

101_1

C:\Users\17839\Desktop\101_sc\

C:\Users\17839\Desktop\101_sc\

- 点乘

$$\vec{a} \cdot \vec{b} = \|\vec{a}\| \|\vec{b}\| \cos \theta$$

最后得到的结果是一个数。向量的点乘可以方便地算出两个向量的夹角余弦，进而得出夹角。

点乘满足交换律，结合律，分配律

- In 2D

$$\vec{a} \cdot \vec{b} = \begin{pmatrix} x_a \\ y_a \end{pmatrix} \cdot \begin{pmatrix} x_b \\ y_b \end{pmatrix} = x_a x_b + y_a y_b$$

- In 3D

$$\vec{a} \cdot \vec{b} = \begin{pmatrix} x_a \\ y_a \\ z_a \end{pmatrix} \cdot \begin{pmatrix} x_b \\ y_b \\ z_b \end{pmatrix} = x_a x_b + y_a y_b + z_a z_b$$

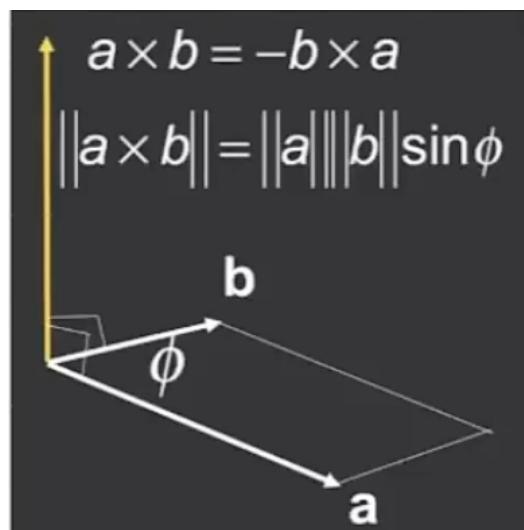
点乘在图形学中的使用：

- 找两个向量之间的夹角。
- 求一个向量在另一个向量上的投影。
- 两个向量点乘的结果可以知道这两个向量是否接近
- 两个向量点乘的结果大于零表示方向向前，小于零表示方向向后。

- 叉乘

两个向量叉乘，叉乘的结果也是一个向量，并且垂直于原本两个向量。

得到结果的方向需要有右手螺旋定则确定，假设是a向量叉乘b向量，就伸出右手，四指从向量a旋转到向量b，此时大拇指的方向就是结果的方向。



$$\vec{x} \times \vec{y} = +\vec{z}$$

$$\vec{a} \times \vec{b} = -\vec{b} \times \vec{a}$$

$$\vec{y} \times \vec{x} = -\vec{z}$$

$$\vec{a} \times \vec{a} = \vec{0}$$

$$\vec{y} \times \vec{z} = +\vec{x}$$

$$\vec{a} \times (\vec{b} + \vec{c}) = \vec{a} \times \vec{b} + \vec{a} \times \vec{c}$$

$$\vec{z} \times \vec{x} = +\vec{y}$$

$$\vec{a} \times (k\vec{b}) = k(\vec{a} \times \vec{b})$$

$$\vec{x} \times \vec{z} = -\vec{y}$$

如果x叉乘y，得到正的z，说明坐标系是右手坐标系。

叉乘在图形学中的使用：

- 可以判断向量的左右（例如向量a叉乘b，如果得到的向量z为正，说明b在a的右侧）
- 可以判断向量的内外（例如三角形abc，点p在内部，依次ab叉乘ap，bc叉乘bp，ca叉乘cp，如果都为正，说明在内部）

变换

缩放

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

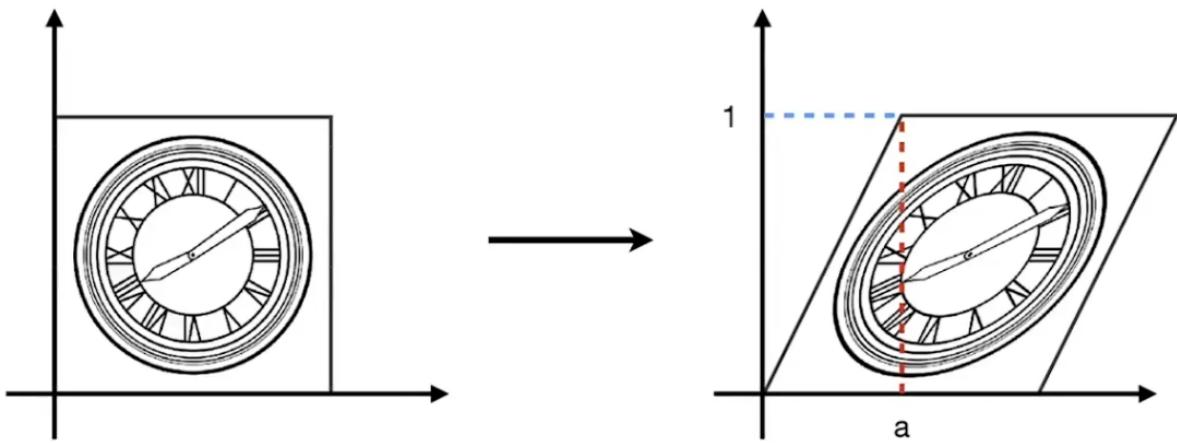
反射（对称）

$$x' = -x$$

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

$$y' = y$$

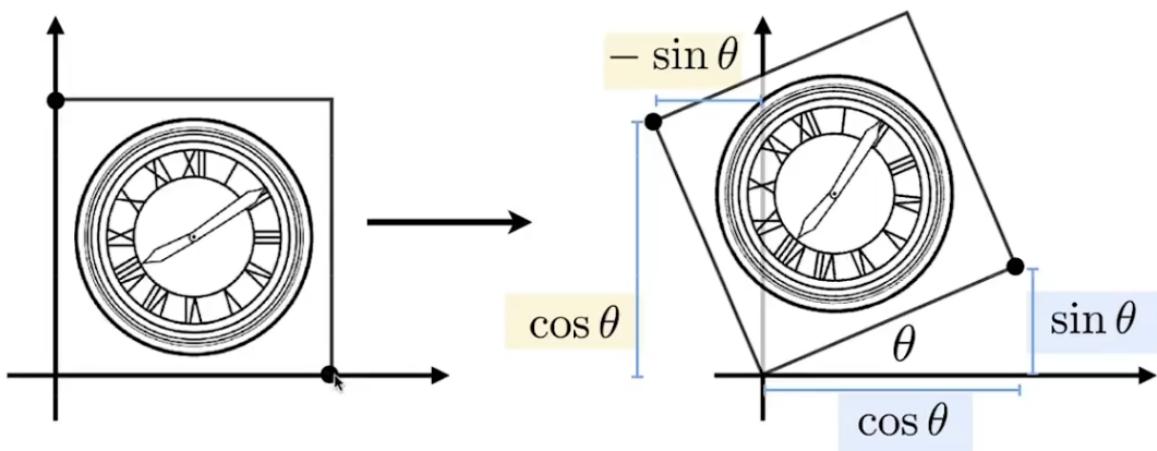
切变



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

旋转

默认：逆时针方向、绕原点旋转



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

平移

由于平移操作无法通过一个变换矩阵和向量相乘得到，因此引入齐次坐标，将所有的操作都能通过一个变换矩阵和向量相乘得到。

$$\begin{pmatrix} x' \\ y' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$$

齐次坐标

- 2D point = $(x, y, 1)^T$
- 2D vector = $(x, y, 0)^T$
- vector + vector = vector
- point - point = vector
- point + vector = point
- point + point = ??

向量时最后加0，可以保证平移不变性，并且对于上图的操作是正确的。

point + point = 这两个点的中点

仿射变换与齐次变换的对应关系：

Affine map = linear map + translation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

Using homogenous coordinates:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

因此：

Scale

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

Rotation

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

Translation



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

组合变换

$$A_n(\dots A_2(A_1(\mathbf{x}))) = \mathbf{A}_n \cdots \mathbf{A}_2 \cdot \mathbf{A}_1 \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$


三维变换

Use homogeneous coordinates again:

- 3D point = $(x, y, z, 1)^T$
- 3D vector = $(x, y, z, 0)^T$

形式如下：

Use 4×4 matrices for affine transformations

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Scale

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

Translation

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

旋转

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

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

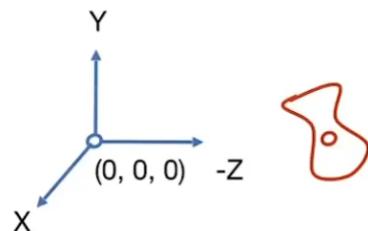
$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos \alpha & -\sin \alpha & 0 & 0 \\ \sin \alpha & \cos \alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

View Transformation

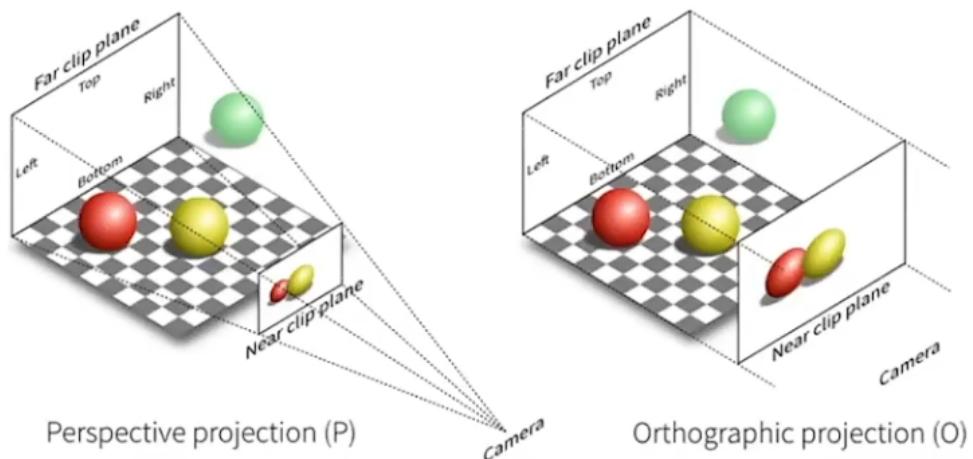
定义一个相机

- 位置 (position)
- 朝向 (look at)
- 向上方向 (up)

默认：相机向上方向为Y轴正向，朝向 -Z



projection Transformation



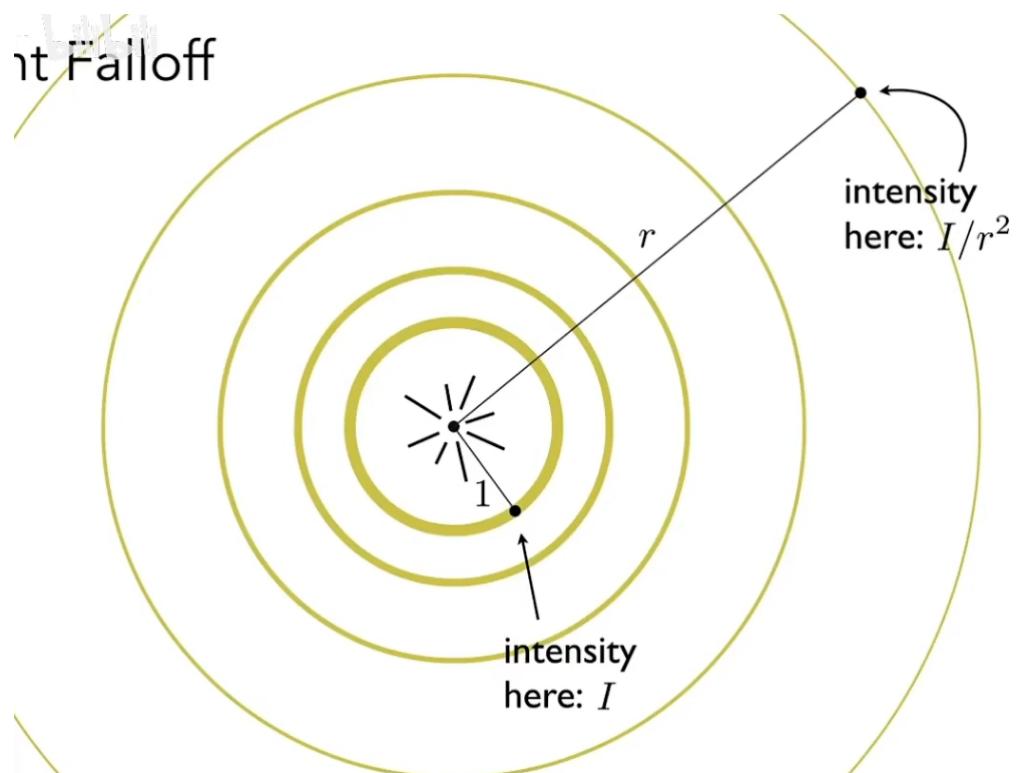
- 正交投影

正交投影: 在世界空间内，我们手动设置一个立方体，然后通过变换矩阵，先平移到原点，再缩放到[-1,1]的标准化空间，就是正交投影

- 透视投影

Shading

- 着色没有阴影，因为着色计算的是局部的区域，不考虑其它区域对光照的影响。
- 设d点受到的光照强度为 I ，则， d 点处接收到的光照强度与 I 和 n 的 \cos 成正比， n 为 d 点处的法向。
- 光源处到 i 处的光强为：



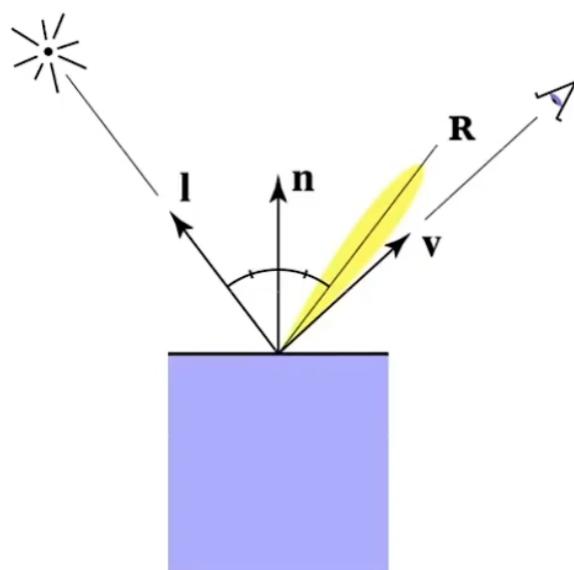
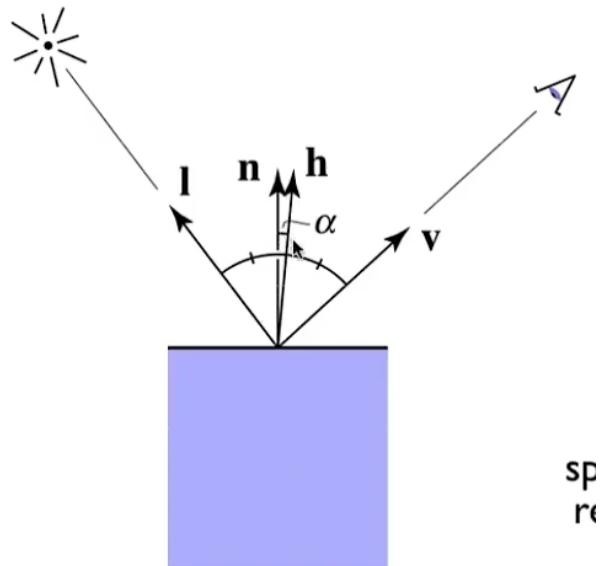
则 Diffuse (Lambertian) Shading 为：

$$L_d = k_d(I/r^2)\max(0, n \cdot L)$$

n 和 L 都是法向。 \max 的作用是：当点乘为负数时，说明没有什么意义，就设为0。

k_d 为一个[0,1]的rgb，1表示所有颜色全部反射，一点也不吸收。(注：反射什么光，显示什么颜色)

Blinn-Phong模型



入射光为 I 向量， 反射光为 R 向量， H 向量为半程向量， V 向量为视角。 n 为法向量

Blinn-Phong：当半程向量 H 和法向量 n 接近时，说明反射光向量和视线向量接近。说明更能看到高光。

$$\nabla \quad \begin{aligned} \mathbf{h} &= \text{bisector}(\mathbf{v}, \mathbf{l}) \\ &\text{(半程向量)} \\ &= \frac{\mathbf{v} + \mathbf{l}}{\|\mathbf{v} + \mathbf{l}\|} \end{aligned}$$

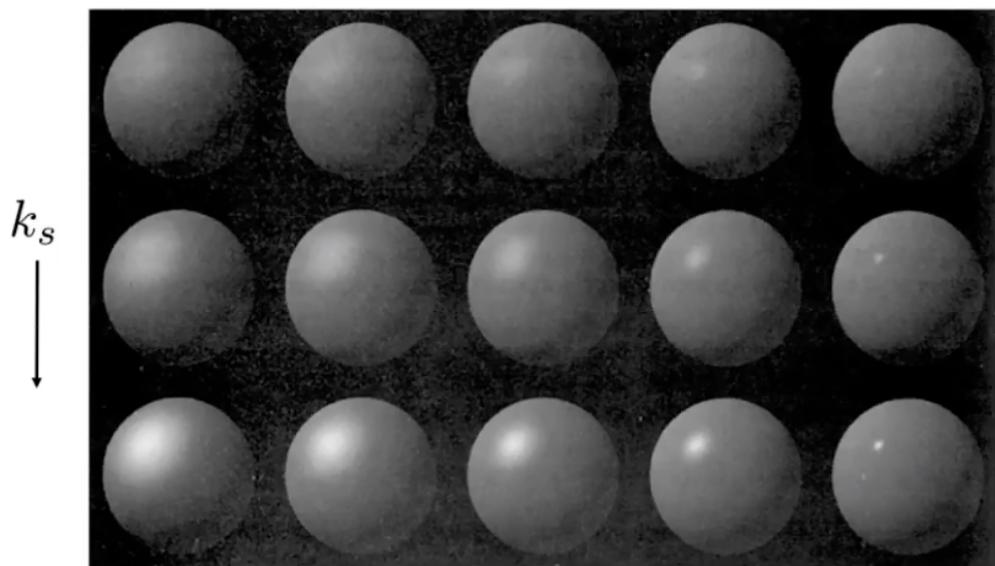
$$L_s = k_s (I/r^2) \max(0, \cos \alpha)^p$$

↑ ↑
 = $k_s (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{h})^p$
 specularly specular
 reflected coefficient
 light

则 (specularly reflected light) Shading 为：

$$L_s = k_d (I/r^2) \max(0, n \cdot h)^p$$

加P次方的原因是 \cos 在0~90度衰减太慢，所以看到的很大的光斑。

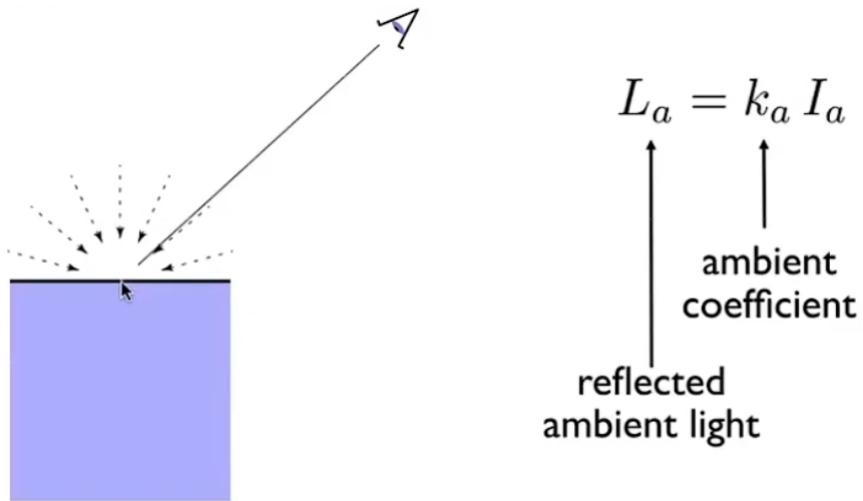


Note: showing
 $L_d + L_s$ together

p —————

Ambient Term

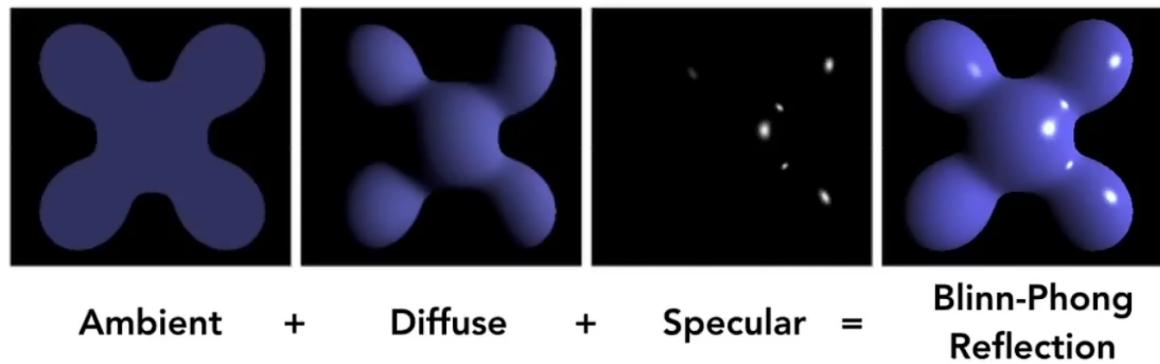
这里假设所有来自环境四面八方的光是一个常量, 和光照方向和观测方向无关。



所以 Ambient Term 为:

$$L_a = k_a I_a$$

Blinn-Phong Reflection Model

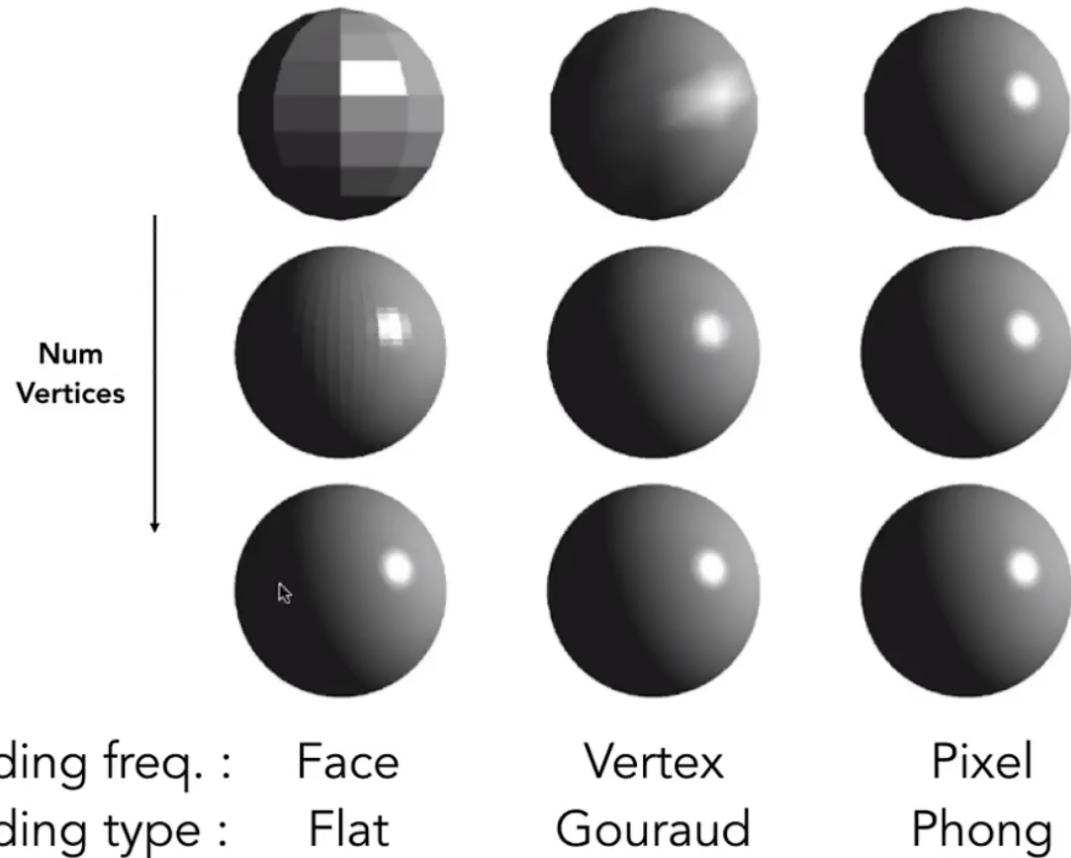


$$\begin{aligned}
 L &= L_a + L_d + L_s \\
 &= k_a I_a + k_d (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{h})^p
 \end{aligned}$$

Shading Frequencies

- 三角形的两条边做一个叉积就可以求得该三角形的法线

有三种shading方式: Face , Vertex or Pixel



Flat shading: 逐面着色

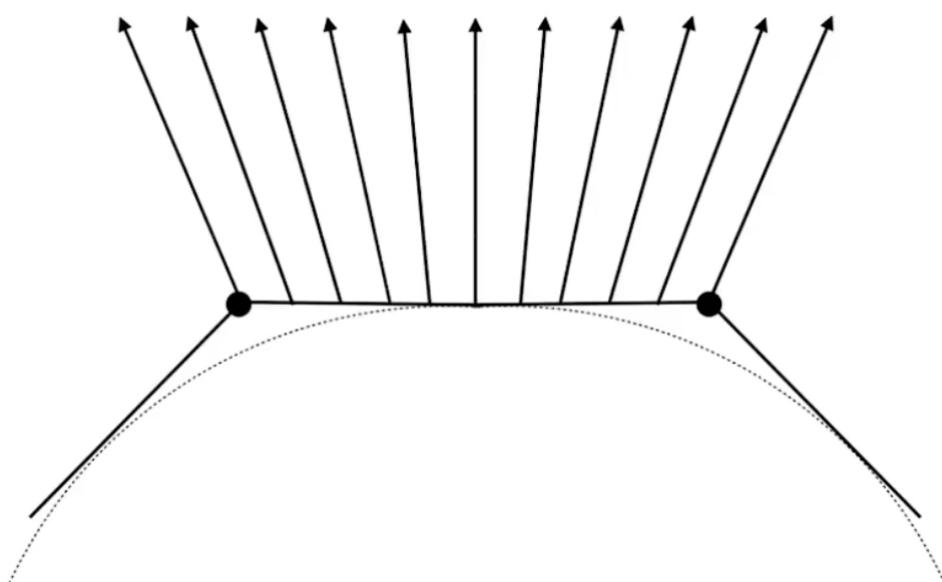
- 直接求每个面的法线，然后对面进行着色即可。

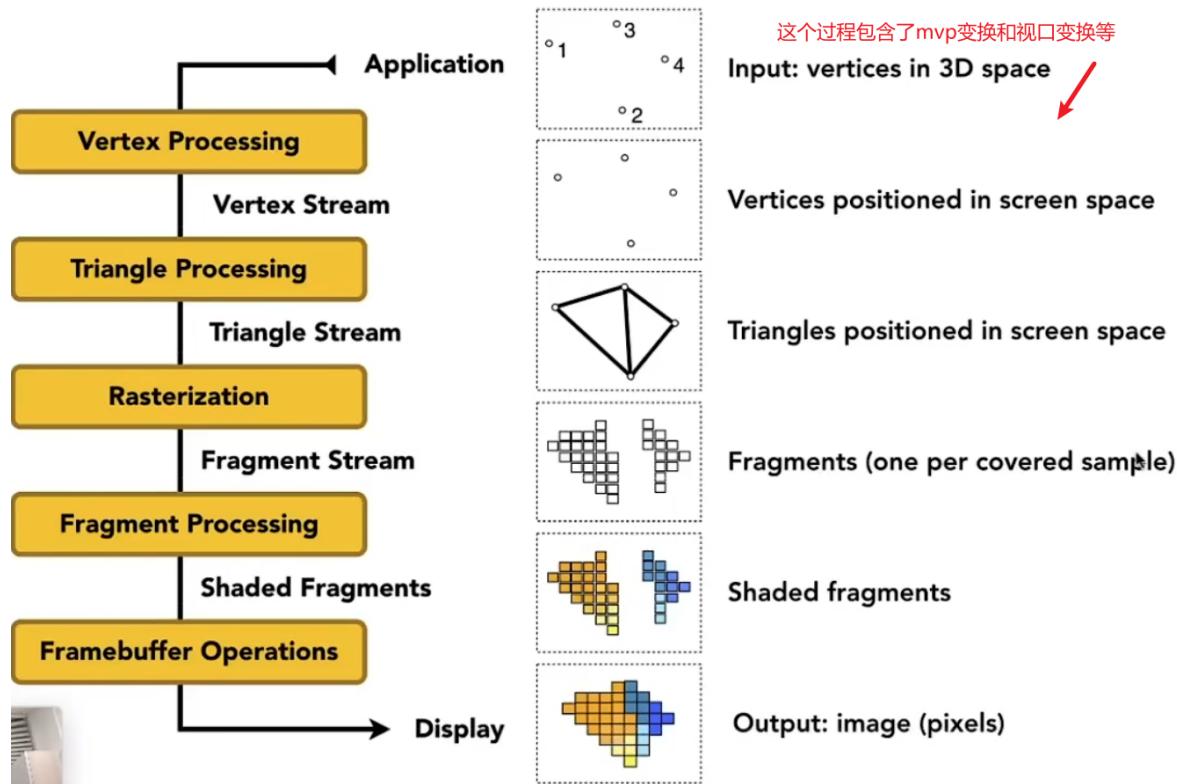
Gouraud shading: 逐顶点着色

- 由于不能直接求每个顶点的法线，因此根据该顶点周围面的法线进行加权求平均，权重为每个面的面积，用平均值作为该顶点的法线。

Phong shading: 逐像素着色

假设已知两个顶点的法线，那么可以根据重心坐标对顶点进行插值，得到顶点之间的像素的法线。





Shader Programs

Shader 是opengl的api，可以编写程序在gpu上运行。简称GLSL.

写shader时定义的是每个顶点执行的操作，因此不需要写for循环。

如果写的shader定义每个顶点的操作，那么这个shader就叫做顶点着色器，如果写的shader定义的是像素的操作，那么这个shader就叫做像素着色器。

texture

texture: Every 3D surface point also has a place where it goes in the 2D image.

纹理上的点定义了3d物体上每个顶点的一些性质。

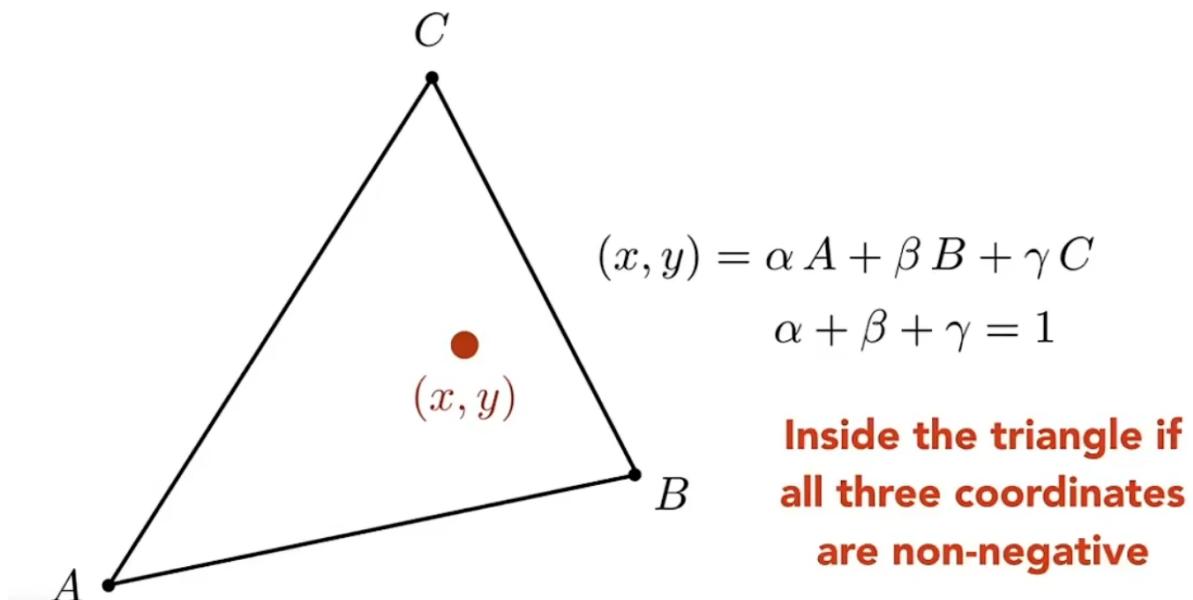
纹理图的横纵坐标分别称为U、V，范围都是在[0,1]

纹理也可以重复使用，如果一张纹理图左右，上下重复时看不到缝隙，称为tiled。比如：Wang tiled。

Barycentric Coordinates (重心坐标)

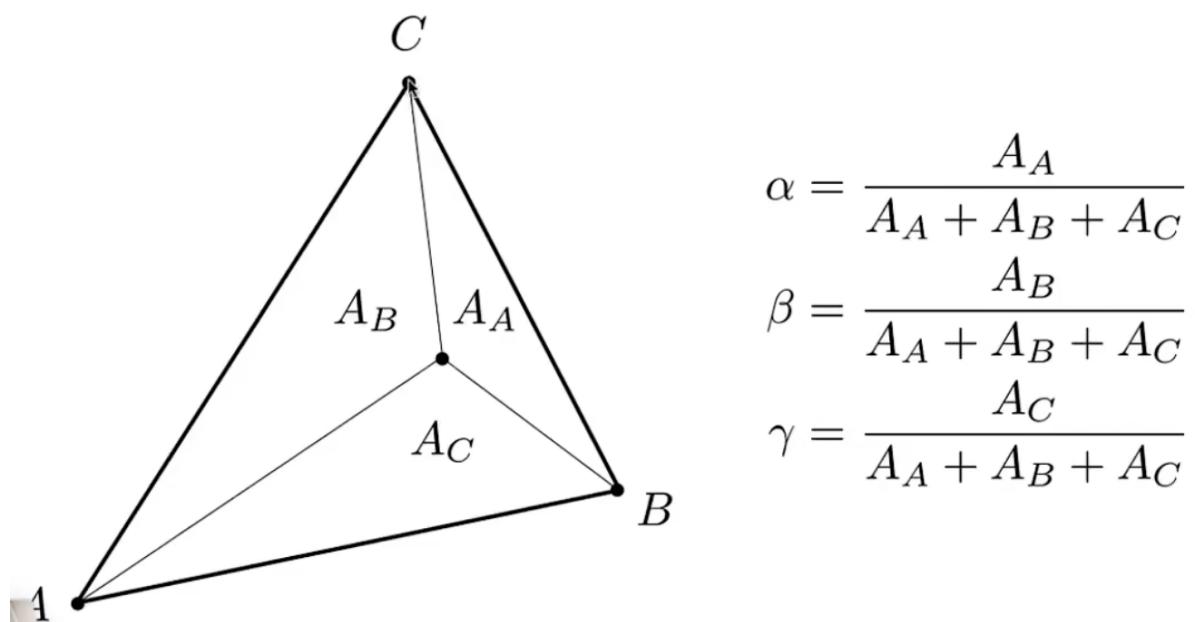
可以利用重心坐标在三角形内进行任意属性的插值：如纹理、颜色、法线

A coordinate system for triangles (α, β, γ)



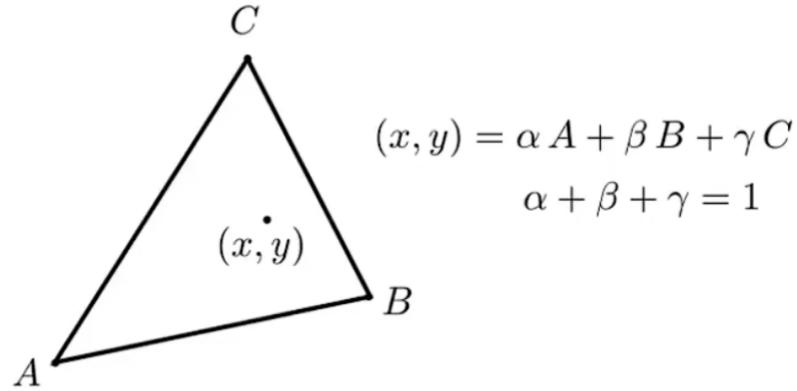
重心坐标是 (α, β, γ) , 而不是 (x, y) , 并且三个值都是非负的, 因此只需要求出两个值即可。

重心坐标计算方法:



A_x 代表对应三角形的面积

也可以不计算面积, 直接套公式:



$$\alpha = \frac{-(x - x_B)(y_C - y_B) + (y - y_B)(x_C - x_B)}{-(x_A - x_B)(y_C - y_B) + (y_A - y_B)(x_C - x_B)}$$

$$\beta = \frac{-(x - x_C)(y_A - y_C) + (y - y_C)(x_A - x_C)}{-(x_B - x_C)(y_A - y_C) + (y_B - y_C)(x_A - x_C)}$$

$$\gamma = 1 - \alpha - \beta$$

对应的实际坐标可以通过以下公式计算：

$$(\alpha, \beta, \gamma) = \left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3} \right)$$

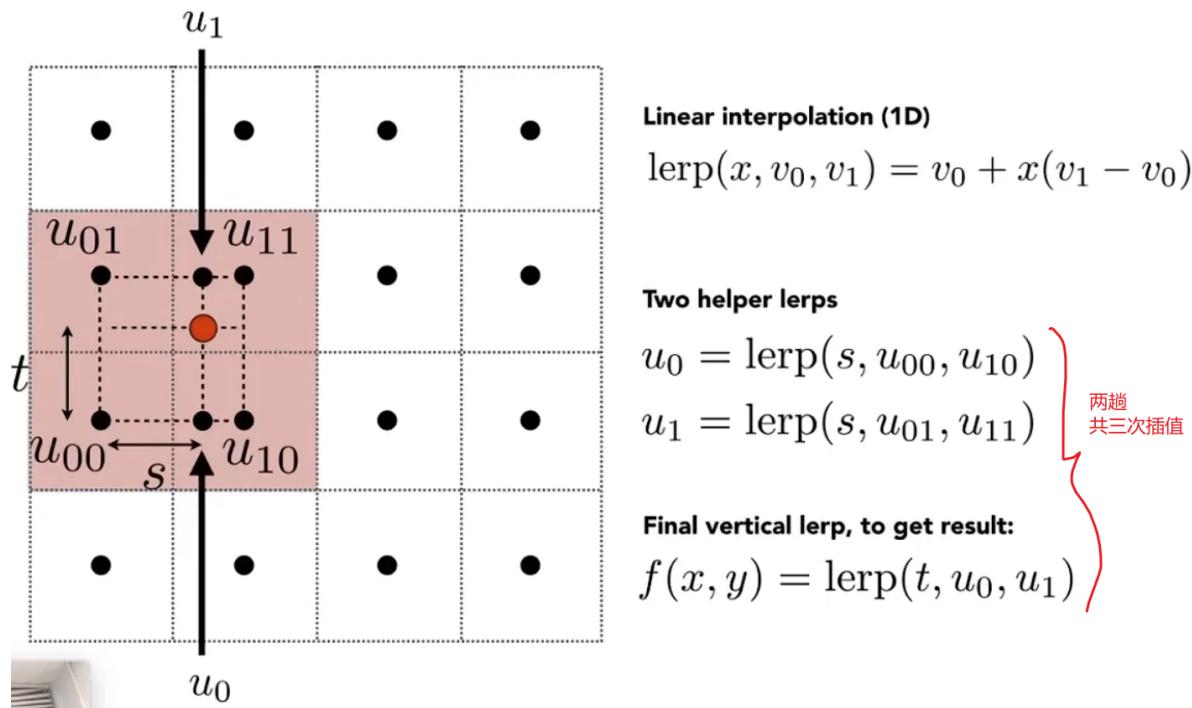
$$(x, y) = \frac{1}{3} A + \frac{1}{3} B + \frac{1}{3} C$$

投影变换后重心坐标会变。

因此在三维空间中的属性，需要在三维空间中做插值，在把结果对应到二维空间，而不能直接在二维空间做。

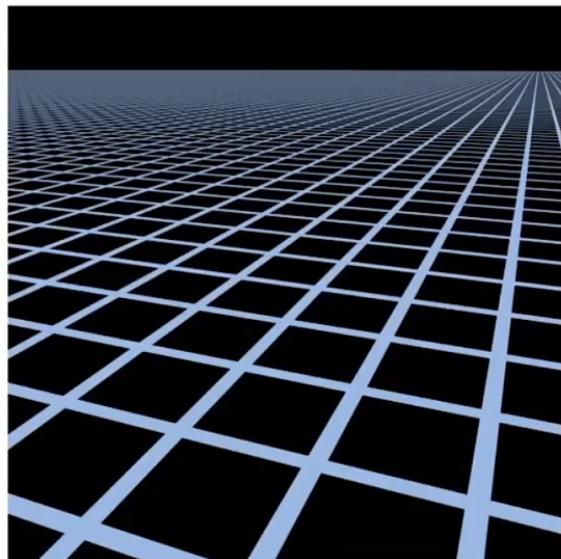
a pixel on a texture ----- a texel (纹理元素)

如果模型大，而需要贴图的纹理小的话，直接贴图会出现锯齿。这种情况，非整型位置可以采用双线性插值 (Bilinear)。

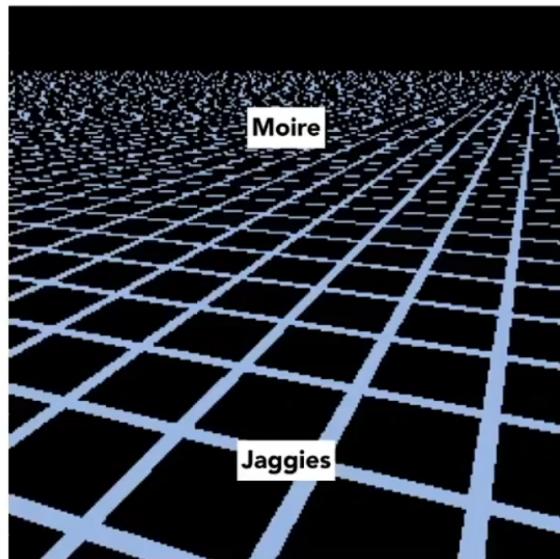


还有一种方法为Bicubic，双三次插值（每次取4个点做三次插值，而不是线性插值）

如果纹理大，而模型小的话会走样现象。



Reference

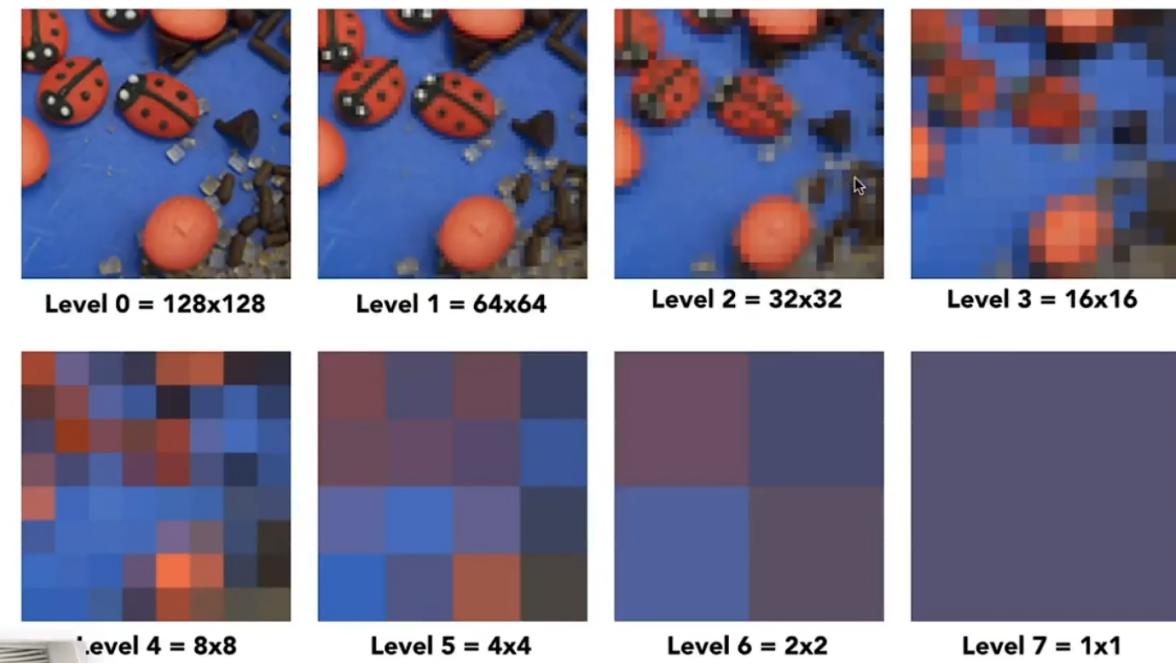


Point sampled

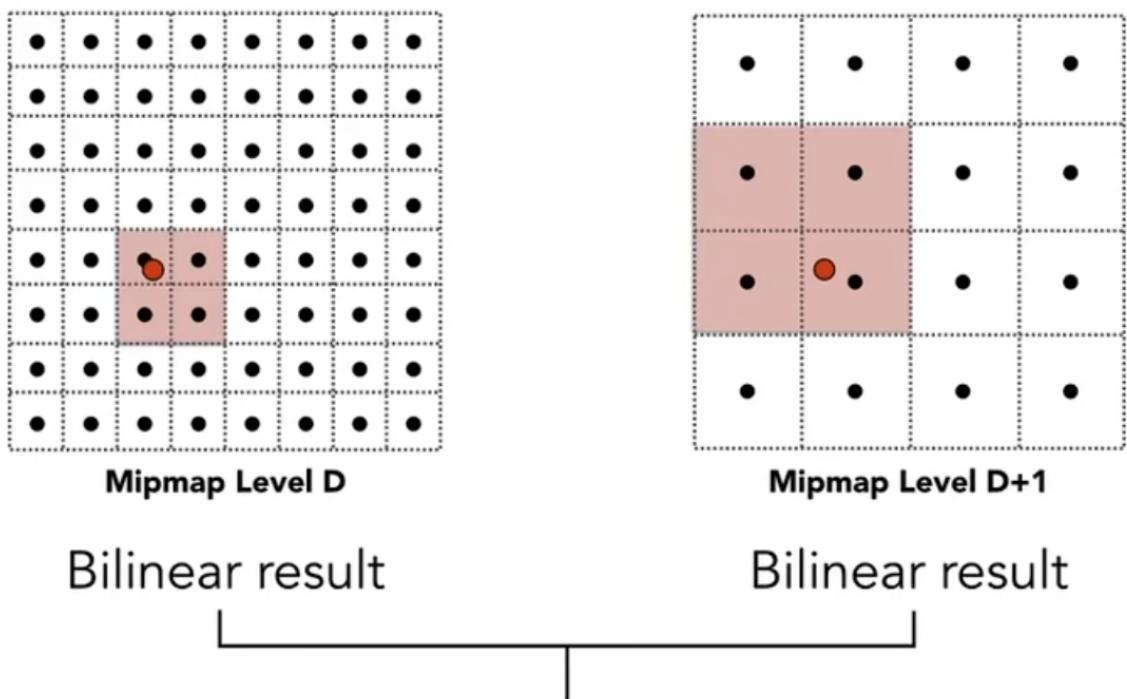
因为相比于大小合适的模型，模型小的话模型上每个顶点所能覆盖的纹理上的区域就变大了，（相当于正常大小的模型采样少了）

可以使用超采样方法缓解，但是花费太大了。

还有一种不同的方法：Mipmap（一种范围查询，很快，只能做近似的、正方形的），因为可以提前计算出每一层的纹理。并且所有层加起来只有原图存储量的1/3



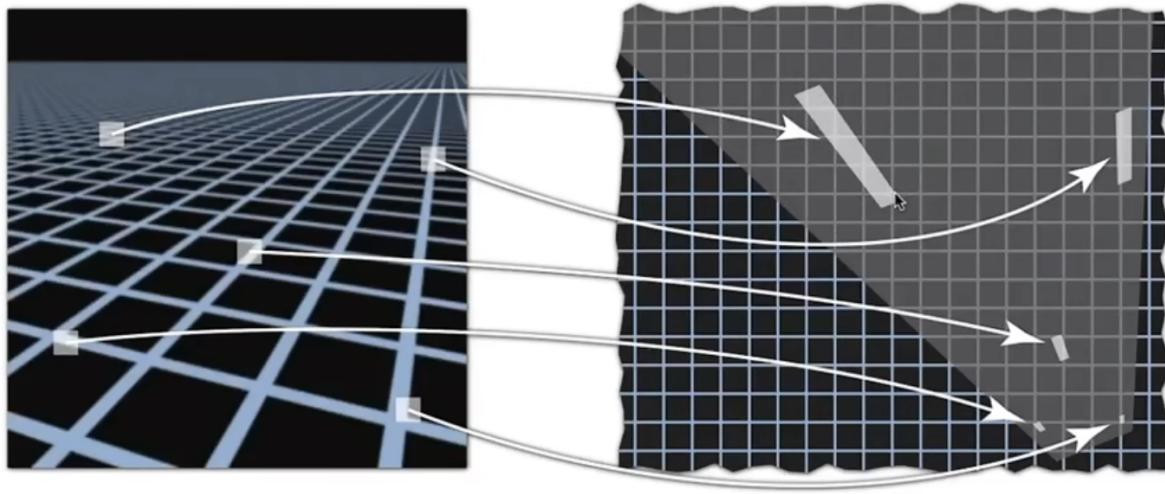
实际使用的使用，会事先计算出每一层的大小，假设纹理为 $n \times n$ ，则共有 $\log_2 n$ 层。计算每个像素对应的纹理贴图时，会得出一个近似的正方形区域，然后根据区域的面积大小去对应的层查找纹理元素的值。比如区域为 $D \times D$ ，那么就去 $\log_2 n$ 层去查找。实际上，大多数时候得出的层数不是整数，所以需要进行三线性插值。如下：



Linear interpolation based on continuous D value

假设得出的层数在D和D+1之间，那么需要先在D层和D+1层上分别做一个双线性插值，然后在层与层之间做一次线性插值。

Mipmap在视线远处会出现Overblur的情况（过分模糊），因为Mipmap是近似正方形，而实际的顶点投影到纹理上可能更接近长方形。



Screen space

Texture space

Geometry

Ray Tracing

光栅化是一种很快的、很近似的渲染方法，质量相对较低。（一般用在实时应用，比如游戏）

光线追踪是一种很慢，但是质量非常高的渲染方法（一般用在离线应用，比如电影）

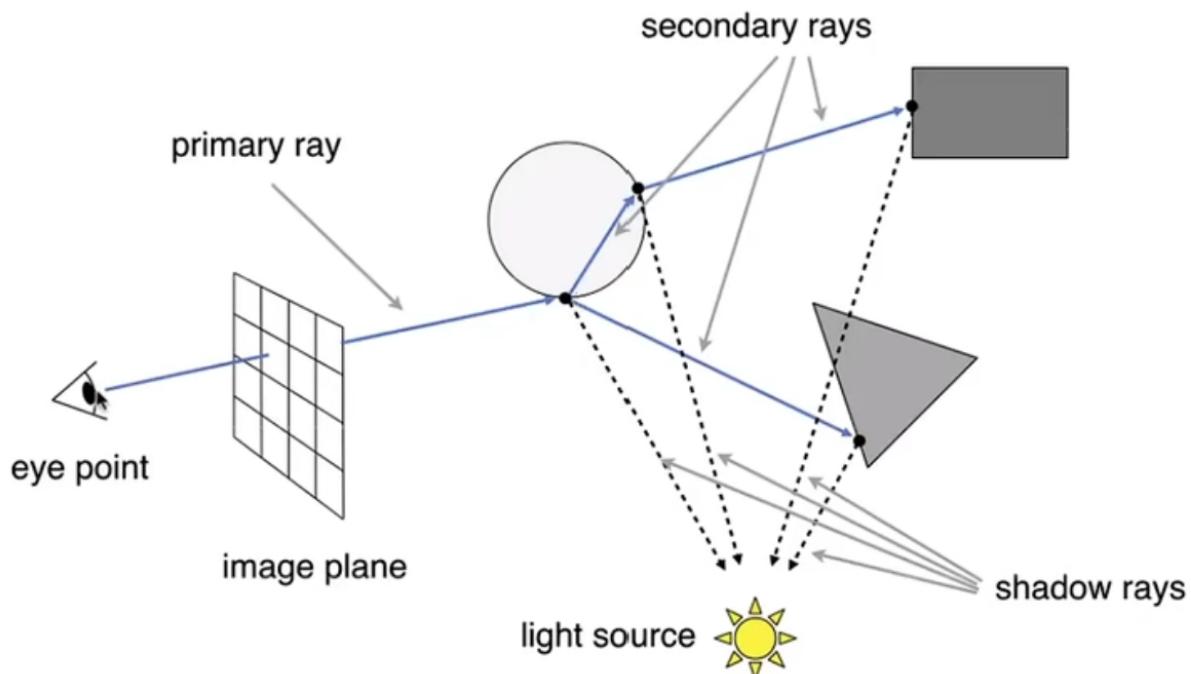
Light Rays

1. 光线是沿直线传播的
2. 光线和光线不会发生碰撞
3. 光线从光源出来，最终到人的眼睛（光线是可逆的）

Whitted-style

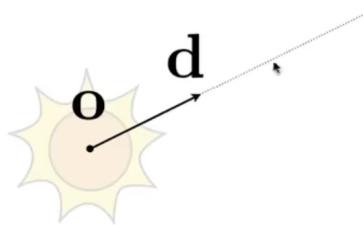
光线投射：

视窗的每个像素从眼睛投射出一个光线，打到距离最近的物体上（因为近的物体会遮挡住远的物体，因此可以不用深度计算），然后看打到的点能否被光源照亮（是否对光源可见），可见的话就直接计算该点的着色，写入像素。



最终像素的结果是所有不被遮挡的shadow rays的累加

Example:



Ray equation:

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

光线和三角形求交：先求光线和平面的交点，然后求三角形是否在平面内。

平面可以定义为：一个法线和一个点

Plane equation:

$$\mathbf{p} : (\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = 0$$

光线与平面的交点为：

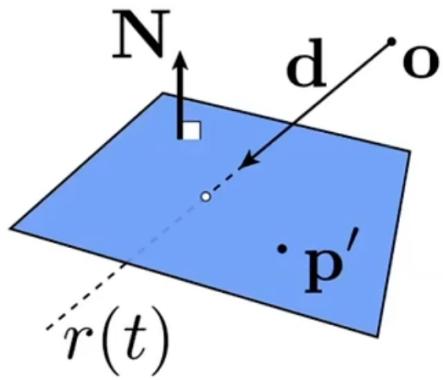
Ray equation:

$$\mathbf{r}(t) = \mathbf{o} + t \mathbf{d}, \quad 0 \leq t < \infty$$

Plane equation:

$$\mathbf{p} : (\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = 0$$

Solve for intersection



Set $\mathbf{p} = \mathbf{r}(t)$ and solve for t

$$(\mathbf{p} - \mathbf{p}') \cdot \mathbf{N} = (\mathbf{o} + t \mathbf{d} - \mathbf{p}') \cdot \mathbf{N} = 0$$

$$t = \frac{(\mathbf{p}' - \mathbf{o}) \cdot \mathbf{N}}{\mathbf{d} \cdot \mathbf{N}}$$

Check: $0 \leq t < \infty$

MT 算法: (Moller Trumbore Algorithm)[可以直接求出结果，不用先和平面求交]

用重心坐标表示平面上的点,

A faster approach, giving barycentric coordinate directly

Derivation in the discussion section!

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

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

Where:

$$\begin{aligned} \bar{\mathbf{E}}_1 &= \bar{\mathbf{P}}_1 - \bar{\mathbf{P}}_0 \\ \bar{\mathbf{E}}_2 &= \bar{\mathbf{P}}_2 - \bar{\mathbf{P}}_0 \\ \bar{\mathbf{S}} &= \bar{\mathbf{O}} - \bar{\mathbf{P}}_0 \end{aligned}$$

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)

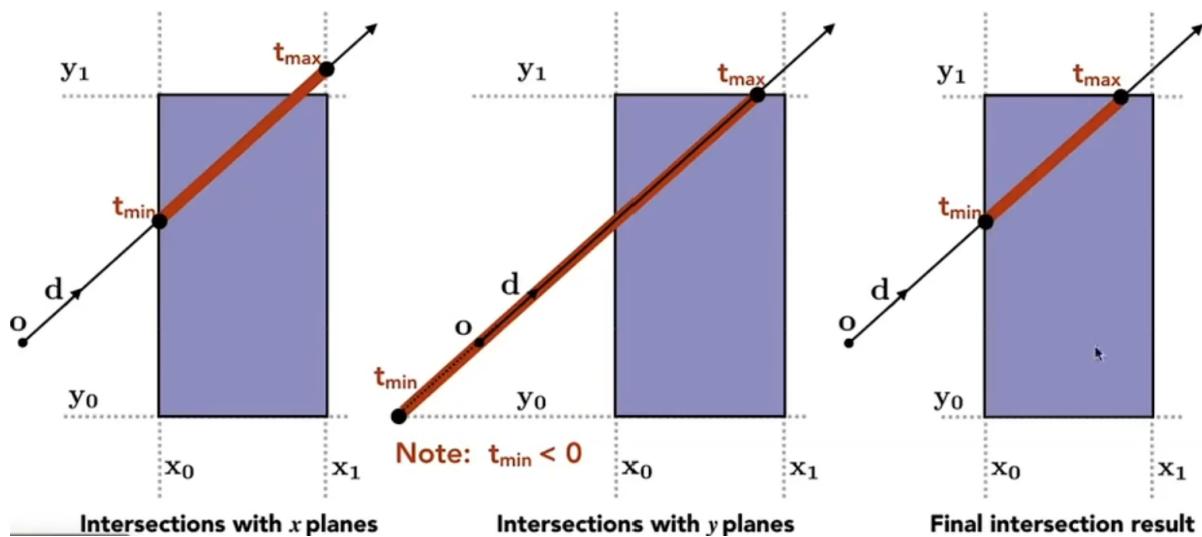
$$\begin{aligned} \bar{\mathbf{S}}_1 &= \bar{\mathbf{D}} \times \bar{\mathbf{E}}_2 \\ \bar{\mathbf{S}}_2 &= \bar{\mathbf{S}} \times \bar{\mathbf{E}}_1 \end{aligned}$$

加速光线和表面求交

先将每个模型用一个包围盒围起来，即使用AABB盒围起来，如果光线与包围盒没有交点，那么和包围盒内的物体也一定没有交点。

对于二维的情况：

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



对于三维：

- Recall: a box (3D) = three pairs of infinitely large slabs
- Key ideas
 - The ray enters the box **only when** it enters all pairs of slabs
 - The ray exits the box **as long as** it exits any pair of slabs
- For each pair, calculate the t_{\min} and t_{\max} (negative is fine)
- For the 3D box, $t_{\text{enter}} = \max\{t_{\min}\}$, $t_{\text{exit}} = \min\{t_{\max}\}$
- If $t_{\text{enter}} < t_{\text{exit}}$, we know the ray **stays a while** in the box (so they must intersect!) (not done yet, see the next slide)

但是，光线是一条射线，因此t可能为负：（有以下这几种情况）

- 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
 - $t_{enter} < t_{exit} \&\& t_{exit} \geq 0$

最终，时间计算如下：

Slabs perpendicular to x-axis

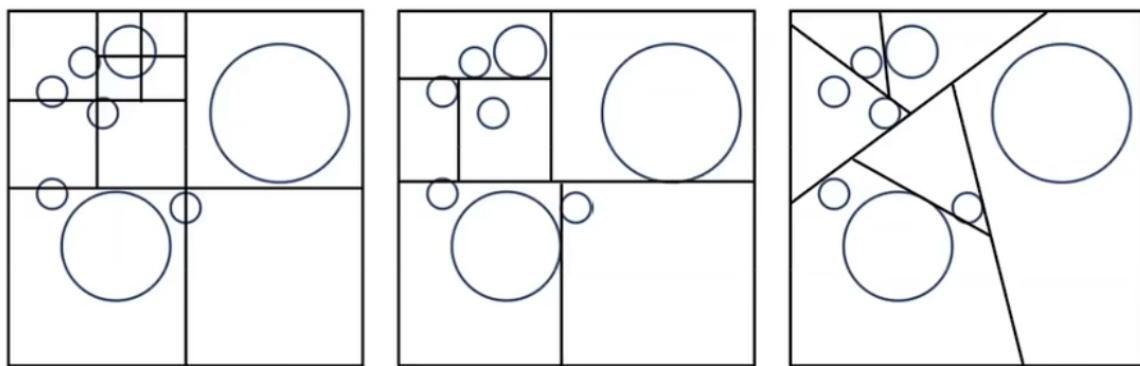
$$t = \frac{\mathbf{p}'_x - \mathbf{o}_x}{\mathbf{d}_x}$$

1 subtraction, 1 division

使用AABBS 加速光线追踪

- Uniform Spatial Partitions (Grids)

但是当场景中存在大量空旷空间时，效率很低，因此提出了多种划分方法：



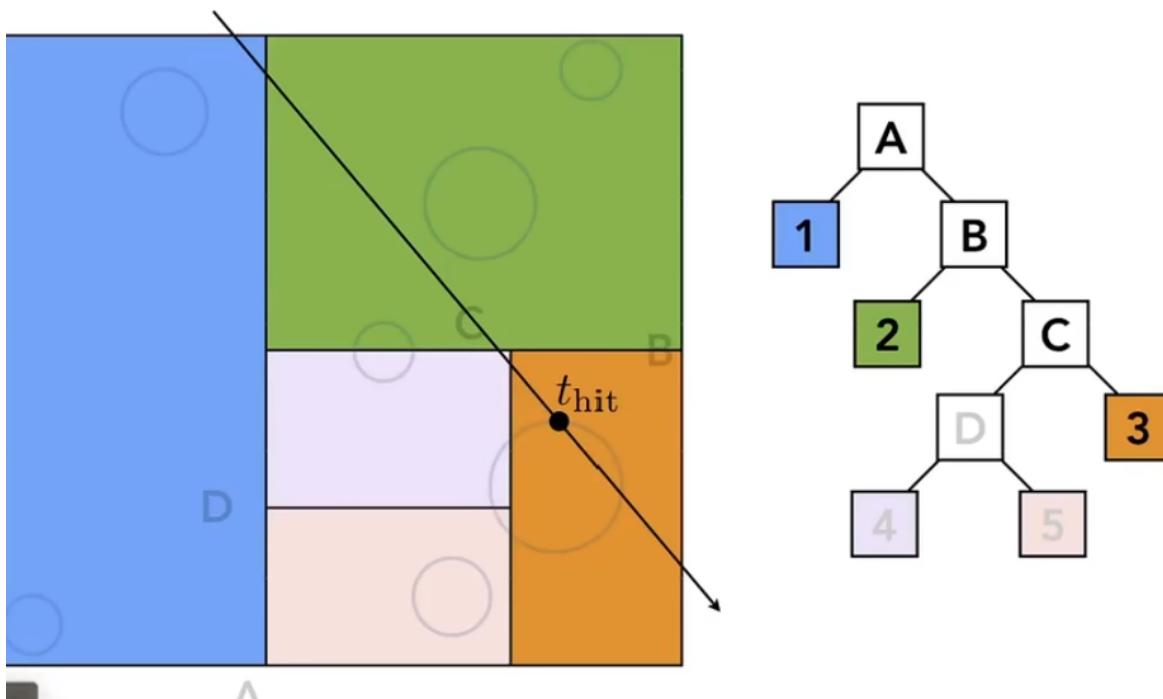
Oct-Tree

KD-Tree

BSP-Tree

KD-Tree：每次划分按照x or y or z 轴进行划分，x, y, z 依次进行。

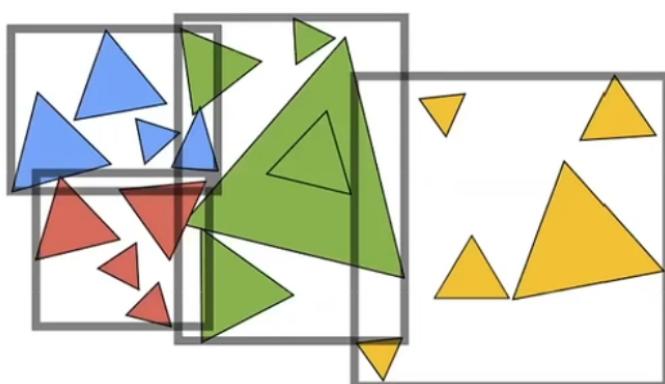
Traversing a KD-Tree



但是KD-Tree的建立比较麻烦，需要考虑三角形和盒子的求交，并且一个物体可能会存储在多个不同的格子里。

- Object Partitions & Bounding Volume Hierarchy (BVH)

划分的是物体，每次将物体划分为两部分，再分别求bounding box



- Find bounding box
- Recursively split set of objects in two subsets
- **Recompute** the bounding box of the subsets
- Stop when necessary
- Store objects in each leaf node

划分要求：

How to subdivide a node?

- Choose a dimension to split
- Heuristic #1: Always choose the longest axis in node
- Heuristic #2: Split node at location of **median** object

BVH的数据结构：

Data Structure for BVHs

Internal nodes store

- Bounding box
- Children: pointers to child nodes

Leaf nodes store

- Bounding box
- List of objects

Nodes represent subset of primitives in scene

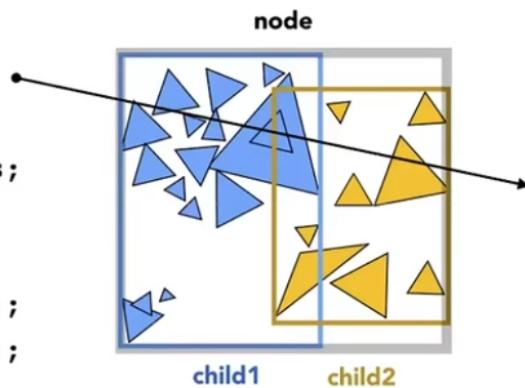
- All objects in subtree

BVH算法思路：

```
Intersect(Ray ray, BVH node) {
    if (ray misses node.bbox) return;
    if (node is a leaf node)
        test intersection with all objs;
        return closest intersection;

    hit1 = Intersect(ray, node.child1);
    hit2 = Intersect(ray, node.child2);

    return the closer of hit1, hit2;
}
```

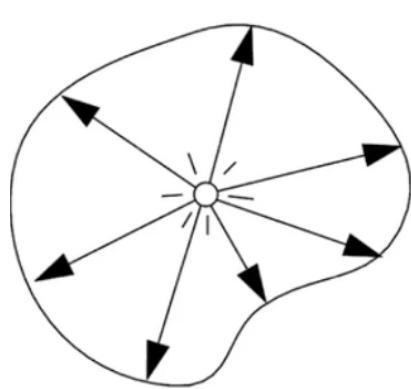


Basic radiometry(辐射度量学)

专业名词定义：

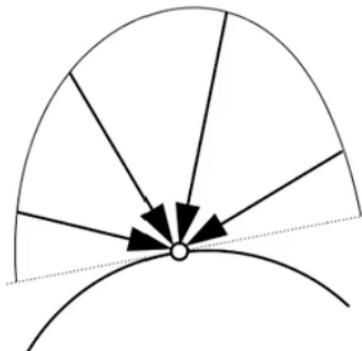
- Radiant Energy and Flux(power)

Flux:光的功率，代表光的亮度



Light Emitted
From A Source

“Radiant Intensity”



Light Falling
On A Surface

“Irradiance”



Light Traveling
Along A Ray

“Radiance”

Radiant intensity: 一个点（物体）往四面八方散发的能量

Irradiance:一个点（物体）接收的来自四面八方的能量

Radiance:光线载传播过程中的能量