

GAMES 101 NOTE

Lecture 2

向量点乘: \cos

向量叉乘: 1、可以判断向量的左右关系;
2、判断点是否在多边形内部

第一点, 叉乘可以用来向量的左右关系。若向量 \vec{a} 在向量 \vec{b} 右边, 通过右手法则, 他们叉乘的结果应该朝上; 而如果结果朝下, 则向量 \vec{a} 在向量 \vec{b} 左边;

第二点, 叉乘可以用来判断一个点是否在多边形内。如果一个点P在一个多边形内, 那么从该多边形所有的点来说, 每条边构成的向量与点p构成的向量叉乘结果都应该是统一的, 反之, 会出现某些叉乘结构不一致, 即算出的向量方向相反。

Lecture 3

齐次坐标: 添加一个维度信息来统一平行变换和其他变换

2D point = $(x, y, 1)$

2D vector = $(x, y, 0)$

Lecture 4

三维变换

3D point = $(x, y, z, 1)$

3D vector = $(x, y, z, 0)$

齐次坐标的定义下, 三维变换的本质就是先做线性变换再做平行变换

几种变换

缩放:

相对于原点缩放矩阵 S 为:

$$\mathbf{S}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

相对于某点 $p(x, y, z)$ 的缩放矩阵 S 为 (先将坐标移到原点, 再缩放, 再移回, 其组合矩阵如下) :

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ (1 - s_x)x & (1 - s_y)y & (1 - s_z)z & 1 \end{bmatrix}$$

平移:

$$\mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

旋转

绕x轴：

$$\mathbf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha & 0 \\ 0 & \sin \alpha & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

绕y轴：

$$\mathbf{R}_y(\alpha) = \begin{pmatrix} \cos \alpha & 0 & \sin \alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \alpha & 0 & \cos \alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

绕z轴：

$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 & 0 \\ \sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

视图变化：

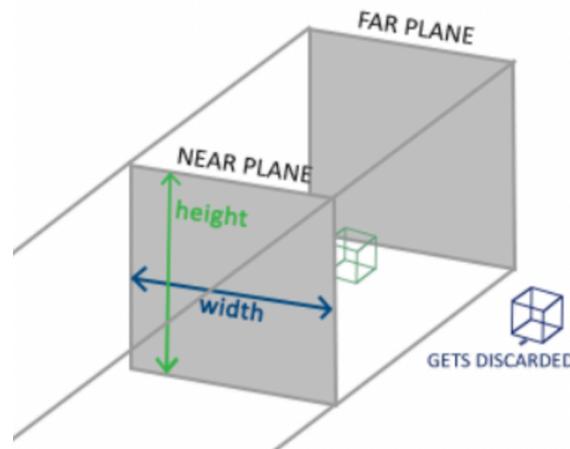
视图变化本质是通过定义一个摄像机来观察物体，所以该问题也可以转换成如何定义一个摄像机。通常一个摄像机由四个部分组成：相机位置，相机朝向，相机上轴和右轴。但是我们一般只需要前三个分量，因为最后一个右轴就可以通过相机朝向和上轴叉乘得到。

两种投影：

投影所做的事是把3D空间物体投影到二维空间，同时通过一个范围约束所有坐标必须处于这段范围内，落到范围外的坐标应该被舍弃掉。

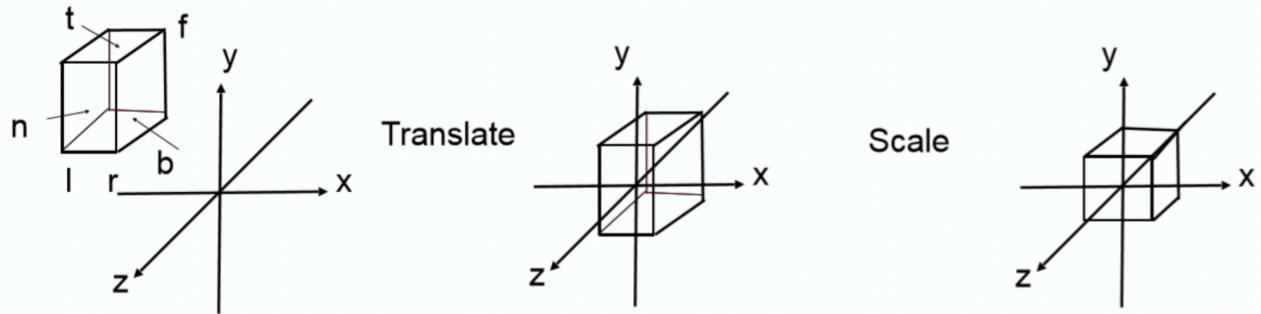
正交投影：

正交投影假设视点无限远，所以构成的范围是一个长方体。它常常需要我们提供长方体的高，宽，近平面距离以及远平面距离。



如何通过一个长方体 $[l, r] \times [b, t] \times [\mathbf{f}, \mathbf{n}]$ 进行正交投影呢？通常有如下步骤：

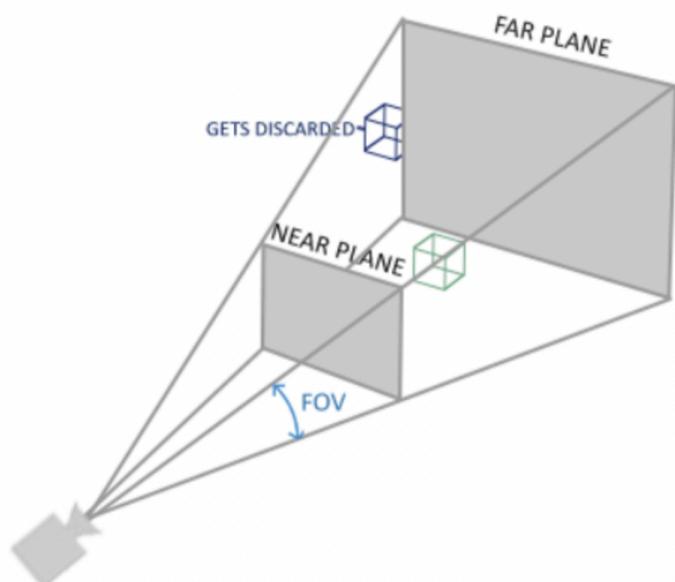
1. 固定相机在原点，使其朝向-z，上轴和y重合；
2. 丢弃z轴；
3. 把正方体规范化到 $[-1, 1]^3$ ，即先平移，后放缩：



$$M_{\text{ortho}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{n-f} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{r+l}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

透视投影：

透视投影更像一个近小远大的截体来规定坐标的范围，它更符合人眼成像，即产生近大远小效果，在图形学中也更常见，使用得更多。它通常需要我们提供截体的FOV角度，宽高比，近平面距离以及远平面距离。



如何进行透视投影?

1. 做“squish”即 $M_{\text{persp} \rightarrow \text{ortho}}$; 通过挤压把远平面上的点都挤压到近平面上;
2. 做正交投影即 M_{ortho} 。

$$M_{\text{persp} \rightarrow \text{ortho}} = \begin{pmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

第三行为 0, 0, n+f, -nf

第二步:

通过正交投影计算透视投影矩阵: $M_{\text{persp}} = M_{\text{ortho}} \cdot M_{\text{persp} \rightarrow \text{ortho}}$

Lecture 5

正交投影:

正交投影比较好理解, 即如何定义一个长方体, 我们只需要知道高, 宽和深坐标。所以需要的参数是height_start,height_end, width_start,width_end, znear (近平面), zfar (远平面) 六个参数

透视投影:

构建透视矩阵需要视野角度fov, 宽高比, znear, zfar四个参数。

视口变换:

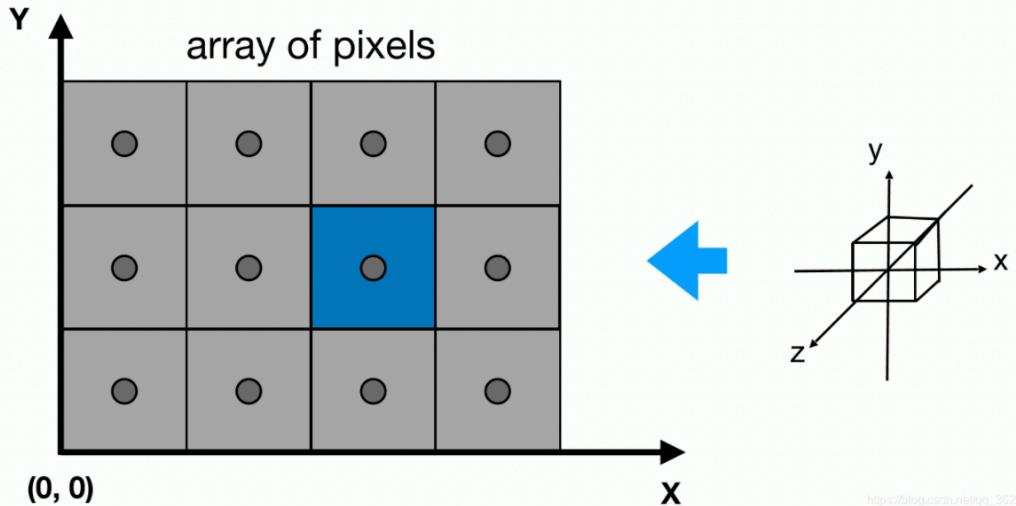
在M (model) V (view) P (projection) 变换之后, 我们已经得到了三维坐标在二维屏幕上的投影, 同时也得到了投影区域(Canonical Cube)。之后所需要的就是视口变换。视口变换是把三维坐标与屏幕上指定的区域进行映射, 简单来说就是在屏幕上规定了一块区域, 然后将我们投影的坐标规定好, 只能画在这块区域中, 这就是视口变换所做的事情。

屏幕空间:

既然要找到这个映射关系, 那么屏幕自身也应该有自己的坐标系, 也就是屏幕空间。我们可以把屏幕上的像素看成是像素的矩阵形式, 它的x, y坐标都是从0,0开始的, 里面每一个坐标都可以看做包含了一个像素。

屏幕空间的标准化

这一步需要把Canonical Cube $\in [-1, 1]^2$ 变换到 $[0, \text{width}] \times [0, \text{height}]$, width和height就是视口变换所需要规定的区域的长和宽。



这一步变换可以看成是二维平面上矩阵的平移和缩放，故有以下变换公式：

$$M_{\text{viewport}} = \begin{pmatrix} \frac{\text{width}}{2} & 0 & 0 & \frac{\text{width}}{2} \\ 0 & \frac{\text{height}}{2} & 0 & \frac{\text{height}}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

光栅化：

在视口变换之后，我们已经在屏幕上规定了一块区域用于映射对应的坐标，所以后面就需要通过光栅化来把三维图元映射成二维像素。光栅化也分为光栅化线和三角形，下面主要讲光栅化三角形

为什么选取三角形：

- ①基础图元：三角形可以看成是基础图元，用它可以构建出不同的图元
- ②平面：一个三角形必定是一个平面
- ③内外定义：三角形是可以通过顶点环绕顺序来描述其内外性的，方便做面剔除等操作
- ④插值：三角形内部插值十分方便

如何确定三角形在屏幕上的像素

当输入三角形顶点坐标时，如何判断对应屏幕上的像素区域？答案是采样

采样

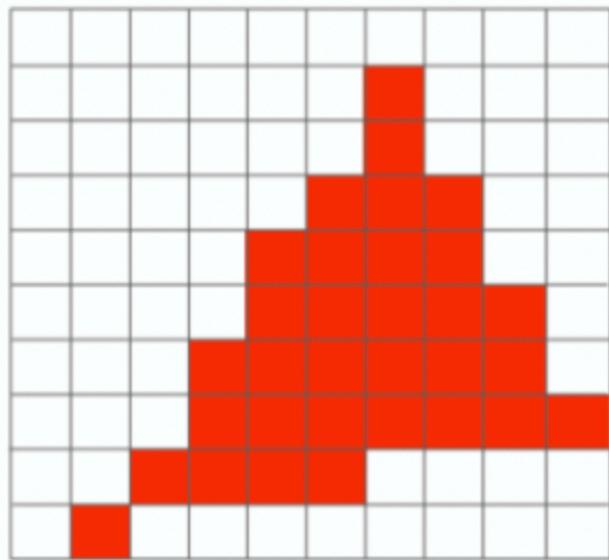
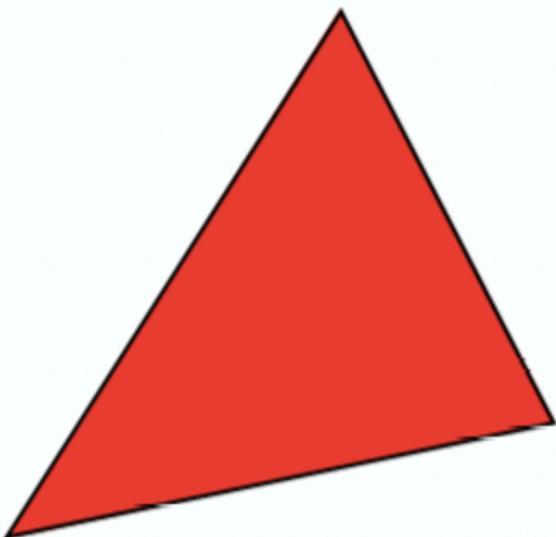
采样就是对方程离散化的一个过程，在这里我们通过对每一个像素进行判断，如果它的中心点在三角形内部，那么它就是属于三角形的，我们应该在屏幕上将其画出来，而判断方法之前也说过，可以通过叉乘来实现

```
1 |   for (int x = 0; x < xmax; ++x)
2 |     for (int y = 0; y < ymax; ++y)
3 |       image[x][y] = inside(tri, x + 0.5, y + 0.5); // 判断点是否在三角形内
```

Lecture 6

反走样：

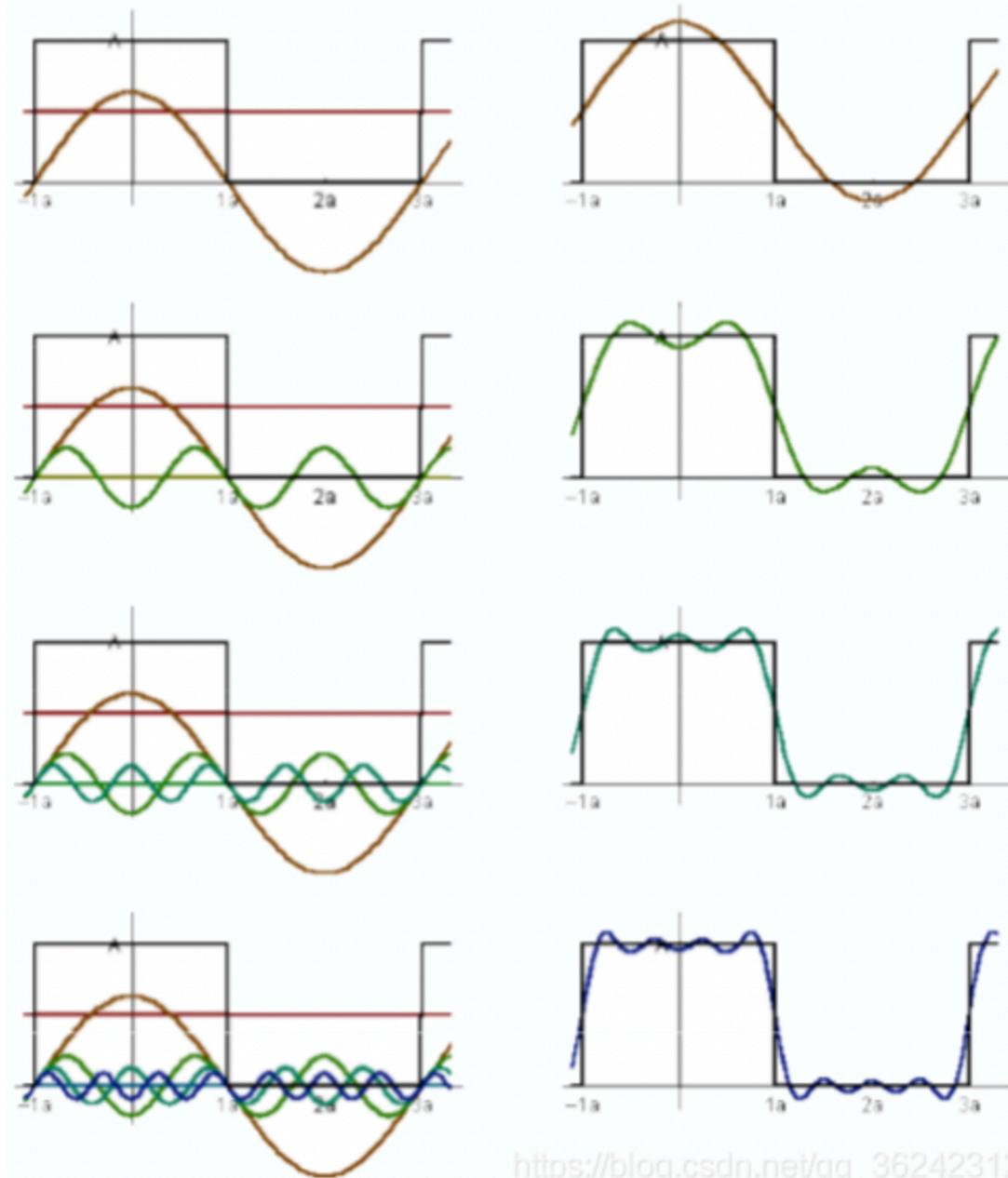
上一节提到了光栅化三角形，而在实际中，屏幕会由于分辨率和采样频率的一些问题，导致三角形在光栅化过程中出现走样（当然，线条也会出现走样）。这是由于三角形的边或者线段在观测中是无限细的，可以看做是带宽无限的信号，而对这些图元进行像素化的过程中，采样频率不足，导致采样过程中丢失高频成分，所以产生的信号失真，在屏幕上就表现为锯齿状的图像，这种现象就称为走样，如下图：



所以我们就需要反走样来减少或消除这种效果。基本上反走样方法可分为两类。第一类是通过提高分辨率即增加采样点(提高采样频率)，比如先在较高分辨率上对光栅进行计算，然后采用某种下采样算法得到较低分辨率的象素的属性，并显示在分辨率较低的显示器上，这类方法有SSAA；另一类反走样是把像素作为一个有限区域，对区域采样来调整像素的颜色或亮度，这种方法类似于图像中的前置滤波 (blur等) ，这类方法有MSAA。

傅立叶变换：

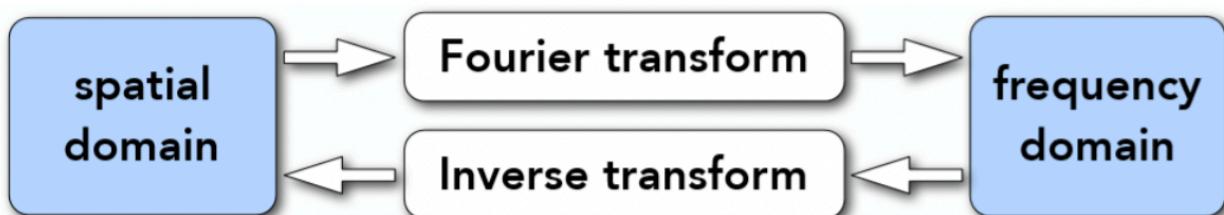
傅里叶变换是假设一个函数能够由不同频率的正弦波 (\cos 和 \sin) 组合得到。而这里的正弦波，我们把它看做是不同的频率分量，由下图所示：



https://blog.csdn.net/qq_36242312

上图中随着正弦波的频率的变换，以及数量的增加，可以看到它们的组合逐渐接近一个 90° 的矩形波。

所以实质上如果想要把一个函数从时域变到频域就可以通过傅里叶变换，而通过其逆变换就可以把这个函数从频域变换到时域，如下图：

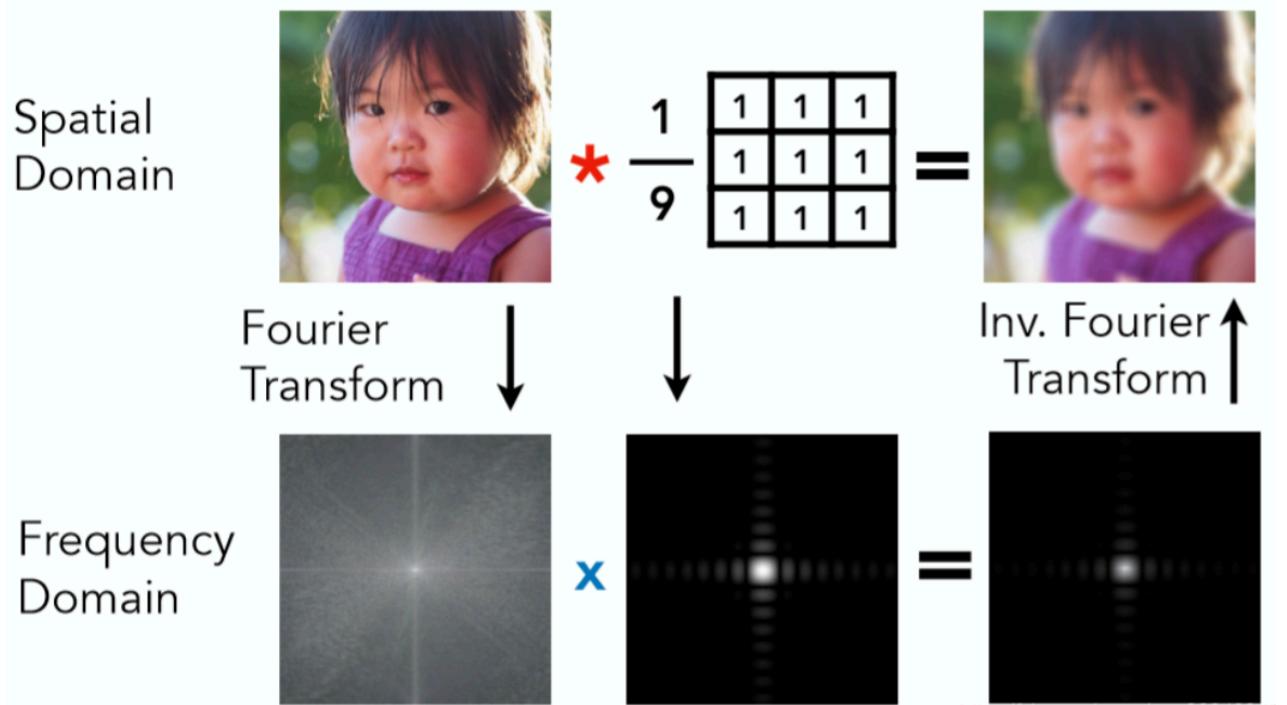


采样与滤波

假设我们以一定的频率对下面的函数进行采样，会发现当函数自身频率越高时，由于采样频率不够，恢复的函数与原来的函数也差异过大。

而滤波恰恰就是能够过滤掉图像（函数）中的某些频率部分。如高通滤波，低通滤波。卷积其实也是滤波的一种形式，它是对信号周围进行加权平均的一种运算。而卷积定理在频域与时域上也十分重要，即：

在函数在时域上的卷积等于其在频域上的积，反之亦然，如下图：

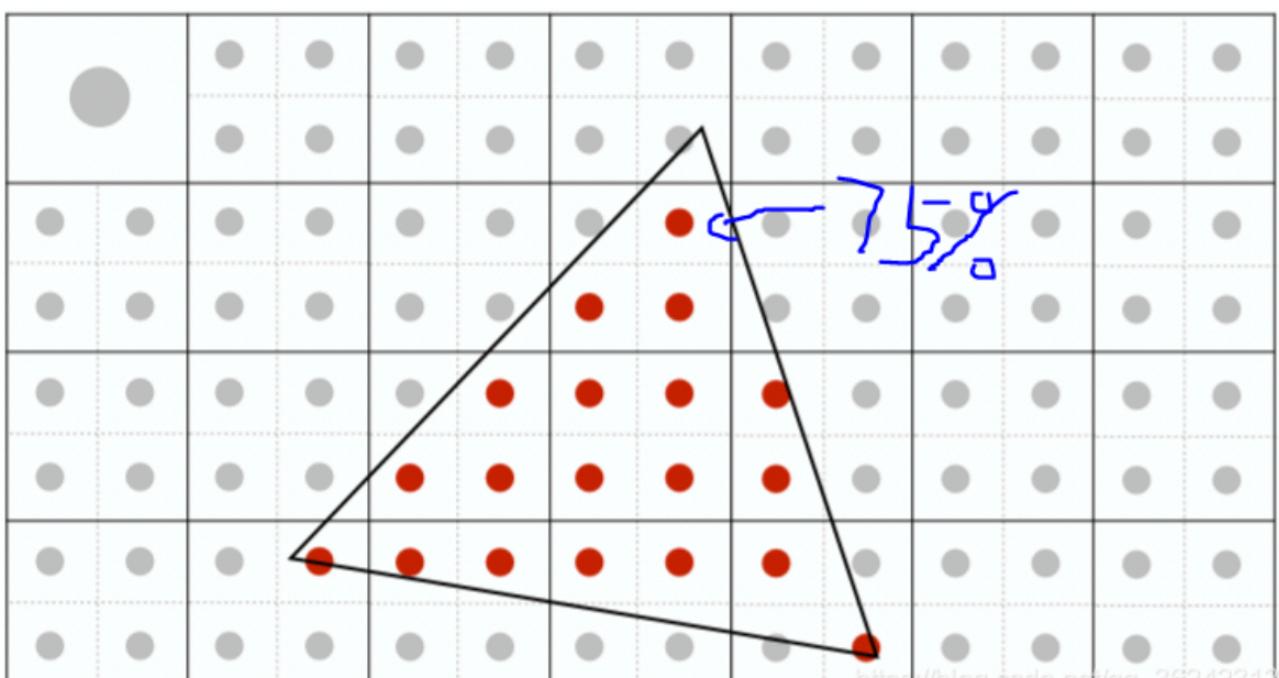


通过超采样来反走样(MSAA)

超采样就是通过对一个像素内的多个位置进行采样并取其平均值来近似1-pixel 滤波器(卷积,blur)的效果

对于4X MSAA来说，其步骤如下：

①假设每个像素中对四个点进行采样



②判断对于一个像素有多少个点在三角形内，然后根据比例对颜色进行“模糊”
可得上图中的一个像素内的颜色应该为原来的75%。

Lecture 7

可见性/遮挡

当在屏幕上画好了三角形后，我们要确定物体与物体之间的遮挡关系，而这种方法通常使用的是Z-buffering（深度缓冲）。

画家算法：

它把需要画的物体按深度大小的顺序进行排序，然后按由远到近的顺序依次画物体。在出现遮挡时，只需要把遮挡物体直接画在遮挡处即可，所以它会不断的覆盖之前绘制的物体。（无法处理深度顺序问题）

Z-buffering

Z-buffering相较画家算法而言更加适用，因为它是按像素的深度大小进行排序。

它的主要思想如下：

为每一个像素存储当前的最小深度值（这里课程指的是深度越小，离视点越近，如果z轴的远近定义不一样，则会存储最大深度值），当扫描到新的像素，如果它的深度更小，则用它对应或插值的颜色值来替代当前需要显示的颜色值，流程图如下：

```
for (each triangle T)
    for (each sample (x,y,z) in T)
        if (z < zbuffer[x,y])                // closest sample so far
            framebuffer[x,y] = rgb;           // update color
            zbuffer[x,y] = z;                 // update depth
        else
            ;                                // do nothing, this sample is occluded
```

从上面也可以看出，Z-buffering同时也需要帧缓冲和深度缓冲来分别存储像素的颜色值和深度值。这里提到的帧缓冲，可以把它理解成一种可以自己定义的缓冲（一块内存区域），即通过定义可以存储颜色，深度，或者颜色与深度，颜色与纹理的不同组合的缓冲区。

shading

shading的字面意思就是通过对物体进行上色

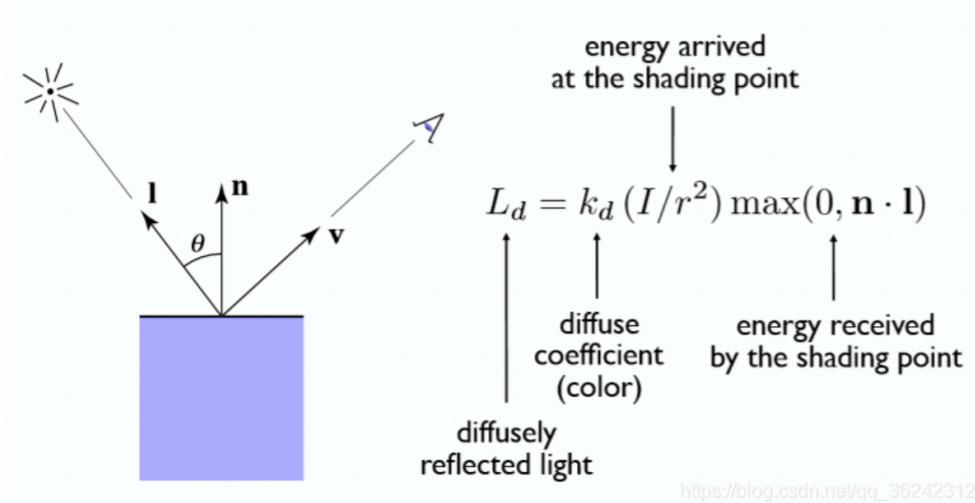
对于这个课程来说：shading的定义是对物体应用材质的过程。而材质可以理解成物体自身对光线的各种反应情况，它反应了物体本身的物理属性。

Blinn–Phong 光照模型

Blinn–Phong 光照模型，它是高光项，漫反射项，环境光项三项组成。

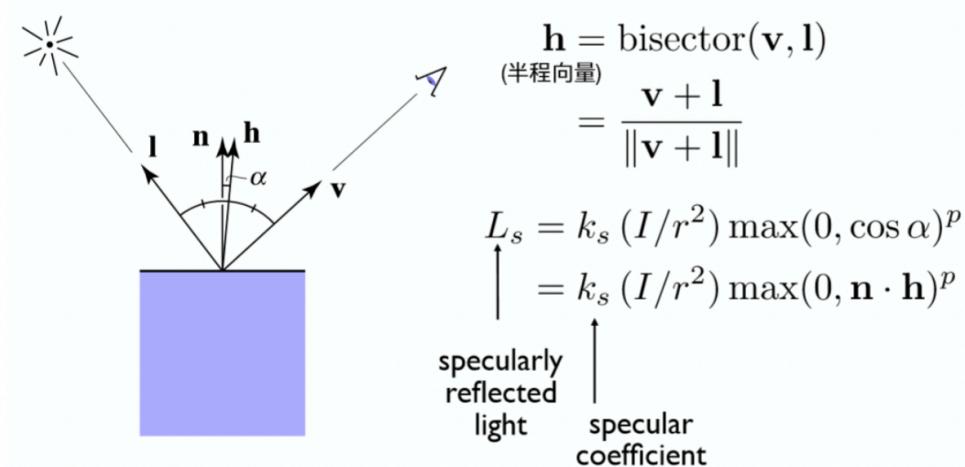
漫反射

首先是漫反射项，它是模拟光源对物体的方向性影响，所以它是独立于视线的，图中方程也没有和视线相关的参数。 I/r^2 表示光的衰减情况， $\max(0, \mathbf{n} \cdot \mathbf{l})$ 计算的是 0 和 法向量与光照向量的较大值，因为当光照向量与法向量夹角大于 90° 时，其结果会为负，没有意义，而 k_d 可以看做是漫反射系数，当像素有颜色时，它是像素的颜色值，当像素有纹理时，它可以是像素的纹理值（实际上还是纹理的颜色值）

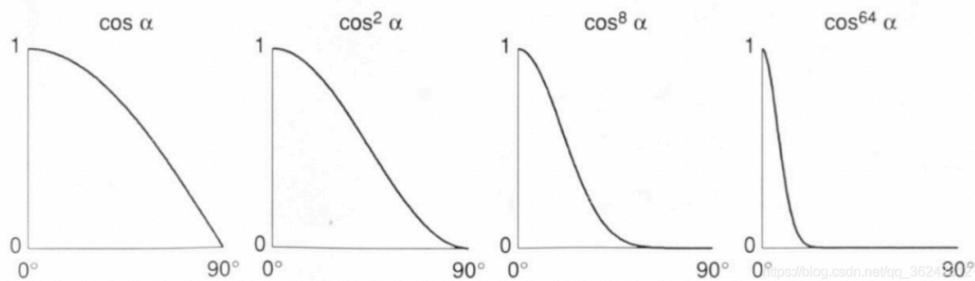


高光

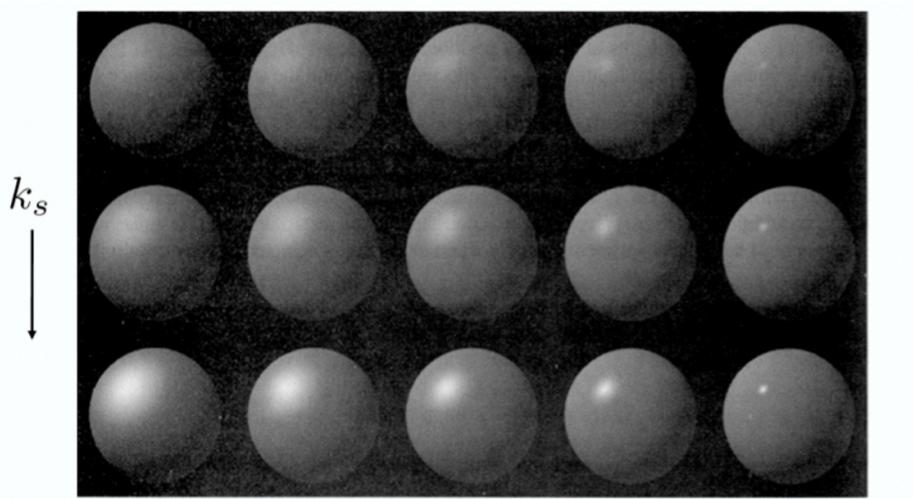
高光用来模拟有光泽物体上面出现的亮点，它是依赖于观察方向的，当视线与光照向量的半程向量与法向量夹角越大时，高光影响越小，当夹角越小时，影响越大。可以这样想象，我们需要看的是一块镜面的反射，当我们的视线与光照向量恰好对称时，高光越强。 k_s 为高光系数， h 为半程向量，这里用半程向量的好处主要是用于解决视线与光照向量在同一侧，从而引起的光断层现象。



注意到这里求max时有一个p系数，它是用来控制reflection lobe（有点像反射区域的意思），由于 $\cos^p a$ 有如下性质：



所以当p很大时，一旦法向量和半程向量的夹角稍微大一点，其反射影响程度都会十分小，表现在图片上也是一个非常小的光点：



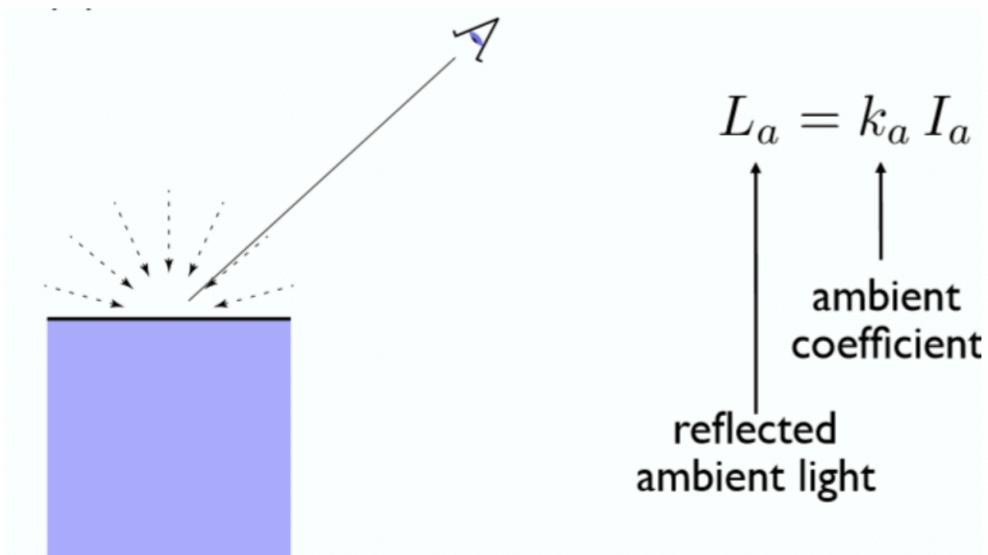
Note: showing
Ld + Ls together

$p \longrightarrow$

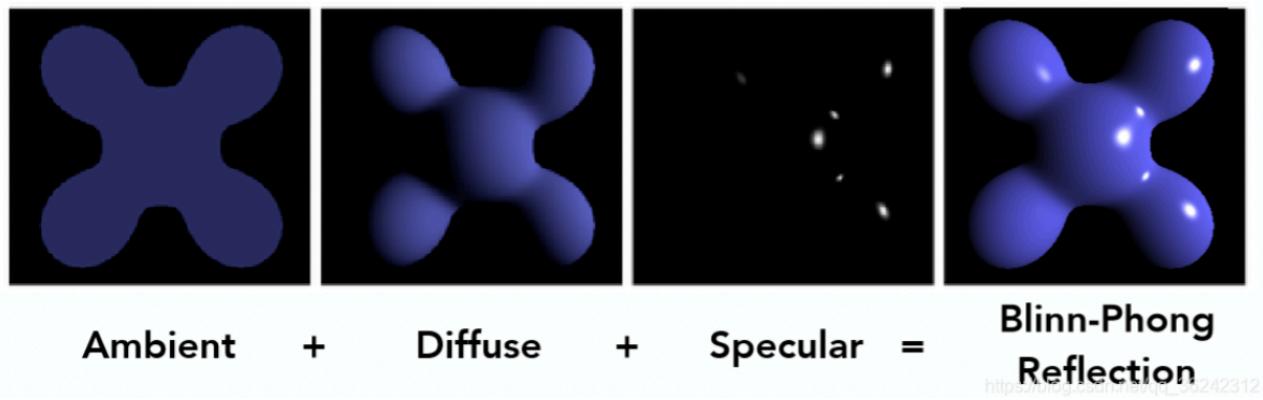
https://blog.csdn.net/qq_36242312

环境光

它是用来模拟即使在黑暗的情况下，世界上通常也仍然有一些光亮（月亮、远处的光），所以物体几乎永远不会是完全黑暗的这种情况。



当计算好了三项之后，其组合结果如下：



Lecture 8

着色频率：

这里主要分成对三角形进行着色，对顶点进行着色，对像素进行着色三种着色方案进行讨论

对三角形进行着色

对每个三角形着色也称为Flat shading，它是指每项属性例如纹理，法向，颜色，都是属于一个三角形的，所以三角形内部不会有插值变化。所以，三角形内部只会根据面法向量对光线反射进行计算，所有的三角形反射都看做一个平面，给人以一种块状的效果（三角形较少时）。

对每个顶点着色

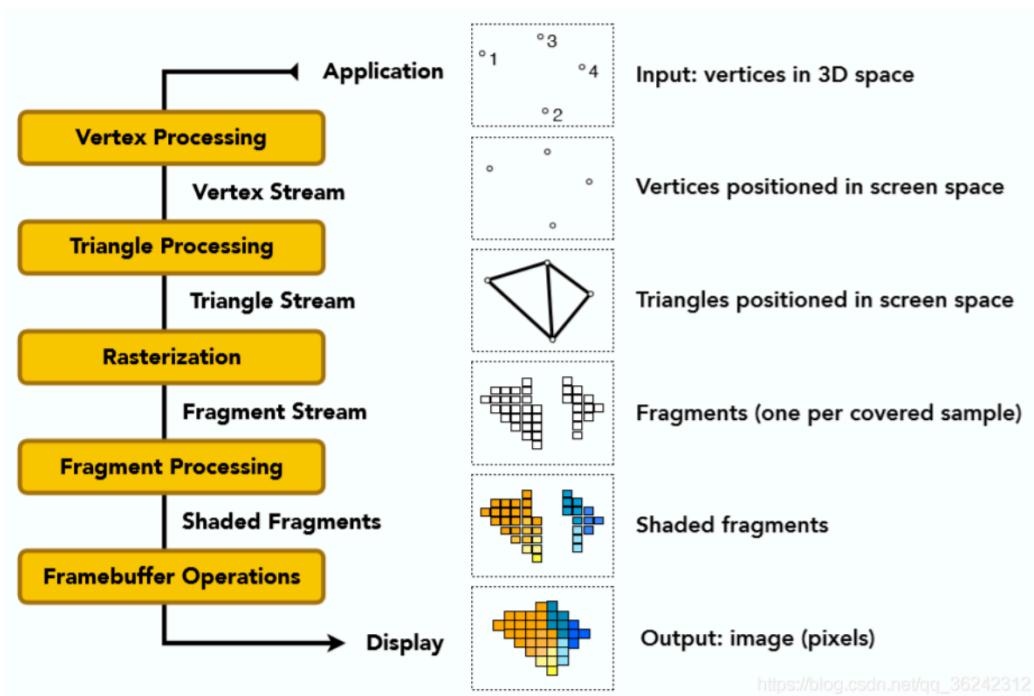
对顶点进行着色也称为gouraud shading，它发生在顶点着色器阶段。对于每个多边形的顶点都存在一个法向量，但是他的着色是先通过这些法向量对顶点计算出光照颜色，然后用光照颜色来进行三角形内部插值

对每个像素着色

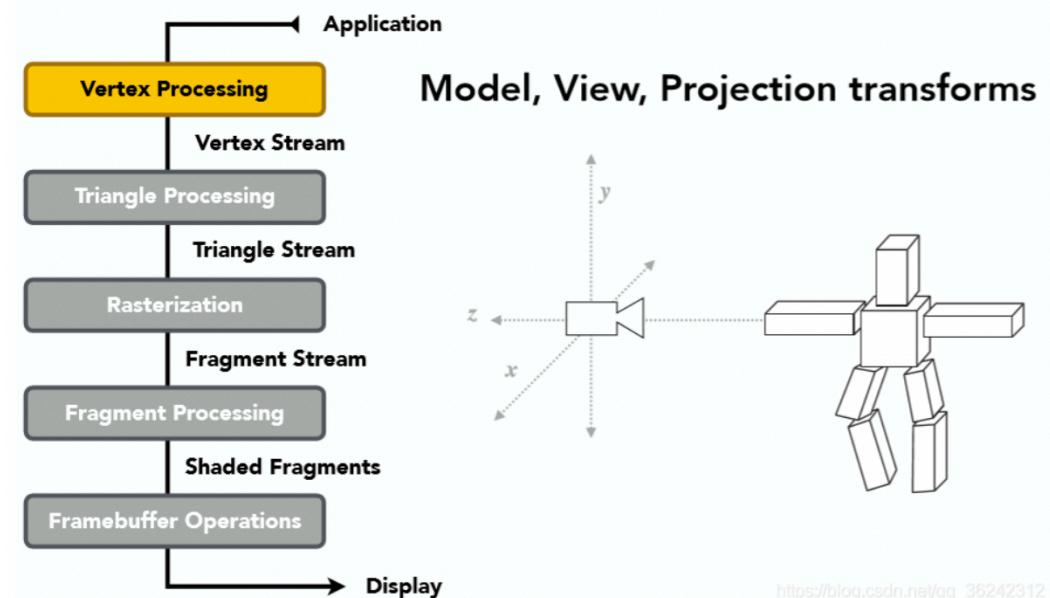
对每个像素着色也称phong shading，它发生在片段着色器阶段。它是通过对多边形每个顶点的法向量进行插值，然后我们通过插值得到的法向量再去计算光照颜色。

由于片段着色器阶段计算的坐标都是处于世界坐标系中，所以在计算光照时，所用的法向量也要转换到世界坐标系下。还记得之前我们把坐标转换到世界坐标是乘以model矩阵，但是法向量是不可以直接乘以model矩阵的，因为它仅仅是一个向量，而且也不是齐次坐标表示，所以这里需要一个法线矩阵（它是model矩阵逆的转置）来进行转换，使法向量也处于世界坐标下。

图形管线（实时渲染管线）



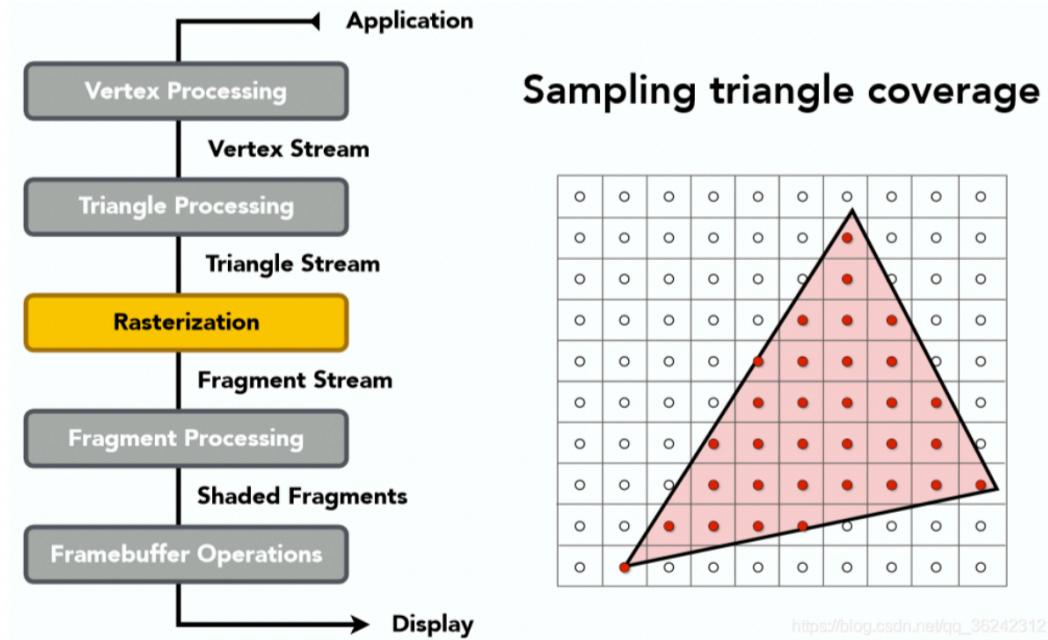
①首先我们有一系列三维点坐标，可能是从某些文件读入。然后我们把它们输入给Vertex Processing阶段（顶点着色器），该阶段主要进行的操作是model, view, projection变换，将三维坐标变换到二维裁剪空间（通过投影丢掉了不可见的区域），也可以进行gouraud shading等操作：



②随后把顶点着色器的输出，作为输入给Triangle Processing阶段，这里主要是进行一些图元的绘制，包括把点连接成三角形，或者制作一些顶点的副本等等

③下一个阶段是Rasterization阶段（光栅化），该阶段将三维图元作为输入，通过采样将其绘制在二维屏幕上，并以片段的形式作为输出

（片段是三维顶点光栅化后的数据集合，还没有经过深度测试，而像素是片段经过深度测试、模板测试、alpha混合之后的结果，片段的个数远远多于像素，因为有的片段会在测试和混合阶段被丢弃，无法被渲染成像素）



④Fragment Processing 把片段作为输入。该阶段主要进行深度测试，计算光照以及纹理映射等，所以该阶段输出的结果基本上已经确定了像素最终的颜色



渲染管线的编程

通常可以在Vertex Processing阶段（顶点着色器）和Fragment Processing阶段（片段着色器）进行编程，这些编程的小程序被称为Shader Programs，里面常用的语言有glsl等

下面的示例程序简单描述了一个着色器的工作，uniform表示从cpu传来的变量，varying 表示从上一个阶段传来的变量，void diffuseShader() 为着色函数，texture2d是内置函数，它表示将纹理myTexture 对应到uv向量，gl_FragColor 可以看做是内置参数，它是屏幕上用来显示的最终颜色。

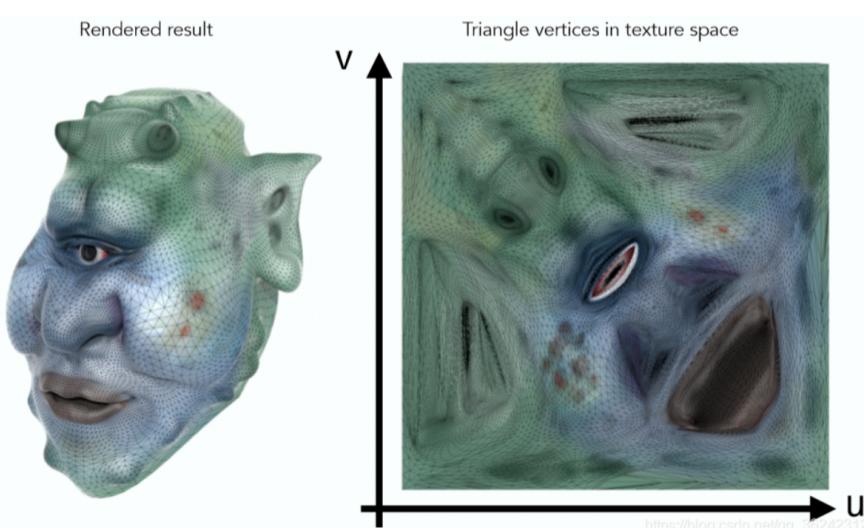
```

1 uniform sampler2D myTexture; // program parameter
2 uniform vec3 lightDir; // program parameter
3 varying vec2 uv; // per fragment value (interp. by rasterizer)
4 varying vec3 norm; // per fragment value (interp. by rasterizer)
5 void diffuseShader()
6 {
7     vec3 kd;
8     kd = texture2d(myTexture, uv); // material color from texture
9     kd *= clamp(dot(-lightDir, norm), 0.0, 1.0); // Lambertian shading
10    gl_FragColor = vec4(kd, 1.0); // output fragment color
11 }

```

纹理映射

纹理映射就是将纹理空间中的纹理像素映射到屏幕空间中的像素的过程。通俗来说可以认为是一张二维纹理把一个三维物体“包裹”了起来，因此三维物体获得了一些表面纹理，纹理也是有坐标的，它的坐标空间是由uv构成的，里面对应的元素是纹素，是计算机图形纹理空间中的基本单元，如下图：



纹素和像素不一样，因为它们是处于不同坐标下的，纹素处于纹理空间，而像素处于屏幕空间。在对三维表面铺设纹理的时候，通过纹理映射技术将纹素映射到恰当的输出图像像素上，这种映射不是简单的一一对应，因为会受到视角的影响，如果以一种斜的姿势观察物体，一个像素对应的纹理区域很可能是比较扭曲的。

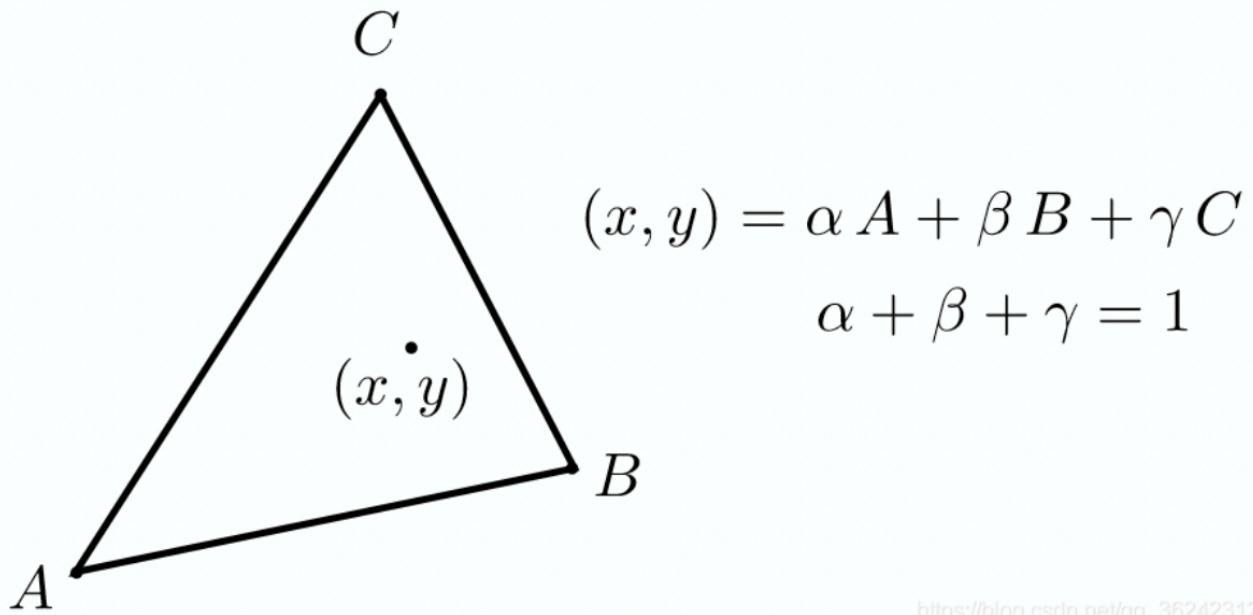
Lecture 9

着色(shading)

重心坐标

上一篇文章说到纹理映射，实质上纹理映射就是把uv坐标下的纹素映射到图像像素的过程。而我们在画三角形的过程中通常只是定义了三个顶点的属性，其内部属性往往需要通过三个顶点插值得来，像法向量，颜色值，深度值，还有纹理坐标等。所以我们需要一个插值的方式来解决这个问题，常用的有利用重心坐标进行插值。

那么重心坐标是什么？简单来说就是找到三角形 ABC 内部的一个点 $P(x, y)$ ，有 $(x, y) = \alpha A + \beta B + \gamma C$ ，且 $\alpha + \beta + \gamma = 1$ ，那么 (α, β, γ) 就是点 P 的重心坐标，如下图：



https://blog.csdn.net/qq_36242312

纹理映射过程

简单的纹理映射的过程：在光栅化过程中，对当前扫描到的点的uv坐标进行采样，得到的颜色直接赋给物体

```
1 | for each rasterized screen sample (x,y):
2 |     (u,v) = evaluate texture coordinate at (x,y)
3 |     texcolor = texture.sample(u,v);
4 |     set sample's color to texcolor
```

纹理分辨率过小

但是这样会带来一些问题，就是当纹理分辨率过小，而需要用来覆盖的物体过大时，多个像素坐标 (x, y) 都会采样到同一个纹素，产生一些锯齿的效果，这时我们就有一些解决方法，如：最近邻插值，双线性插值，双三次插值

最近邻插值

假设红点是像素坐标点对应的纹理像素坐标，那么最近邻插值就是把离它最近的纹素分配给当前像素点

双线性插值

双线性插值就是找它周围四个点对应的纹素进行插值（双线性插值就是在横坐标（竖）上先做两次插值，再在竖坐标（横）上做一次插值即可）

双三次插值

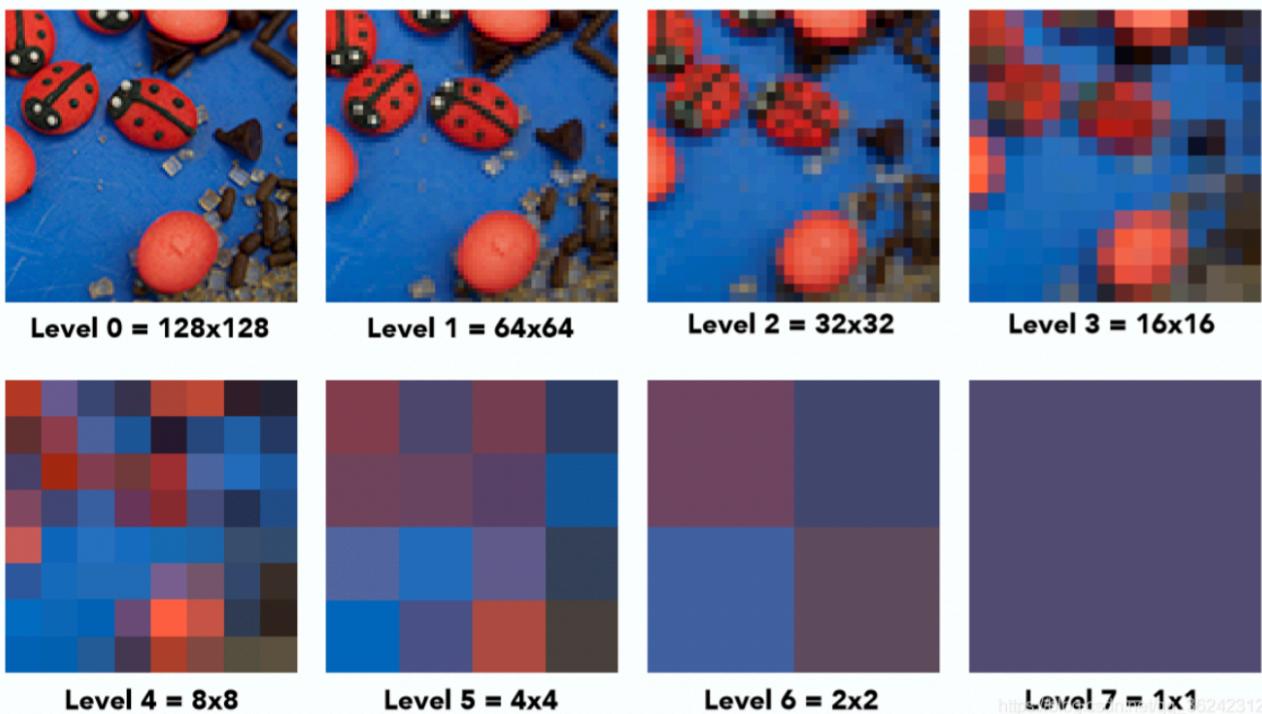
这里的双三次不同于三维空间中的三线性插值，应该叫做双三次插值（Bicubic），它是对二维空间的插值，考虑了邻近十六个采样点的插值，具体方法同双线性插值

Mipmap

它是一种快速，近似，仅限于正方形范围的范围查询

Mipmap就是通过一张纹理，生成许多高层（分辨率每次缩小到一半）的纹理，如下图

"Mip" comes from the Latin "multum in parvo", meaning a multitude in a small space



假设我们定义Mipmap的层级为D，那么如何确定某个像素P对应的Mipmap层级呢？

首先找到该点和相邻像素点在纹理上的映射点，计算边长L，所以层级可以由

$$D = \log_2 L$$

算出（因为随着层数增加，纹理分辨率是以4倍的速度减小，所以L是以2倍的速度增加的）

Lecture 10

隐式几何：公式

显式几何：点

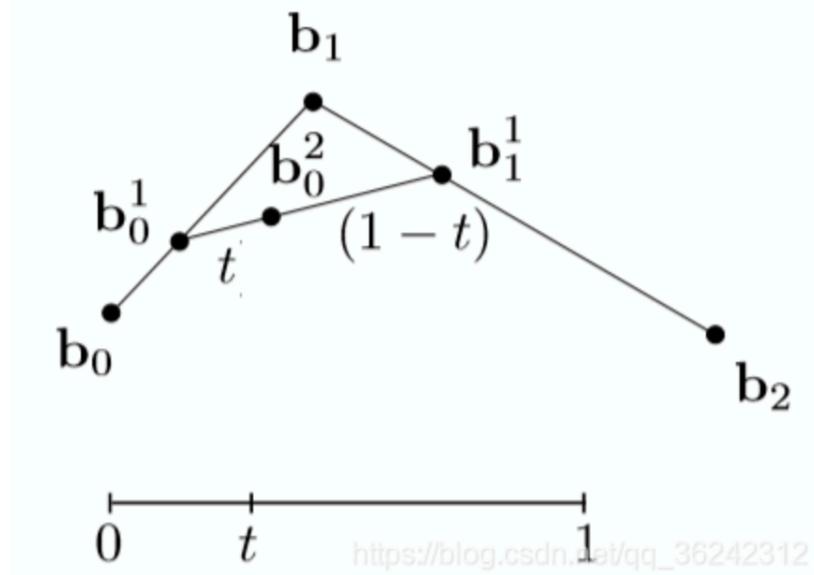
Lecture 11

贝塞尔曲线

贝塞尔曲线本质就是对线段插值，它是一个类似递归求解的过程。Casteljau

Casteljau 算法描述了如何求解贝塞尔曲线，下面是该算法对应三个控制点的贝塞尔曲线求解过程：

已知三点 b_0, b_1, b_2 ，和插值参数 t ，第一次递归插值得到 $b_0 \rightarrow b_1$ 线段上的 b_0^1 和 $b_1 \rightarrow b_2$ 线段上的 b_1^1 。第二次递归插值对 $b_0^1 \rightarrow b_1^1$ 进行插值，发现他们与插值结果 b_0^2 构成的线段位于同一条直线上，所以递归结束，得到贝塞尔曲线结果： $b_0 \rightarrow b_0^1 \rightarrow b_0^2 \rightarrow b_1^1 \rightarrow b_2$



由上图可以总结出求解一趟贝塞尔曲线的过程：

- ①给定起始点和终止点（已知插值参数 t ）
- ②遍历每个控制点和该点的下一点（如 $b_0 \rightarrow b_1$ ）形成的线段，对其进行插值。将插值后得到的点的集合中的第一个点和最后一个点替换为起点和终点。当起点和终点以及最终插值结果的点 (b_0^2) 是在同一条直线 (如 $b_0^1 \rightarrow b_0^2 \rightarrow b_1^1$) 时，递归结束回到③，否则回①
- ③结束，记录最终插值结果的点

一趟贝塞尔曲线的结果只是针对于当前 t 得到的结果，它只是一个点。所以我们在 t 的范围内遍历 t ，得到所有贝塞尔曲线的结果，将这些点和起始点、终止点连接起来即得到完整的曲线。

Lecture 12

Loop Subdivision

这是一种针对三角形网格进行细分的方案，主要步骤分为两步：

- ①添加顶点
- ②更新顶点

网格简化

边塌缩、二次度量误差

Shadow mapping

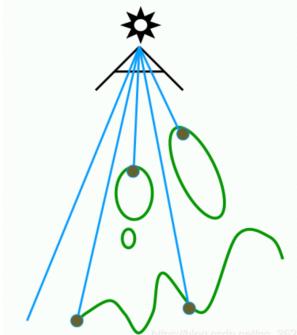
Shadow mapping 是图形学中计算阴影的过程。它是一种图像空间的算法，即在计算阴影的过程中不需要知道场景中的几何关系。

理解 Shadow mapping 中最核心的一点是：如果物体不在阴影中，那么它肯定能被相机看见且被灯光照射到。

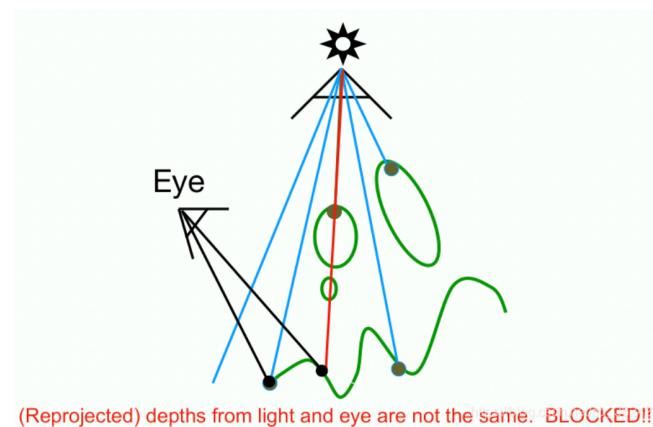
阴影按类型可分为硬阴影和软阴影，硬阴影是由点光源产生的，软阴影是由范围光源产生的。

硬阴影：

1. 第一步从光源处出发，假设光源处有一个相机，计算光源处看向场景的深度，渲染成深度图保存；



2. 第二步从实际相机出发，计算相机看向场景的深度后，投影回光源处的相机空间（常称为光源空间），并与第一步得到的深度进行比较。如果第二步得到的深度大于第一步得到的深度，那么说明该点在阴影内，反之不在。



这里存在几个问题

①为什么需要投影回光源空间?

因为两个相机看到的深度需要处于同一空间下进行比较

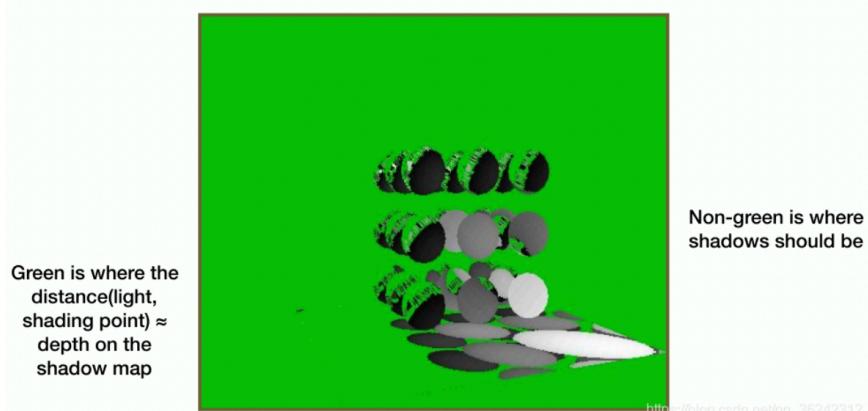
②如何将第二步看到的顶点变换到光源空间?

将第二步中得到的处于世界坐标系下的顶点乘以光源空间下的 PV矩阵即可 (投影和视角)

③在计算阴影时的比较问题

如图所示, 非绿色部分表示阴影, 图中阴影部分明显出现了较多的噪声。这是因为深度缓冲区的深度采用的是浮点表示, 而浮点比较会存在精度问题, 无法直接判断相等。

- Comparing $\text{Dist}(\text{light, shading point})$ with shadow map



同时, 由于深度图的分辨率的影响, 会使得多个片段从同一个深度值采样, 这还会造成失真。常见做法是在判断一个片段是否为阴影时添加一个 bias

软阴影

软阴影是由范围光源产生的。相比硬阴影, 软阴影的边界更加模糊。

Lecture 13

光线追踪

为什么需要光线追踪?

这里常用之对比的是光栅化。光栅化是把场景中的物体独立出来, 并且再将物体独立成一个个的三角形面片进行渲染, 通常 GPU 绘制三角形面片是并行的, 所以对于物体来说, 它是不知道自己周围的物体, 同样的, 对于三角形面片来说, 它也是不知道自己周围的三角形。所以光栅化无法处理一类需要全局信息进行渲染的效果, 如: 软阴影、Glossy 反射以及间接照明:

相比较光栅化, 光线追踪是从相机出发, 计算相机发出的光线(视线)与场景产生的结果, 这里的光线可能会与场景中的物体碰撞后生成另外一些光线, 然后生成的光线也需要继续做计算。所以对于光线追踪来说, 相机产生的光线是知道整个场景信息的。

光栅化与光线追踪的对比:

光栅化快, 实时, 是一种近似算法

光纤追踪慢, 离线, 是一种准确的算法

光线

光线追踪需要光线的概念，课程以三个假设引入了光线的概念：

1. 光线沿着直线传播
2. 光线之间不会碰撞
3. 光线的可逆性。光线在场景中传播，产生反弹，假设反弹至相机，那么从相机视角来看，一定可以沿着传播的路径找到光源位置

常规的着色过程

从视点发射一条光线，通过成像平面（某个像素）投射至物体，然后从物体发出光线至光源检查其是否在阴影中，如果不在阴影中，就可以通过某些着色方案对这个像素进行着色，如 Blinn Phong 等。

Recursive (Whitted-Style) Ray Tracing

该方法有几点不同于常规的着色方案：

光源：光线不同，这里假设光线在碰撞到物体后还会产生折射和反射。

着色过程：每发生一次折射或者反射（弹射点）都计算一次着色，前提是该点不在阴影内。

如图所示，该算法的过程如下：

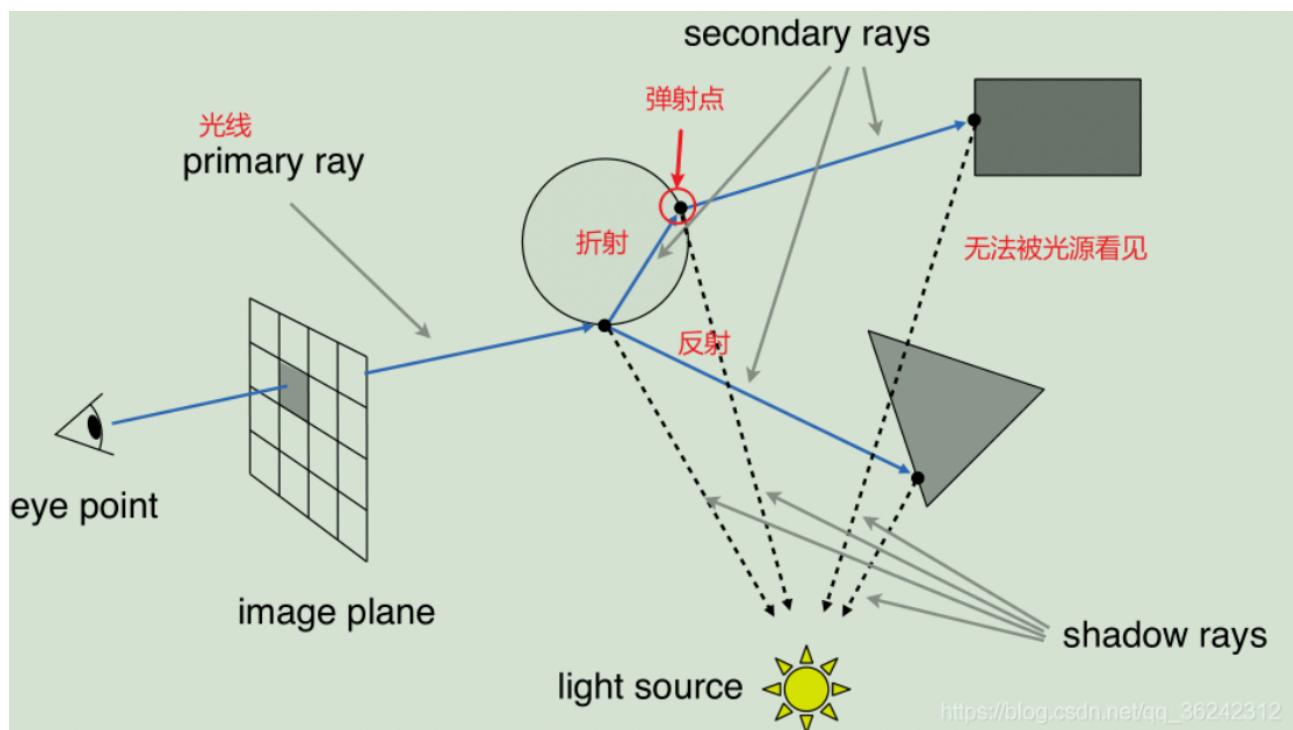
从视点从成像平面发出光线，检测是否与物体碰撞

碰撞后生成折射和反射部分

递归计算生成的光线

所有弹射点都与光源计算一次着色，前提是该弹射点能被光源看见

将所有着色通过某种加权叠加起来，得到最终成像平面上的像素的颜色



光线与三角形求交

光线与三角形相交最简单的便是将光线方程带入平面方程，求解；

这里有一种加速三角形与光线相交判断的方案：

平面方程的一般式表达为：

$$Ax + By + Cz + D = 0$$

如果光线在三角形内，那么这个点一定可以用重心坐标进行表示，那么进行求解便可得到答案：

A faster approach, giving barycentric coordinate directly

Derivation in the discussion section!

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

Where:

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

Recall: How to determine if the “intersection” is inside the triangle?

Hint:
(1-b1-b2), b1, b2 are barycentric coordinates!

Cost = (1 div, 27 mul, 17 add)

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

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

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

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

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

https://blog.csdn.net/qq_36242312

光线-表面相交加速

很快的一种方案是不用判断光线与每个三角形的相交情况，而是判断光线与包围物体表面的盒子来粗略判定光线是否与物体相交：

它等同于用更简单的模型来替代原有复杂的模型，代替原有模型求解与光线的相交

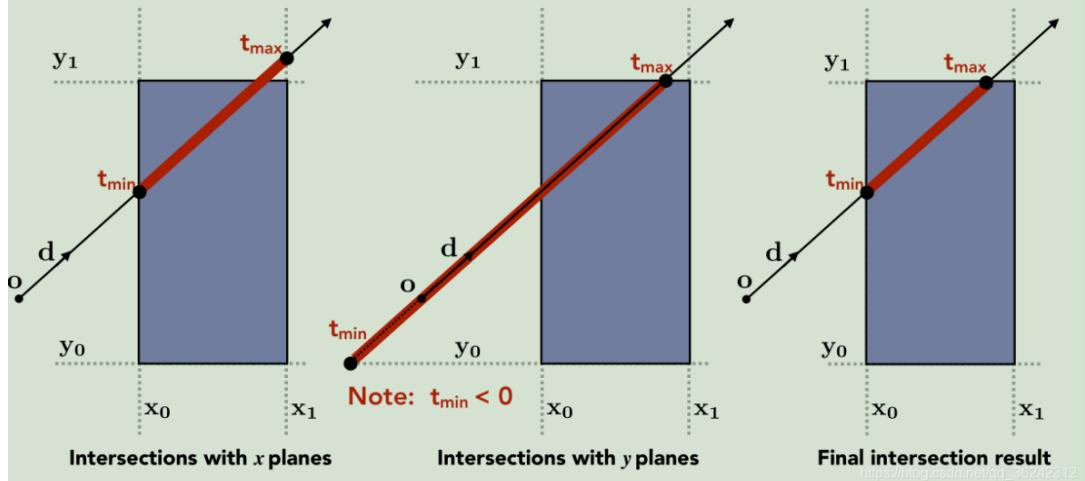
课程介绍了 AABB 包围盒（轴对称包围盒），它是轴对齐的。通常在判断物体相交情况的时候，可以先检测物体是否与 AABB 包围盒是否相交，如果相交再进一步做判断（更精细的相交判断）。

那么如何判断光线与 AABB 包围盒相交？

这里给了一个 2D 的例子进行解释：

Ray Intersection with Axis-Aligned Box

2D example; 3D is the same! Compute intersections with slabs and take intersection of t_{min}/t_{max} intervals



需要记住的关键概念是：

1. 光线需要从插槽内（图中 $y = y_0$ 和 $y = y_1$ 构成的）进入包围盒
2. 光线需要从插槽内（图中 $x = x_0$ 和 $x = x_1$ 构成的）离开包围盒

满足上述条件光线才会和物体相交

对于三维包围盒来说，有如下公式成立：

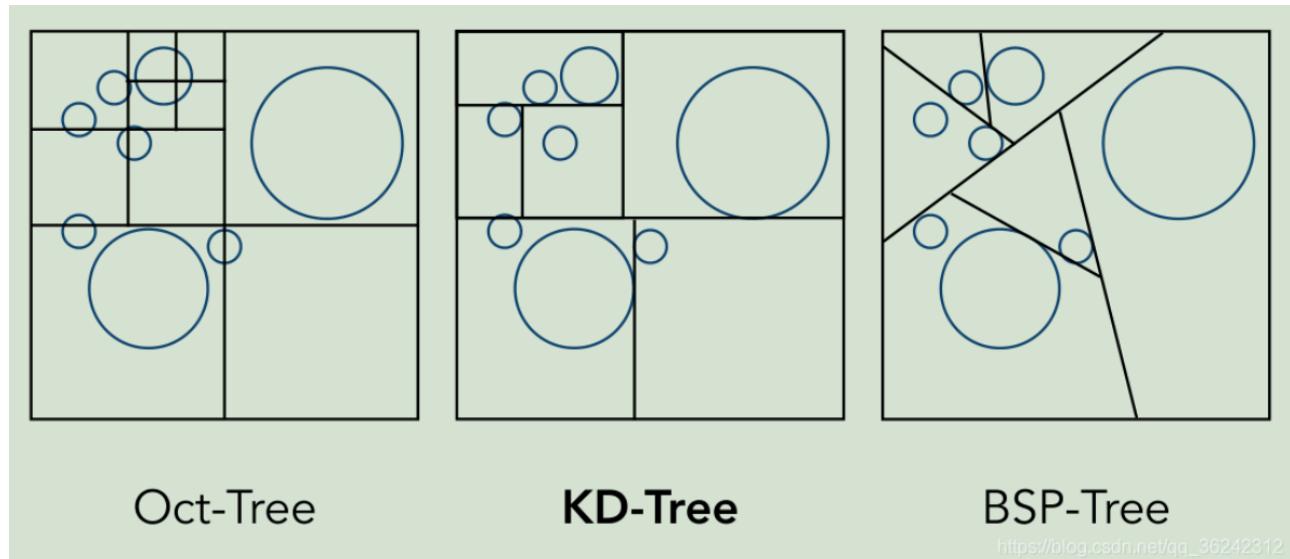
$$t_{enter} = \max \{t_{min}\}, t_{exit} = \min \{t_{max}\}$$

因为总是要寻找到插槽内位于包围盒上的点，只有当 $t_{enter} < t_{exit}$ 时，才能说光线与包围盒相交。

- However, ray is not a line
 - Should check whether t is negative for physical correctness!
- What if $t_{exit} < 0$?
 - The box is “behind” the ray — no intersection!
- What if $t_{exit} \geq 0$ and $t_{enter} < 0$?
 - The ray's origin is inside the box — have intersection!
- In summary, ray and AABB intersect iff
 - $t_{enter} < t_{exit} \&& t_{exit} \geq 0$

Lecture 14

光追加速



Oct-Tree: 每次迭代都将区域重新切分为均匀四块，按一定规则停止切分（如切分得到的四块区域中，三块都没有物体；或四块区域都还有物体，但是此时的区域已经较小）

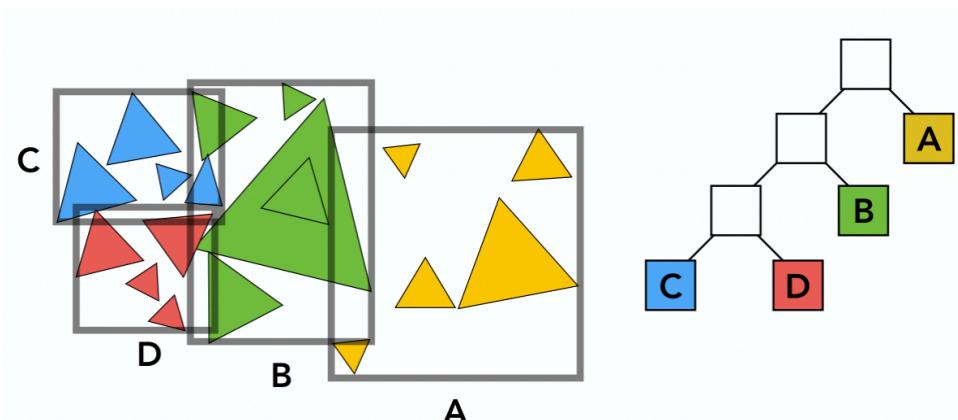
KD-Tree: 总是沿某个轴进行切分，每次划分总会在原来的区域上生成两块新的区域（这里沿轴的次序是由交替进行的，如二维中总是接着 x/y 或 y/x 的次序进行交替切分）

BSP-Tree: 每次都是沿着一定方向进行切分（非水平或竖直）

Object Partitions & Bounding Volume Hierarchy (BVH)

为了解决上述问题，人们提出了另外一种划分方案，即在场景中对物体进行划分，此时就不用考虑三角形与包围盒的求交问题

它本质是将一个场景用一个包围盒包住，然后按照一定划分方案将盒子划分成不同的子区域，不同子区域都需要包含三角形，最终划分到叶子节点时，每个叶子节点就包含了一些三角形，即包含了对应的一些物体：



辐射度量学

Radiant flux, intensity, irradiance, radiance

Radiant Flux

是单位时间的能量（功率），它简单描述了一个发光体在单位时间所发出的能量/接收的能量

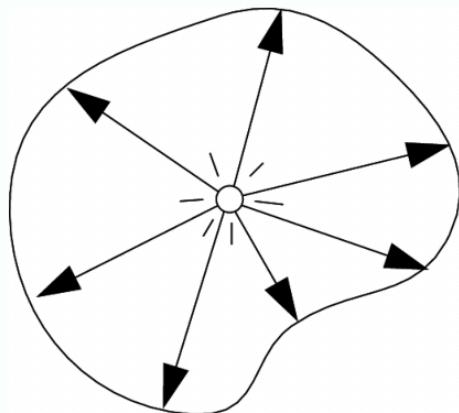
$$\Phi \equiv \frac{dQ}{dt} [W = Watt] [lm = lumen]^*$$

Radiant Intensity

由于光源是往不同方向发射光的，所以我们需要一种方法来描述在不同方向上，光源在单位时间发射的能量（即不同方向光源产生的功率），这里采用了 Radiant Intensity 来进行描述，如图所示（假设 ω 为球面上的某个方向向量）

Definition: The radiant (luminous) intensity is the power per unit **solid angle (?)** emitted by a point light source.

(立体角)



$$I(\omega) \equiv \frac{d\Phi}{d\omega}$$

$$\left[\frac{W}{sr} \right] \left[\frac{lm}{sr} = cd = candela \right]$$

The candela is one of the seven SI base units.
https://blog.csdn.net/qq_36242312

Solid Angle 描述了球面面积与半径的平方之比

Lecture 15

irradiance

irradiance 与 Radiant intensity 不同，它描述的是接收到的功率：即单位区域面积、单位时间内接收到的辐射能量：

$$E(\mathbf{x}) \equiv \frac{d\Phi(\mathbf{x})}{dA}$$

这里的面积指的是与光线所垂直的面积，即如果表面与光线存在夹角，需要对其进行投影。

相较于 Radiant intensity，irradiance 在传播过程中是在衰减的，因为 Radiant intensity 只和角度有关。但是当光源离物体表面越远，角度是不变的，但辐射到的面积却是增大的，所以 irradiance 是逐渐衰减的。

radiance

radiance 是指单位立体角、单位投影面积所辐射的能量。为什么需要 radiance？是因为需要和 irradiance 进行联系，以描述场景中物体接收到的能量以及它向周围所辐射出去的能量。

BRDF

前面定义了每单位表面如何接收能量的，以及该表面所反射能量的计算方式。那么当表面接收到能量后，它们又是如何将能量从不同方向反射出去的呢？这里就引入了 BRDF 来进行解决。由于不同方向反射的能量是不同的，所以反射出去的能量可以看成是一个分布，BRDF 解决的正是从某个方向辐射到表面的能量，它所辐射到其他方向的能量分布问题，它描述了反射方向上的能量分布（这一切的前提都是在物体表面会把接收的能量辐射出去的前提下）

定义物体表面单位面积在单位入射立体角 ω_i 、单位时间下通过光源入射所接收到的能量 $dE_i(\omega_i)$ ，假设它反射了能量，定义该单位表面沿某个单位出射立体角 ω_r 反射的能量为 $dL_r(\omega_r)$ ，人们就将 BRDF 定义为它们两者的比值：

$$f_r(\omega_i \rightarrow \omega_r) = \frac{dL_r(\omega_r)}{dE_i(\omega_i)} = \frac{dL_r(\omega_r)}{L_i(\omega_i) \cos \theta_i d\omega_i} \left[\frac{1}{sr} \right]$$

上述过程只是寻找某个单位入射立体角方向的反射能量分布，那么当所有方向都对物体表面辐射了能量，就需要将这些能量一并考虑进反射方程。所以对于某个单位表面来说，它沿某个出射立体角方向所反射的能量就是光沿所有方向辐射给该表面的能量，定义如下：

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

渲染方程

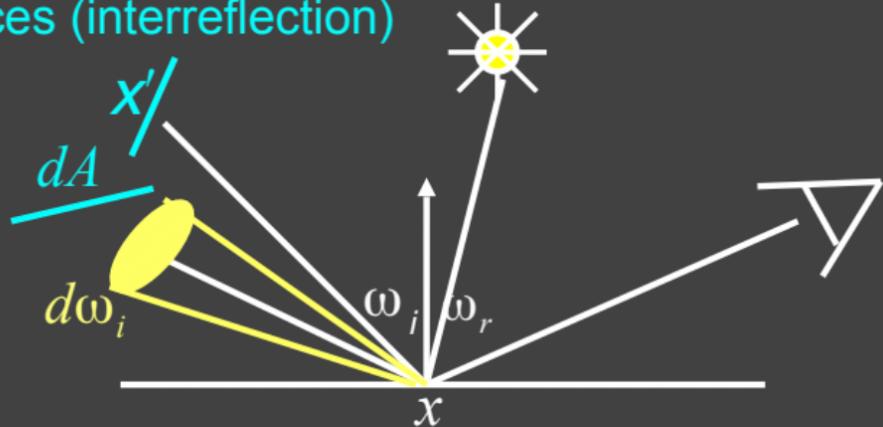
渲染方程简单理解就是：物体自反光 + 物体反射光

上一节已经计算出了物体沿各个方向通过入射光计算得到的物体反射光，而物体自发光可以通过人为规定的材质来进行定义，所以渲染方程就可以定义如下：

物体自发光	物体反射光
$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_{\Omega^+} L_i(p, \omega_i) f_r(p, \omega_i, \omega_o) (n \cdot \omega_i) d\omega_i$	

Rendering Equation

Surfaces (interreflection)



$$L_r(x, \omega_r) = L_e(x, \omega_r) + \int_{\Omega} L_r(x', -\omega_i) f(x, \omega_i, \omega_r) \cos \theta_i d\omega_i$$

Reflected Light
(Output Image)

UNKNOWN

GAMES101

Emission

KNOWN

Reflected
Light

UNKNOWN

30

BRDF

KNOWN

Cosine of
Incident angle

KNOWN

https://blog.csdn.net/cqq_36242312

Lecture 16

Monte Carlo Integration

是一种近似求解积分的方法，它在所求定积分的函数中随机取样多次，分别对取样得到的 $f(x)$ 值和定积分上下限的矩形求面积，然后将这些面积平均，以近似求解定积分。

②数学定义

定义积分： $\int_a^b f(x) dx$ 和随机变量： $X_i \sim p(x)$ ，那么 Monte Carlo estimator 的计算方式就如下：

$$F_N = \frac{1}{N} \sum_{i=1}^N \frac{f(X_i)}{p(X_i)}$$

当然，采样次数越多，该方法求解的定积分结果就越准确。

Path Tracing

为什么需要 Path Tracing ? 是为了解决 Whitted–Style Ray Tracing 中的一些问题，让渲染更加真实。

问题①：Whitted–Style Ray Tracing 在处理光照反射时总是沿着镜面反射方向进行，这适用于镜面材质，但却不适用于 Glossy 之类的材质：

问题②：Whitted–Style Ray Tracing 不考虑物体表面漫反射会进一步反射，但实际上漫反射物体仍会继续反射光线：

相较于这些问题，渲染方程的结果是对的，所以我们只需要按照渲染方程来求解物体表面的着色问题即可：

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

不考虑自发光项，那么渲染方程就可以写成：

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

为了求解这个在球面上的积分，这里就需要 Monte Carlo 方法来近似求解：

$$\begin{aligned} L_o(p, \omega_o) &= \int_{\Omega^+} L_i(p, \omega_i) f_r(p, \omega_i, \omega_o) (n \cdot \omega_i) d\omega_i \\ &\approx \frac{1}{N} \sum_{i=1}^N \frac{L_i(p, \omega_i) f_r(p, \omega_i, \omega_o) (n \cdot \omega_i)}{p(\omega_i)} \end{aligned}$$

1. 随机采样 N 次入射方向
2. 根据采样的入射方向，针对能够反射到光源的部分，计算
 $L_i(p, \omega_i) f_r(p, \omega_i, \omega_o) (n \cdot \omega_i)$
3. 累加该次求解结果
4. 平均，完成着色

问题①：物体表面在进行反射的过程中，反射的光线数目应该为多少？如果反射的光线数目为 N，反射次数为 M，那么对于一个像素上的射线的计算量都有可能为： N^M ，这个计算量是相当大且呈指数增长的。为了减少计算量，人们将反射的光线数目设为1，而在每个像素处打入多条射线来解决这个问题，这一步可以在一个循环里面做，即对于一个像素选取不同的位置打入射线以求解着色。

```
ray_generation(camPos, pixel)
    Uniformly choose N sample positions within the pixel
    pixel_radiance = 0.0
    For each sample in the pixel
        Shoot a ray r(camPos, cam_to_sample)
        If ray r hit the scene at p
            pixel_radiance += 1 / N * shade(p, sample_to_cam)
    Return pixel_radiance
```

问题②：着色何时停止，即当光线反射多少次后，不再进行反射？这里采用了 Russian Roulette (RR) 来解决这个问题。当满足概率为 P 时，返回除以该概率的着色结果： L_o/P ，当概率为 $(1-P)$ 时，就停止反射光线，通过这种方式，我们仍然能够得到正确的着色结果

$$E = P * (L_o/P) + (1 - P) * 0 = L_o$$

```
shade(p, wo)
    Manually specify a probability P_RR
    Randomly select ksi in a uniform dist. in [0, 1]
    If (ksi > P_RR) return 0.0;

    Randomly choose ONE direction wi~pdf(w)
    Trace a ray r(p, wi)
    If ray r hit the light
        Return L_i * f_r * cosine / pdf(wi) / P_RR
    Else If ray r hit an object at q
        Return shade(q, -wi) * f_r * cosine / pdf(wi) / P_RR
```

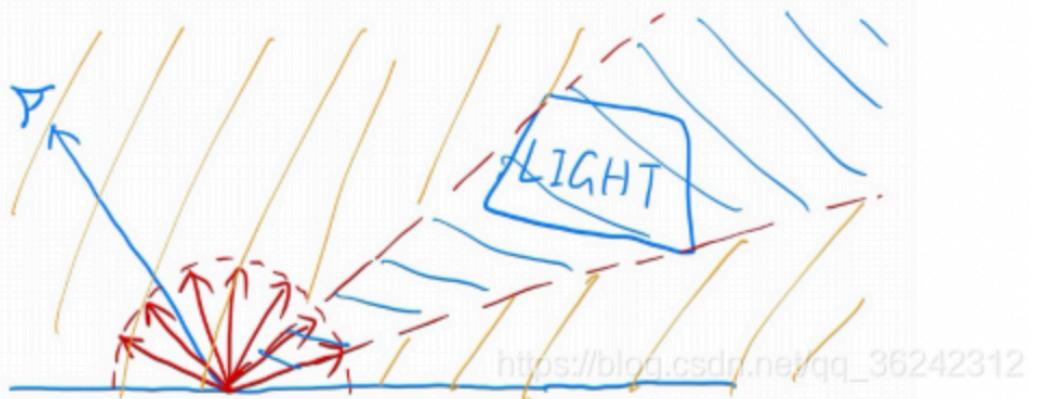
https://blog.csun.net/qq_36245512

问题③：光线在反射的过程中打到光源是存在一定概率的，它往往与光源自身大小相关，如果光源过小，会使得大量光线无法打到光源

这里的做法就是将物体表面接收到的 radiance 分解为两部分：来自光源和来自其他反射物体。对于来自光源的部分，直接在光源上进行采样即可，采样过程就是换积分域的过程，在采样完后我们还需要判断光源能否作用在物体表面

1. light source (direct, no need to have RR)

2. other reflectors (indirect, RR)



问题④：在对光源进行采样的过程中，可能会存在光源是无法打到物体的，这时需要判断一下

One final thing: how do we know if the sample on the light is not blocked or not?

Contribution from the light source.

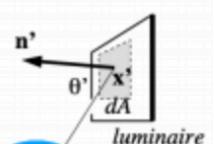
L_dir = 0.0

Uniformly sample the light at x' ($\text{pdf_light} = 1 / A$)

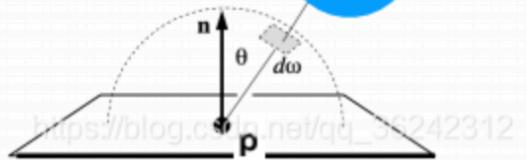
Shoot a ray from p to x'

If the ray is not blocked in the middle

L_dir = ...



Now path tracing is finally done!



1. 给定射线
2. 判断射线是否击中场景中的物体；若击中就记录下该位置，并进入 3 和 4
 3. 对光源进采样，判断光源能否对该位置进行作用
 4. 对反射后的光线进行采样，判断该光线能否反射到其他物体，如果可以，递归回到 1；否则进入 5
5. 返回光源对该位置的作用和其他物体对该位置的作用

Lecture 17