $\boldsymbol{p}$, 入射方向 $\boldsymbol{\omega}_i$ 和出射方向 $\boldsymbol{\omega}_o$,BRDF 定义为:

$$f_r(\boldsymbol{p},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o)=\frac{\mathrm{d}L_o(\boldsymbol{p},\boldsymbol{\omega}_o)}{L_i(\boldsymbol{p},\boldsymbol{\omega}_i)\cos\theta_i\mathrm{d}\boldsymbol{\omega}_i}$$

其中 θ_i 是入射方向与表面法线之间的夹角.

基于物理的 BRDF 有两个重要性质:

定理 5.6 BRDF 的性质

(1) Helmholtz 可逆性:BRDF 在入射和出射方向上是对称的, 即

$$f_r(\boldsymbol{p},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o)=f_r(\boldsymbol{p},\boldsymbol{\omega}_o,\boldsymbol{\omega}_i)$$

(2) 能量守恒: 对于任意入射方向 $\boldsymbol{\omega}_i$, 出射光的总能量不大于入射光的能量, 即

$$\forall\boldsymbol{\omega}_i,\quad \int_\Omega f_r(\boldsymbol{p},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o)\cos\theta_o\mathrm{d}\boldsymbol{\omega}_o\leqslant 1$$

双向透射分布函数 BTDF

描述透射光分布的**表面双向透射分布函数 (Bidirectional Transmittance Distribution Function, BTDF)** 与 BRDF 类似, 其定义为:

$$f_t(\boldsymbol{p},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o)=\frac{\mathrm{d}L_o(\boldsymbol{p},\boldsymbol{\omega}_o)}{L_i(\boldsymbol{p},\boldsymbol{\omega}_i)\cos\theta_i\mathrm{d}\boldsymbol{\omega}_i}$$

其中各符号的定义相同, 但入射光和出射光的方向分居表面两侧.

需要注意的是, BTDF 并不满足 Helmholtz 可逆性. 可以证明, 如果入射光和出射光的介质的折射率分别为 η_i 和 η_o, 那么 BTDF 函数满足如下关系:

$$\eta_i^2 f_t\left(\boldsymbol{p},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o\right)=\eta_o^2 f_t\left(\boldsymbol{p},\boldsymbol{\omega}_o,\boldsymbol{\omega}_i\right)$$

双向散射分布函数 BSDF

为了方便起见, 我们把 BRDF 和 BTDF 合并为一个统一的函数, 称为**双向散射分布函数 (Bidirectional Scattering Distribution Function, BSDF)**. BSDF 定义为:

$$f(\boldsymbol{p},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o)=\begin{cases}f_r(\boldsymbol{p},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o), & \text{如果}\boldsymbol{\omega}_i,\boldsymbol{\omega}_o\text{在表面同侧}\\ f_t(\boldsymbol{p},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o), & \text{如果}\boldsymbol{\omega}_i,\boldsymbol{\omega}_o\text{在表面异侧}\end{cases}$$

这样, 在 $\boldsymbol{p}$ 沿 $\boldsymbol{\omega}_o$ 出射光的辐射率就可以表示为

$$L_o(\boldsymbol{p},\boldsymbol{\omega}_o)=\int_{S^2} f(\boldsymbol{p},\boldsymbol{\omega}_i,\boldsymbol{\omega}_o)L_i(\boldsymbol{p},\boldsymbol{\omega}_i)\left|\cos\theta_i\right|\mathrm{d}\boldsymbol{\omega}_i$$

这里的积分区域 S^2 为 $\boldsymbol{p}$ 为球心的球面. 这是综合考虑同面的反射光和异面的透射光后的结果.

双向散射表面反射率分布函数 BSSRDF

在更复杂的情形下, 光线入射到表面后, 可能会在物体内部传播一段距离后, 再从物体的另一个位置出射. 为了描述这种现象, 我们引入**双向散射表面反射率分布函数 (Bidirectional Surface Scattering Reflectance Distribution Function, BSSRDF)**. BSSRDF 定义为:

$$S(\boldsymbol{p}_i, \boldsymbol{\omega}_i, \boldsymbol{p}_o, \boldsymbol{\omega}_o) = \frac{\mathrm{d}L_o(\boldsymbol{p}_o, \boldsymbol{\omega}_o)}{\mathrm{d}\Phi(\boldsymbol{p}_i, \boldsymbol{\omega}_i)} = \frac{\mathrm{d}L_o(\boldsymbol{p}_o, \boldsymbol{\omega}_o)}{L_i(\boldsymbol{p}_i, \boldsymbol{\omega}_i)\cos\theta_i \mathrm{d}\boldsymbol{\omega}_i \mathrm{d}A_i}$$

于是, 计算出射光的方程由二重积分变成了四重积分:

$$L_o(\boldsymbol{p}, \boldsymbol{\omega}_o) = \int_A \int_{H^2(\boldsymbol{n})} S(\boldsymbol{p}_i, \boldsymbol{\omega}_i, \boldsymbol{p}_o, \boldsymbol{\omega}_o) L_i(\boldsymbol{p}_i, \boldsymbol{\omega}_i) \cos\theta_i \mathrm{d}\boldsymbol{\omega}_i \mathrm{d}A_i$$

其中 A 是 $\boldsymbol{p}_o$ 的邻域 (一般而言 S 随着 $\boldsymbol{p}_i$ 与 $\boldsymbol{p}_o$ 的远离而减小, 因此只要考虑其附近的一个区域即可).

由于积分维度升高, BSSRDF 的计算量也大大增加. 因此在实际应用中, 我们通常只在需要考虑次表面散射的情况下使用 BSSRDF, 否则都使用 BSDF.

3.5.3 渲染方程

渲染方程的基本形式

我们已经通过辐射度量学对反射/折射光的物理描述. 现在再考虑物体本身的发光项 $L_e(\boldsymbol{p}, \boldsymbol{\omega}_o)$, 我们就得到了下述方程:

$$L_o(\boldsymbol{p}, \boldsymbol{\omega}_o) = L_e(\boldsymbol{p}, \boldsymbol{\omega}_o) + \int_{S^2} f(\boldsymbol{p}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L_i(\boldsymbol{p}, \boldsymbol{\omega}_i) \left|\cos\theta_i\right| \mathrm{d}\boldsymbol{\omega}_i$$

定义 5.7 渲染方程 上述方程称为**渲染方程 (Rendering Equation)**.

在空间中, 每一束入射光 $L_i(\boldsymbol{p}, \boldsymbol{\omega}_i)$ 都来源于另一物体上另一点 $\boldsymbol{p}'$ 沿方向 $-\boldsymbol{\omega}_i$ 的出射光 $L_o(\boldsymbol{p}', -\boldsymbol{\omega}_i)$. 因此, 渲染方程实际上是递归定义的.

渲染的本质问题, 实际上就是求解上述积分方程. 直接求解是非常困难的, 因此我们通常采取各种数值方法进行计算.

渲染方程的面积分形式

我们现在把渲染方程从一点的形式继续扩展至整个场景. 假定出射光由 $\boldsymbol{p}'$ 指向 $\boldsymbol{p}$, 则对于 $\boldsymbol{p}'$ 的渲染方程为

$$L_o(\boldsymbol{p}', \boldsymbol{\omega}_o) = L_e(\boldsymbol{p}', \boldsymbol{\omega}_o) + \int_{S^2} f(\boldsymbol{p}', \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L_i(\boldsymbol{p}', \boldsymbol{\omega}_i) \left|\cos\theta_i\right| \mathrm{d}\boldsymbol{\omega}_i$$

前面我们已经提到, 每一束入射光 $L_i(\boldsymbol{p}', \boldsymbol{\omega}_i)$ 都来源于另一点 $\boldsymbol{p}''$ 沿方向 $-\boldsymbol{\omega}_i$ 的出射光 $L_o(\boldsymbol{p}'', -\boldsymbol{\omega}_i)$. 定义**路径追踪函数** $t(\boldsymbol{q}, \boldsymbol{\omega})$ 表示从点 $\boldsymbol{q}$ 沿 $\boldsymbol{\omega}$ 方向上与空间中各物体的第一个交点 $\boldsymbol{q}'$, 那么 $\boldsymbol{p}'$ 处的入射光的辐射率可以表示为

$$L_i(\boldsymbol{p}', \boldsymbol{\omega}_i) = L_o(t(\boldsymbol{p}', \boldsymbol{\omega}_i), -\boldsymbol{\omega}_i)$$

此外, 如果场景不是封闭的, 并且光线 $(\boldsymbol{q},\boldsymbol{\omega})$ 与任何物体都不相交, 就定义 $t(\boldsymbol{q},\boldsymbol{\omega})=\boldsymbol{\Lambda}$, 并且 $L_o(\boldsymbol{\Lambda},\boldsymbol{\omega})=0$. 由此, 我们可以把渲染方程改写为下述形式:

$$L_o(\boldsymbol{p}',\boldsymbol{\omega}_o)=L_e(\boldsymbol{p}',\boldsymbol{\omega}_o)+\int_{S^2} f(\boldsymbol{p}',\boldsymbol{\omega}_i,\boldsymbol{\omega}_o)L_o(t(\boldsymbol{p}',\boldsymbol{\omega}_i),-\boldsymbol{\omega}_i)\left|\cos\theta_i\right|\mathrm{d}\boldsymbol{\omega}_i$$

现在, 我们就无需考虑入射辐射率 L_i, 而仅需要考虑出射辐射率 L_o. 当然, 它出现在等式的两边, 并且积分号中也存在, 因此我们的任务仍然不简单.

上述方程比较复杂的原因之一是路径追踪函数 $t(\boldsymbol{p},\boldsymbol{\omega})$ 是隐式的. 我们尝试把上述积分改写成面积分. 记

$$L(\boldsymbol{p}'\to\boldsymbol{p})=L_o(\boldsymbol{p}',\boldsymbol{\omega}_o)$$

其中 $\boldsymbol{\omega}_o$ 即由 $\boldsymbol{p}'$ 指向 $\boldsymbol{p}$ 的向量. 我们把 $\boldsymbol{p}'$ 处的 BSDF 函数写成类似的形式:

$$f(\boldsymbol{p}''\to\boldsymbol{p}'\to\boldsymbol{p})=f(\boldsymbol{p}',\boldsymbol{\omega}_i,\boldsymbol{\omega}_o)$$

其中 $\boldsymbol{\omega}_i$ 由 $\boldsymbol{p}'$ 指向 $\boldsymbol{p}''$.

现在, 最重要的步骤是把对角度的积分转换为对面积的积分. 根据立体角的定义可知 p' 处入射立体角的微元 $\mathrm{d}\boldsymbol{\omega}_i$ 与 $\boldsymbol{p}''$ 处的面积微元 $\mathrm{d}A(\boldsymbol{p}'')$ 的关系为

$$\mathrm{d}\boldsymbol{\omega}_i=\frac{\left|\cos\theta'\right|}{\left|\left|\boldsymbol{p}''-\boldsymbol{p}'\right|\right|^2}\mathrm{d}A(\boldsymbol{p}'')$$

其中 θ' 为 $\boldsymbol{\omega}_i$ 与 $\boldsymbol{p}''$ 处法向量 $\boldsymbol{n}''$ 的夹角. 定义可见函数 $V(\boldsymbol{q}\leftrightarrow\boldsymbol{q}')$, $\boldsymbol{q},\boldsymbol{q}'$ 之间没有阻挡时取 1, 否则取 0. 把上述关系连同渲染方程中原有的角度项记为

$$G(\boldsymbol{p}''\leftrightarrow\boldsymbol{p}')=V(\boldsymbol{p}''\leftrightarrow\boldsymbol{p}')\frac{\left|\cos\theta\cos\theta'\right|}{\left|\left|\boldsymbol{p}''-\boldsymbol{p}'\right|\right|^2}$$

这样, 渲染方程最终可以写做如下形式:

$$L(\boldsymbol{p}'\to\boldsymbol{p})=L_e(\boldsymbol{p}'\to\boldsymbol{p})+\int_A f(\boldsymbol{p}''\to\boldsymbol{p}'\to\boldsymbol{p})L(\boldsymbol{p}''\to\boldsymbol{p}')G(\boldsymbol{p}''\leftrightarrow\boldsymbol{p}')\mathrm{d}A(\boldsymbol{p}'')$$

其中 A 是场景中的全部表面.

渲染方程的路径积分形式

我们把光传输方程展开如下:

$$\begin{aligned}L(\boldsymbol{p}_1\to\boldsymbol{p}_0)=&L_e(\boldsymbol{p}_1\to\boldsymbol{p}_0)\\&+\int_A L_e(\boldsymbol{p}_2\to\boldsymbol{p}_1)f(\boldsymbol{p}_2\to\boldsymbol{p}_1\to\boldsymbol{p}_0)G(\boldsymbol{p}_2\leftrightarrow\boldsymbol{p}_1)\mathrm{d}A(\boldsymbol{p}_2)\\&+\int_A\int_A L_e(\boldsymbol{p}_3\to\boldsymbol{p}_2)f(\boldsymbol{p}_3\to\boldsymbol{p}_2\to\boldsymbol{p}_1)G(\boldsymbol{p}_3\leftrightarrow\boldsymbol{p}_2)\mathrm{d}A(\boldsymbol{p}_3)\mathrm{d}A(\boldsymbol{p}_2)\\&+\cdots\end{aligned}$$

等式右边的每一项都表示一条长度递增的路径. 更简洁地, 定义

$$P(\bar{\boldsymbol{p}_n}) = \underbrace{\int_A \cdots \int_A}_{n-1\text{重积分}} L_e(\boldsymbol{p}_n \to \boldsymbol{p}_{n-1}) \left(\prod_{i=1}^{n-1} f(\boldsymbol{p}_{i+1} \to \boldsymbol{p}_i \to \boldsymbol{p}_{i-1}) G(\boldsymbol{p}_{i+1} \leftrightarrow \boldsymbol{p}_i) \right) \mathrm{d}A(\boldsymbol{p}_2) \cdots \mathrm{d}A(\boldsymbol{p}_n)$$

表示具有 $n+1$ 个点的路径 $\boldsymbol{p}_n \to \boldsymbol{p}_{n-1} \to \cdots \to \boldsymbol{p}_1 \to \boldsymbol{p}_0$ 所贡献的辐射率. 于是上述无穷求和可以写做

$$L(\boldsymbol{p}_1 \to \boldsymbol{p}_0) = \sum_{n=1}^{\infty} P(\bar{\boldsymbol{p}}_n)$$

现在, 对于给定的 n, 只需要在空间中随机地采样 n 个点, 就可以估计出 $P(\bar{\boldsymbol{p}}_n)$ 的值.

3.5.4 蒙特卡洛积分

我们回到前面的渲染方程的角度积分形式.

$$L_o(\boldsymbol{p}, \boldsymbol{\omega}_o) = L_e(\boldsymbol{p}, \boldsymbol{\omega}_o) + \int_{S^2} f(\boldsymbol{p}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L_i(\boldsymbol{p}, \boldsymbol{\omega}_i) \left|\cos\theta_i\right| \mathrm{d}\boldsymbol{\omega}_i$$

直接求解上述积分显然是很麻烦的. 我们更希望用采样的方式解决积分问题. 我们先从简单的情形考虑采样的办法. 例如, 对于函数 $f(x)$ 和积分 $\int_a^b f(x)\mathrm{d}x$. 我们在 $[a,b]$ 区间采样了 N 个点 $x_1, \cdots, x_N$, 采样的概率密度函数为 $p(x)$. 令

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$

于是有

$$E(F_N) = \frac{1}{N} \sum_{i=1}^{N} \int_a^b \left(\frac{f(x_i)}{p(x_i)} \right) p(x_i) \mathrm{d}x_i = \int_a^b f(x) \mathrm{d}x$$

于是我们可以把 F_N 作为上述积分的估计值. 同样地, 在 S^2 中随机地选取 N 个立体角 $\omega_1, \cdots, \omega_N$, 则有

$$L_o(\boldsymbol{p}, \boldsymbol{\omega}_o) = L_e(\boldsymbol{p}, \boldsymbol{\omega}_o) + \frac{1}{N} \sum_{j=1}^{N} \frac{f(\boldsymbol{p}, \boldsymbol{\omega}_j, \boldsymbol{\omega}_o) L_i(\boldsymbol{p}, \boldsymbol{\omega}_j) \left|\cos\theta_j\right|}{p(\boldsymbol{\omega}_j)}$$

于是渲染的过程就可以写做递归的形式:

```
def shade(p, wo)
    Randomly choose N directions wi
    Lo = 0.0
    for each wi
        Trace a ray r(p, wi)
        if r hit a light
            Lo += (1 / N) * L_i * f_r * cos / p(wi)
        else if r hit an obj at q
            Lo += (1 / N) * shade(q, -wi) * f_r * cos / p(wi)
    return Lo
```

第四章　仿真

4.1 物理模拟

万事万物的运动都遵循一定的原理. 经过物理学家们长久以来的努力, 我们已经可以用一系列物理方程描述物体的运动, 但这些方程往往复杂而难以求解. 于是, 人们开始寻求这些方程的近似解, 并且希望这些近似解足够精确, 以至于可以用来模拟现实世界中的物理现象. 这种通过数值方法求解物理方程, 并用计算机进行模拟的过程, 就称为**物理模拟 (Physical Simulation)**.

一般而言, 物理模拟的范围主要是宏观低速物体的运动, 因此主要使用牛顿运动定律进行运动的计算. 牛顿运动定律为

$$\boldsymbol{f} = m\boldsymbol{a} = m\frac{\mathrm{d}\boldsymbol{v}}{\mathrm{d}t} = m\frac{\mathrm{d}^2\boldsymbol{x}}{\mathrm{d}t^2}$$

在上述公式中, 我们的研究对象是一个质点, 并且它仅受一个力. 这个例子实在过于简单, 无法代表一般的情形. 通常而言, 物体包含无穷多粒子, 并且往往存在多个外力以及复杂的内力. 为了将这样的复杂的系统变为数值求解问题, 我们首先需要对物体进行离散化处理.

因此, 物理模拟的过程大致可以分为三个步骤: **空间离散化**, **时间离散化**以及**数值计算**. 下面就来介绍物理模拟在各种模型下的实现方法.

4.2 弹簧质点模型

在本节中, 我们将以弹簧质点模型为例介绍物理模拟的一般方法. 同时, 这一模型也是计算机图形学中常用的物理模拟模型之一, 研究其仿真办法也具有相当的实际意义.

4.2.1 空间离散化

通过一定的方法, 我们可以把物体抽象成由有限个质点组成的系统. 单个质点的状态由其位置 $\boldsymbol{x}_i$ 和速度 $\boldsymbol{v}_i$, 而为了考虑其运动状态的变化, 我们还需通过其受的力 $\boldsymbol{f}_i$ 和质量 m_i 得出其加速度. 因此, 一个质点模型可以由列表 $\{\boldsymbol{x}_i, \boldsymbol{v}_i, \boldsymbol{f}_i, m_i\}_{i=1}^N$ 表示.

在抽象成质点的情形下, 弹簧质点模型假定质点之间存在弹簧以传递内力. 质点 i 受的外力可以写做

$$\boldsymbol{f}_i = \sum_{j\in\Omega(i)} \boldsymbol{f}_{ij} + f_i^{\mathrm{ext}}$$

其中 f_i^{ext} 表示质点 i 受到的除去弹簧弹力之外的外力. 质点 i 与其邻接的质点 $j \in \Omega(i)$ 通过弹簧相连, 弹簧对质点 i 的弹力 $\boldsymbol{f}_{ij}$ 可以通过胡克定律计算:

$$\boldsymbol{f}_{ij} = -k_{ij}(|\boldsymbol{x}_i - \boldsymbol{x}_j| - l_{ij})\frac{\boldsymbol{x}_i - \boldsymbol{x}_j}{|\boldsymbol{x}_i - \boldsymbol{x}_j|}$$

其中 k_{ij} 为弹簧的弹性系数, l_{ij} 为弹簧的原长.

4.2.2 时间离散化

物体的运动状态是随时间连续变化的. 同样地, 我们也需要对时间进行离散化. 一般而言, 我们会在时间区间上均匀采样, 采样间隔为 Δt. 我们把 N 个质点的位置 $\boldsymbol{x}$ 和速度 $\boldsymbol{v}$ 写做 $3N$ 维的堆叠向量的形式, 在第 k 次采样 t_k 时有:

$$\boldsymbol{x}(t_k) = \begin{bmatrix} \boldsymbol{x}_1(t_k) \\ \cdots \\ \boldsymbol{x}_n(t_k) \end{bmatrix}, \quad \boldsymbol{v}(t_k) = \begin{bmatrix} \boldsymbol{v}_1(t_k) \\ \cdots \\ \boldsymbol{v}_n(t_k) \end{bmatrix}$$

其中 $\boldsymbol{x}_i(t)$ 表示 t 时刻质点 i 的位置, $\boldsymbol{v}_i(t)$ 同理. 根据简单的运动学知识, 我们可以得到下一次采样时的位置矩阵 $\boldsymbol{x}(t_{k+1})$ 和 $\boldsymbol{v}(t_{k+1})$:

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_k) + \int_{t_k}^{t_{k+1}} \boldsymbol{v}(t)\mathrm{d}t$$

$$\boldsymbol{v}(t_{k+1}) = \boldsymbol{v}(t_k) + \boldsymbol{M}^{-1} \int_{t_k}^{t_{k+1}} \boldsymbol{f}(t, \boldsymbol{x}(t), \boldsymbol{v}(t))\mathrm{d}t$$

其中 $\boldsymbol{M} = \mathrm{diag}\{m_1, m_1, m_1, \cdots, m_N, m_N, m_N\}$ 为质点的质量矩阵.

现在, 我们的目标就是求解上述积分方程. 典型的计算时间积分的办法有**显式欧拉 (Explicit Euler)** 积分和**隐式欧拉 (Implicit Euler)** 积分.

显式欧拉积分

在复杂的情形中, 上述积分方程可能不存在解析表达. 好在当取样的时间间隔较小时, 我们可以将被积函数视作常数, 取值即为每个时间间隔开始的时刻, 于是即有

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_k) + \boldsymbol{v}(t_k)\Delta t$$

$$\boldsymbol{v}(t_{k+1}) = \boldsymbol{v}(t_k) + \boldsymbol{M}^{-1}\boldsymbol{f}(t_k)\Delta t$$

这样, 对于弹簧质点系统的模拟就归结于上述迭代过程. 显式欧拉积分的优点在于计算和逻辑简单, 每一步只需要计算当前时刻的力即可.

然而, 这样的简单近似也有其缺点. 当时间步长较大时, $\boldsymbol{v}(t)$ 和 $\boldsymbol{f}(t)$ 可能在时间间隔内变化较大, 从而导致解得的 $\boldsymbol{x}(t)$ 和 $\boldsymbol{v}(t)$ 偏离实际情况, 使得系统能量升高. 这又进一步导致 $\boldsymbol{v}(t)$ 和 $\boldsymbol{f}(t)$ 的偏离, 最终可能导致系统发散, 无法继续模拟下去.

解决上述问题的一种办法是使用更小的时间步长, 但这会大大增加计算量, 反而得不偿失. 因此, 我们需要一种更稳定的办法.

隐式欧拉积分

隐式欧拉积分的推导 与显式欧拉积分不同, 隐式欧拉积分在计算下一个时间步长的状态时, 使用的是每个时间间隔结束的时刻的力. 即有

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_k) + \boldsymbol{v}(t_{k+1})\Delta t$$

$$\boldsymbol{v}(t_{k+1}) = \boldsymbol{v}(t_k) + \boldsymbol{M}^{-1}\boldsymbol{f}(t_{k+1})\Delta t$$

这样, 我们需要解一个隐式方程组. 将 $\boldsymbol{v}(t_{k+1})$ 代入 $\boldsymbol{x}(t_{k+1})$ 中可得

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_k) + \left(\boldsymbol{v}(t_k) + \boldsymbol{M}^{-1}\boldsymbol{f}(t_{k+1})\Delta t\right)\Delta t$$

为了方便考虑, 我们把力 $\boldsymbol{f}$ 分成内力和外力两部分:

$$\boldsymbol{f}(t_{k+1}) = \boldsymbol{f}_{\text{int}}(t_{k+1}) + \boldsymbol{f}_{\text{ext}}$$

其中 $\boldsymbol{f}_{\text{ext}}$ 作为外力与时间和质点的位置无关. 于是进一步整理可得

$$\boldsymbol{x}(t_{k+1}) = \left[\boldsymbol{x}(t_k) + \Delta t\boldsymbol{v}(t_k) + (\Delta t)^2\boldsymbol{M}^{-1}\boldsymbol{f}_{\text{ext}}\right] + (\Delta t)^2\boldsymbol{M}^{-1}\boldsymbol{f}_{\text{int}}(t_{k+1})$$

等号右边前半部分全部为已知量, 不妨记作 $\boldsymbol{y}(t_k)$. 于是方程组即为

$$\boldsymbol{x}(t_{k+1}) - \boldsymbol{y}(t_k) - (\Delta t)^2\boldsymbol{M}^{-1}\boldsymbol{f}_{\text{int}}(t_{k+1}) = \boldsymbol{0}$$

接下来, 我们需要将内力 $\boldsymbol{f}_{\text{int}}(t_{k+1})$ 表示成位置 $\boldsymbol{x}(t_{k+1})$ 的函数. 对于质点 i 与 j 之间的弹簧, 其弹性势能

$$E_{ij} = \frac{1}{2}k_{ij}\left(||\boldsymbol{x}_j - \boldsymbol{x}_i|| - l_{ij}\right)^2$$

于是

$$\boldsymbol{f}_{ij} = -\frac{\partial E_{ij}}{\partial \boldsymbol{x}_i}$$

从而对于质点 i, 其内力 $\boldsymbol{f}_{i,\text{int}}$ 就可以写作

$$\boldsymbol{f}_{i,\text{int}} = -\sum_{j\in\Omega(i)}\frac{\partial E_{ij}}{\partial \boldsymbol{x}_i} = -\frac{\partial E}{\partial \boldsymbol{x}_i}$$

其中 E 为弹簧系统的总能量 (与 i 不相连的弹簧的势能对 $\boldsymbol{x}_i$ 的梯度为 $\boldsymbol{0}$, 因此可以并入求和项中), 其自变量为各质点的位置向量 $\boldsymbol{x}$. 将所有质点的内力堆叠起来, 我们就有

$$\boldsymbol{x}(t_{k+1}) - \boldsymbol{y}(t_k) + (\Delta t)^2\boldsymbol{M}^{-1}\frac{\partial E(\boldsymbol{x}(t_{k+1}))}{\partial \boldsymbol{x}} = \boldsymbol{0}$$

于是, 可以构造辅助函数

$$g(\boldsymbol{x}) = \frac{1}{2(\Delta t)^2}(\boldsymbol{x} - \boldsymbol{y}(t_k))^{\text{t}}\boldsymbol{M}(\boldsymbol{x} - \boldsymbol{y}(t_k)) + E(\boldsymbol{x})$$

则有

$$\frac{\partial g(\boldsymbol{x})}{\partial \boldsymbol{x}} = \boldsymbol{x}(t_{k+1}) - \boldsymbol{y}(t_k) + (\Delta t)^2\boldsymbol{M}^{-1}\frac{\partial E(\boldsymbol{x}(t_{k+1}))}{\partial \boldsymbol{x}} = \boldsymbol{0}$$

于是, 目标就转化为最小化问题

$$\boldsymbol{x}(t_{k+1}) = \arg\min_{\boldsymbol{x}} g(\boldsymbol{x})$$

最小化问题的数值求解 我们采取牛顿法解决上述问题. 牛顿法的基本思想是在每轮迭代中使用一个二次函数逼近目标函数, 然后求解该二次函数的极小值作为下一次迭代的点. 迭代过程每次通过尝试解 $\boldsymbol{x}^{(i)}$ 计算下一次的解 $\boldsymbol{x}^{(i+1)}$. 对 $g(\boldsymbol{x})$ 在 $\boldsymbol{x}^{(i)}$ 处进行二阶泰勒展开, 有

$$g(\boldsymbol{x}) = g(\boldsymbol{x}^{(i)}) + \nabla g(\boldsymbol{x}^{(i)}) \cdot (\boldsymbol{x} - \boldsymbol{x}^{(i)}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{x}^{(i)})^{\mathrm{t}} \boldsymbol{H}_g(\boldsymbol{x}^{(i)})(\boldsymbol{x} - \boldsymbol{x}^{(i)}) + o(||\boldsymbol{x} - \boldsymbol{x}^{(i)}||^3)$$

其中 $\nabla = \dfrac{\partial}{\partial \boldsymbol{x}}$, $\boldsymbol{H}_g$ 为 g 的 Hessian 矩阵. 忽略高阶无穷小量, 下一轮迭代的解 $\boldsymbol{x}^{(i+1)}$ 应当为上述二次函数的极小值点. 于是对上式两边求梯度并代入 $\boldsymbol{x}^{(i+1)}$ 可得

$$\boldsymbol{0} = \nabla g(\boldsymbol{x}^{(i+1)}) = \nabla g(\boldsymbol{x}^{(i)}) + \boldsymbol{H}_g(\boldsymbol{x}^{(i)})(\boldsymbol{x}^{(i+1)} - \boldsymbol{x}^{(i)})$$

于是下一轮迭代的解即为线性方程组

$$\boldsymbol{H}_g(\boldsymbol{x}^{(i)})(\boldsymbol{x}^{(i+1)} - \boldsymbol{x}^{(i)}) = -\nabla g(\boldsymbol{x}^{(i)})$$

的解. 对于弹簧质点系统, 我们认为 $g(\boldsymbol{x})$ 的性质比较良好, 只需一次迭代就能求得较为准确的解. 于是就有

$$\boldsymbol{H}_g(\boldsymbol{x}(t_k))(\boldsymbol{x}(t_{k+1}) - \boldsymbol{x}(t_k)) = -\nabla g(\boldsymbol{x}(t_k))$$

根据 $g(\boldsymbol{x})$ 的定义可知

$$\nabla g(\boldsymbol{x}(t_k)) = \frac{1}{(\Delta t)^2}\boldsymbol{M}(\boldsymbol{x}(t_k) - \boldsymbol{y}(t_k)) + \nabla E(\boldsymbol{x}(t_k))$$

$$\boldsymbol{H}_g(\boldsymbol{x}(t_k)) = \frac{1}{(\Delta t)^2}\boldsymbol{M} + \boldsymbol{H}_E(\boldsymbol{x}(t_k))$$

其中 $\boldsymbol{H}_E(\boldsymbol{x})$ 为能量函数 $E(\boldsymbol{x})$ 的 Hessian 矩阵.

对于 $\nabla E(\boldsymbol{x}(t_k))$, 我们可以把它写成对各个质点的位置 $\boldsymbol{x}_i$ 的梯度 $\nabla_i E(\boldsymbol{x}) = \dfrac{\partial E(\boldsymbol{x})}{\partial \boldsymbol{x}_i}$ 的堆叠向量, 并且我们已经知道质点 i 受到的内力之和等于能量的负梯度, 于是

$$\nabla E(\boldsymbol{x}(t_k)) = \begin{bmatrix} \nabla_1 E(\boldsymbol{x}) \\ \cdots \\ \nabla_n E(\boldsymbol{x}) \end{bmatrix}_{\boldsymbol{x}=\boldsymbol{x}(t_k)} = -\begin{bmatrix} \boldsymbol{f}_{1,\mathrm{int}}(t_k) \\ \cdots \\ \boldsymbol{f}_{n,\mathrm{int}}(t_k) \end{bmatrix}$$

接下来考虑 $\boldsymbol{H}_E(\boldsymbol{x}(t_k))$. 根据弹簧质点模型的定义, 系统的总势能为各个弹簧势能之和:

$$E(\boldsymbol{x}) = \sum_{(i,j)} E_{ij}(\boldsymbol{x})$$

其中 (i,j) 表示以弹簧相连的质点. 于是

$$\boldsymbol{H}_E(\boldsymbol{x}) = \sum_{(i,j)} \boldsymbol{H}_{E_{ij}}(\boldsymbol{x})$$

其中 $\boldsymbol{H}_{E_{ij}}(\boldsymbol{x})$ 为单个弹簧势能 $E_{ij}(\boldsymbol{x})$ 的 Hessian 矩阵. 根据定义 $E_{ij} = \dfrac{1}{2}k_{ij}\left(||\boldsymbol{x}_j - \boldsymbol{x}_i|| - l_{ij}\right)^2$ 有

$$\boldsymbol{H}_{ij} := \frac{\partial^2 E_{ij}(\boldsymbol{x})}{\partial \boldsymbol{x}_i^2} = k_{ij}\frac{(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^{\mathrm{t}}}{||\boldsymbol{x}_i - \boldsymbol{x}_j||^2} + k_{ij}\left(1 - \frac{l_{ij}}{||\boldsymbol{x}_i - \boldsymbol{x}_j||}\right)\left(\boldsymbol{I} - \frac{(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^{\mathrm{t}}}{||\boldsymbol{x}_i - \boldsymbol{x}_j||^2}\right)$$

$$\frac{\partial^2 E_{ij}(\boldsymbol{x})}{\partial \boldsymbol{x}_j^2} = \boldsymbol{H}_{ij}, \quad \frac{\partial^2 E_{ij}(\boldsymbol{x})}{\partial \boldsymbol{x}_i \partial \boldsymbol{x}_j} = -\boldsymbol{H}_{ij}$$

对于 $E_{ij}(\boldsymbol{x})$ 而言, 其值仅与 $\boldsymbol{x}$ 中第 i 个向量 $\boldsymbol{x}_i$ 和第 j 个向量 $\boldsymbol{x}_j$ 有关. 于是将 $E_{ij}(\boldsymbol{x})$ 的 Hessian 矩阵 $\boldsymbol{H}_{E_{ij}}(\boldsymbol{x})$ 视作由 $N \times N$ 个 3×3 子矩阵组成的矩阵, 只有 $(i,i),(j,j),(i,j),(j,i)$ 四个子矩阵非零, 且分别为 $\boldsymbol{H}_{ij}, \boldsymbol{H}_{ij}, -\boldsymbol{H}_{ij}, -\boldsymbol{H}_{ij}$. 这就求得了 $\boldsymbol{H}_E$, 于是再代入前述公式即可得隐式欧拉积分的迭代公式.

过程总结 最后, 我们把隐式欧拉积分求解弹簧质点系统的过程总结如下:

1. 遍历系统中的所有弹簧, 然后进行如下两个操作:

a. 根据弹力的计算公式

$$\boldsymbol{f}_{ij} = -k_{ij}(|\boldsymbol{x}_i - \boldsymbol{x}_j| - l_{ij})\frac{\boldsymbol{x}_i - \boldsymbol{x}_j}{|\boldsymbol{x}_i - \boldsymbol{x}_j|}$$

计算弹簧所连接的两个质点 i,j 的内力 $\boldsymbol{f}_{ij}, \boldsymbol{f}_{ji}$, 并将其累加到质点 i,j 的总内力 $\boldsymbol{f}_{i,\text{int}}, \boldsymbol{f}_{j,\text{int}}$ 上.

b. 根据弹簧的 Hessian 矩阵计算公式

$$\boldsymbol{H}_{ij} = \frac{\partial^2 E_{ij}(\boldsymbol{x})}{\partial \boldsymbol{x}_i^2} = k_{ij}\frac{(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^{\mathrm{t}}}{||\boldsymbol{x}_i - \boldsymbol{x}_j||^2} + k_{ij}\left(1 - \frac{l_{ij}}{||\boldsymbol{x}_i - \boldsymbol{x}_j||}\right)\left(\boldsymbol{I} - \frac{(\boldsymbol{x}_i - \boldsymbol{x}_j)(\boldsymbol{x}_i - \boldsymbol{x}_j)^{\mathrm{t}}}{||\boldsymbol{x}_i - \boldsymbol{x}_j||^2}\right)$$

计算质点 i,j 对应的 Hessian 子矩阵, 并将其累加到系统总能量的 Hessian 矩阵 $\boldsymbol{H}_E$ 的对应位置上.

完成上述两步后即可求出 ∇E 和 $\boldsymbol{H}_E$.

2. 根据 $\boldsymbol{y}(t_k)$ 的定义

$$\boldsymbol{y}(t_k) = \boldsymbol{x}(t_k) + \Delta t \boldsymbol{v}(t_k) + (\Delta t)^2 \boldsymbol{M}^{-1}\boldsymbol{f}_{\text{ext}}$$

求出 $\boldsymbol{y}(t_k)$.

3. 根据 $\nabla g(\boldsymbol{x}(t_k))$ 和 $\boldsymbol{H}_g(\boldsymbol{x}(t_k))$ 的定义

$$\nabla g(\boldsymbol{x}(t_k)) = \frac{1}{(\Delta t)^2}\boldsymbol{M}(\boldsymbol{x}(t_k) - \boldsymbol{y}(t_k)) + \nabla E(\boldsymbol{x}(t_k))$$

$$\boldsymbol{H}_g(\boldsymbol{x}(t_k)) = \frac{1}{(\Delta t)^2}\boldsymbol{M} + \boldsymbol{H}_E(\boldsymbol{x}(t_k))$$

求出 $\nabla g(\boldsymbol{x}(t_k))$ 和 $\boldsymbol{H}_g(\boldsymbol{x}(t_k))$.

4. 求解线性方程组

$$\boldsymbol{H}_g(\boldsymbol{x}(t_k))(\boldsymbol{x}(t_{k+1}) - \boldsymbol{x}(t_k)) = -\nabla g(\boldsymbol{x}(t_k))$$

得到 $\boldsymbol{\delta} = \boldsymbol{x}(t_{k+1}) - \boldsymbol{x}(t_k)$.

5. 根据隐式欧拉的基本公式

$$\boldsymbol{x}(t_{k+1}) = \boldsymbol{x}(t_k) + \boldsymbol{\delta}$$

$$\boldsymbol{v}(t_{k+1}) = \frac{\boldsymbol{\delta}}{\Delta t}$$

计算更新后的位置 $\boldsymbol{x}(t_{k+1})$ 和速度 $\boldsymbol{v}(t_{k+1})$. 如此就完成了一个时间间隔内的模拟.

4.3 流体模拟

4.3.1 流体的基本性质

定理 3.1 Navier-Stokes 方程 流体的运动遵循 Navier-Stokes 方程, 其表达式为

$$\rho\frac{\mathrm{D}\boldsymbol{v}}{\mathrm{D}t} = -\nabla p + \rho\boldsymbol{g} + \mu\nabla^2\boldsymbol{v}$$

其中 $\dfrac{\mathrm{D}v}{\mathrm{D}t}$ 为流体的随体导数, 其定义为

$$\frac{\mathrm{D}\boldsymbol{v}}{\mathrm{D}t} = \frac{\partial\boldsymbol{v}}{\partial t} + (\boldsymbol{v}\cdot\nabla)\boldsymbol{v}$$

定理 3.2 不可压性质 流体中的质量守恒方程为

$$\frac{\partial\rho}{\partial t} = -\nabla\cdot(\rho\boldsymbol{v})$$

一般的流体模拟中总是假设流体不可压缩/膨胀, 于是密度 ρ 在各处和任意时刻均为常数. 于是上式即为

$$\nabla\cdot\boldsymbol{v} = 0$$

即流体的速度场是无散的, 也即流入任意一点的流体体积总是等于流出该点的流体体积.

4.3.2 描述流体的两种方式

上述 N-S 方程是以**欧拉视角**描述流体的运动的. 欧拉视角是指将流体的所有物理量看成空间上的一个场, 然后描述这个场随时间的变化. 这好比在水中插有无限多的木桩, 每个木桩上装有检测水的流速, 密度, 压强等的传感器, 我们描述的是所有传感器上的数值变化.

另一种描述流体的方式是**拉格朗日视角**, 即将流体看成由无数个质点组成, 然后描述每个质点随时间的运动. 这好比在水中放入无数悬浮的小球, 其上也装有传感器, 我们描述的是每个小球的位置, 速度等随时间的变化.

4.3.3 光滑粒子流体

我们首先采用拉格朗日视角来描述流体的运动.

定理 3.3 光滑粒子流体模型 光滑粒子流体 (Smoothed Particle Hydrodynamics, SPH) 模型将流体看成有限个流体微团 (视作粒子) 组成的系统. 每个粒子 i 具有位置 $\boldsymbol{x}_i$, 速度 $\boldsymbol{v}_i$, 质量 m_i.

为了根据上述信息计算流体的宏观性质, 例如求解 N-S 方程必不可少的密度场和压强场, 我们采用一种经典的办法: **核函数**.

核函数

我们已经经历了很多采样问题以从离散的样本点重建连续函数. 对于流体而言, 如果采取简单的近邻法划分空间并计算目标函数, 往往会在边界上造成函数值的跳变, 因而结果交叉. 因此, 我们需要采取一定的办法使得这种不连续性被平滑. 一种自然的想法就是考虑目标点附近的所有粒子对该点性质的影响, 并且距离越近的粒子影响越大. 这就可以通过引入核函数 $W(\boldsymbol{r},h)$ 进行加权平均而解决.

一般而言, 核函数 W 需要满足归一化性质, 即在 d 维空间下有

$$\int_{\mathbb{R}^d} W(||\boldsymbol{x}||)\mathrm{d}\boldsymbol{x}=1$$

一种常用的核函数是三次样条函数:

$$W(\boldsymbol{r},h)=\sigma\begin{cases}6(q^3-q^2)+1, & 0\leqslant q\leqslant 1/2\\ 2(1-q)^3, & 1/2<q\leqslant 1\\ 0, & q>1\end{cases}$$

其中 $q=\dfrac{||\boldsymbol{r}||}{h}$, h 为核函数的半径; σ 为归一化系数, 在三维空间中有 $\sigma=\dfrac{8}{\pi h^3}$. 在实践中, 一般取 h 为初始状态下粒子平均间距的 $1\sim 3$ 倍左右.

流体性质的计算

有了核函数之后, 我们就可以根据粒子的位置和质量计算流体的密度和压强等性质. 流体的密度场可计算如下:

$$\rho(\boldsymbol{x})=\sum_i m_i W(||\boldsymbol{x}-\boldsymbol{x}_i||)$$

类似地, 流体的压强场及其梯度可计算如下:

$$p(\boldsymbol{x})=\sum_i p_i\frac{m_i}{\rho_i}W(||\boldsymbol{x}-\boldsymbol{x}_i||)$$

$$\nabla p(\boldsymbol{x})=\sum_i p_i\frac{m_i}{\rho_i}\nabla W(||\boldsymbol{x}-\boldsymbol{x}_i||)$$

其中 $p_i=k(\rho_i-\rho_0)^\gamma$, k,γ 为流体的常数, ρ_0 为流体的静密度. p_i 表示粒子 i 处的压强, 其随 ρ_i 的增加而增加. 于是在 N-S 方程中 $-\nabla p$ 项就会将粒子从密度高的区域推向密度低的区域以维持近似不可压的状态.

4.3.4 欧拉网格流体

使用拉格朗日法进行流体模拟十分直观, 便捷且高效, 但其对物理模型做了过于简化的处理, 因此模拟结果往往不够准确也不够真实. 欧拉法则将流体视为一种连续介质, 并且由于 N-S 方程本身的形式就是欧拉视角, 欧拉法也能够更自然地对其进行处理.

空间离散化

现在, 我们不再考虑组成流体的粒子的具体运动, 而是将整个容器划分为若干小单元格 (例如边长为 Δx 的正方形/立方体网格). 场景中的各种物质与物理量均存储在网格中.

标记网格 为了表示流体在空间中的分布, 我们可以使用**标记网格 (Marker-and-Cell grid, MAC grid)** 方法. 该方法在每个网格单元的中心存储一个标记变量 m, 用于表示该单元中是否存在流体. 如果存在流体, 则 $m = 1$; 否则 $m = 0$. 通过这种方式, 我们就可以表示流体的形状和位置.

数值求解

求解 N-S 方程常用的办法是分裂法. 对于无粘性的流体, 将 N-S 方程拆分成三个更简单的偏微分方程, 在一个时间步内对它们依次求解. 对 N-S 方程变形可得

$$\frac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} + \boldsymbol{g} + \frac{\mu}{\rho}\nabla^2 u - \frac{1}{\rho}\nabla \boldsymbol{p}$$

于是一个时间步内的步骤即为:

1. **对流 (Advection)**
   求解 $\dfrac{\partial \boldsymbol{u}}{\partial t} = -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u}$ 以更新 $\boldsymbol{u}$.
2. **外力 (External Force)**
   求解 $\dfrac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{g}$ 以更新 $\boldsymbol{u}$.
3. **粘度和扩散 (Viscosity or Diffusion)**
   求解 $\dfrac{\partial \boldsymbol{u}}{\partial t} = \dfrac{\mu}{\rho}\nabla^2 \boldsymbol{u}$ 以更新 $\boldsymbol{u}$.
4. **压强投影 (Pressure Projection)**
   求解 $\dfrac{\partial \boldsymbol{u}}{\partial t} = -\dfrac{1}{\rho}\nabla p$ 以更新 $\boldsymbol{u}$, 并且保证更新后的速度场满足不可压性质 $\nabla \cdot \boldsymbol{u} = 0$. 这也是这一步放在最后的原因.

现在, 我们就来考虑每一步的微分方程的求解方法.

对流 对流方程可以表示为以下的一般形式:

$$\frac{\mathrm{D}\phi}{\mathrm{D}t} = 0, \quad \text{i.e.} \quad \frac{\partial \phi}{\partial t} = -(\boldsymbol{u} \cdot \nabla)\phi$$

其中 ϕ 是流体的标量场, 例如速度 $\boldsymbol{u}$ 的各个分量. 求解对流过程的算法可以表示如下:

$$\phi^{(n+1)} = \mathrm{advect}(\boldsymbol{u}, \Delta t, \phi^{(n)})$$

其中 $\phi^{(n)}$ 表示当前的标量场,$\phi^{(n+1)}$ 表示对流之后的标量场.

如果我们直接用差分的方式将上面的微分方程离散化, 会导致严重的问题: 已经证明这样的方法是无条件不稳定[1]的. 因此, 研究者们提出了一个更加直观稳定的办法: 使用拉格朗日视角解决对流问题, 即**半拉格朗日**

[1] 即无论时间步长取多小, 数值解都会震荡发散

方法 (Semi-Lagrangian Method).

假定我们需要求解标量场 ϕ 在网格点 $\boldsymbol{x}_g$ 在第 $n+1$ 个时间步的值 $\phi^{(n+1)}(\boldsymbol{x}_g)$. 在拉格朗日的视角下, 我们可以寻找在前一步时是何处的流体粒子在速度场 $\boldsymbol{u}$ 的作用下运动到了 $\boldsymbol{x}_g$. 设该粒子在第 n 个时间步的位置为 $\boldsymbol{x}_p$. 在拉格朗日视角下, 物理量 ϕ 由流体粒子携带, 因此有

$$\phi^{(n+1)}(\boldsymbol{x}_g) = \phi^{(n)}(\boldsymbol{x}_p)$$

于是目标转化为求解粒子位置 $\boldsymbol{x}_p$. 根据粒子的运动方程有

$$\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}t} = \boldsymbol{u}$$

在一个时间步长 Δt 内, 可以用显式欧拉积分求出 $\boldsymbol{x}_p$, 即

$$\boldsymbol{x}_p = \boldsymbol{x}_g - \Delta t \boldsymbol{u}(\boldsymbol{x}_g)$$

为了提高精度, 我们也可以使用二阶 Runge-Kutta 方法, 即在计算 $\boldsymbol{x}_p$ 的过程中将时间步长继续细分:

$$\boldsymbol{x}_{\mathrm{mid}} = \boldsymbol{x}_g - \frac{1}{2}\Delta t \boldsymbol{u}(\boldsymbol{x}_g), \quad \boldsymbol{x}_p = \boldsymbol{x}_{\mathrm{mid}} - \frac{1}{2}\Delta t \boldsymbol{u}(\boldsymbol{x}_{\mathrm{mid}})$$

继续增加采样点, 将时间步长划分为更小的间隔也是可行的.

现在, 我们已经获取了 $\boldsymbol{x}_p$. 一般而言, 求得的 $\boldsymbol{x}_p$ 并不在网格点上, 因此需要使用插值的方法计算 $\phi^{(n)}(\boldsymbol{x}_p)$. 在二维情形下可以使用双线性插值的办法; 而在三维情形下可以使用三线性插值的办法.

如果求得的 $\boldsymbol{x}_p$ 在液体内部, 那么直接使用插值的结果即可; 但有时 $\boldsymbol{x}_p$ 可能在液体的外部. 造成这一意外的可能原因主要有两种, 其解决办法分别如下:

1. $\boldsymbol{x}_p$ 确实在液体的外部, 这意味着有新的流体流入. 此时, 我们应当知道外部流体的性质, 因此直接用外部流体的性质赋值即可.
2. $\boldsymbol{x}_p$ 本应在液体的内部, 但由于数值计算的误差等原因导致其落在了外部. 此时, 我们可以寻找流体边界上距离 $\boldsymbol{x}_p$ 最近的点 $\boldsymbol{x}_b$, 然后使用插值的方法计算 $\phi^{(n)}(\boldsymbol{x}_b)$ 作为结果.

外力 求解外力方程实际上非常简单, 因为外力项 $\boldsymbol{g}$ 通常是一个常量 (例如重力). 因此, 可以直接用差分的形式写出新的速度场

$$\boldsymbol{u}^{new} = \boldsymbol{u} + \boldsymbol{g}\Delta t$$

粘度 粘度方程中包含一个拉普拉斯算子 ∇^2. 我们可以使用有限差分的方法对其进行离散化 (这和图像的泊松编辑是类似的). 在二维情形下, 对于网格点 (i, j) 有

$$\boldsymbol{u}_{ij}^{new} = \boldsymbol{u}_{ij} + \frac{\mu}{\rho}\Delta t \frac{\boldsymbol{u}_{(i-1)j} + \boldsymbol{u}_{(i+1)j} + \boldsymbol{u}_{i(j-1)} + \boldsymbol{u}_{i(j+1)} - 4\boldsymbol{u}_{ij}}{(\Delta x)^2}$$

如果需要更高的精度, 可以将 Δt 继续细分为更小的时间间隔, 然后按照上述公式多次迭代.

压强投影 最后, 我们需要求解压强投影方程. 将其写作对时间离散的形式就有

$$\boldsymbol{u}^{new} - \boldsymbol{u}^* = -\frac{\Delta t}{\rho}\nabla p$$

这里 $\boldsymbol{u}^*$ 是经过前三个步骤所得的速度场, 它可能已经失去了无旋的性质. 而求得的速度场 $\boldsymbol{u}^{new}$ 需要满足不可压性质, 于是 $\nabla \boldsymbol{u}^{new} = \boldsymbol{0}$. 于是对上式两边求梯度可得

$$\nabla \boldsymbol{u}^* = \frac{\Delta t}{\rho}\nabla^2 p$$

然后将上式空间离散化即可得

$$p_{(i-1)j} + p_{(i+1)j} + p_{i(j-1)} + p_{i(j+1)} - 4p_{ij} = \frac{\rho \Delta x}{\Delta t}(u_{(i+1)j} - u_{ij} + v_{i(j+1)} - v_{ij})$$

式中右边的各 p_{ij} 是未知量, 而左边的各 u_{ij}, v_{ij} 均为已知量. 于是上述方程即可写作线性方程组 $\boldsymbol{Ap} = \boldsymbol{b}$ 的形式. 我们现在来考虑体系的边界条件.

1. **Neumann 边界条件**

   对于流体与固体的边界, 我们要求流体不能穿透固体. 设边界处法向量为 $\boldsymbol{n}$, 则有

   $$\boldsymbol{u} \cdot \boldsymbol{n} = 0$$

   代入压强投影方程可得

   $$\left(\boldsymbol{u}^* - \frac{\Delta t}{\rho}\nabla p\right) \cdot \boldsymbol{n} = 0$$

   即

   $$\nabla p = \frac{\rho}{\Delta t}\boldsymbol{u}^* \cdot \boldsymbol{n}$$

   这就是 Neumann 边界条件.

4.4 动画原理

在上一节中, 我们主要讨论各种物理现象的模拟方法. 然而对于具有能动性的对象, 如人和动物, 我们往往需要通过动画来表现其运动. 本节将介绍动画的基本原理和技术.

4.4.1 旋转的四元数表示

四元数的基本概念

我们在**几何变换**一节中介绍了用矩阵表示物体旋转的方法. 现在我们介绍另一种常用的表示旋转的方法: 四元数.

定义 4.1 四元数

定义四元数 $q = a + b\mathrm{i} + c\mathrm{j} + d\mathrm{k}$, 其中 $a, b, c, d \in \mathbb{R}$, 且 $\mathrm{i}, \mathrm{j}, \mathrm{k}$ 满足以下乘法规则:

$$\mathrm{i}^2 = \mathrm{j}^2 = \mathrm{k}^2 = \mathrm{ijk} = -1, \quad \mathrm{ij} = -\mathrm{ji} = \mathrm{k}, \quad \mathrm{jk} = -\mathrm{kj} = \mathrm{i}, \quad \mathrm{ki} = -\mathrm{ik} = \mathrm{j}.$$

由于四元数虚部的运算规则与三维空间中的叉乘类似, 因此我们可以将四元数 q 表示为 $q = [w, \boldsymbol{v}]$, 其中 $\boldsymbol{v} = (b, c, d)^{\mathrm{t}} \in \mathbb{R}^3$.

定理 4.2 四元数的乘法 四元数 $q_1 = [w_1, \boldsymbol{v}_1], q_2 = [w_2, \boldsymbol{v}_2]$ 的乘积为

$$q_1 q_2 = [w_1 w_2 - \boldsymbol{v}_1 \cdot \boldsymbol{v}_2, w_1 \boldsymbol{v}_2 + w_2 \boldsymbol{v}_1 + \boldsymbol{v}_1 \times \boldsymbol{v}_2]$$

证明. 根据四元数的定义和向量叉积的定义即可得出上式. □

并且由此不难得出四元数的运算满足结合律和分配律, 但不满足交换律.

定义 4.3 四元数的模长与单位四元数

定义四元数 q 的**模长**为 $||q|| = \sqrt{q \cdot q} = \sqrt{w^2 + \boldsymbol{v} \cdot \boldsymbol{v}}$.

定义**单位四元数**为模长为 1 的四元数. 单位四元数可以表示为 $q = [\cos\theta, \boldsymbol{u}\sin\theta]$, 其中 $\boldsymbol{u}$ 是三维的单位向量.

定义 4.4 四元数的共轭与逆元

定义四元数 $q = [w, \boldsymbol{v}]$ 的共轭 $q^* = [w, -\boldsymbol{v}]$ 为其虚部取相反数的结果.

根据四元数乘法的定义不难得出

$$qq^* = [w^2 + \boldsymbol{v} \cdot \boldsymbol{v}, \boldsymbol{0}] = ||q||^2$$

于是可以定义四元数 q 的**逆元**为

$$q^{-1} = \frac{q^*}{||q||^2}$$

并且总有 $qq^{-1}=q^{-1}q=[1,\mathbf{0}]$. 对于单位四元数而言总有 $q^*=q^{-1}$.

用四元数表示旋转

定理 4.5 旋转的四元数表示 设 $\boldsymbol{v}$ 是三维空间内的单位向量, $q=[\cos\theta,\boldsymbol{u}\sin\theta]$ 为单位四元数. 令 $v=[0,\boldsymbol{v}]$, 则 qvq^* 的实部为 0, 虚部为 $\boldsymbol{v}$ 绕旋转轴 $\boldsymbol{u}$ 旋转弧度 2θ 后得到的向量.

证明. 由四元数的乘法定义可得

$$\begin{aligned}qvq^* &= [\cos\theta,\boldsymbol{u}\sin\theta][0,\boldsymbol{v}][\cos\theta,-\boldsymbol{u}\sin\theta]\\ &= [-\boldsymbol{u}\cdot\boldsymbol{v}\sin\theta,\cos\theta\boldsymbol{v}+\sin\theta(\boldsymbol{u}\times\boldsymbol{v})][\cos\theta,-\boldsymbol{u}\sin\theta]\\ &= [0,\boldsymbol{v}\cos 2\theta+(\boldsymbol{u}\times\boldsymbol{v})\sin 2\theta+\boldsymbol{u}(\boldsymbol{u}\cdot\boldsymbol{v})(1-\cos 2\theta)]\end{aligned}$$

由此可见 qvq^* 的实部为 0. 而其虚部正是 $\boldsymbol{v}$ 绕旋转轴 $\boldsymbol{u}$ 旋转弧度 2θ 后得到的向量 (根据我们在旋转矩阵的推导中的结论可知). □

4.4.2 运动学

前向运动学

以人体为例, 我们可以把各个关节视作节点, 各个骨骼视作连接节点的边. 这样, 人体就可以表示为一个带有根节点的树形结构. 我们只需要知道根节点的位置和各个关节的旋转角度, 就可以计算出各个节点的位置.

以刚体铰链模型 $P_0\cdots P_n$ 为例展示前向运动学的过程. 节点 $P_0,\cdots,P_n$ 的初始位置记作 $\boldsymbol{p}_0^{\mathrm{st}},\cdots,\boldsymbol{p}_n^{\mathrm{st}}$, 向下一个关节的位移即为 $\boldsymbol{l}_i=\boldsymbol{p}_i^{\mathrm{st}}-\boldsymbol{p}_{i-1}^{\mathrm{st}}(i=1,\cdots,n)$, 每个节点上都带有一个局部坐标系 $\mathcal{C}_i$.

现在, 我们为除去 P_n 外的每个关节 P_j 指定一个旋转矩阵 $\boldsymbol{R}_j^{\mathrm{loc}}(j=1,\cdots,n-1)$(这里的旋转矩阵是基于该关节关联的局部坐标系 $\mathcal{C}_j$ 所定义的), 每个关节 P_j 处的旋转带动其后的所有关节 (包括其上带的局部坐标系) 和骨骼旋转. 因此, 节点 P_j 的总的旋转矩阵 $\boldsymbol{R}_j^{\mathrm{tot}}$ 即为它的局部坐标系 $\mathcal{C}_j$ 相对于根节点的坐标系 $\mathcal{C}_0$ 的旋转与它本身的旋转的复合. 而 $\mathcal{C}_j$ 的旋转事实上就是由 P_{j-1} 节点的旋转定义的, 因此 $\boldsymbol{R}_j^{\mathrm{tot}}=\boldsymbol{R}_{j-1}^{\mathrm{tot}}\boldsymbol{R}_j^{\mathrm{loc}}$.

现在, 我们就可以从根节点开始计算每个子节点的位置 $\boldsymbol{p}_j^{\mathrm{ed}}(j=1,\cdots,n)$. 根节点的位置 $\boldsymbol{p}_0^{\mathrm{ed}}=\boldsymbol{p}_0^{\mathrm{st}}$ 已知, 而每个节点的位置可以通过其父节点的位置加上旋转后的位移向量得到, 即

$$\boldsymbol{p}_j^{\mathrm{ed}}=\boldsymbol{p}_{j-1}^{\mathrm{ed}}+\boldsymbol{R}_{j-1}^{\mathrm{tot}}\boldsymbol{l}_j$$

逆向运动学

逆向运动学 (Inverse Kinematics, IK) 的任务与前向运动学恰恰相反, 其需要根据给定的部分关节的位置 $\boldsymbol{p}_i$(包括但是不仅限于末关节)，反向求解出一组关节局部旋转 $\boldsymbol{R}_i^{\mathrm{loc}}$ 的解. 逆向运动学可以被用于机械臂或者是虚拟角色的控制, 也可以帮助修复一些质量较差, 没有满足部分接触要求的动作捕捉数据.

对于简单的, 具有两个骨骼的系统, IK 问题有解析的解. 然而, 对于更复杂的系统, 我们只能通过数值方法

求解 IK 问题. 下面介绍两种常用的方法: **循环坐标推演 (Cyclic Coordinate Descent, CCD)** 和**前向后向抵达 (Forward and Backward Reaching, FABR)**.

CCD IK CCD 的思想是从末端关节开始, 依次调整每个关节的旋转, 使得末端关节逐渐接近目标位置.

具体地, 假定当前铰接刚体系统共有 $n+1$ 个关节与 n 个连接关节的骨骼, 我们希望末端的关节位置 $\boldsymbol{p}_n$ 距离目标 t 尽可能接近. 于是遍历第 $n-1,\cdots,0$ 个关节, 每次求解关节 i 处的旋转矩阵 $\boldsymbol{R}_i^{\text{loc}}$ 使得调整后 $\boldsymbol{p}_i$, $\boldsymbol{p}_n$ 和 t 三点共线[2], 然后将更新后的 $\boldsymbol{R}_i^{\text{loc}}$ 应用于第 i 个关节及其后续的所有关节, 更新它们的位置. 重复上述过程直到末端关节 $\boldsymbol{p}_n$ 足够接近目标 t 或者达到最大迭代次数为止.

FABR IK FABR 的思想是通过两次遍历关节链, 根据骨骼长度调整关节位置, 使得末端关节接近目标位置.

具体地, 假定当前铰接刚体系统共有 $n+1$ 个关节与 n 个连接关节的骨骼, 我们希望末端的关节位置 $\boldsymbol{p}_n$ 距离目标 t 尽可能接近.

首先是 Backward 步骤的计算. 我们将末端关节 $\boldsymbol{p}_n$ 移动到目标位置 t, 记 $\boldsymbol{p}_n^{\text{back}}=\boldsymbol{t}$. 接着, 我们从末端关节向根节点遍历每个关节 $i(i=n-1,\cdots,0)$, 将 $\boldsymbol{p}_i$ 更新到 $\boldsymbol{p}_i^{\text{back}}$ 使得

$$\left|\boldsymbol{p}_{i+1}^{\text{back}}-\boldsymbol{p}_i^{\text{back}}\right|=|\boldsymbol{l}_{i+1}|,\quad \boldsymbol{p}_i,\boldsymbol{p}_i^{\text{back}},\boldsymbol{p}_{i+1}^{\text{back}}\text{共线}$$

于是可得

$$\boldsymbol{p}_i^{\text{back}}=\boldsymbol{p}_{i+1}^{\text{back}}+\frac{\boldsymbol{p}_i-\boldsymbol{p}_{i+1}^{\text{back}}}{\left|\boldsymbol{p}_i-\boldsymbol{p}_{i+1}^{\text{back}}\right|}|\boldsymbol{l}_{i+1}|$$

如此往复直到根节点位置[3]. 现在, 我们得到了一组新的关节位置 $\left\{\boldsymbol{p}_i^{\text{back}}\right\}_{i=1}^n$.

接下来是 Forward 步骤的计算. 记 $\boldsymbol{p}_0^{\text{for}}=\boldsymbol{p}_0^{\text{st}}$. 接着, 我们从根节点向末端遍历每个关节 $i(i=1,\cdots,n)$, 将 $\boldsymbol{p}_i^{\text{back}}$ 更新到 $\boldsymbol{p}_i^{\text{for}}$ 使得

$$\left|\boldsymbol{p}_i^{\text{for}}-\boldsymbol{p}_{i-1}^{\text{for}}\right|=|\boldsymbol{l}_i|,\quad \boldsymbol{p}_i^{\text{back}},\boldsymbol{p}_i^{\text{for}},\boldsymbol{p}_{i-1}^{\text{for}}\text{共线}$$

于是可得

$$\boldsymbol{p}_i^{\text{for}}=\boldsymbol{p}_{i-1}^{\text{for}}+\frac{\boldsymbol{p}_i^{\text{back}}-\boldsymbol{p}_{i-1}^{\text{for}}}{\left|\boldsymbol{p}_i^{\text{back}}-\boldsymbol{p}_{i-1}^{\text{for}}\right|}|\boldsymbol{l}_i|$$

如此往复直到末端节点位置. 现在, 我们得到了一组新的关节位置 $\left\{\boldsymbol{p}_i^{\text{for}}\right\}_{i=1}^n$. 这组新的关节满足骨骼长度约束, 并且 $\boldsymbol{p}_n$ 相比之前更接近目标 $\boldsymbol{t}$.

重复上述的 Backward 和 Forward 步骤直到末端关节 $\boldsymbol{p}_n$ 足够接近目标 t 或者达到最大迭代次数为止.

4.4.3 动作捕捉

动作捕捉 (Motion Capture, Mocap) 是通过设备记录人和物体运动的方法. 就角色来说, 我们需要一些专业的动捕演员, 让其穿戴特定的设备进行表演. 在每一时刻, 演员的肢体运动甚至面部表情会都会被设备记录下来, 转化成骨骼的旋转数据, 记录完毕后将其交给后续的处理流程. 我们接下来介绍一些常用的动作捕捉设备及其原理.

[2]确切地说, 是令 $\boldsymbol{p}_n-\boldsymbol{p}_i$ 和 $\boldsymbol{t}-\boldsymbol{p}_i$ 同向. 在 `glm` 库中, 求解将向量 `a` 旋转到向量 `b` 对应的四元数可以用 `glm::rotate` 函数实现.

[3]实际上根节点的位置不用更新.

外骨骼

外骨骼是一种基于机械结构的动作捕捉设备. 演员需要穿戴一个带有多个传感器的外骨骼装置, 这些传感器通常安装在关节处, 用于测量关节的旋转角度. 通过读取这些传感器的数据, 我们可以实时获取演员各个关节的旋转信息, 从而重建出演员的动作.

外骨骼的优点是可以提供高精度的关节旋转数据, 且不受外部环境光线和遮挡的影响. 然而, 外骨骼设备通常较为笨重, 限制了演员的动作自由度.

光学动作捕捉

光学动作捕捉系统是目前最常用的动作捕捉技术, 通常使用多个摄像头来跟踪演员身上的发光或反光的标记点. 摄像头捕捉到标记点的位置后, 通过三角测量法计算出标记点在三维空间中的位置. 然后, 通过对这些位置数据进行处理, 可以推断出演员的骨骼姿态和动作.