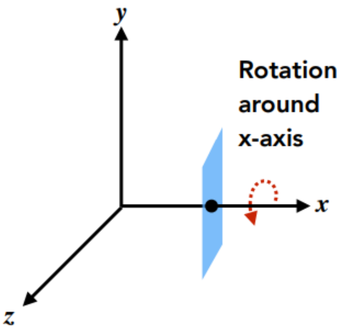
Rotation around x-, y-, or z-axis

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

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

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



Anything strange about R_y?

3D Rotations

Compose any 3D rotation from $\mathbf{R}_x, \mathbf{R}_y, \mathbf{R}_z$?

$$\mathbf{R}_{xyz}(\alpha, \beta, \gamma) = \mathbf{R}_x(\alpha)\,\mathbf{R}_y(\beta)\,\mathbf{R}_z(\gamma)$$

三维上旋转都可以拆分成 xyz 上的旋转，角度对应轴为拇指的右手定则

二维上 xoy 旋转，$\begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$ 其实就是绕 z 旋转，所以看起来是逆时针

v 绕着 n 旋转，角度 α 是 以 n 为大拇指的右手法则方向
旋转是在垂直 n 的平面上画弧，对应角为 α

罗德里格斯旋转公式可以表示为：

$$\mathbf{v}_{\text{rot}} = \mathbf{v}\cos(\alpha) + (\mathbf{n} \times \mathbf{v})\sin(\alpha) + \mathbf{n}(\mathbf{n} \cdot \mathbf{v})(1 - \cos(\alpha))$$

其中：

- $\mathbf{v}_{\text{rot}}$ 是旋转后的向量。
- $\mathbf{v}$ 是原始向量。
- α 是旋转角度（以弧度为单位）。
- $\mathbf{n}$ 是代表旋转轴的单位向量。
- $\mathbf{n} \times \mathbf{v}$ 是 $\mathbf{n}$ 和 $\mathbf{v}$ 的叉积。
- $\mathbf{n} \cdot \mathbf{v}$ 是 $\mathbf{n}$ 和 $\mathbf{v}$ 的点积。

解释：

1. **第一项 $\mathbf{v}\cos(\alpha)$：**
   - 这一部分代表原始向量 $\mathbf{v}$ 在自身方向上的投影，并乘以 $\cos(\alpha)$。它给出了 $\mathbf{v}$ 在原始方向上的分量，该分量根据旋转角度 α 进行缩放。
2. **第二项 $(\mathbf{n} \times \mathbf{v})\sin(\alpha)$：**
   - 这一项给出了旋转的垂直分量。叉积 $\mathbf{n} \times \mathbf{v}$ 生成一个与 $\mathbf{n}$ 和 $\mathbf{v}$ 都垂直的向量，$\sin(\alpha)$ 通过缩放这个垂直方向来实现基于 α 的旋转量。
3. **第三项 $\mathbf{n}(\mathbf{n} \cdot \mathbf{v})(1 - \cos(\alpha))$：**
   - 这一部分考虑了沿旋转轴 $\mathbf{n}$ 的向量分量。由于沿旋转轴的向量不随旋转改变方向，这一项确保该分量不会受旋转影响，只是根据角度适当地缩放。

旋转的可视化：

- 想象你有一个向量 $\mathbf{v}$，并希望将其绕某个轴 $\mathbf{n}$ 旋转角度 α。
- 公式将旋转分解为以下几个分量：
  - 一部分沿着旋转轴（保持不变），
  - 一部分垂直于旋转轴（发生旋转），
  - 另一部分确保围绕旋转轴的旋转量正确。

1. 旋转角度 α 的定义：

- α 是从原始向量 $\mathbf{v}$ 到旋转后的向量 $\mathbf{v}_{\text{rot}}$ 的角度。
- 旋转轴 $\mathbf{n}$ 是一个固定的单位向量，表示旋转的方向。旋转发生在垂直于该轴的平面上。
- α 以弧度为单位，表示旋转的角度大小，范围通常为 $[0, 2\pi]$ 或 $[0°, 360°]$。

2. 旋转方向：右手法则

为了确定旋转的方向，我们使用 **右手法则**：

- 伸出右手，大拇指指向旋转轴 $\mathbf{n}$ 的方向，手指弯曲的方向就是旋转的方向。
- 如果你用右手的大拇指沿着 $\mathbf{n}$ 指向（即轴的方向），那么其他四指的弯曲方向就是向量 $\mathbf{v}$ 围绕 $\mathbf{n}$ 旋转的方向。
  - 如果角度 α 是正的（例如 $\alpha > 0$），那么旋转是按照右手法则的方向，即顺着你手指弯曲的方向旋转。
  - 如果角度 α 是负的（例如 $\alpha < 0$），那么旋转方向相反，即逆着你手指弯曲的方向旋转。

3. 具体旋转的几何解释：

- $\mathbf{n}$ 表示旋转的轴，向量 $\mathbf{v}$ 将围绕这根轴在垂直于 $\mathbf{n}$ 的平面上旋转。
- α 表示旋转的角度，描述了 $\mathbf{v}$ 到 $\mathbf{v}_{\text{rot}}$ 的旋转程度。
- **旋转的轨迹**：假设 $\mathbf{v}$ 不与 $\mathbf{n}$ 平行，则 $\mathbf{v}$ 在垂直于 $\mathbf{n}$ 的平面内沿着圆弧旋转。旋转的中心是轴 $\mathbf{n}$，角度 α 是该圆弧所对应的圆心角。

4. 如何在空间中确定旋转：

设想一个三维空间中的例子：

- 假设 $\mathbf{n}$ 是沿着 z 轴的单位向量，即 $\mathbf{n} = (0, 0, 1)$。
- 向量 $\mathbf{v}$ 是 $(1, 0, 0)$，即沿 x 轴的单位向量。
- 如果我们围绕 z 轴旋转 $\alpha = 90°$（即 $\frac{\pi}{2}$ 弧度），那么 $\mathbf{v}$ 会从 x 轴的方向旋转到 y 轴的方向，最终变为 $\mathbf{v}_{\text{rot}} = (0, 1, 0)$。

Perspective Projection

- Solve for A and B

$$An + B = n^2 \qquad\qquad A = n + f$$
$$Af + B = f^2 \qquad\qquad B = -nf$$

- Finally, every entry in $M_{persp\text{->}ortho}$ is known!
- What's next?
  - Do orthographic projection (M_{ortho}) to finish
  - $M_{persp} = M_{ortho} M_{persp \to ortho}$

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

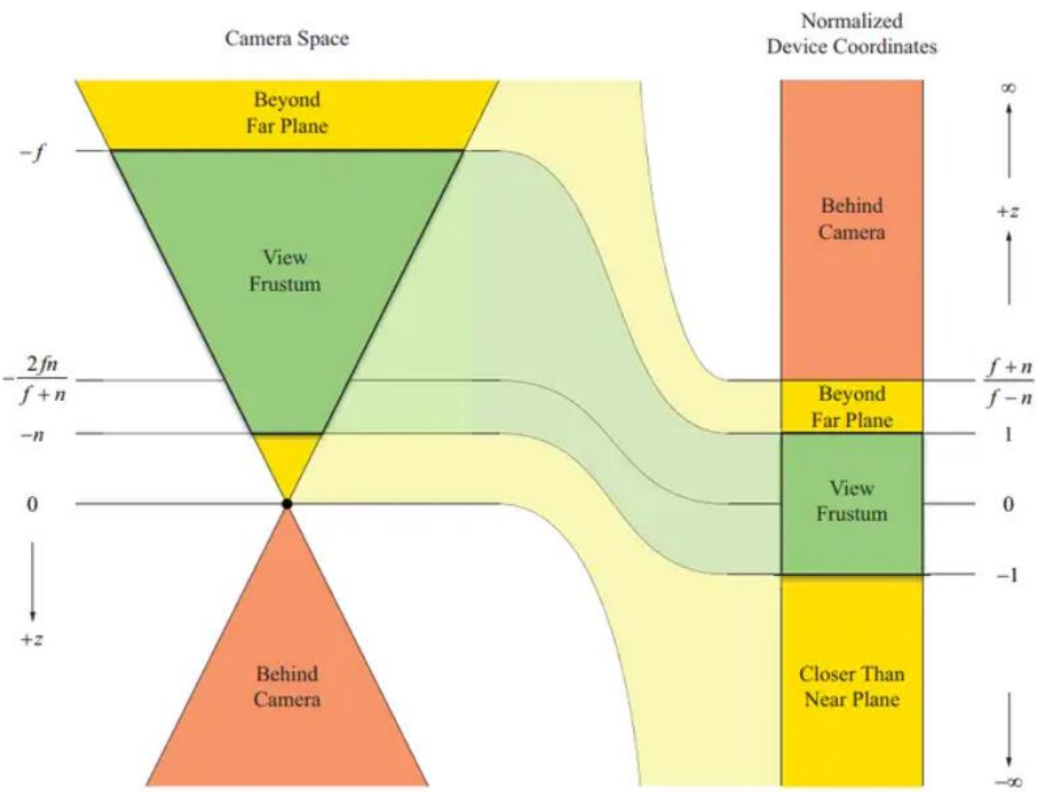
n 0 0 0
0 n 0 0
0 0 n+f -nf
0 0 1 0
上面是从远到近
然后压缩到【-1，1】
还得
1/width，0，0，0
1，1/top，0，0，
1，0，1/near-far，0， z 也要压缩到【0，1】
这里可能不对，但是 homework1 效果却没问题
0，0，0，1
俩个相乘

```cpp
Eigen::Matrix4f get_projection_matrix(float eye_fov, float aspect_ratio,
                                      float zNear, float zFar)
{
    // Students will implement this function

    Eigen::Matrix4f projection = Eigen::Matrix4f::Identity();

    // TODO: Implement this function
    // Create the projection matrix for the given parameters.
    // Then return it.
    Eigen::Matrix4f p1,p2;
    p1<<zNear,0,0,0,
        0,zNear,0,0,
        0,0,zNear+zFar,-zNear*zFar,
        0,0,1,0;
    float top=-abs(zNear)*tan((eye_fov/2.0)/180.0*acos(-1));
    float width=aspect_ratio*top;
    p2<<1.0/width,0,0,0,
        0,1.0/top,0,0,
        0,0,1.0/zNear,0,
        0,0,0,1;
    projection=p2*p1*projection;
```

参数是 相机视角大小、宽高比、近、远

$$\begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

https://www.songho.ca/opengl/gl_projectionmatrix.html



https://zhuanlan.zhihu.com/p/509902950

光栅化

```
static bool insideTriangle(int x, int y, const Vector3f* _v)
{
    // TODO : Implement this function to check if the point (x, y) is inside the triangle
    Eigen::Vector3f point((float)x+0.5,(float)y+0.5,0);
    Eigen::Vector3f v0=_v[0];
    Eigen::Vector3f v1=_v[1];
    Eigen::Vector3f v2=_v[2];
    int flag0=(v1-v0).cross(point-v0).z();
    int flag1=(v2-v1).cross(point-v1).z();
    int flag2=(v0-v2).cross(point-v2).z();
    if(flag0>0&&flag1>0&&flag2>0) return true;
    if(flag0<0&&flag1<0&&flag2<0) return true;
    return false;
}
```

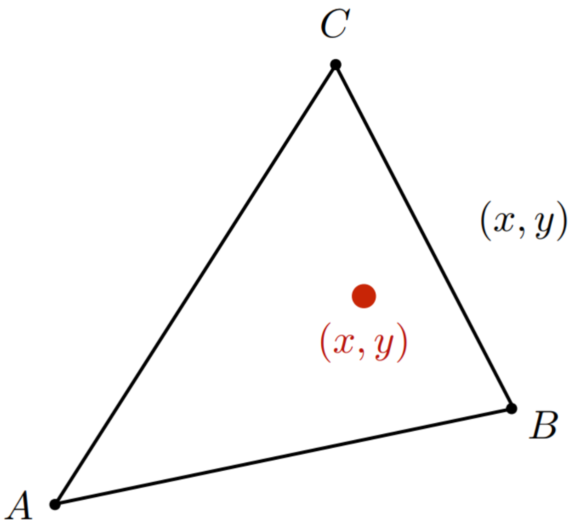
通过叉乘判断点是否在三角形内，三角形是经过 mvp 再放大到屏幕以后的三角形

```
if(insideTriangle(x,y,t.v)){
    // If so, use the following code to get the interpolated z value.
    auto[alpha, beta, gamma] = computeBarycentric2D(x+0.5, y+0.5, t.v);
    float w_reciprocal = 1.0/(alpha / v[0].w() + beta / v[1].w() + gamma / v[2].w());
    float z_interpolated = alpha * v[0].z() / v[0].w() +
                           beta * v[1].z() / v[1].w() +
                           gamma * v[2].z() / v[2].w();
    z_interpolated *= w_reciprocal;
    // TODO : set the current pixel (use the set_pixel function) to the color of the tria
    int index=get_index(x,y);
    if(z_interpolated<depth_buf[index]){
        depth_buf[index]=z_interpolated;
        frame_buf[index]=t.getColor();
    }
}
```

点在三角形内，计算出 z，根据 z-buffer 对像素上色

Barycentric Coordinates

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



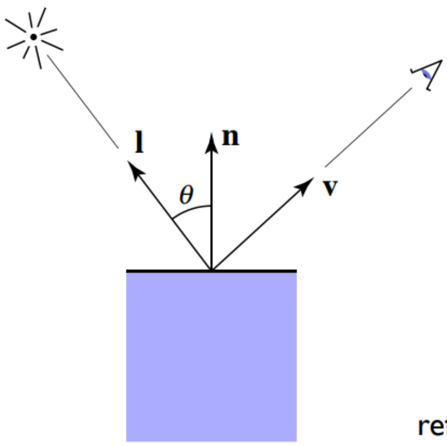
$$(x, y) = \alpha A + \beta B + \gamma C$$
$$\alpha + \beta + \gamma = 1$$

Inside the triangle if all three coordinates are non-negative

漫反射

Recap: Lambertian (Diffuse) Term

Shading independent of view direction



energy arrived at the shading point

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

diffusely reflected light

diffuse coefficient (color)

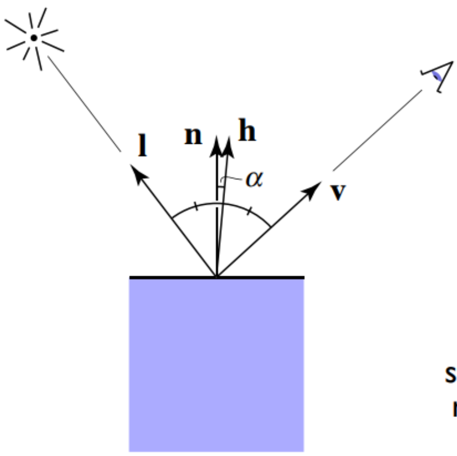
energy received by the shading point

镜面反射

Specular Term (Blinn-Phong)

V close to mirror direction ⇔ **half vector** near normal

- Measure "near" by dot product of unit vectors



$$\mathbf{h} = \mathrm{bisector}(\mathbf{v}, \mathbf{l}) = \frac{\mathbf{v}+\mathbf{l}}{\|\mathbf{v}+\mathbf{l}\|}$$

(半程向量)

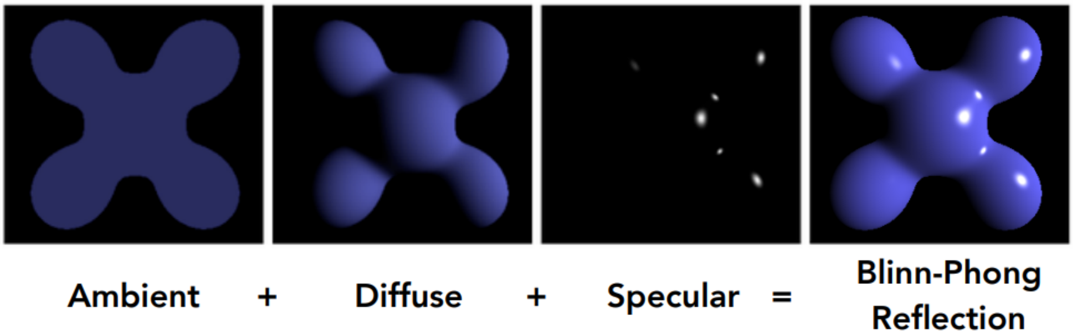
$$L_s = k_s\,(I/r^2)\max(0, \cos\alpha)^p = k_s\,(I/r^2)\max(0, \mathbf{n}\cdot\mathbf{h})^p$$

specularly reflected light

specular coefficient

向量是单位向量

Blinn-Phong Reflection Model



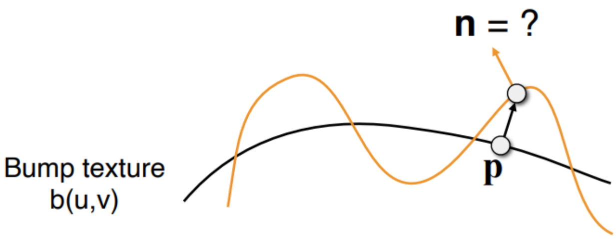
$$
\begin{aligned}
L &= L_a + L_d + L_s \\
&= k_a\, I_a + k_d\, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s\, (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{h})^p
\end{aligned}
$$

模拟出来的凹凸感

Bump Mapping

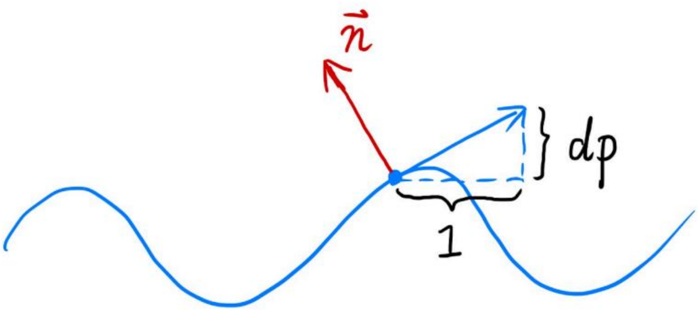
Adding surface detail without adding more triangles

- Perturb surface normal per pixel (for shading computations only)
- "Height shift" per texel defined by a texture
- How to modify normal vector?



How to perturb the normal (in flatland)

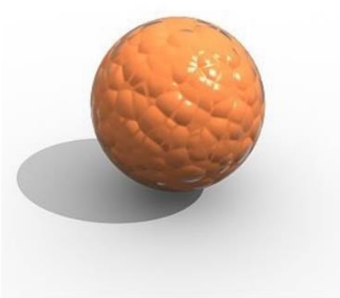
- Original surface normal n(p) = (0, 1)
- Derivative at p is dp = c * [h(p+1) - h(p)]
- Perturbed normal is then n(p) = **(-dp, 1)**.normalized()



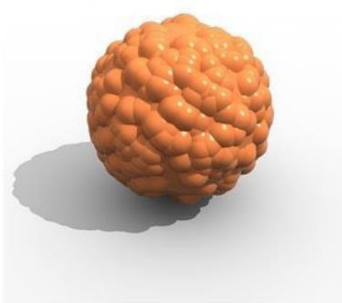
真实的凹凸感

Textures can affect shading!

- **Displacement mapping** — a more advanced approach
  - Uses the same texture as in bumping mapping
  - Actually **moves the vertices**



Bump / **Normal** mapping

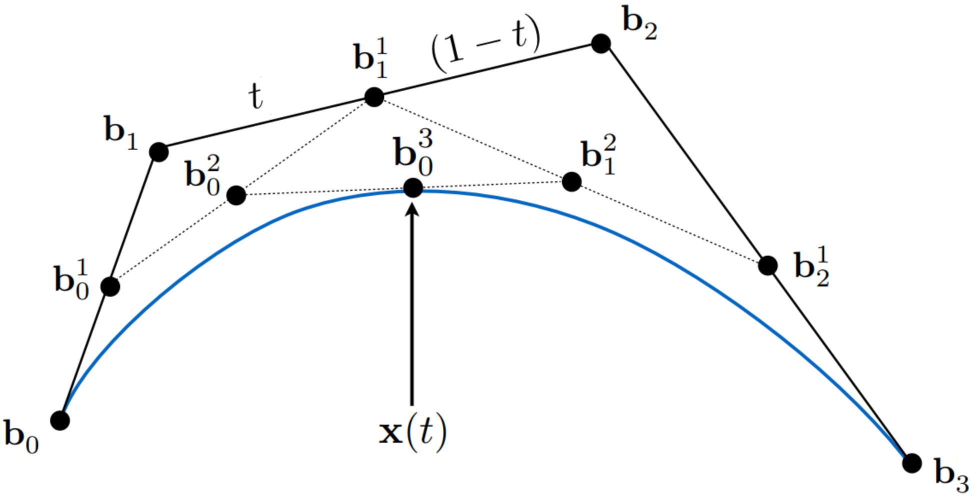


Displacement mapping

Cubic Bézier Curve – de Casteljau

Four input points in total

Same recursive linear interpolations



$$\mathbf{b}^n(t) = \mathbf{b}_0\,(1-t)^3 + \mathbf{b}_1\,3t(1-t)^2 + \mathbf{b}_2\,3t^2(1-t) + \mathbf{b}_3\,t^3$$

Bézier Curve – General Algebraic Formula

Bernstein form of a Bézier curve of order n:

$$\mathbf{b}^n(t) = \mathbf{b}_0^n(t) = \sum_{j=0}^{n} \mathbf{b}_j B_j^n(t)$$

Bézier curve order n

(vector polynomial of degree n)

Bernstein polynomial

(scalar polynomial of degree n)

Bézier control points

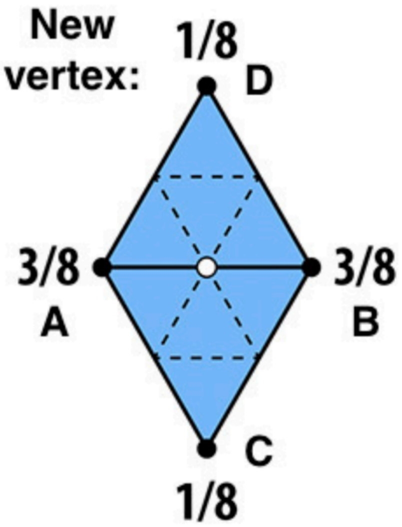
(vector in R^N)

Bernstein polynomials:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

Loop Subdivision — Update

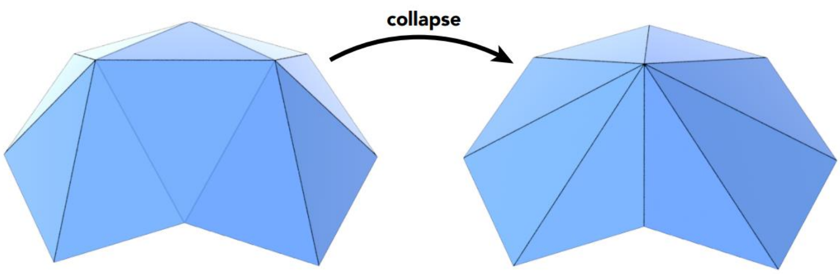
For new vertices:



Update to:
3/8 * (A + B) + 1/8 * (C + D)

Collapsing An Edge

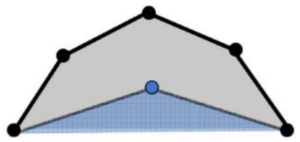
- Suppose we simplify a mesh using edge collapsing



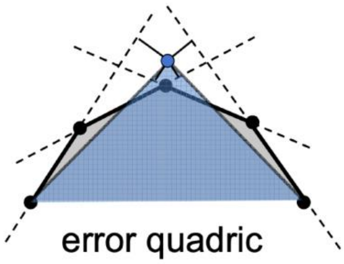
Quadric Error Metrics

（二次误差度量）

- How much geometric error is introduced by simplification?
- Not a good idea to perform local averaging of vertices
- Quadric error: new vertex should minimize its sum of square distance (L2 distance) to previously related triangle planes!



average

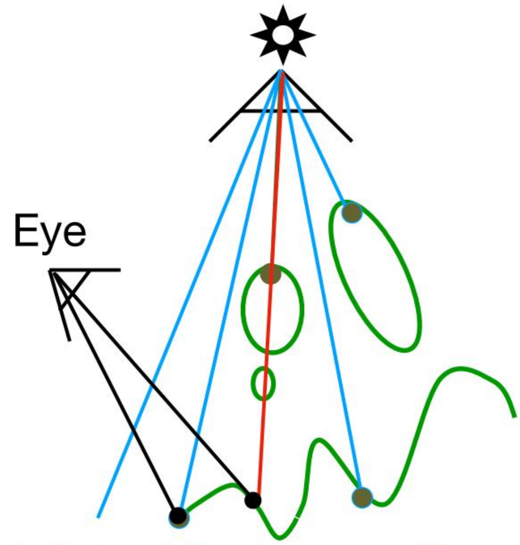


error quadric

shadow maps

Pass 2B: Project to light

- Project visible points in eye view back to light source



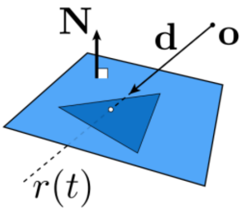
(Reprojected) depths from light and eye are not the same. BLOCKED!!

Ray Intersection With Triangle

Triangle is in a plane

- Ray-plane intersection
- Test if hit point is inside triangle

Many ways to optimize…



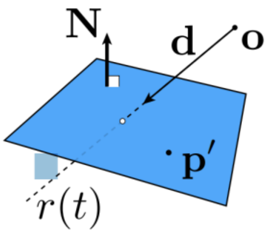
Ray Intersection With Plane

Ray equation:

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

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:** $0 \le t < \infty$

对于 AABB，N 是（0,0,1）等，能更简化的计算 t，所以使用 AABB

Möller Trumbore Algorithm

A faster approach, giving barycentric coordinate directly

Derivation in the discussion section!

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

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

Where:

$$\vec{\mathbf{E}}_1 = \vec{\mathbf{P}}_1 - \vec{\mathbf{P}}_0$$
$$\vec{\mathbf{E}}_2 = \vec{\mathbf{P}}_2 - \vec{\mathbf{P}}_0$$
$$\vec{\mathbf{S}} = \vec{\mathbf{O}} - \vec{\mathbf{P}}_0$$
$$\vec{\mathbf{S}}_1 = \vec{\mathbf{D}} \times \vec{\mathbf{E}}_2$$
$$\vec{\mathbf{S}}_2 = \vec{\mathbf{S}} \times \vec{\mathbf{E}}_1$$

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

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

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

Ray Intersection with Axis-Aligned Box

- 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, $t_{enter} = \mathbf{max}\{t_{min}\}$, $t_{exit} = \mathbf{min}\{t_{max}\}$
- If $t_{enter} < t_{exit}$, we know the ray **stays a while** in the box (so they must intersect!) (not done yet, see the next slide)

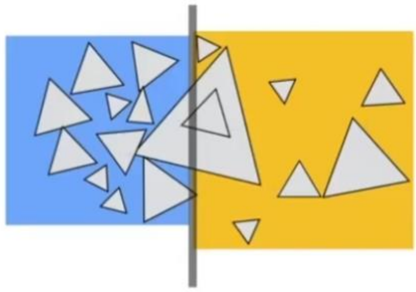
Ray Intersection with Axis-Aligned Box

- 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 $t_{exit} >= 0$ and $t_{enter} < 0$?
  - The ray’s origin is inside the box — have intersection!
- In summary, ray and AABB intersect iff
  - $t_{enter} < t_{exit}$ && $t_{exit} >= 0$

Spatial vs Object Partitions

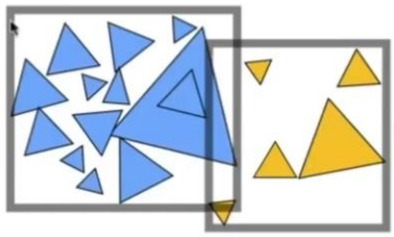
Spatial partition (e.g.KD-tree)

- Partition space into non-overlapping regions
- An object can be contained in multiple regions



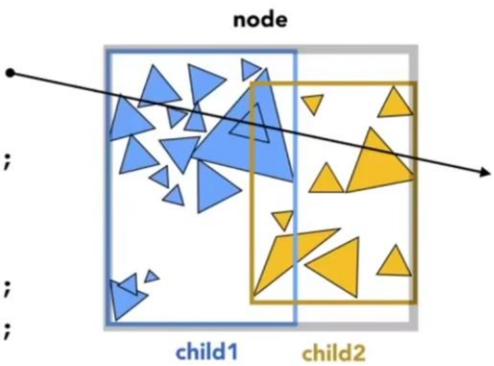
Object partition (e.g. BVH)

- Partition set of objects into disjoint subsets
- Bounding boxes for each set may overlap in space



BVH Traversal

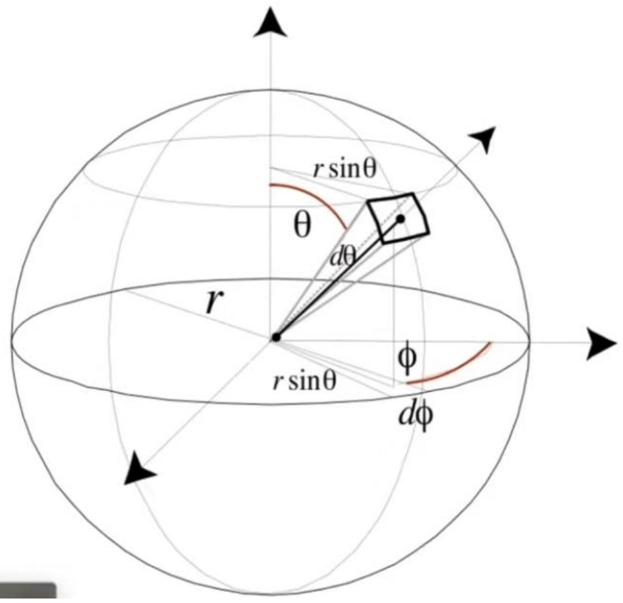
```
Intersect(Ray ray, BVH node) {
  if (ray misses node.bbox) return;

  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;
}
```



递归到最后 一个包围盒只包含一个三角形

先判断和包围盒相交，递归到最后再返回与三角形的交点，再取俩个交点距离小的

Differential Solid Angles

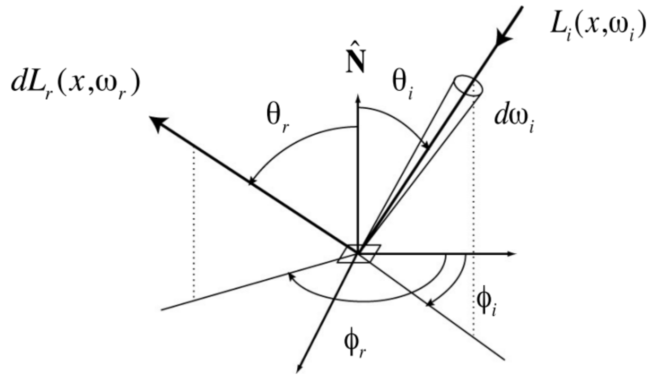


$$
\begin{aligned}
\mathrm{d}A &= (r\,\mathrm{d}\theta)(r\sin\theta\,\mathrm{d}\phi) \\
&= r^2\sin\theta\,\mathrm{d}\theta\,\mathrm{d}\phi
\end{aligned}
$$

$$
\mathrm{d}\omega = \frac{\mathrm{d}A}{r^2} = \sin\theta\,\mathrm{d}\theta\,\mathrm{d}\phi
$$

BRDF

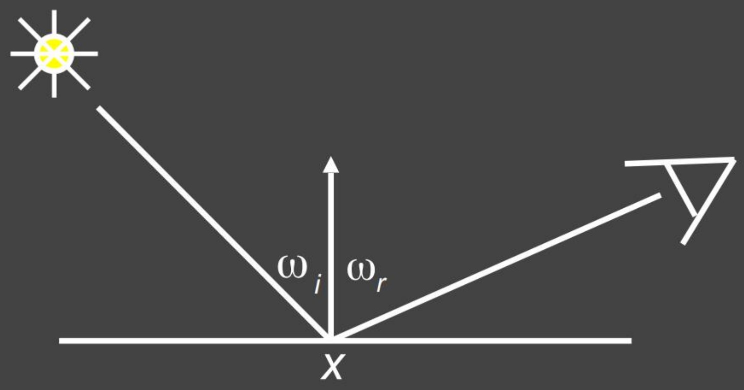
The Bidirectional Reflectance Distribution Function (BRDF) represents how much light is reflected into each outgoing direction ω from each incoming direction



$$
f_r(\omega_i \to \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)} = \frac{\mathrm{d}L_r(\omega_r)}{L_i(\omega_i)\cos\theta_i\,\mathrm{d}\omega_i} \quad \left[\frac{1}{\mathrm{sr}}\right]
$$

反射到一个半球，这里是观察其中一个角度

Reflection Equation



$$L_r(x,\omega_r) = L_e(x,\omega_r) + L_i(x,\omega_i)\ f(x,\omega_i,\omega_r)\ (\omega_i, n)$$

Reflected Light (Output Image) | Emission | Incident Light (from light source) | BRDF | Cosine of Incident angle

Ray Tracing

$$L = E + KE + K^2E + K^3E + \ldots$$

Emission directly
From light sources

Direct Illumination
on surfaces

Indirect Illumination
(One bounce indirect)
[Mirrors, Refraction]

(Two bounce indirect illum.)

Shading in
Rasterization

Motivation: Whitted-Style Ray Tracing

Whitted-style ray tracing:

- Always perform specular reflections / refractions
- Stop bouncing at diffuse surfaces

Whitted Style 只考虑了镜面反射和折射

Whitted-Style Ray Tracing is Wrong

But the rendering equation is correct

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

But it involves

- Solving an integral over the hemisphere, and
- Recursive execution

渲染方程正确，但是需要求解半球上的积分，以及递归执行

所以引入概率，变成每个像素多次采样求平均

A Simple Monte Carlo Solution

We want to compute the radiance at p towards the camera

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

Monte Carlo integration: $\displaystyle\int_a^b f(x)\,\mathrm{d}x \approx \frac{1}{N}\sum_{k=1}^{N} \frac{f(X_k)}{p(X_k)} \qquad X_k \sim p(x)$

What's our "f(x)"? $\quad L_i(p, \omega_i) f_r(p, \omega_i, \omega_o)(n \cdot \omega_i)$

What's our pdf? $\quad p(\omega_i) = 1/2\pi$

(assume uniformly sampling the hemisphere)

Path Tracing

From now on, we always assume that
only 1 ray is traced at each shading point:

```
shade(p, wo)
    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)
    Else If ray r hit an object at q
        Return shade(q, -wi) * f_r * cosine / pdf(wi)
```

This is **path tracing**! (FYI, Distributed Ray Tracing if N != **1**)

1 ray 是为了防止反射时 ray 数量成指数增长，
但是依然存在问题，shade（）递归不会停止，所以引入俄罗斯轮盘赌

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

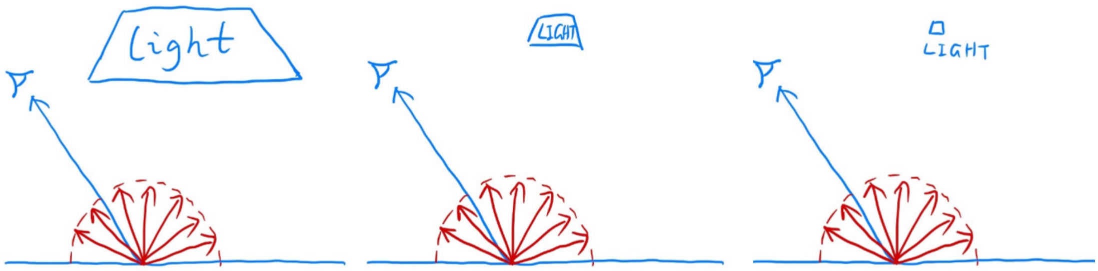
Solution: Russian Roulette (RR)

```
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
```

要么光直射，要么物体反射

Sampling the Light

Understanding the reason of being inefficient



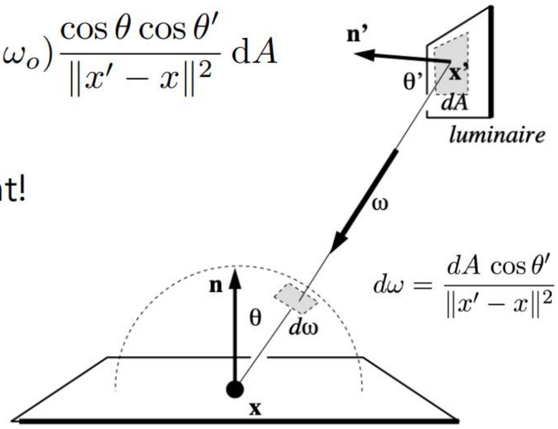
Every 5 rays, 500 rays, 50000 rays,

there will be 1 ray hitting the light. So **a lot of rays are "wasted"** if we uniformly sample the hemisphere at the shading point.

对半球 ray 采样，但是光源面积小导致 ray 浪费，
所以第一部分光源直射改成对光源采样

Then we can rewrite the rendering equation as

$$L_o(x,\omega_o) = \int_{\Omega^+} L_i(x,\omega_i) f_r(x,\omega_i,\omega_o) \cos\theta \, \mathrm{d}\omega_i$$
$$= \int_A L_i(x,\omega_i) f_r(x,\omega_i,\omega_o) \frac{\cos\theta \cos\theta'}{\|x'-x\|^2} \, \mathrm{d}A$$



Now an integration on the light!

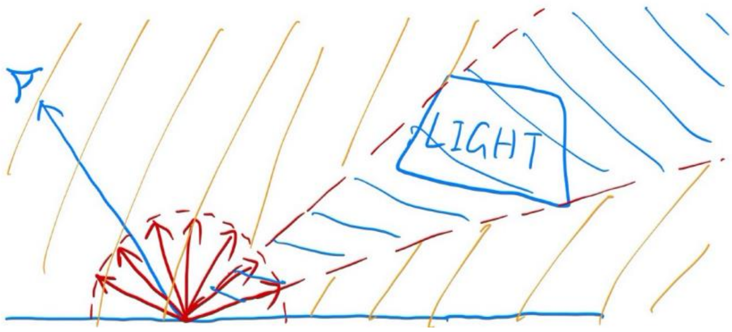
Monte Carlo integration:

"f(x)": everything inside

Pdf: 1 / A

Now we consider the radiance coming from two parts:

1. light source (direct, no need to have RR)
2. other reflectors (indirect, RR)

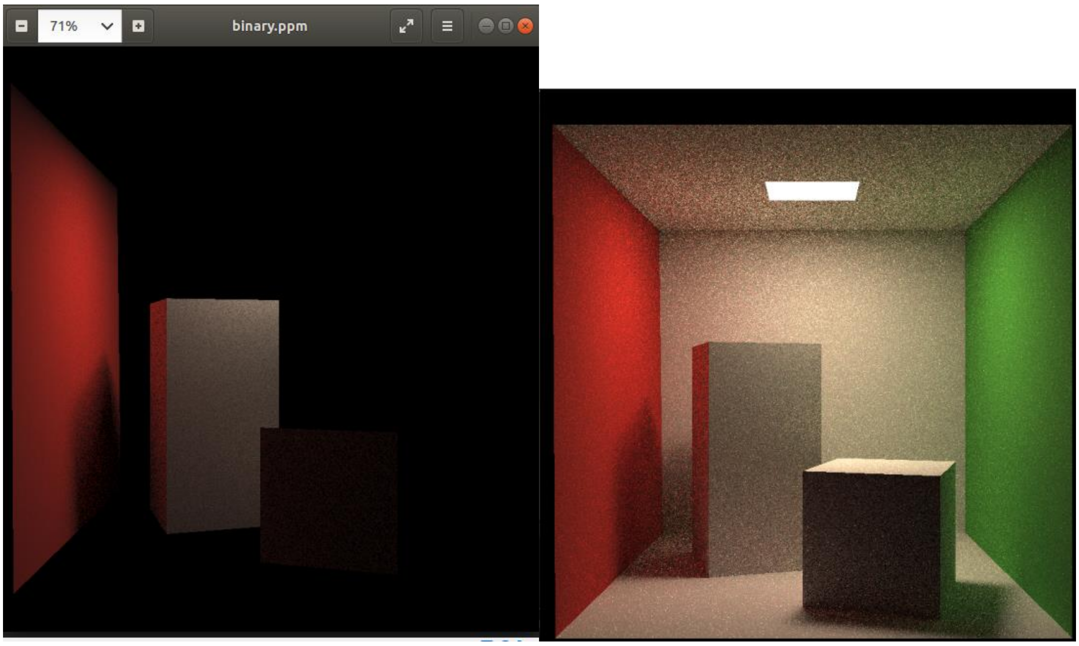


so:

Sampling the Light

```
shade(p, wo)
    # Contribution from the light source.
    Uniformly sample the light at x' (pdf_light = 1 / A)
    L_dir = L_i * f_r * cos θ * cos θ' / |x' - p|^2 / pdf_light

    # Contribution from other reflectors.
    L_indir = 0.0
    Test Russian Roulette with probability P_RR
    Uniformly sample the hemisphere toward wi (pdf_hemi = 1 / 2pi)
    Trace a ray r(p, wi)
    If ray r hit a non-emitting object at q
        L_indir = shade(q, -wi) * f_r * cos θ / pdf_hemi / P_RR

    Return L_dir + L_indir
```

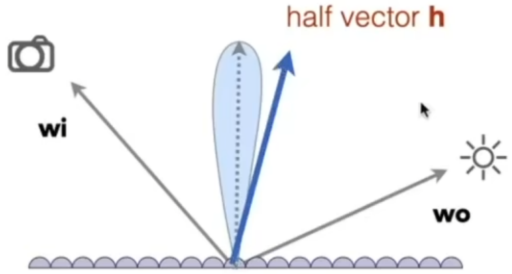
先对光源采样，
再俄罗斯轮盘赌，
对半球内物体采样反射光



作业七中第一次得到的结果发现大片的黑色，检查以后才明白是在求解光与场景的交点，使用 BVH，当 t_exit=t_enter 时，错误的认为是光线与 AABB 相切，而认为不相交
其实此时的 AABB 是一个平面，应当返回相交从而进一步求交点

Microfacet BRDF

- What kind of microfacets reflect wi to wo?
  (hint: microfacets are mirrors)



$$f(\mathbf{i},\mathbf{o}) = \frac{\overset{\text{Fresnel term}}{\boxed{\mathbf{F}(\mathbf{i},\mathbf{h})}}\,\overset{\text{shadowing-masking term}}{\boxed{\mathbf{G}(\mathbf{i},\mathbf{o},\mathbf{h})}}\,\overset{\textbf{distribution of normals}}{\boxed{\mathbf{D}(\mathbf{h})}}}{4(\mathbf{n},\mathbf{i})(\mathbf{n},\mathbf{o})}$$