

# 计算机图形学

---

班级：07111903 学号：1120191334 姓名：周永扬

## 计算机图形学

- 1、概述
- 2、光栅化
  - 2.1 变换
  - 2.2 采样
  - 2.3 深度检测
  - 2.4 着色
    - 2.4.1 漫反射
    - 2.4.2 镜面反射
    - 2.4.3 环境光
    - 2.4.4 光照衰减
    - 2.4.5 Phong光照模型
    - 2.4.6 Blin-Phong光照模型
    - 2.4.7 着色方式
    - 2.4.8 纹理采样
    - 阴影
- 3、几何
  - 3.1 曲面方程
    - 3.1.1 隐式曲面
    - 3.1.2 显式曲面
  - 3.2 CSG体素构造法
  - 3.3 贝塞尔曲线
    - 3.3.2 二阶贝塞尔曲线
    - 3.3.2 三阶贝塞尔曲线
    - 3.3.3 贝塞尔曲线通项
    - 3.3.4 贝塞尔曲面
  - 3.4 网格变换
    - 3.4.1 曲面细分
    - 3.4.2 网格简化
- 4、光线追踪
  - 4.1 光线追踪基本理论与算法
    - 4.1.1 光线投射
    - 4.1.2 Whitted-Style 光线追踪
    - 4.1.3 光线追踪加速
    - 4.1.4 BVH划分
  - 4.2 辐射度量学
    - 4.2.1 辐射强度 (Radiant Intensity)
    - 4.2.2 辐照度 (Irradiance)
    - 4.2.3 辐射亮度(Radiance)
    - 4.2.4 双向反射分布函数 (Bidirectional Reflectance Distribution Function)
  - 4.3 路径追踪
- 5、其他
  - 5.1 材质
    - 5.1.1 漫反射材质
  - 5.2 微表面模型
  - 5.3 颜色
- 6、总结

## 1、概述

此报告将从光栅化、几何、光线追踪三个方面的理论学习来进行阐述。在概述部分，将简要的阐述个人对于这三个部分的一些理解。以及对于计算机图形学这一计算机科学分支的整体认识。由于希望在撰写读书报告的同时能够充分学习图形学的知识，将采用类似于“讲述”的视角。

光栅化可以说是图形学的基础，一切数学计算、代码实现在最终走需要通过带像素这一最小单位在物理层面进行展现。光栅化作为图形学中一种十分有效的渲染手段，在实时渲染领域运用即为广泛。在我看来，对于一个通过数学方式定义的虚拟空间，通过坐标变换，投影，二维化处理得到屏幕空间的坐标后。在进行几何组合，顶点着色，片元着色，深度检测等操作后，才能在屏幕中展现一个符合透视规律的形象。快速的光栅化算法配合GPU中大量单元的计算，能够实现流水线化的渲染效果，每一帧都实现画面的绘制，最终达到24帧，30帧乃至当下的60帧，144帧等高帧率实时动态效果。光栅化本质上就是将几何数据进行坐标变换、几何离散的过程。

通过几何手段建立模型、模拟运动，实现曲线、曲面同样重要。几何表达式所表示的信息具有连续性，通过显示、隐式方程进行建模也各有优点。各种各样的几何建模方法为我们提供了足够的创造基础，通过当代高度图形化的建模软件，如3Ds Max, Maya, SolidWorks来进行建模使得获得具有足够细节，高精度的模型已不再是难事。OpenGL, DirectX等不同平台的API也提供了导入模型的方法，配合合理的渲染方法就能够建立具有真实感的虚拟场景。模型是场景的可视化呈现，合理的模型，纹理贴图，凹凸贴图……配合真实的材质，光线算法就能够建立足以以假乱真的虚拟场景。

光线追踪可以说是一种“高代价，高回报”的渲染方法，其大量的计算量，复杂的计算过程注定完全的光线追踪当下难以完整运用到实时渲染的领域（虽然当下硬件厂商Nvidia, AMD的GPU中已经能够支持光线追踪，Unity, UE等游戏引擎同样支持光线追踪特性，但仍然是光栅化方法、光线追踪方法以及计算着色器的混合渲染管线，但效果已经达到了相当惊人的程度）。但是在离线渲染这一方面，通过光线追踪得到的图像真实感极强，特别是通过改进的路径追踪方法，进行基于物理的渲染，完全能够达到照片级别的渲染效果。我在学习中看到的一个典型例子是康奈尔箱（Cornell Box）在真实物理环境下拍照得到的效果与通过路径追踪算法得到的相似度近乎100%。

动画是计算机图形学能够广泛使用的一个领域，特别是在近年来很火的CG动画领域（包括游戏CG），影视特效等都是基于图形学制作的。同样，实时动画也可用于仿真、模拟方面（北理的数字仿真团队）。在娱乐方面，通过计算机模拟场景、动画、特效、粒子等能够产生符合人类感知，物理定律的科幻场景或是奇幻特效；在工业生产方面，通过仿真模拟能够大幅度降低进行真正物理实验的成本，给研究做出拥有参考意义的效果展示；在数字表演方面，通过计算机进行可视化的模拟演练能够精确控制大量参演者的实时信息，实现实时的全方位掌控。

我认为图形学从大方向来分能够分为渲染、几何、模拟三个方面，但各类小方向数不胜数。图形学广泛应用与游戏、影视、仿真模拟……图形学的知识帮助我们能够通过计算机图像来模拟真实世界中的一切，特别是相较于计算机出现初期，一切均显示在二维的显示器方面，当下多种多样的人机交互设备，图形显示设备，渲染设备……为我们提供了创建一个真正虚拟世界的可能性。从图形学的角度来说，万事万物均可虚拟，虚拟的光线，虚拟的场景，虚拟的交互体验带来的是真正意义上的“以假乱真”，当下各类图形设备发展迅速，算力的大幅提升也使许多算法变为可能。可以说，计算机图形学是计算机科学与现实世界的链接器，透过这门学科，虚拟世界与现实世界得以相连。

---

## 2、光栅化

光栅化（Rasterization）是把顶点数据转换为片元的过程，具有将图转化为一个个栅格组成的图像的作用，特点是每个元素对应帧缓冲区的一个像素。

从效果上来讲，光栅化实现了将几何图元转变为二维图像，即整个光栅化分为两个部分。一、决定窗口坐标中的哪些整形栅格趋于被基本图元占用。二、为每一个基本图元分配颜色值和深度值。两部完成后就形成了片元。

从计算机渲染角度来看，光栅化的过程实在图形渲染管线进行的。整个渲染流程的起点是顶点数据集（Vertex Data），其中包含了顶点相关的所有数据，比如：位置，法线，纹理映射坐标，颜色……

之后，顶点数据集会被传入顶点着色器（Vertex Shader）。这一部分会将顶点数据进行变换，即从输入的模型空间坐标逐步转换到观察坐标，世界坐标，裁剪坐标。（如果采用顶点着色方法，顶点颜色的计算也会在这一部分实现）在完成这一部分后，Vertex Shader可以将需要的数据（如法线，纹理坐标）传入之后的片元着色器。

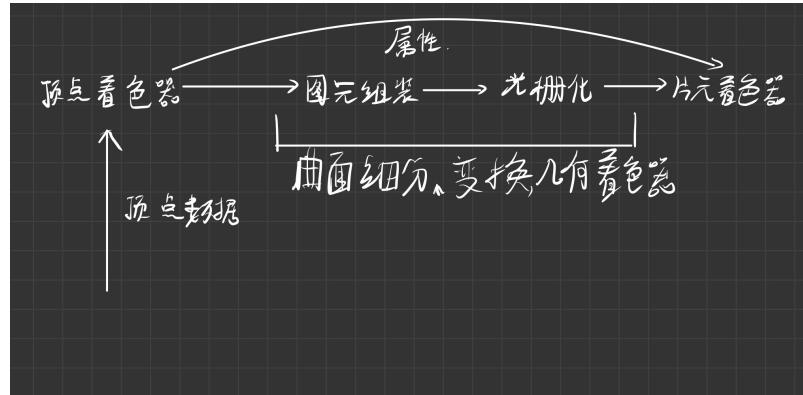
下一步将进行曲面细分，通过一些特殊算法来增加三角面的数量（如Loop subdivision）方法以及（catmull-clark subdivision）方法（在几何部分会说明这两个部分）。曲面细分可以实现物体细节的动态变化，使得细节需要显露时出现，不需要隐藏降低计算量。

几何着色器（Geometry Shader）是一个可选的渲染阶段，对于这一个部分不是特别了解，就仅简略说明。几何着色器处于顶点着色器之后，输入为完整的图元，输出为一个或多个图元，是将输入的点、线拓展为多边形。

图元组装部分用来将得到的顶点数据组装为指定的图元，组装阶段会进行边缘的裁剪，图元背面的剔除来进行优化，减少光栅化进程的图元数目。同时进行屏幕映射

经过图元组装、屏幕映射后，将进行光栅化处理。即将连续的物体转化为离散的像素点的过程。包括组装三角形、遍历三角形两阶段。首先确定图元覆盖的像素片段，通过顶点属性进行重心坐标插值得到片段各个位置的属性信息，后将这些信息送至片段着色器计算颜色。

片段着色器（Fragment Shader）用来确定每一个像素的最终颜色。在这一阶段会计算光照效果，阴影处理。通过此过程，每一个像素将被着色，从而形成画面。



## 2.1 变换

首先，常用的变换共有以下几种形式：1、平移变换，2、缩放变换，3、旋转变换。这是所有空间变换的基本组成部分，各类复杂的空间变换都可通过若干简单的变换组合得到。

在图形学空间中，点和向量是两个基本要素，一个表示位置，一个表示方向。我们以  $(x,y,z,1)$  来表示点， $(x,y,z,0)$  来表示向量。这是合理的，两个点相减即可使四维向量的w分量为0，即变换为一个向量的形式。对于齐次变换矩阵的推导，这里就不再过多说明，只说明其形式。

### 1、平移

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1)$$

### 2、旋转(以绕Z轴旋转为例)

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1)$$

### 3、缩放

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1)$$

通过以上三种类型的矩阵，只要注意变换顺序，更换坐标系等方法，能够实现绝大多数空间变换。

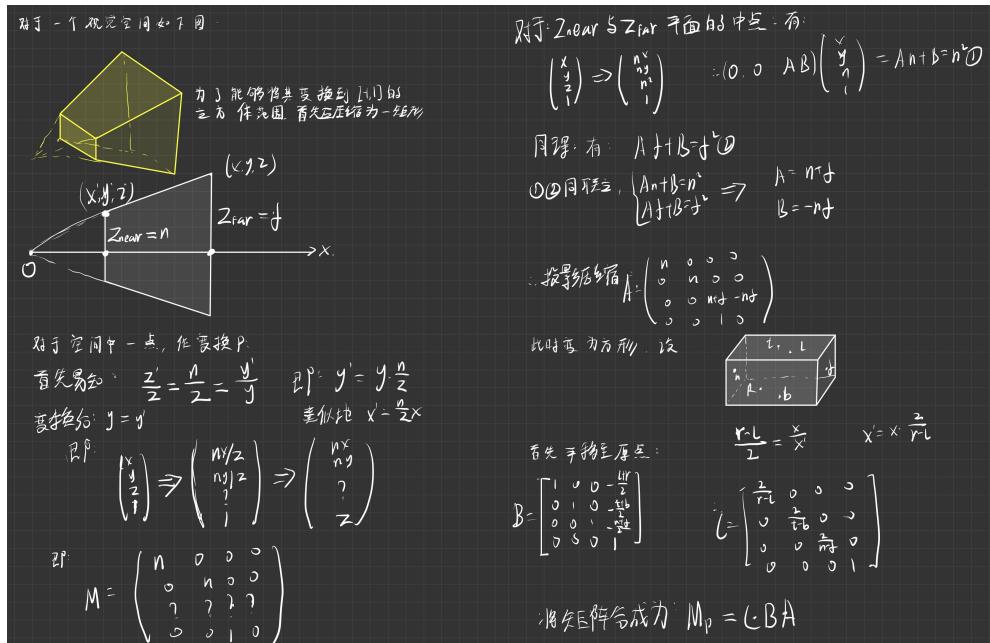
说完了以上的基础内容，下面就是具体的光栅化变换流程。

对于光栅化过程中每一个顶点的位置，起初我们都是定义在模型空间中的，但是最终我们需要显示到屏幕上，那就需要实现从模型空间到最终屏幕的变换过程。

1、首先对于一个模型，我们需要将它转换到世界空间中，即需要对模型进行缩放、旋转、平移来改变模型在空间中的位置。这是第一个变换矩阵model，实现的是将模型由模型空间变换到世界空间中去。

2、之后，对于我们观察者而言，需要将这些模型转移到观察空间，即以“观察者”为中心的空间。对于“观察者”我们需要定义一个坐标系统，即三个相互垂直的坐标轴。front指向观察方向，后拥有一个右向量，一个上向量来组成观察坐标系统的参照标准。

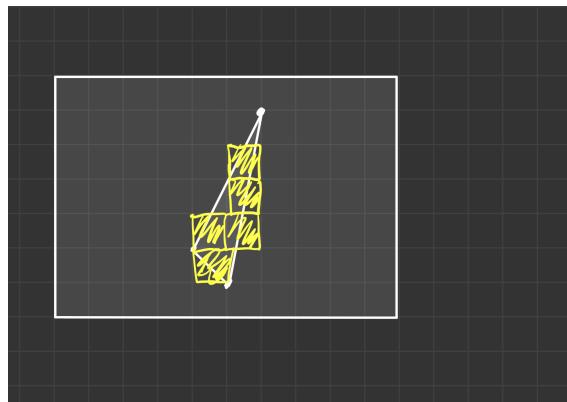
3、在转移到观察空间后，还需要确立我们的观察方式，由于画面最终显示到屏幕这一二维的“纸”上，所以所有模型此时需要经过某种变换压缩便于最后去除z值，从而映射到x, y坐标屏幕上。而这类变换常有两种。1、正交投影变换，简单的不考虑z，将所有顶点在x, y平面“拍平”。这种变换简单，但是相当“反直觉”，与我们正常人眼的观察截然不同。但此类投影效果在工程上则常用，如工程上常见的三视图效果就是如此。2、透视投影变换，透视法是符合观察效果的。透视法的视觉范围是一个棱台空间（某种情况也可认为是一个视锥）。若想得到符合常理的图像，那么就需要使用透视投影进行变换。透视投影变换将会把所有坐标变换到xyz均位于[-1,1]的范围内，之后根据z的数值进行zBuffer判断是否显示。下面进行透视投影这一过程的推导。



通过压缩+平移+缩放，就可以实现整个空间到NDC (Normalized Device Coordinates) 中。最后通过坐标映射、深度检测等就能够实现到屏幕空间的转换。

## 2.2 采样

采样 (Sampling) 是将图元离散化到由像素表示的过程。对于一个图元和屏幕中的像素点而言，直观的想法是如果像素位于图元内，那么就给它进行该图元部分的染色，否则就不染色。这种做法很简单。拿一个三角形图元（已经完成了基本的变换）为例，我们只要求出它x, y坐标的范围，通过计算这个方形范围内每一个像素中心是否在三角形内即可，如果是就把它添加到需要染色的部分。但是问题在于。仅仅如此简单的进行像素的选取，可能导致严重的走样现象的产生。



如图所示，如果这样简单的处理，像这种长、窄的图元走样现象将十分严重。产生这种走样的原因是采样频率过低（采样点过少），试想，如果使用更多，更小的像素块，就能够有效的减小这种走样现象。但显然这种方法不是很可取的，因为对于一台显示设备而言，它的像素点是基于硬件实现的，其像素点最多个数也无法超过一定空间内能够容纳的极限。那么就需要通过其他方法来处理。这种处理走样的方式称为反走样（Anti-aliasing）。

一种直观的反走样方式是模糊处理。即对于一个像素点，我们不简单的通过是否该像素的中心处于图形中而直接上色，而是综合其轴为各个点的颜色进行平均处理，实现边缘的柔化。对于一个像素点，我们可以将这个像素虚拟的分割为多个等大的小正方形，即将一个像素分为了 $n^2$ 个“小像素”，之后判断这 $n^2$ 个小像素有多少在图元中，计算出在图元中小像素所占所有小像素的比例，这一比例来对颜色做加权，即可实现模糊。这种方法称之为MSAA(Multi-Sampling Anti Aliasing)。

此外还可以通过对每一个像素进行多次采样，即每一个像素分割为多个像素，采样做颜色混合，这样也能进行反走样，但代价就比较高昂，应为将一个像素采样 $n^2$ 次，整体的计算代价也将变为 $n^2$ 。称之为SSAA(Super Sampling Anti Aliasing)。当然还有其他的一些反走样方法，但我也只是粗略了解，就不再赘述了。

### 2.3 深度检测

我们希望的虚拟环境是一个三维空间，即拥有X,Y,Z三个维度坐标的立体空间。但不管怎样，最终是现在我们面前的是显示屏幕，它仍然是一个2维化的平面。之前在变换部分提到了NDC空间是一个三维空间，那既然我们通过NDC空间的X,Y坐标直接映射到屏幕上，那这个Z坐标的用处是什么？

在三维空间中，物体和物体在空间中的非接触关系可认为有：1、完全无关。2、部分阻挡。3、完全阻挡。即3维空间中的物体会出现前后交叉阻挡的情况，而不简简单单是二维空间的覆盖。但最终实现于屏幕上时，我们希望能看到的就是这种覆盖的效果。那么Z这一坐标决定的就是物体之间的先后关系了。在渲染的过程中，我们可以通过zBuffer的方法来进行深度检测，从而实现正确地展示物体间的先后关系。

对于每一个像素点，我们不止缓存它的颜色，同样缓存它目前显示这一颜色在空间中的深度。如果之后又新的颜色同样映射到这一像素，那么就去判断缓存深度与当前深度的大小，将较浅的颜色实际显示即可。以下可以给出一段伪代码。

```

1 if(obj.deep < zBuffer[pixel_now].deep)
2 {
3     zBuffer[pixel_now].deep = obj.deep; //深度更新
4     zBuffer[pixel_now].color = obj.color; //颜色缓存更新
5 }
```

通过zBuffer方法就能够实现符合物理深度规律的显示效果。zBuffer由于是实时更新画面深度的，时间复杂度也较低为O(n)，在世界空间中所有的图元映射到像素时，都需要完成这一过程。

## 2.4 着色

对于世界空间中的各种物体，我们能够看到他们展现出不同的颜色。但显然，如果在周围环境中没有光线存在，就不会有光线通过反射、折射进入到“观察者”的“眼睛”中，自然对于观察者而言就无法看到颜色了。

那么，我们可以这样认为：对于一个物体而言，我们能够观察到它的颜色取决于：1、外界光源。2、物体表面材质对于光线的作用。3、光能量的变化。首先对于世界空间中，如果期望能够“照亮”空间中的所有物体，我们需要定义一个物理意义上的光源。下面首先来探讨光源的设计。

光源顾名思义就是光线的来源之初。但就如同我们生活中所熟知的，光并不只是一种单纯的电磁波，而是由多种不同波长的电磁波复合而成的。对于图形学而言，我们认为光线是由红 (Red)，绿 (Green)，蓝 (Blue) 光的三原色组成，人眼能够分辨的绝大多数颜色都能够通过RGB三个通量的线性组合得到，不难写出：

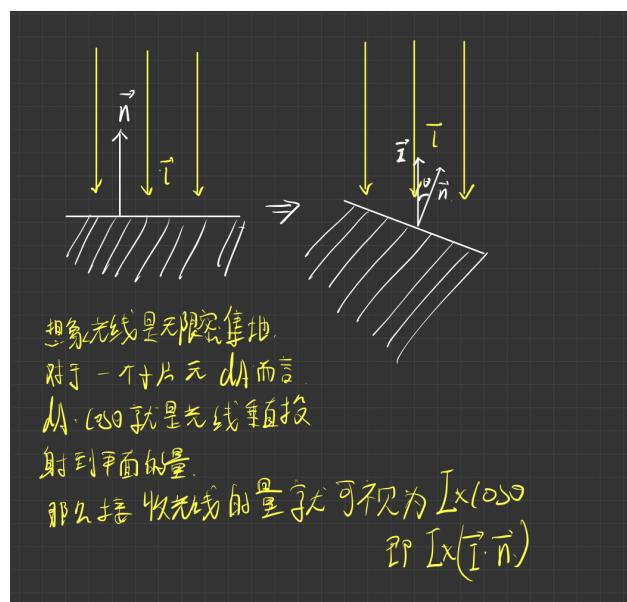
$$Color = \alpha * R + \beta * G + \gamma * B$$

那么，对于一个物理表面，我们只要得到能够反应其与光线交互性质的一个三维向量 ( $\alpha, \beta, \gamma$ ) 即可通过将所有光源的效果线性叠加得到表面的颜色了。

明白了物体表面颜色的来历，下一步就是解决如何得到这一组三维向量。首先一个感性的认知是：物理世界中的绝大多数物体都能够借由漫反射来交互光线，而对于表面十分光滑的物体，则因能够将入射的平行光反射为平行光而显得高亮。那么实际上，对于一个物体，我们就需要考虑它的漫反射效应 (diffuse) 以及镜面反射效应 (specular) 。

### 2.4.1 漫反射

对于漫反射而言，我们认为一道光线与平面相交，会均匀地发散到周围的空间中，即对于一个表面，它在世界中的颜色就是所有光线在其表面发生漫反射地叠加。我们认为单位面积表面垂直投射的光线具有相同的能力 (来自同一光源的每一道光带来的影响都是相同的)。以一个小平面为例说明如何量化平行光的效果。



我们不妨假设：物体表面材质为material, 光线为light, 那么漫反射部分就可表示为：

$$diffuse = material.diffuse * light.diffuse * dot(normal, lightDir) * power$$

对于本式中的前两个因子，取决于我们设定的物体材质与光线性质。第三项为着色面法向量与光线方向的点乘 (都是单位向量)，最后一项为光线的强度。

## 2.4.2 镜面反射

对于表面比较光滑的物体，当平行光投射在其表面时，会发生镜面反射，形成高光的效果。对于观察者而言，如果当反射光线正好进入观察者眼睛时，就能够看到这种高光效果。那么我们很自然地就能够通过从物体表面到观察者的方向 ( $\text{viewDir}$ ) 以及光线到物体表面的方向 ( $\text{lightDir}$ ) 来判断是否形成高光效果。根据反射定律，反射光线与入射光线关于法线两边对称，如果  $\text{viewDir}$  与  $\text{lightDir}$  的角平分线与平面法线  $\text{normal}$  角度相近，就可以视为大量的光线进入了观察者视野。通过下面的式子可以合理的表示这一效果。

$$\text{specular} = \text{light. specular} * \text{material. specular} * ((\overrightarrow{\text{lightDir}} + \overrightarrow{\text{viewDir}}) \cdot \overrightarrow{\text{normal}})^m * \text{Intensity}$$

本式中的前两个因子表示光线和物体材质的高光属性，后为半程向量和法向量的夹角的  $m$  次方，当  $m$  越大时，意味着高光范围越小，只有与法向量夹角十分接近的项才能够产生足够的镜面反射光。

## 2.4.3 环境光

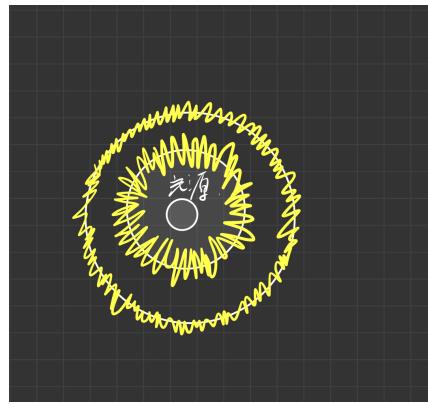
在Phong以及Blinn-Phong模型中，由于光线并没有“真正”被反射，所以所有物体表面的颜色都是光源效果叠加而成，只有直接光照的效果。但现实中，不论是哪一种反射，光线均会不停歇的弹射，比如一间屋子只要有一束光线存在，许多光线无法直射的地方也会显示出颜色，而不是完全的黑色。那么我们就需要通过某种手段“模拟”这种光线无限弹射引起的整体光亮现象。这就是环境光 (ambient)。对于环境光，我们通常只给它一个很小的数值，保证所有物体不会是纯黑的即可。这样比较符合在客观世界的认知规律。

$$\text{ambient} = \text{light. ambient} * \text{material. ambient}$$

## 2.4.4 光照衰减

对于光源而言，认为它用于一个固定的发光强度  $\text{Intensity}$ 。对于以光源为中心的球形表面，仍然是一个二维平面。由于能量守恒的原因，光线在距离中心不同的二维球面上做面积分，即对于各个方向强度于整个二维球面积分得到的结果应该是一个定值，即光源的发光功率。那么易知在距离中心为  $r$  的位置，各个点的强度应为： $\text{Intensity}/r^2$ 。（如图，在距离中心近处，功率更大，远处则相反）

对于光照衰减方式，同样有其他的衰减函数用来表示距光源不同距离的效果。可以引用更多的次方项计算出不同的效果，这里不再说明。



在说明完以上四部分内容后，就可以详细说明具体的光照模型以及着色方式了。

## 2.4.5 Phong光照模型

Phong光照模型是一个简单的光照模型，仅使用环境光，镜面反射，漫反射来实现光照效果。其中其漫反射、环境光与上文提到的计算方式一致。但在计算镜面反射时，不使用半程向量而是使用光线方向与法线的夹角来替代。

## 2.4.6 Blin-Phong光照模型

直接使用上文提到的三项线性组合即可。

## 2.4.7 着色方式

### 1、平面着色 (Flat Shading)

平面着色以平面为基本着色单元。当需要着色时，仅仅根据平面三角形三个顶点的法线求出平面的法线，然后直接将平面根据这一个法向量进行着色计算。有点是计算快捷，耗费资源少，但是相对效果也比较差。

### 2、逐顶点着色 (Gouraud Shading)

这一方法首先需要计算出各个顶点的颜色。每个顶点根据自己的位置、属性、法线等计算该顶点处应具有的颜色。之后对于三角形片面内的像素则是直接通过求取其中心在三角形重心坐标系下的坐标来对顶点进行线性插值得到。这种方法比第一种好不少，但同样存在细节上比较差的问题。

### 3、逐像素着色 (Phong Shading)

顾名思义，这一种方法是对平面的每一个像素单独着色。具体做法是计算顶点的位置，法向量，通过线性插值得到每一个像素的法向量，再更具光照模型对这一像素进行光照计算。由于需要对每一个像素都单独计算着色，这种方法的代价比较大，需要消耗大量的计算量，但效果也最好。

以上三种着色方式各有优缺点，但如果随着顶点个数不断增加，效果则会越来越接近，但对于计算量的要求差距还是很大的。当模型足够精细时，就可以选择Flat Shading或者Gouraud Shading来进行着色，同样可以取得不错的效果。

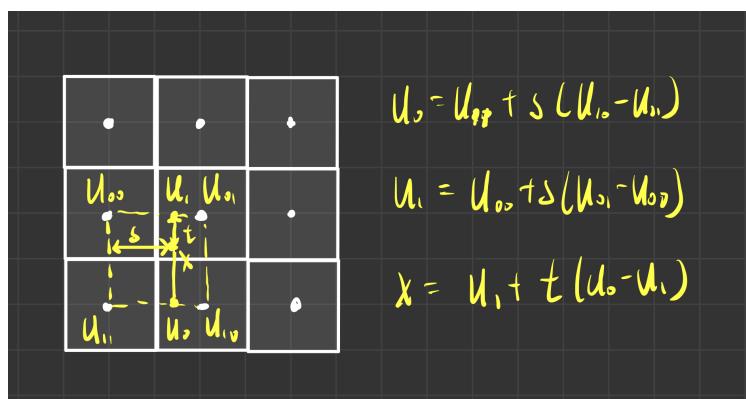
## 2.4.8 纹理采样

对于一些物体，我们建立其模型结构后，希望能够为它盖上一层“蒙皮”，来实现真实的表面效果。但如果想要一点一点去定义模型表面的颜色则十分困难，一种简单的方式是使用一张图片对应的覆盖到物体表面即可，这种图片就是纹理贴图。对于每一个像素，它在纹理上都应该对应一片区域，如果我们希望纹理能够正确“贴”上，那么就需要确定这种对应关系。

对于三角形图元来说，有一个特性是在重心坐标系统下，三角形内的每一个点都可以通过三个顶点坐标的线性组合来表示。例如：对于一个三角形，其顶点 $p_i$ 的坐标为 $x_i, y_i$ ，那么三角形内的每一个点都可以表示为： $p = \alpha * p1 + \beta * p2 + \gamma * p3$ 。如果我们能够找到三角形三个顶点在纹理贴图上所对应的坐标，那么对于这个三角形内任何一点的对应纹理坐标 $u_i, v_i$ 都可以通过三个顶点的线性插值得到。

对于纹理采样时，可能会出现纹理放大以及缩小的情况。

纹理放大现象是一种易于处理的情况。这指的是一个像素点不能刚好对应到一个纹理颜色时的情况，如果直接选择最近的纹理区域进行采样，就会出现边缘的锯齿情况。处理方法就是对于一个像素中心，在寻找到它对应的纹理坐标后，对其最近的4个像素值进行双线性插值进行平均化处理，以此来使得像素之间的过渡显得平缓。



纹理缩小现象就比较难以处理了。对于一个像素，它如果覆盖了很多纹理像素时，如果不加以处理，会导致摩尔纹的出现，相邻像素间的颜色出现了较大差异，导致失真。一种方法是超采样，通过增加采样点进行混合，降低这种情况。另一种方法是Mipmap，通过将原纹理等在面积上进行25%的压缩，实现对于过大的像素覆盖区域能够找到一个合理的缩放等级来使纹理贴图能够比较好的贴合像素。从而实现平滑的过度。对于不能正好找到对应缩放等级的像素，则可以根据其大小对应该的两个缩放等级进行线性插值来得到合适的纹理。更好的做法是进行各向异性过滤，即对水平、竖直方向分别进行0.5倍的压缩，并存储压缩过的纹理，在需要进行mipmap时，直接调用相应的纹理进行插值计算即可。

好的纹理采样能够在保证计算量不大量增长的情况下显著提升真实感。实际上，对于任何纹理而言，其本质都是一个二维平面，我们所确定的仅仅是将这一平面按一定要求映射到物体表面。实际上，对于一个纹理，我们希望它能够显示出与其图像相近的物理效果（比如：我们不希望一个木头箱子能够像金属箱一样拥有大量镜面反射）那么就要理解不同贴图的含义。

1、漫反射贴图。顾名思义，漫反射贴图影响物体的漫反射效果，它通过贴图的“颜色”值来代替漫反射项中的漫反射常数。一个物体表面的漫反射就决定它在视觉中的颜色，因此漫反射贴图就是物体本身的颜色属性。

2、镜面反射贴图。当光滑的表面（如金属表面）接收到光照时，就会发生镜面反射的情况。同样，我们使用镜面反射贴图来替代物体的镜面反射系数，以此来实现物体的区域性高光效果。

3、法线贴图。法线贴图在原物体的每个表面都做法线，这个法线不一定与物体实际的真实法线方向一致。通过法线贴图，可以表现物体表面具有不同的高度效果，从而实现视觉上的凹凸感。虽然实际上物体表面仍然是光滑平整的，但通过法线贴图就能够从视觉角度使得其具有高度信息，实现高细节的精确光照和反射效果。

4、置换贴图。相较于法线贴图而言，置换贴图同样表现得是物体的高度信息，但置换贴图会将模型从原有的网格进行替换以及细分，从物理层面改变模型，实现真实的凹凸感。但缺点是会产生大量的额外计算。

### 阴影

在光栅化的部分，我们一直关注那些应该被照亮、计算的部分，对于阴影则一直没有提到。由于光栅化计算的仅仅是来自光源的直接光照，那么与我们认知相同的是，如果一个位置没有被光照射到，它就应该是所谓的阴影区域。实际问题是如何在光栅化的过程种找出那些阴影区域。

首先我们知道，如果观察空间中，“观察者”仅能看到最前方与观察者视线相交的位置。对于光源同理，仅有与光线第一个相交的位置才能够被照亮。那么观察者能够看到的条件就是1、该位置被光源照亮。2、该位置能被观察者直视。

在zBuffer部分，我们提到深度检测就是记录了每一个像素当前的最浅深度的一张二维表。如果我们将观察者视为一个“光源”，视线就是“光线”，那么我们也能够通过同样的方法得到这样一张深度二维表。将两张表进行比对，如果同一个像素在两张表的记录深度一致，那该像素就是可见的，否则就是阴影中的。

通过实现一张Shadow Map，我们就能够轻松处理阴影效果。值得注意的是，如果环境中存在多个光源，需要将每一个光源的效果都与Shadow Map进行计算处理，只要能够被至少一个光源照亮，那么该像素就是可见的。

到此光栅化部分就已简单的说完了。其实对于光栅化而言，更多会应用于实时渲染，光栅化拥有足够的渲染速度，不错的渲染效果。但是，对于完全的“真实感”而言，光栅化的效果还是不够的，我们需要更好的方法。

## 3. 几何

在图形学的世界中，一切可见物体均有点、线、面组成。且一个重要的点是，在图形学中，我们永远能看到的是物体的表面。所有的实际立体，均是由一个又一个面组成的。那么，如何构造出符合我们需要的点、线、面就是构造场景、物体的关键了。

### 3.1 曲面方程

最理想的一种构造曲面方式是通过数学函数表达式来进行定义。如果们能够得到一个曲面的函数表达式，那么对于绘制曲面就十分简单了，只需要通过带入表达式求值即可。但曲面方程分为两个大类。1、隐式曲面。2、显示曲面。

#### 3.1.1 隐式曲面

以三维物体为例，一个隐式曲面方程可以表达为： $x, y, z$ 的计算组合形式。即实际曲面的每一个点都通过真正的坐标 $x, y, z$ 来定义。例如一个单位球： $f(x, y, z) = x^2 + y^2 + z^2 - 1$ , 当 $f(x, y, z) = 0$ 时点 $(x, y, z)$ 就在曲面上。否则不属于曲面。

隐式曲面的缺点显而易见，我们无法简单的找到所有曲面上的点，对于任何一个点，我们都需要满足方程 $f(x, y, z) = 0$ ，确定曲面上的所有点是相当困难的

不过同样它也具有优点，那就是判断一个点与曲面的关系变得简单。如果知道一个点 $p(x, y, z)$ 只需将其带入到方程中，如果等于0，则该点就位于曲面上，如果小于0，则位于曲面内，如果大于0则位于曲面外。

#### 3.1.2 显示曲面

显示曲面方程的特点是不直接表现点的坐标，而是通过参数进行计算映射。显示曲面方程是一个由 $(u, v)$ 映射到 $(x, y, z)$ 的过程。在一个二维平面上定义了 $(u, v)$ 面的形状，再通过方程记录映射关系，即可通过这样的一个参数方程方法实现曲面。

对于这样的一个参数方程，采样是很容易的。通过参数的变化范围可以快速确定物体表面的所有点。这是显示曲面方程的优点。

相对地，它的缺点就是难以判断一个点与该曲面的关系。因为我们很难从 $(x, y, z)$ 转换为 $(u, v)$ 中。毕竟两个方程使用的不是同一个坐标体系。

### 3.2 CSG体素构造法

CSG (Constructive Solid Geometry) 方法的基本思路是自底向上，由简单物体进行一系列计算逐步地构造复杂的模型。

基本的体素就是如立方体，球体，圆柱体……简单的立体造型。对于这些立体造型之间，我们可以将他们摆放在适当的位置就行一系列布尔运算。物体与物体间可以求并、求交、求差……当大量的体素以及计算构造成一个布尔表达树时，最终位于树根的就是我们得到的模型。

由于CSG方法时从最基本的体素开始，一步步向上构造的，这也就赋予了它足够的创造力，能够由简到繁构造出足够复杂、精细的模型。

### 3.3 贝塞尔曲线

贝塞尔曲线是一种特殊的曲线。一条贝塞尔曲线可以由n个控制点得到。贝塞尔曲线有如下特性：1、起点为 $p_0$ ,终点为 $p_3$ 。 $\rightarrow$  2、初始切线方向为： $(p_1 - p_0)$ , 终点方向为： $(p_n - p_{n-1})$ 。 $\rightarrow$

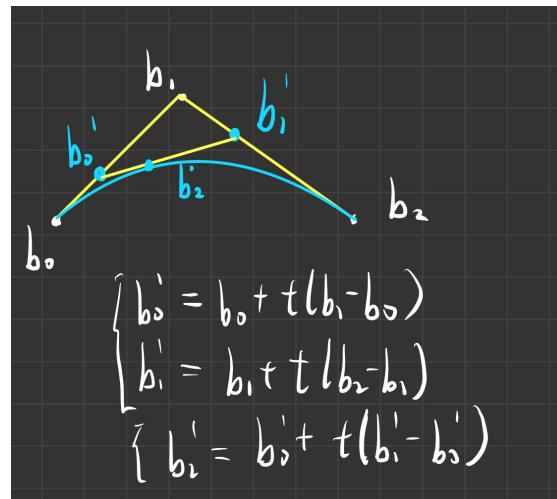
根据控制点的不同，可以构成不同阶的贝塞尔曲线。通过最简单的2阶贝塞尔曲线来介绍其构成方式。

#### 3.3.2 二阶贝塞尔曲线

设平面中有三个点 $b_0, b_1, b_2$ ,以这三个点为控制点来构造贝塞尔曲线。贝塞尔曲线的核心思想仍然是一种线性插值方法。我们假设有一个时间为 $t(0 \leq t \leq 1)$ ，在t时刻，线段 $b_{k-1}, b_k$ 上有一点 $b_k^1$ ,该店满足以下关系式：

$$b_k^1 = b_{k-1} + t(b_k - b_{k-1})$$

由此我们可以确定两个点 $b_0^1, b_1^1$ , 对于这两个点构成的线段, 可以使用同样的方式构成下一个点 $b_0^2$ , 此时已经没有更多的新产生的线段可以用于插值了。那么这个 $b_0^2$ 就是我们需要的最后一个点。根据时间 $t$ 的变化, 该点会在平面中处于不同位置, 故以 $t$ 为参数的参数方程就是这条贝塞尔曲线的方程。

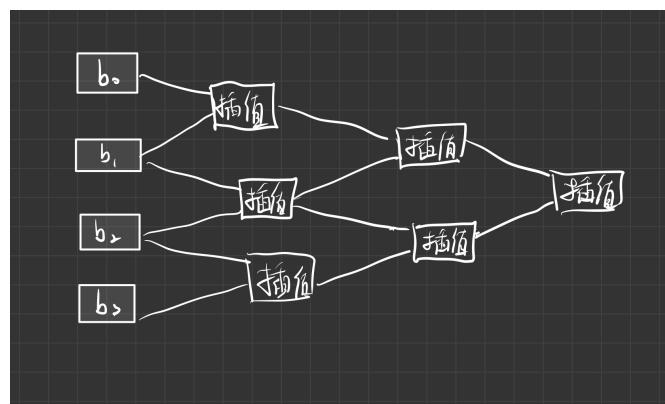


很容易发现的是, 这是一个递归的方法, 每次新产生的点将构成新的新段, 一致递归下去直到产生最后一个点为止。那么很自然, 我们可以通过更多的控制点来形成贝塞尔曲线。使其能够更能够达到我们希望的效果。

### 3.3.2 三阶贝塞尔曲线

在刚才已经说明了二阶贝塞尔曲线的形成, 如果再多一个控制点, 即使用四个控制点来形成贝塞尔曲线, 会是怎样的呢?

在二阶贝塞尔曲线中, 我们共计算了两层。同样, 在三阶贝塞尔曲线中, 我们需要计算三层。由原来的4个点可以得到3个新的点, 在由此继续递归, 即可类似于二阶贝塞尔曲线得到最终的点 $b_0^3$ 。下面给出一个三阶贝塞尔曲线的递归计算过程:



在前面的推导中, 我们可以发现, 两个点可以保证起止位置一定, 三个点可以保证一阶导数一定, 四个点可以保证二阶导数一定。那么对于n个点的贝塞尔曲线是否有类似的性质呢? 下面进行数学上的推导。

### 3.3.3 贝塞尔曲线通项

以二阶为起始：

$$b_0'(t) = (1-t)b_0 + tb_1 \quad ①$$

$$b_1'(t) = (1-t)b_1 + tb_2 \quad ②$$

$$b_2'(t) = (1-t)b_2 + tb_3 \quad \text{代入} ①, ②$$

$$b_3'(t) = (1-t)^2 b_0 + (1-t)(b_1 + t(1-t)b_2) + t^2 b_3$$

$$= (1-t)^2 b_0 + 2t(1-t)b_1 + t^2 b_3$$
  

以n阶：

$$b_0^n = b_0^{n-1} + b_1^{n-1}$$

$$b_1^n = b_0^{n-2} + b_1^{n-2} + b_2^{n-2} + b_3^{n-3}$$

$$= b_0^{n-2} + 2b_1^{n-1} + b_2^{n-3} \quad \text{其中 } B_j^n(t) = \binom{n}{j} t^j (1-t)^{n-j}$$

$$b_0^n(t) = \sum_{j=0}^n b_j \cdot B_j^n(t)$$

由此，我们在知道所有基础控制点的情况下，可以通过公式得出任意阶贝塞尔曲线的参数方程。通过这种方法，可以构造出各种各样的复杂曲线。

已经能够构造任意阶的贝塞尔曲线了？那实际中是否是越高阶越好呢？答案是否定的，如果我们希望生成一条拥有大量曲折的贝塞尔曲线，那么需要的控制点数量将是十分可怕的。

理想的一种做法是通过三阶贝塞尔曲线进行阶段构造。在前文中，我们已经认识到对于三阶贝塞尔曲线来说，它能够保持起点、一阶导数、二阶导数确定。也就是说在曲线连接处如果以上一条贝塞尔曲线的末尾两个点为新起点的话，就能够保证下一组贝塞尔曲线与上一组的连续性。实际上我们常用的也是这一种方法。通过每4个相邻顺序的控制点来分组进行构造贝塞尔曲线，能够保证在曲线光滑连续的情况下同样能有较好的控制效果。这种构造贝塞尔曲线的方法就是分段贝塞尔曲线。

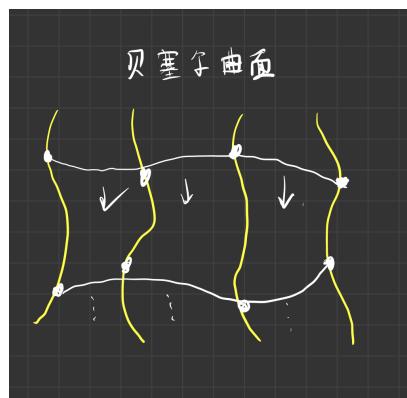
分段贝塞尔曲线在计算量和实现效果上实现了比较好的平衡。此外还有样条曲线等方式，这里就不再介绍了。

### 3.3.4 贝塞尔曲面

在上一部分已经说明了贝塞尔曲线，但我们的目的是通过一个个曲面来构造立体模型，下一步就是如何通过曲线构成贝塞尔曲面。

在贝塞尔曲线中，通过n个控制点得到的贝塞尔曲线在最后会得到一个表示贝塞尔曲线的绘图点，该点是一个以  $t$  为参数的方程。本质上，曲面可以看成是在两个维度上足够多的贝塞尔曲线交织而成。如果我们在一个方向（例如X方向）已经得到了若干条贝塞尔曲线，之后只需要在Y方向上再次构造即可得到贝塞尔曲面。

直观的来说，得到n条贝塞尔曲线时，最终其实得到的就是绘图点。当我们把这n个绘图点作为新的控制点时，会在一个新的方向时得到一条贝塞尔曲线，当参数  $t$  由0到1时，这一条新的贝塞尔曲线与直线方向的贝塞尔曲线就会得到贝塞尔曲面。



贝塞尔曲面的构建可视为是由两个方向的贝塞尔曲线的共同构成。整个贝塞尔曲面的核心仍然是主方向上控制点的选取，它控制着整个贝塞尔曲面的弯曲程度以及走向。同样在构造贝塞尔曲面时，通常可以根据设置贝塞尔曲线参数方程的方法直接通过所有的控制点得到贝塞尔曲面的中心绘图点。

### 3.4 网格变换

在网格模型的实际使用过程中，根据环境的不同我们对于其表面的网格数量常常有着不同的要求。在模型距离观察者比较近时，我们希望能够观察到更加平滑、具有细节的表面；而在模型距离较远时，当模型表面网格变少时则不会大幅度影响观感，我们就希望能够减少模型表面网格的数量，以此来降低计算量。

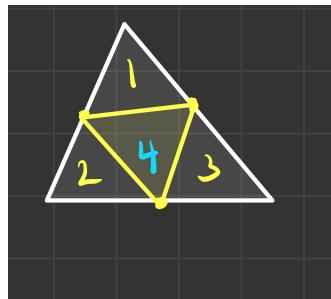
那么，对于一个确定的网格模型我们就需要有细分以及粗分的方法。

#### 3.4.1 曲面细分

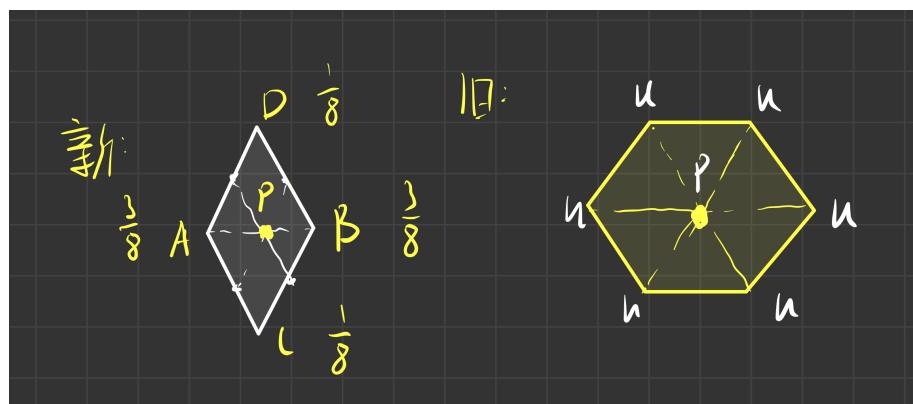
曲面细分最终希望得到的是更平滑的表面，更多的网格。反映到模型的基本构成单位——顶点，要求就是构造出更多的顶点，形成更多的三角形。

##### 1、Loop Subdivision

以一个最简单的三角形图元为例。当我们找到一个三角形三条边的中点，并将其相连时，原本的一个三角形就会被新得到的三条线分割为4个三角形，这就是一种朴素的曲面细分思想。具体效果如下图所示。



在完成这一部分后，需要面对的问题就是如何处理旧的点与新的点的位置问题，这里不加证明的给出计算公式。



对于在处理过程中新产生的顶点，找到共享该点所在边的两个端点  $A, B$ ，以及该边所对的两个顶点  $C, D$ ，以  $P = 3/8A + 3/8B + 1/8C + 1/8D$  进行计算，得到的坐标就是新加入顶点的坐标。

在加入新顶点的过程中，由于新面的加入，旧顶点的位置同样需要改变。对于如图所示的情形：我们以下面的方式计算：

$$P = (1 - nu) * \text{original\_position} + u = \text{neighbor\_position\_sum}$$

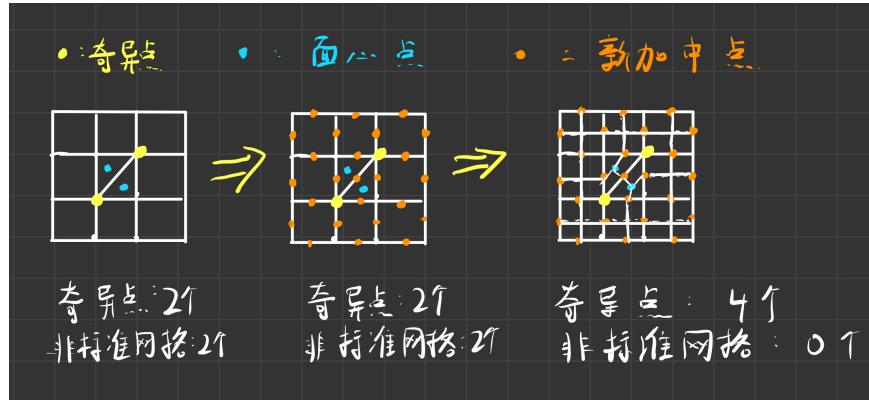
其中： $n$  表示该顶点的度，对于  $u$  则分两种情况1、 $n = 3, u = 3/16$  否则  $u = 3/(8n)$  通过这种方法就可以实现曲面的细分。

对于Loop Subdivision而言，有一个巨大的缺陷是该方法仅仅对于三角形网格有效，如果网格模型是其他多变形，则无法使用这种方法了。

## 2、Catmull-Clark Subdivision

此方法对于多边形网格同样有效。这里仅作一个简单的说明。

对于一个网格模型中的所有顶点而言，如果该点的度为4，那么就是一个平凡点，否则认为这是一个**奇异点**，其中**奇异点**周围的网格必然有一部分不是标准的四边形网格（Non-quad-face）。Catmull方法的核心就是找到所有**奇异点**以及这些**非四边形网格的中心点**。之后在所有网格体的所有边都添加一个新的中点，后将所有新添加的中点以及**非四边形网格中的中心点**相连，就可以形成新的细分网格。在这一个过程中，所有的**非四边形网格**都被修正为四边形网格，同时，**奇异点**的个数也确定为初始个数加上面中心点的个数。具体变化如图所示。

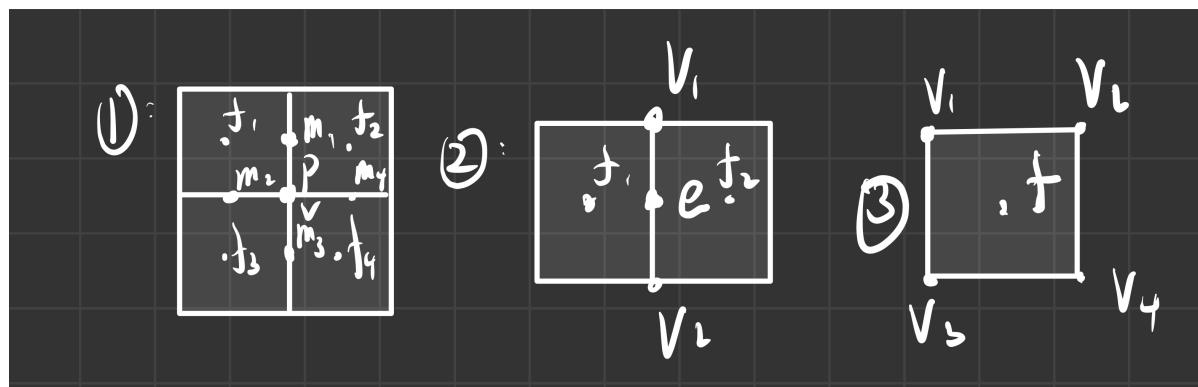


在这之后，就可以直接在所有标准网格的边上增加终点连接，进行细分了。接下来考虑的问题是，如何更新细分前后新顶点以及旧顶点的坐标。在这方面，我们共有3种特殊的点需要处理：1、顶点。2、边中点，3、面心点。下面直接给出这三种点的坐标更新方式。

对于顶点： $v = \frac{f_1+f_2+f_3+f_4+2(m_1+m_2+m_3+m_4)+4p}{16}$  其中 $m$ 代表边中点， $p$ 代表旧顶点

对于边终点： $e = \frac{v_1+v_2+f_1+f_2}{4}$

对于面中心： $f = \frac{v_1+v_2+v_3+v_4}{4}$



到此为止，就说明了两种曲面细分方法以及实现方式。

### 3.4.2 网格简化

说完了网格细分方式，下面就是网格简化方式。在细分过程中，通过增加新顶点为主要思想来增加边和面，而在网格简化则相反，我们希望能在尽可能保持物体原貌的情况下，减小构建其形体的网格数目。那么在构造过程中，首先想到的就是我们从影响整体效果最小的网格开始消去，以求消去该部分后造成的影响尽可能小。

在网格简化的核心单位是边，当我们将边塌缩时，与它相邻的顶点和面也都会进行收缩。这里使用的方法是二次误差度量法。在进行边塌缩时，我们希望的是在消去一部分边后，剩余的轮廓形状与原形状能够尽可能保持一致。

二次误差指的是边塌缩后得到顶点到所有与其相关边的距离平方和。当这个距离平方和最小时，塌缩的这条边后形成的最佳顶点的位置，就是对于整个模型效果影响最小的情况。

具体方法就是计算每条边塌缩后的二次度量误差，每次都去塌缩最小的那一条几个。问题在于，之前塌缩的边会对所有与它相连的边造成影响，即我们在塌缩过程中需要动态的更新所有边的二次度量误差，同时需要每次得到最小的情况。这时需要用一个小顶堆来处理即可。

通过以上方法，就可以实现动态地进行网格简化，越不明显地部分进行地简化越多即可实现在显示效果以及计算量上的平衡。

## 4、光线追踪

渲染的两大方式，一为光栅化，二为光线追踪。前者计算量小，显示效果不错，故在实时渲染领域大展拳脚。后者则是一种高代价，高回报的渲染方式。大量的计算量、时间消耗，换来的是真实、符合物理效果的渲染结果。这一原因也使得光线追踪难以在实时渲染领域运用（虽然当下已经有部分光线追踪技术能够实时渲染），但在追求高质量而对时间要求没有那么严格的离线渲染领域，光线追踪方法就成为了主流。在这一部分，将说明传统的光线追踪方法及其实现过程，实现过程优化。以及当下在工业界流行的路径追踪方法和其中需要了解到的辐射度量学理论。在最后，还将说明以下基于物理的渲染方法，微表面模型等。

### 4.1 光线追踪基本理论与算法

了解光线理论，我们需要知道为什么需要光线追踪。光线追踪耗时巨大，但能够尽可能真实地模拟光线在场景中传播的过程，让画面尽可能符合真实的物理规律。而且在影视、效果图等方面只要求最后高质量的图像产出，并不要求实时地动态效果，那么使用光线追踪技术就变得合情合理。

在光线追踪的过程中，我们所说的光线是几何意义上的，也就是说我们需要忽略一部分显示光线的特性。几何化的光线有如下三个要求。

- 1、光沿直线传播，并不会同一种介质内部的变化而扭曲。
- 2、光线与光线之间不会产生碰撞，每一条光线都是独立的。
- 3、光线从光源发出进入到观察者的“眼睛”，且在光路逆向的情况下物理效果不变。

在渲染时，我们认为显示在屏幕上的图像就是观察者眼睛所能看到的所有物体。而我们同样知道光路是可逆的。光线是从光源发出，经过一系列的散射，进入了观察者的眼睛。同样，我们在渲染的意义上也可以认为观察者的眼睛发出了若干条检测射线，经过若干次散射，最终到达光源的这一部分成为了光线。有了这一概念后，下面就可以说明**光线投射方法**。

#### 4.1.1 光线投射

在显示设备上最终进行显像的是一个又一个像素，这是组成图像的最小单元。同时一个像素仅能在同一时刻显示出一种颜色。由此，我们就可以知道，现实中的连续化的视觉图像会被转化为一个个像素组成的离散化的点阵。而对于像素而言，又可以视为是观测者的“摄像机”。根据上一部分所说明的部分，我们可以将每一个像素当成一个“摄像机”。那么想让它拍摄到场景中的画面，我们就可以以观察者的眼睛为起点，向每一个像素中心发射光线，之后判断该光线是否与场景中的物体表面相交。如果相交则求出交点，通过光照模型计算交点处的光照效果即可。下面给出伪代码。

```
1 vec3 RayCasting (eye_pos, pixel)
2 {
3     dir = normalize(pixel - eye_pos);
4     if(intersection(dir, scene))
5     {
6         return(calculateColor());
7     }
8     else
9     {
10        return vec3(0,0,0);
11    }
```

通过光线投射，貌似其实现与光栅化最终也没有什么不同，计算的仅仅是来自光源的直接光照效果，同样没有考虑物体之间散射、折射产生的间接光照的效果。但不同的是，在这里我们引入了光线的概念，同样在光线投射的过程中，有一部分物体时没有办法与光线相交的，这也自然地解决了阴影的问题。

#### 4.1.2 Whitted-Style 光线追踪

在介绍了光线投射的概念后，就是光线追踪了。我们引入光线追踪的最终目的是能够通过这种方法实现间接光照的计算，达到更好的真实感效果。

光线投射的光线碰到物体表面后，我们希望它能继续的是：发生表面反射、折射，继续传播。光线追踪实际实现的就是这一部分。光线在触碰到物体表面后并不停下来，而是以此为一个新的起点，确定散射方向继续传播，不断进行下去，永不停止。显而易见，这是一个递归的过程，既然如此，我们就一定要设定一个终止条件防止其无限递归下去。当下我们能够设置的就是递归的深度，通过这一点来控制光线发生散射折射的次数。在递归到最深返回时，我们就可以再次通过当前的交点计算这一点的光照效果，然后再将其返回上一次，层层叠加，最终计算出光线传播过程中每一个交点处的光照效果。

需要明白的是，再光线追踪过程中，一条光线仅仅能反射到一个方向，也就是说这一说的反射一定是镜面反射。对于一个确定的接触表面，一条光线只有三种散射情况：1、仅反射。2、反射、折射。3、不再额外散射。对于这三种情况，需要分别计算继续传播光线得到的颜色。下面给出伪代码。

```
1 vec3 castRay(ray, depth)
2 {
3     if(deptm > maxDepth)
4     {
5         //大于最大递归深度
6         return vec3(0.0, 0.0, 0.0);
7     }
8     if(intersection(ray, dir))//发生碰撞
9     {
10        switch(case)
11        case1:{//碰撞表面既能反射又能够折射
12            //反射颜色
13            reflectionColor = castRay(ray_next, depth+1);
14            //折射颜色
15            refractionColor = castRay(ray_next_2, depth+1);
16            //最终碰撞点的颜色
17            hitColor = reflectionColor * kr + refractionColor * (1-kr);
18            break;
19        }
20        case2:{//仅计算反射颜色
21            hitColor = castRay(ray_next, depth+1);
22            break;
23        }
24        default:{//都不发生的表面
25            //分别计算漫反射项与高光项即可，这里需要计算所有光源的共同贡献
26            hitColor = diffuseColor() + specularColor();
27            break;
28        }
29    }
30 }
31 }
32 return hitColor;
33 }
```

其中的一些点就是计算光线的反射方向，折射方向等等，这些计算与具体的物体材质以及物理性质有关，这里就不再过多赘述了。

再伪代码中，光线由一个Ray类的对象进行表示。真正的难点便是如何判断光线与物体相交以及求出交点，下面就将说明这个问题。

对于一个几何化的光线，我们可以为它建模如下：

```

1 class Ray{
2     vec3 orgPos;      //光线起点
3     vec3 dir;         //光线方向
4     float t;          //发出“时间”
5 }
```

这样我们就得到了一个光线类。对于任意时刻，我们就可以得到光线传播到的坐标，即：

$$r(t) = \vec{o} + t * \vec{d} \quad (0 < t < \infty)$$

对于三角网格而言，我们直接求光线是否与其相交还是困难的，不过我们容易得到的是判断光线是否与三角形所在的那个平面相交。

对于一个三角形平面，通过叉乘很容易得到其平面法向量，再通过三角形三个顶点中的任意两个构成的直线，就能够得到其所在平面的方程：

$$p : (p - p') * N = 0$$

$$ax + by + cz + d = 0$$

在同时拥有光线方程与平面方程后，我们就能够求解了。如果光线与平面相交于p点，那么不难得出：

$$(p - p') * N = (o + td - p') * N = 0$$

$$t = \frac{(p' - o) * N}{d * N} \quad (0 < t < \infty)$$

由此我们就能得到光线与平面相交的交点p，之后，只需要判断这个交点是否在三角形内即可。这里可以使用在光栅化部分提到的方法，即通过重心坐标的线性组合三者是否均大于1来求解。对于三角形而言，同样具有一个特性，三角形所在平面的所有点都可以通过三角形重心坐标来进行线性表出，故可以写出下面的式子。

$$\vec{O} + t\vec{D} = (1 - b_1 - b_2)\vec{P}_0 + b_1\vec{P}_1 + b_2\vec{P}_2$$

只要  $b_1, b_2, 1 - b_1 - b_2$  均位于  $[0, 1]$  即可说明光线与三角形  $P_0 P_1 P_2$  相交。

根据这个式子，使用克莱默法则可以进行快速计算。

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\vec{S}_1 \cdot \vec{E}_1} \begin{bmatrix} \vec{S}_2 \cdot \vec{E}_1 \\ \vec{S}_1 \cdot \vec{S}_2 \\ \vec{S}_1 \cdot \vec{D} \end{bmatrix}$$

$$\vec{E}_1 = \vec{P}_1 - \vec{P}_0$$

$$\vec{E}_2 = \vec{P}_2 - \vec{P}_0$$

$$\vec{S} = \vec{O} - \vec{P}_0$$

$$\vec{S}_1 = \vec{D}_1 \times \vec{E}_2$$

$$\vec{S}_2 = \vec{D}_2 \times \vec{E}_1$$

这样，就可以判断光线是否与三角形网格相交了。

#### 4.1.3 光线追踪加速

光线追踪过程需要对所有像素点发射一条光线，并追踪每一条光线的散射流程，当场景内物体十分空旷时，有的光线根本不可能与物体表面相交，那么这一部分光线就完全不需要考虑与物体表面相交的情况，由此就可以大大减少计算时间。我们可以通过某种方法来初步判断光线所经过的地方是否有可能发生相交。

对于任意物体，我们使用6个平面可以将其完全包裹起来，只要找到该物体在X,Y,Z三个坐标轴上的最大、最小值即可构建这些平面，由于它们是与轴平行的，就称之为**轴对齐包围盒(Axis-Aligned Bounding Box, 后称为AABB)**，对于一条光线，如果它都不进入AABB，那它就更不可能与其内部的物体相交。下一个就是如何判断光线与AABB相交的情况。

对于X,Y,Z三个方向的任意一组平面，我们可以计算出光线与其相交的两个时间 $t_{min}, t_{max}$ ，记较小的为 $t_{min}$ ，较大的为 $t_{max}$ ，那么，不难发现，光线进入AABB的初始时间为3个 $t_{min}$ 的最大者，记为 $t_{enter}$ ，光线离开AABB的结束时间为3个 $t_{max}$ 的最小者，记为 $t_{exit}$ 。

如果 $t_{exit} > t_{enter}$ 且 $t_{exit} > 0, t_{enter} > 0$ ，在这段事件中，光线必然是经过AABB的。即它可能与内部物体相交。

如果 $t_{exit} < 0$ ，说明光线传播方向早已越过物体，即光线在AABB后方传播，这种情况下不可能相交。

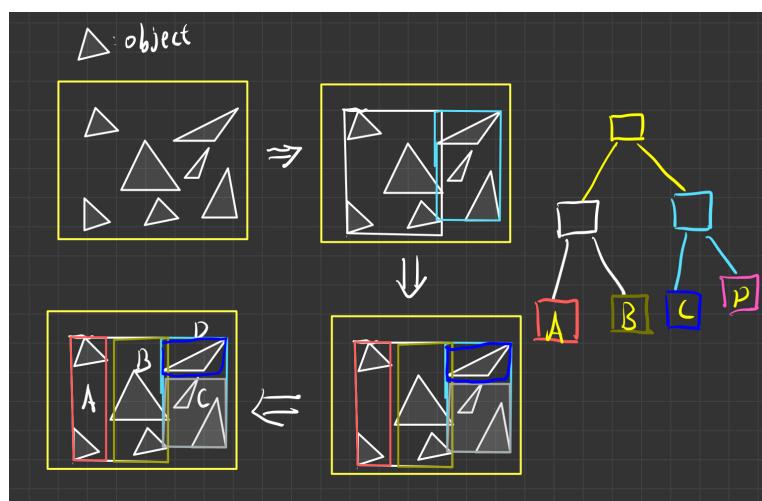
如果 $t_{exit} \geq 0$ 且 $t_{enter} < 0$ ，则说明光线在AABB内部，这种情况下必然会有相交。

总结：如果 $t_{enter} < t_{exit} \&& t_{exit} \geq 0$ ，那么光线与AABB内的物体才可能相交，需要计算交点并判断，否则便不必计算。

#### 4.1.4 BVH划分

在上一部分，提到了使用AABB来加速光线追踪过程。那么如何在整个场景中合理建立AABB才是有效的呢。这就要提到划分方式。

具体的划分方式有很多种，如八叉树(Oct-Tree), KD-Tree, BSP-Tree.....，这里就仅以BVH进行说明。对于场景中的所有物体，在BVH中的最小叶单位就是物体。我们希望做到的就是将物体尽可能地进行均分到一个个AABB中，并且减少各个AABB在空间中相交的部分。以此划分，只需要逐层判断即可。如果进入某AABB，只需要判断它地两个子AABB是否相交，如果相交就继续递归，直到抵达叶子节点的AABB，如果仍相交就可以开始计算交点了，如果不与AABB相交，那么该AABB中的所有物体都可以不进行计算。这就是BVH的意义，下面给出一个BVH划分的过程图。



在场景中建立BVH树之后，就能够开始判断了。就像上面提到的，遍历判断BVH树的方法类似于遍历二叉树，只不过是加了判断条件，只有通过AABB检测的才会被遍历。下面给出伪代码的形式。

```
1 Intersect(Ray ray, BVH node){  
2     //光线未发生相交的情况  
3     if(ray misses node.bbox)
```

```

4         return;
5     //叶子节点
6     if(node is a leaf node)
7     {
8         //对节点的所有物体求交测试
9         //返回最近的求交
10    }
11
12    //左右分别求交
13    hitLeft = intersect(ray, node.leftChild);
14    hitRight = intersect(ray, node.rightChild);
15    return closer(hitLeft, hitRight);
16
17 }

```

值得注意的是，为了保证 $logn$ 的查找效率，我们希望BVH左子树和右子树中的物体数目尽可能接近。那么每次划分的位置就应该是所有坐标的中位数，在这一部分可以通过使用快速线性选择算法，具体算法就不再说明了。

在说明了光线追踪流程以及加速方法后，Witted-Style光线追踪就告一段落了。下面将在此基础上说明路径追踪。

## 4.2 辐射度量学

在之前的光栅化，光线追踪部分，我们对光线的认知仅仅是一条具有颜色的几何射线。但在物理现实中，光线则是具有能量的。在光线传播、散射过程中，能量都在不断地发生变化，影响着现实中的光照效果。

想要真实地模拟光线，那就需要从物理的角度来重新定义光线，即从辐射度量地角度来认识光线。

### 4.2.1 辐射强度 (Radiant Intensity)

光源地发光实际是能量向外辐射的过程。给出的定义是：**辐射强度是单位立体角的辐射功率**。即：  
 $I(\omega) = \frac{d\Phi}{d\omega}$ 。

立体角简单的说就是球体表面的一个区域面积与半径平方的比值。即： $\Omega = A/r^2$  那么单位立体角  
 $d\omega = \frac{dA}{r^2} = \sin(\theta)d\theta d\phi$  这里的 $\theta$ 以及 $\phi$ 指的是球坐标下的两个角度。

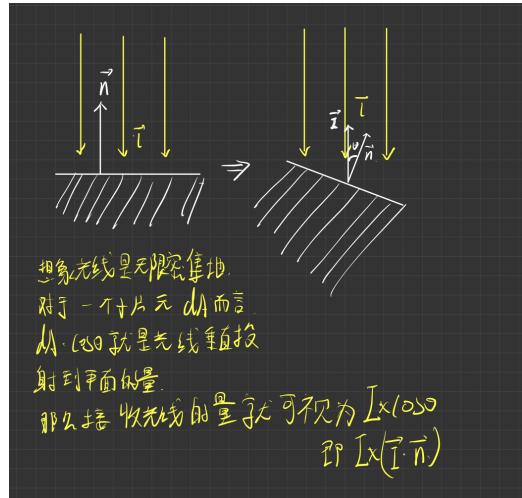
对于一个点光源而言，其表面积为 $4\pi r^2$ ，那其整个表面的立体角就是 $4\pi$ ，假设光源的辐射总功率为 $\Phi$ ，那么不难得到： $I = \frac{\Phi}{4\pi}$

Radiant Intensity 从直观上讲就是光源的强度。反应了光源本身的光亮程度。其强度并不随距离改变，因为不论光源辐射的有多远，同样立体角上的光辐射量是相同的。

### 4.2.2 辐照度 (Irradiance)

使用翻译软件得到的结果是辐照度，不过我个人认为称其为**辐射通量密度**更加合适。Irradiance在定义上指的是**辐射表面单位投影面积上的能量**。我们假设辐射到表面 $dA$ 的总辐射量为 $d\Phi(x)$ 。那么它的Irradiance就是： $E(x) = \frac{d\Phi(x)}{dA}$  实际上就是指能量流过一个封闭面的能流密度。这里的面积指的是垂直于辐射流量方向的面积。

在光栅化部分，提到过一个与漫反射有关的兰伯特模型，我们使用光线与表面法线的夹角作为光照的修正量，在辐射度量学理论下它同样合理。



我们假设辐射到表面的总辐射量为 $\Phi$ ，光线与法线夹角为 $\theta$ 。不难得得到：

$$E(x) = \frac{\Phi}{S} * \cos\theta$$

这里得到的结果与在兰伯特模型中的结果一致，即光照方向与法线放下夹角越大，实际光照产生的效果就越小。符合我们的认知规律。

在说明一个能量流失的例子。我们直观中都明白对于一个点光源，距离光源中心越远的地方实际光照的效果就越弱，也就是辐射的能量就越小。对于以光源为中心，半径为 $r$ 的球形区域，根据能量守恒可知，各个球面的总能量是相同的，都是 $\Phi$ 。但是半径越大的球面其面积越大，根据Irradiance的定义式可知： $E(x) = \frac{\Phi}{4\pi r^2}$  令 $r=1$ 处的 $E_0 = \frac{\Phi}{4\pi}$  则半径为 $r$ 处的则为 $E' = \frac{E_0}{r^2}$ ，即辐射通量密度随着半径的增长而下降，同样符合物理规律。

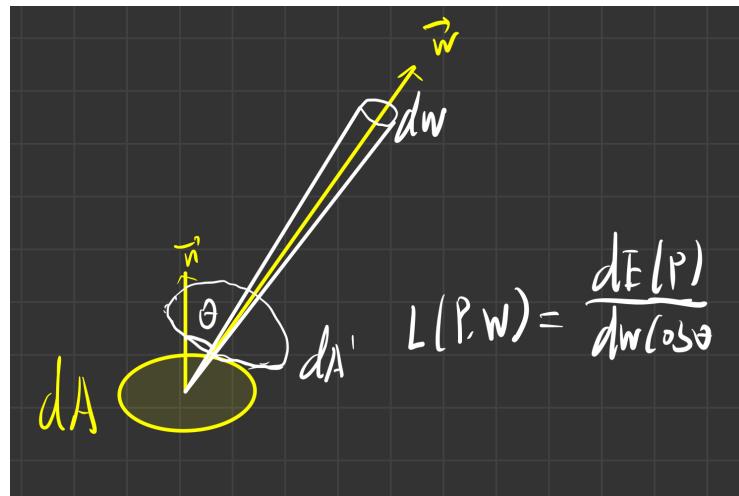
#### 4.2.3 辐射亮度(Radiance)

下一个概念是Radiance，定义是**单位立体角上单位投影面积的能量**。在前面我们提到：Irradiance：**单位投影面积的能量**。Intensity：**单位立体角的能量**。那么，我们可以说Radiance就是**单位立体角的 Irradiance**。或者说是**单位投影面积上的辐射强度**。

对于一个表面 $dA$  我们想求出这个表面向某个方向 $\omega$ 的辐射亮度，那么我们就能把它表示为：

$$L(p, \omega) = \frac{dE(p)}{d\omega \cos\theta} \text{ 或者 } L(p, \omega) = \frac{dI(p, \omega)}{dA \cos\theta} \text{ 两种形式。}$$

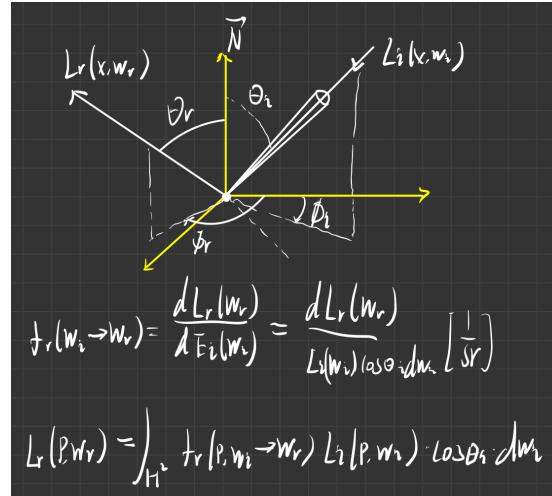
根据定义我们同样能够得出，如果将Radiance对一个半球面对 $d\omega$ 进行积分得到的就是辐射通量。可以说Radiance表现的是一个表面辐射通量密度在某个方向的一部分。



#### 4.2.4 双向反射分布函数 (Bidirectional Reflectance Distribution Function)

双向反射分布函数 (后称为BRDF) , 反映的是从每个方向  $\omega$  进入的Radiance在某点p处后向各个方向 $\omega_r$  反射的分布情况。通俗来说, 就是通过一个函数来定义每一道进入的Radiance如何向各个方向散射。

各个方向的Radiance进入一个小区域后, 我们就能够通过积分计算出该处的irradiance, 所以BRDF也可视为是irradiance与Radiance之间的作用。



对于某个立体角入射的radiancne, 在入射到的微小表面, 就是一个微分的irradiance。因为Irradiance是单位面积上所有立体角的radiancne的积分, 那其中一个立体角的radiancne就是irradiance的微分了。由此就定义了每个立体角入射的radiancne是以怎样的一个比例散射出去变为新的radiancne的。

同时, 需要考虑物体表面的所有radiancne要么是来自于其他物体的反射, 要么是来自自身的发光。那整个式子可以写为:

$$L_0(p, \omega_0) = L_e(p, \omega_0) + \int_{H^2} L_i(p, \omega_i) f_r(p, \omega_i, \omega_0) (n \cdot \omega_i) d\omega_i$$

即:  $\omega_0$  方向出射的radiancne等于自发光向  $\omega_0$  方向的出射以及各个方向 入射radiancne经过BRDF作用的总和。

总而言之, BRDF确定的就是从不同方向入射来的radiancne如何反射出去的过程。

#### 4.3 路径追踪

在简单的说明了辐射度量学后, 就可以通过路径追踪的方式来进行渲染了。

路径追踪的核心是在上一部分提到的渲染方程。

$$L_0(p, \omega_0) = L_e(p, \omega_0) + \int_{H^2} L_i(p, \omega_i) f_r(p, \omega_i, \omega_0) (n \cdot \omega_i) d\omega_i$$

在这个部分中有一个关于半球面的积分, 是对于各个立体角radiancne经过BRDF作用的积分。想要解出它, 可以使用蒙特卡洛积分法。

$$L_0(p, \omega_0) = \int_{H^2} L_i(p, \omega_i) f_r(p, \omega_i, \omega_0) (n \cdot \omega_i) d\omega_i$$

我们随机选定任意方向  $\omega_i$  其中选则中的概率密度为  $p(\omega_i)$  一共选择N次, 就可以将积分式转换为一个求和式。

$$L_0(p, \omega_0) \approx \frac{1}{N} \sum_{i=1}^N \frac{L_i(p, \omega_i) f_r(p, \omega_i, \omega_0) (n \cdot \omega_i)}{p(\omega_i)}$$

由此就可以通过设定一个概率分布函数以及若干采样点来实现数值积分。但此公式存在一个问题, 路径追踪的过程中, 会有多个散射点p, 如果对于每个散射点p我们都进行N次计算, 显然每进行一层递归, 产生的光线数量将是指数增长的。比如: 一束光线撞击一个表面, 我们产生100道光线计算积分, 这100道光线又会再次产生10000道, 将导致计算量爆炸。如果希望计算量不发生爆炸, 必须使得N = 1。但对于

一个点，如果N = 1的话，积分效果几乎难以近似，替代的方案就是由光线追踪每一个像素进行一次光线投射改为进行N次光线投射，将每一个点的次数固定为常数阶即可。

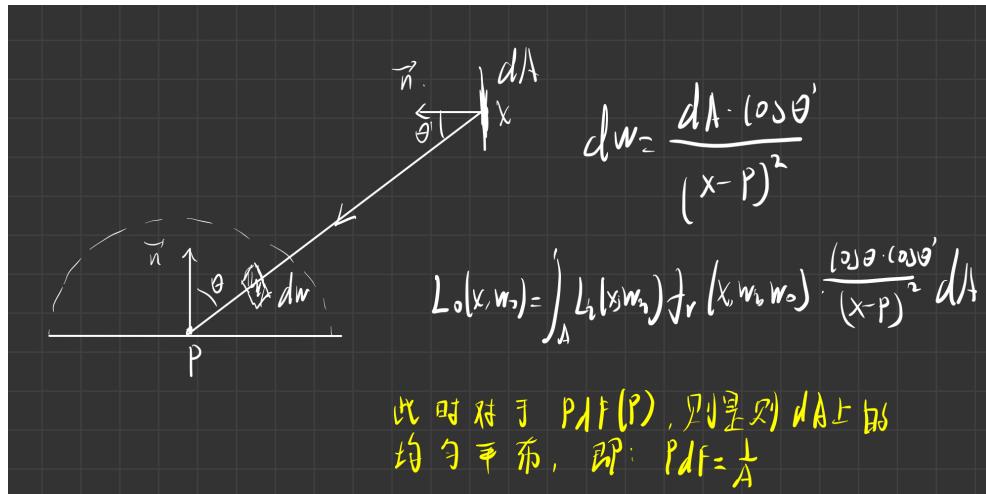
而路径追踪与光线追踪的相同点是，光线在击中可散射表面时，会发生散射。那么对于一个表面的 radiance 就有两个部分需要考虑。1、直接由光线照射得到。2、由其他物体散射得到的部分。对于第二种情况，我们需要继续递归，追踪光线路径。而对于第一种情况则不用递归，直接将radiance加入即可。那么就可以得到如下的伪代码：

```

1 shade(p, wo)
2 {
3     随机选择一个方向: wi 概率密度为 pdf(wi);
4     rayTracing(ray(p, wi));
5     if 光线集中光源:
6         return L_i * f_r * cos(theta)/pdf(wi)
7     else if 光线集中物体于点q:
8         return shade(q, -wi) * f_r * cos(theta)/pdf(wi);
9 }
```

此时已经实现了光线的路径追踪，但这是一个递归的算法，如果不设置停止条件，就无法完成。但事实上我们知道光线的散射是不会停止的，那么我们就需要通过某种方法使光线停止递归的同时能达到无限散射的效果。通过概率论我们知道数学期望  $E = \sum p_i * v_i$  如果我们设  $L_0$  在某次递归过程中，有概率P继续递归，则有  $1-P$  不在递归。不再递归的返回值就是0，递归的返回值就是  $L_0 / P$ ，数学期望恰好是：  
 $E = P * (L_0 / P) + (1 - P) * 0 = L_0$  通过增加一个概率判断的方法，就解决了无限递归的问题。

下一个问题则是对于一个点，我们发射出去的很多光线可能什么都没有撞击到，这部分计算就被浪费了。所以我们希望计算哪些能够击中光源的光线。这就是光线采样。即将积分过程由点P转换到光源的发射点X进行积分。



之后，给出完整路径追踪的伪代码：

```

1 shade(p, wo)
2 {
3     pdf_light = 1/A; //A为采样点x处的面积
4     //直接光照贡献
5     L_dir = L_i * f_r * cos(theta) * cos(theta') / |x-p|^2 / pdf_light;
6
7     //间接光照贡献
8     L_indir = 0;
9     randomTest = random(0, 1); (0 <= P <= 1)
10    if(randomTest > P)
11    {
12        pdf_hemi = 1/2pi; //半球表面均匀采样
13        rayTracing(ray(p, wi));
```

```

14     if(ray hit object at q)
15     {
16         L_indir = shade(q, -wi) * f_r * cos(theta)/pdf_hemi/P;
17     }
18 }
19 return L_dir + L_indir;
20
21
22 }

```

到此，就给出了路径追踪的全部说明。也是光线追踪渲染的最后一个部分。通过光线追踪、辐射度量学、路径追踪三个部分说明了我对于光线追踪这一渲染方式的理解与学习，这一部分还有不少地方感觉自己仍是一知半解，还有不少问题需要通过实践和进一步的学习来加深理解。

## 5、其他

这一个部分用来说明我学到的一些其他相关于图形学但不多的部分，这一部分的内容对我而言有些比较困难，难以系统的进行说明，故通过自己的理解与认识简单说明。

### 5.1 材质

在光线追踪部分已经提到BRDF是被辐射表面的自身性质，决定的是表面如何反射radiance。这是一种物体本身的属性，与其表面的材料、组成、光滑程度、各向性相关。对于这些性质，可以用材质来描述。即材质（Material） == BRDF。

#### 5.1.1 漫反射材质

对于漫反射而言，就是将光照点的irradiance均匀的散射到周围。即对于辐照到该处的所有方向radiance都需要积分，之后均匀散射。对于漫反射，我们假设对于散射点来说，空间中各个方向的radiance都是一样的。那么对于之前的渲染方程就会得到：

$$L_0(w_0) = \pi f_r L_i$$

而我们又认为漫反射之后，各个方向的radiance是均匀的。那么就可以令  $f_r = \frac{\rho}{\pi}$  那么各个方向的  $L_0 = \rho * L_i$  这就通过定义BRDF展现了漫反射的物体特性。

对于镜面反射材质、折射材质的计算比较困难，这里就不再给出了。

### 5.2 微表面模型

对于一个表面上有许多不平整小凹凸的表面，在我们观察者与其距离足够远时，往往难以观察到这种微小的凹凸。但是其光学反射性质又确实是存在的。如果对于足够远但表面细节丰富的物体仍进行精细的计算，得到的效果与计算的代价是不相符合的。对于这种情况，就可以引入为表面模型BRDF。微表面模型实际上反映的是物体表面的法线分布情况。对于一个表面，它的法线分布如果十分接近，即法线法向分布在一个小的均匀范围内，在整体观察时，就如同整个平面的法向量与这一分布接近，整个表面就会显示出高光的性质。反之，如果法线分布十分分散，与整个平面的法线相差较大，就难以显示出镜面反射的效果，而是会表现漫反射的效果。

在这里可以说一下各向异性以及各项同性。对于一个表面如果其法线分布在各个表面上都均匀，那它就是各项同性的，光照效果同样会显示出在各个方向相似的效果。但如果其法线分布集中在某一个方向上，则在不同的方向观察会得到不同的效果，光照在不同方向上的影响也截然不同。这就是各向异性。

### 5.3 颜色

对于颜色，我们使用RGB(A)通道来进行表示。但对于完全一致的自然光，为何在它照射到不同物体上时，我们能够观察到各种不同的颜色呢？这与光的波长以及细胞有关。

人眼底部有着能够感知光线的细胞，特别是对于红、绿、蓝三种颜色光的波长各有一种锥细胞，它们对于各自接近的波长的光有着比较好的吸收效果。也就是说白光照射到物体表面，会有不同波长的成分被吸收，再次反射到人眼的光则又因为细胞的感光能力不同而被吸收一部分。最终显示出的颜色就往往是由RGB三种颜色组合而成的。这也是为什么使用RGB通道来表示颜色的原因，它与人体的感光比较接近。

特别地是，不同人眼内的三种感光细胞分布是不均匀的，也就是说每一个人看到的颜色或多或少都会存在一些差异。但因为大体上还是红绿蓝三种颜色的光，那么大家对于颜色还是能分辨清的。

## 6. 总结

在学习图形学的过程中，也深深感受到了这门学科的深不可测。计算机图形学用数字与颜色构建出一个栩栩如生、难辨真假的虚拟世界。就目前所学而言，如果说使用计算机编程是对现实世界的抽象，那么图形学就是在用计算机重新还原现实世界。

计算机图形学涉及各个方面知识，在数学、光学、物理学……各种学科都有一定的要求。牵涉的方面同样极多，从几何、建模、光照、颜色、渲染……几乎每一个领域都能够单独拿出来学习很久。作为一门综合性如此之强的学科，它也自然在当下的生产生活中占据了一席之地。

学习图形学的过程是困难的，各类原理的理解、数学的推导再到优化、实现。可以说每一步都要付出百分百的精力。但回报同样值得，虚拟即是现实，现实亦是虚拟。当真实的图像呈现在虚拟的屏幕上时，这一切都是值得的。使用计算机，可以实现某种意义上的“创造”，它赋予的是打造一个符合幻想的虚拟世界的能力，虽然当下这还有很多限制，无法生成完全真假难辨的实时画面，打造一个虚拟世界，但它已经有了这个可能，随着将来技术的进步，算力的增加，算法的改进，实现真正的实时地，真实地渲染已是触手可及。

在未来，我也希望自己能够像大多图像学人一样，做到一个“T”。多方面涉猎，单方面深入，从社会生产生活也好，从科学研究也好，能够在这一领域做出一些自己的成果。经过这大半个学期的学习，感觉自己也仅仅是堪堪踏入了图形学的门槛，对于知识的理解还完全不够。“雄关漫道真如铁 而今迈步从头越”。学习路上困难重重，希望自己能够坚持求索，不断努力。

周永扬

2021/5/9